Deep C++

by Olve Maudal

Programming is hard. Programming correct C++ is particularly hard. Indeed, it is uncommon to see a screenful containing only well defined and conforming code. Why do professional programmers write code like this? Because most programmers do not have a deep understanding of the language they are using. While they sometimes know that certain things are undefined or unspecified, they often do not know why it is so. In this presentation we will study small code snippets in C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of this wonderful but dangerous programming language.

Half day lecture for students at NITH in Oslo
Monday, April 22, 2013

JAWS

Programming is hard

Programming is hard

A quick motivation for seeking a deep understanding of C++

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
```

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
```

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
```

Are you ready? Have you written it down?

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
2
```

Are you ready? Have you written it down?

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
2
5
```

Are you ready? Have you written it down?

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
2
5
6
```

Are you ready? Have you written it down?

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
2
5
6
$
```

Are you ready? Have you written it down?

Exercise

What does the following code snippet print?
Write the output on a piece of paper.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
$ c++ foo.cpp
$ ./a.out
2
5
6
$
```

Are you ready? Have you written it down?

It is very difficult to work with C++ if you don't understand exactly why you get this particular result.

Here is a very brief explanation.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
2
5
6
```

Here is a very brief explanation.

A string literal is of type “char *”, when looking for a good match of type, the compiler selects a type that just needs a simple type conversion rather than an implicit construction.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

```
2
5
6
```

Here is a very brief explanation.

A string literal is of type “char *”, when looking for a good match of type, the compiler selects a type that just needs a simple type conversion rather than an implicit construction.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

A for loop is just a convenient way of writing a while loop. For this loop you can think:

```
int i = 5; while(i<7) { std::cout << i << std::endl ; ++i };
```

```
2
5
6
```

Here is a very brief explanation.

A string literal is of type "char *", when looking for a good match of type, the compiler selects a type that just needs a simple type conversion rather than an implicit construction.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

A for loop is just a convenient way of writing a while loop. For this loop you can think:

```
int i = 5; while(i<7) { std::cout << i << std::endl ; ++i };
```

When comparing signed and unsigned integers of the same rank, the unsigned "wins" when one of the values needs a type conversion. In this case -1 is converted to a very big number.

```
2
5
6
```

Here is a very brief explanation.

```
#include <iostream>
#include <string>

void f(const std::string&) { std::cout << 1 << std::endl; }
void f(const void*) { std::cout << 2 << std::endl; }

int main()
{
    f("Hello");
    for (int i=5; i<7; ++i)
        std::cout << i << std::endl;
    unsigned int j = 42;
    if (j > -1)
        std::cout << 9 << std::endl;
}
```

A string literal is of type “char *”, when looking for a good match of type, the compiler selects a type that just needs a simple type conversion rather than an implicit construction.

A for loop is just a convenient way of writing a while loop. For this loop you can think:
`int i = 5; while(i<7) { std::cout << i << std::endl ; ++i };`

When comparing signed and unsigned integers of the same rank, the unsigned “wins” when one of the values needs a type conversion. In this case -1 is converted to a very big number.

```
2
5
6
```

Was your answer correct? Did you know all of this?

There are hundreds of “strange” cases like this in C++. A lot of things looks “strange” in C++ before you have a deep insight into the language. But as you learn more about the history, spirit and motivation for the C++ language, you gradually improve your conceptual model of the language so it becomes easier to really understand what is going on.

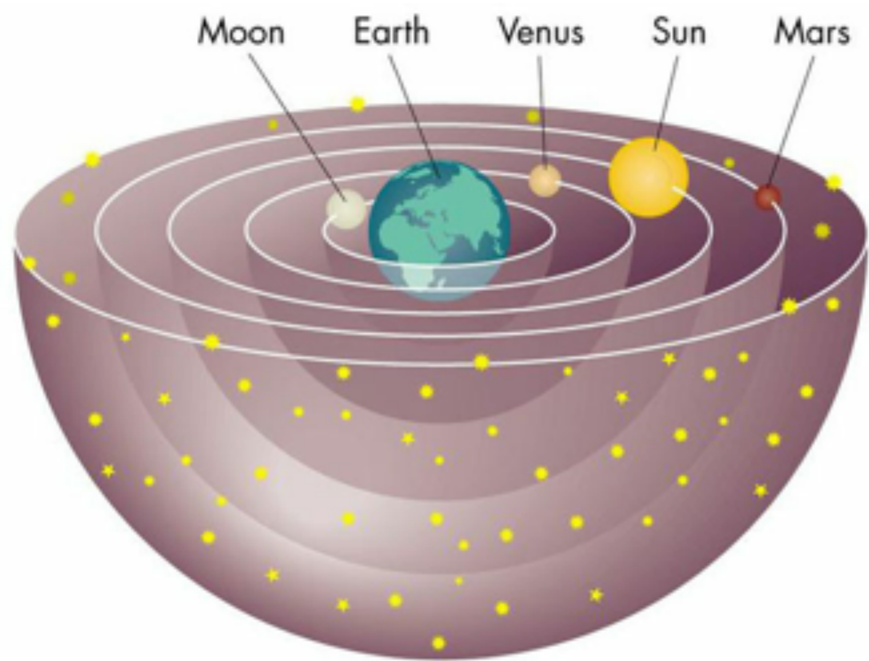
There are hundreds of “strange” cases like this in C++. A lot of things looks “strange” in C++ before you have a deep insight into the language. But as you learn more about the history, spirit and motivation for the C++ language, you gradually improve your conceptual model of the language so it becomes easier to really understand what is going on.

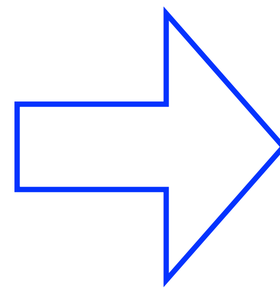
Even very experienced C++ programmers get a bit confused when presented code like in the first exercise. The big difference between mediocre C++ programmers and really good C++ programmers is that the latter category knows very well that they do **not** have a complete understanding of C++ and therefore they are always a bit cautious. Also, really good C++ programmers always try to seek a deeper understanding of the language. You can learn something new about C++ every day if you have the right attitude.

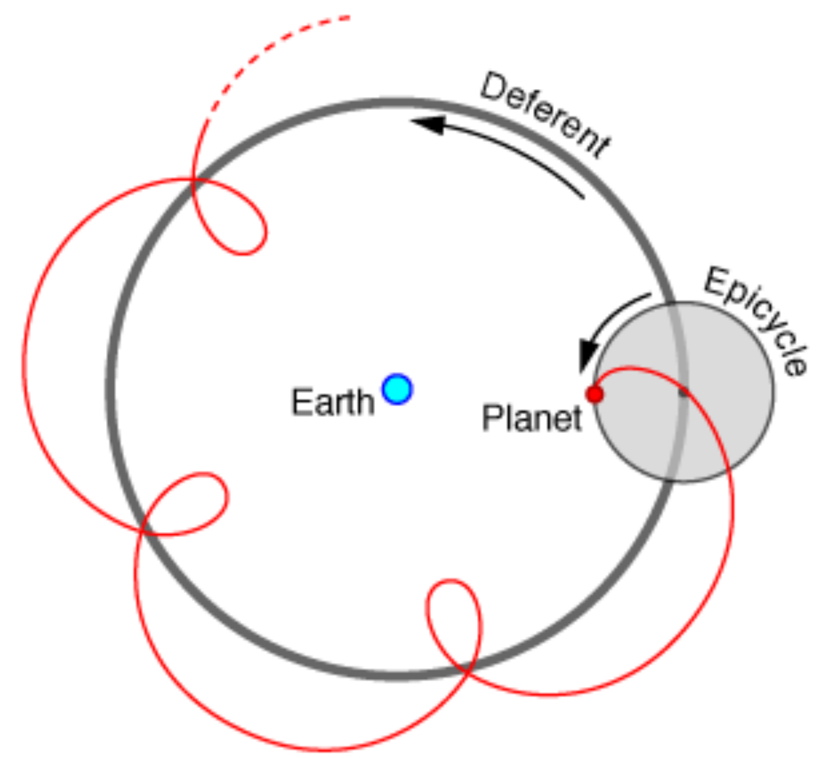
There are hundreds of "strange" cases like this in C++. A lot of things looks "strange" in C++ before you have a deep insight into the language. But as you learn more about the history, spirit and motivation for the C++ language, you gradually improve your conceptual model of the language so it becomes easier to really understand what is going on.

Even very experienced C++ programmers get a bit confused when presented code like in the first exercise. The big difference between mediocre C++ programmers and really good C++ programmers is that the latter category knows very well that they do **not** have a complete understanding of C++ and therefore they are always a bit cautious. Also, really good C++ programmers always try to seek a deeper understanding of the language. You can learn something new about C++ every day if you have the right attitude.

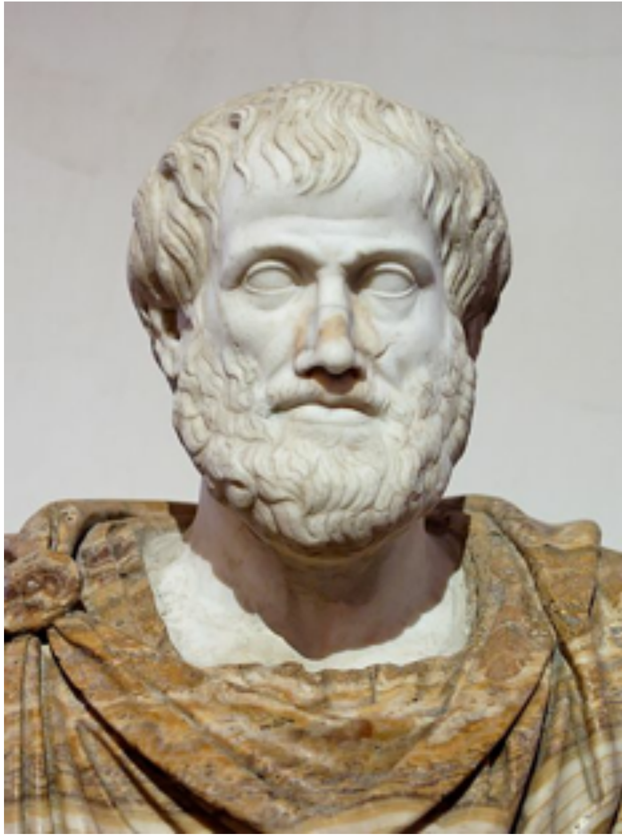
It is common to meet programmers that are convinced that they really understand C++, but where their strange explanations on even very simple concepts reveals that they are using an invalid conceptual model to reach their conclusions. This reminds me very much about...

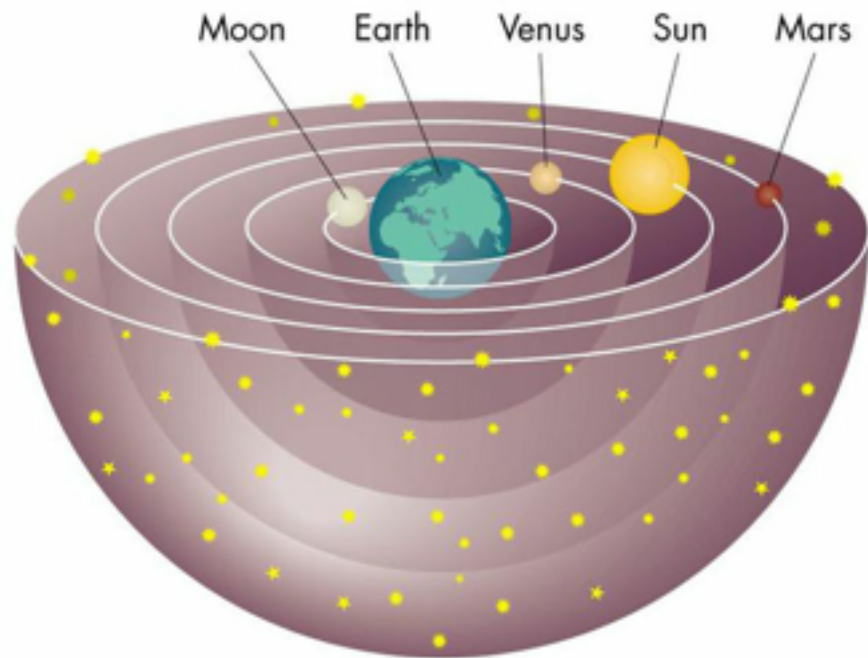

Moon
Earth
Venus
Sun
Mars




Deferent
Epicycle
Earth
Planet

Aristotle (384 BC – 322 BC)

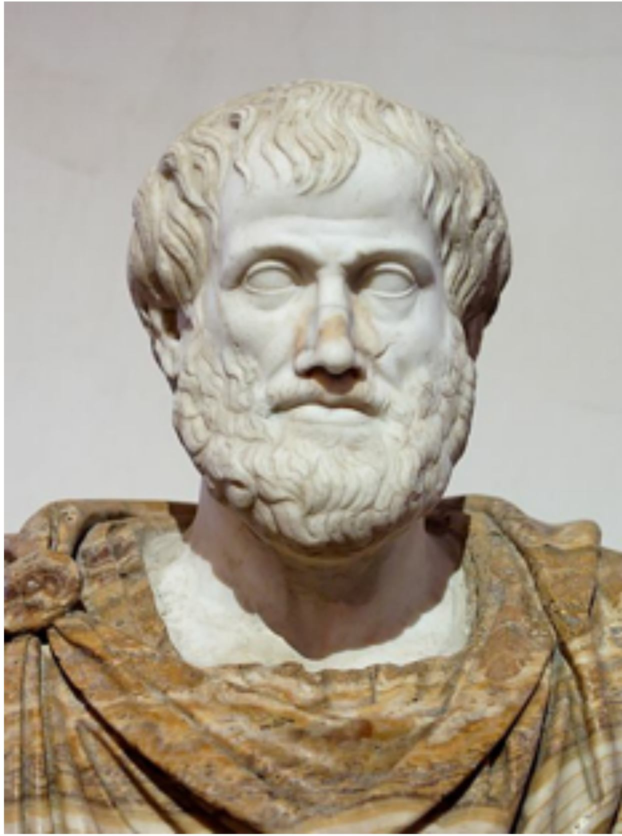


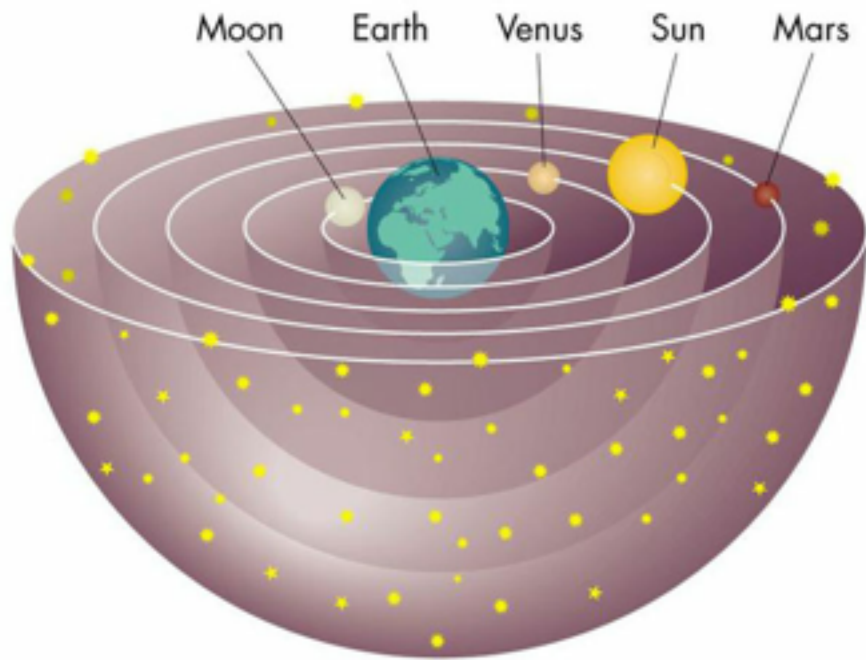


Aristotles Universe

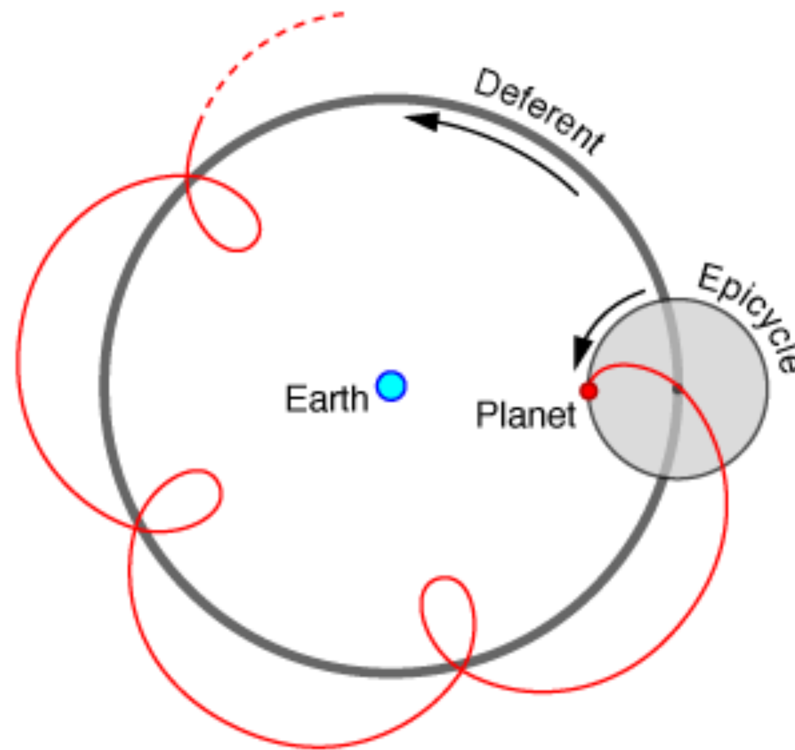
Aristotle (384 BC – 322 BC)

Ptolemy (90 AD – 168 AD)



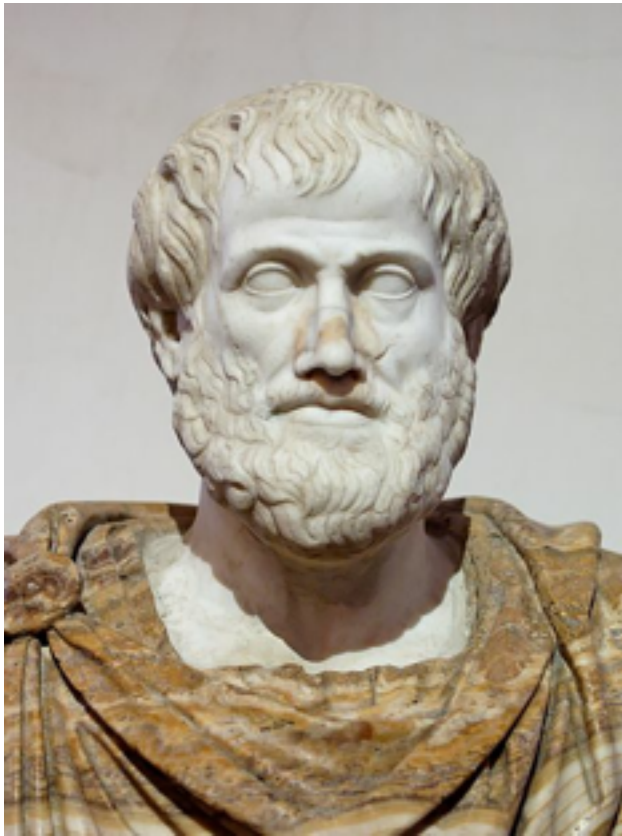


Aristotles Universe



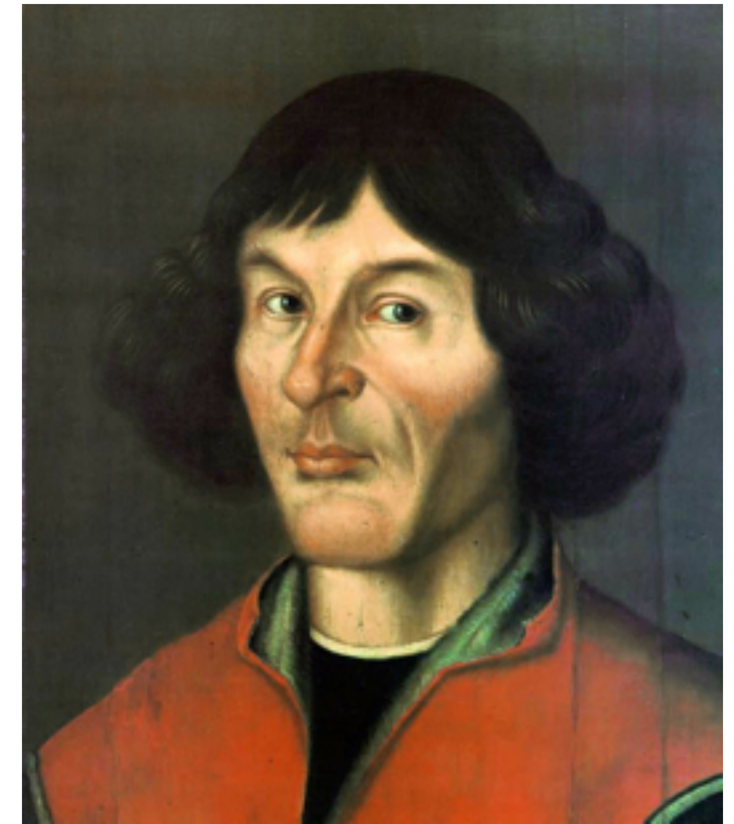
Ptolemy Universe

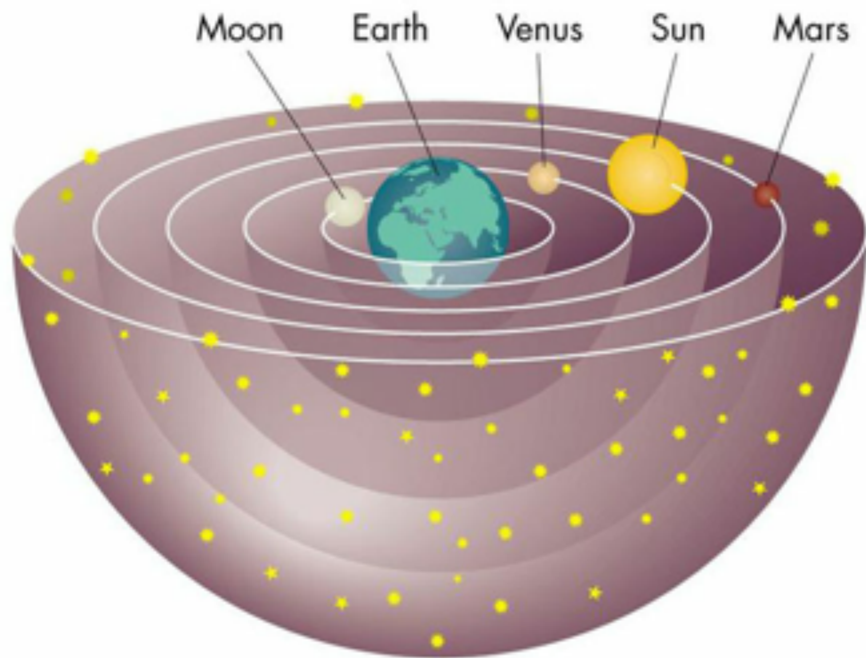
Aristotle (384 BC – 322 BC)



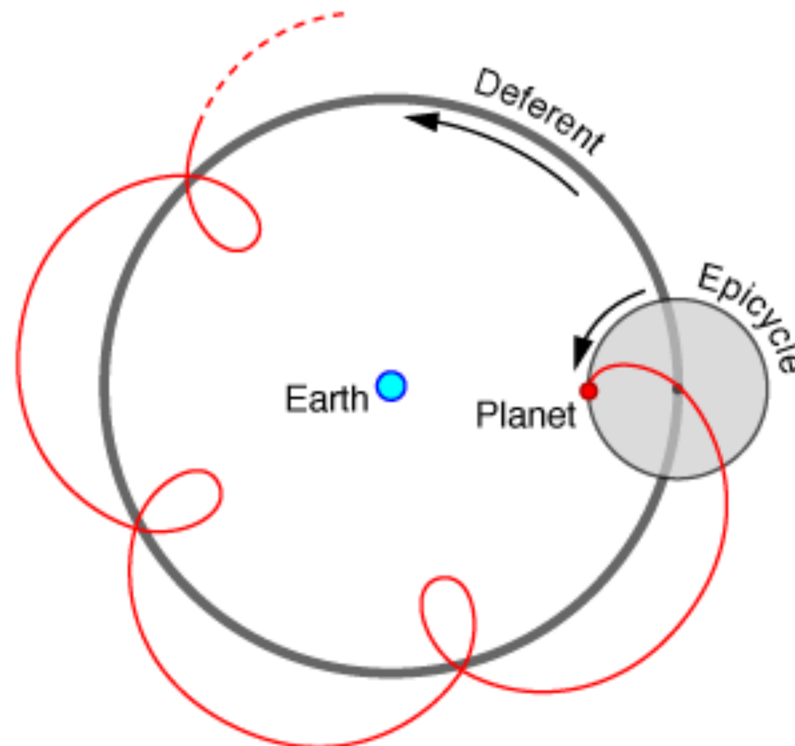
Ptolemy (90 AD – 168 AD)

Copernicus (1473 – 1543)

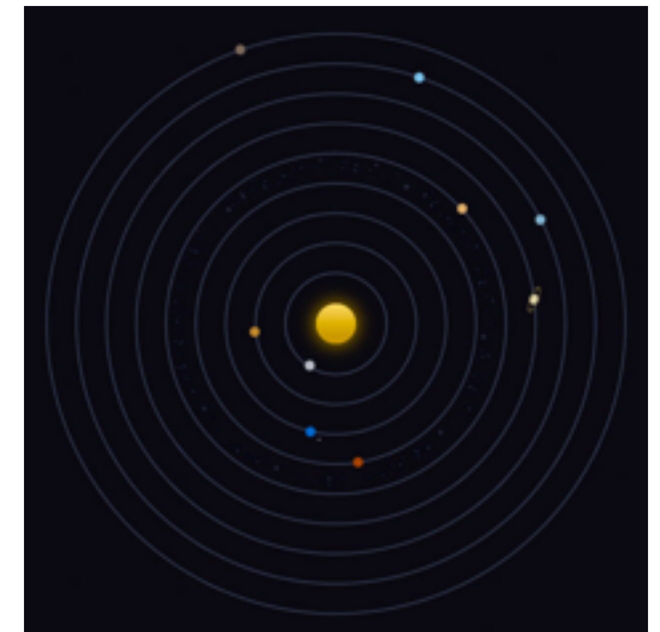




Aristotles Universe

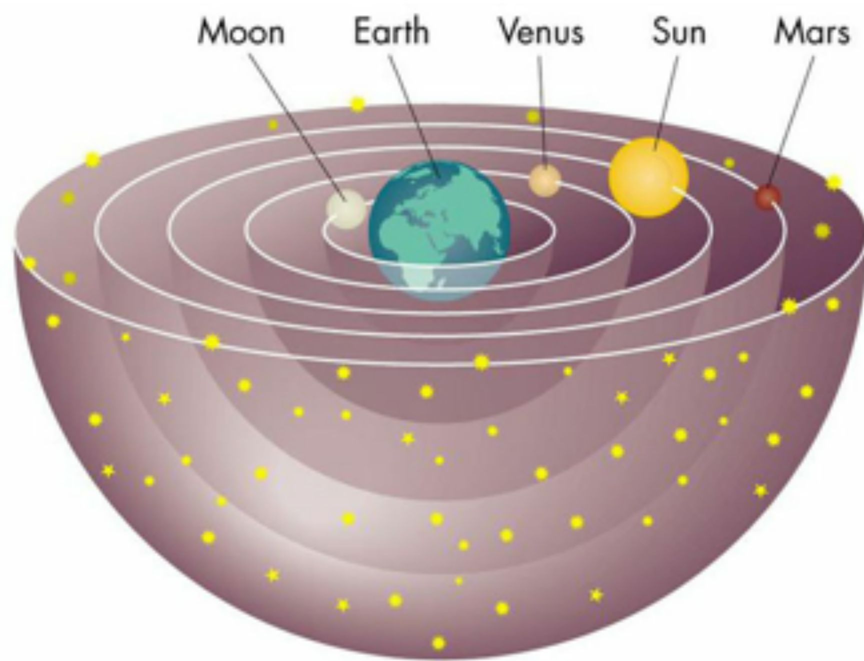


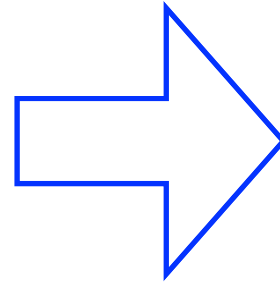
Ptolemy Universe

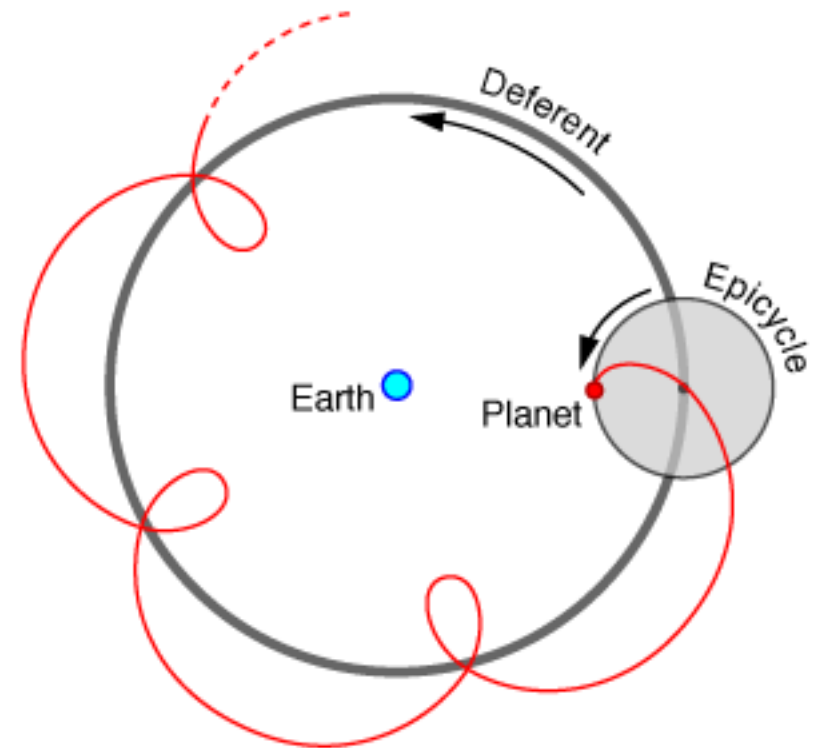


The Solar System

Strange explanations are often symptoms of having an invalid conceptual model!







About Behavior

About Behavior

Let’s look at some basic stuff first

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
```

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
```

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
```

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

```cpp
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
```

```cpp
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
```

```
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
```

```cpp
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
5
```

```
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
5
6
```

```
#include <iostream>

void foo()
{
    static int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
5
6
```

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

garbage, garbage, garbage?

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

garbage, garbage, garbage?

It is better to initialize explicitly.

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
```

garbage, garbage, garbage?

It is better to initialize explicitly.

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
```

garbage, garbage, garbage?

It is better to initialize explicitly.

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
1
```

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
1
2
```

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

void foo()
{
    static int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No. In C++, variables with static storage duration are initialized to their default value, in this case 0

I agree, in this case. But you still need to know that it is so. And it is very useful to know *why* it is so

```
$ c++ foo.cpp
$ ./a.out
1
2
3
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

1, 1, 1?

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
```

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
$ ./a.out
```

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
$ ./a.out
1
```

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
$ ./a.out
1
2
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
$ ./a.out
1
2
3
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

```
$ c++ foo.cpp
$ ./a.out
1
2
3
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

No, variables with automatic storage duration is not initialized implicitly

Maybe. Let's try it on my machine.

any plausible explaination for this behavior?

```
$ c++ foo.cpp
$ ./a.out
1
2
3
```

```cpp
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

This is **undefined behavior**!

First of all, it is important to understand that, you might as well get:

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
[FORMATTING HD]
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
[FORMATTING HD]
```

or even

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
[FORMATTING HD]
```

or even

```
$ c++ foo.cpp
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
[FORMATTING HD]
```

or even

```
$ c++ foo.cpp
[FORMATTING HD]
```

This is **undefined behavior**!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

First of all, it is important to understand that, you might as well get:

```
$ c++ foo.cpp
$ ./a.out
Happy birthday!
```

or

```
$ c++ foo.cpp
$ ./a.out
[FORMATTING HD]
```

or even

```
$ c++ foo.cpp
[FORMATTING HD]
```

“When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose” (from comp.std.c)

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

OK, let’s add some flags

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

OK, let’s add some flags

```
$ c++ -Wall -Wextra foo.cpp
```

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

OK, let’s add some flags

```
$ c++ -Wall -Wextra foo.cpp
$ ./a.out
```

I don't need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -Wall -Wextra foo.cpp
$ ./a.out
1
```

OK, let's add some flags

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -Wall -Wextra foo.cpp
$ ./a.out
1
2
```

OK, let’s add some flags

I don’t need to know about this, because my compiler find bugs like this

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -Wall -Wextra foo.cpp
$ ./a.out
1
2
3
```

OK, let’s add some flags

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -Wall -Wextra foo.cpp
$ ./a.out
1
2
3
```

I don't need to know about this, because my compiler find bugs like this

Lousy compiler!

OK, let's add some flags

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
```

Pro tip:
Compile with
optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
```

Pro tip:
Compile with
optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
foo.cpp:6: warning: 'a' is used
uninitialized in this function
```

Pro tip: Compile with optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
foo.cpp:6: warning: 'a' is used
uninitialized in this function
42
```

Pro tip:
Compile with
optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
foo.cpp:6: warning: 'a' is used
uninitialized in this function
42
-1
```

Pro tip:
Compile with
optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
foo.cpp:6: warning: 'a' is used
uninitialized in this function
42
-1
38911
```

Pro tip:
Compile with
optimization!

```
#include <iostream>

void foo()
{
    int a;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ -O -Wall -Wextra foo.cpp
foo.cpp: In function 'void foo()':
foo.cpp:6: warning: 'a' is used
uninitialized in this function
42
-1
38911
$
```

In C++. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?

In C++. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?

Because C++ is a braindead programming language?

In C++. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?

Because C++ is a braindead programming language?

Because C++ (and C) is all about execution speed. Setting static variables to default values is a one time cost, while defaulting auto variables is a signficant runtime cost.

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
```

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
```

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

```
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

```
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

```
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

or

```cpp
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

```
#include <iostream>

int foo(int a) {
    std::cout << a;
    return a;
}

int bar(int a, int b) {
    return a + b;
}

int main() {
    int i = foo(3) + foo(4);
    std::cout << i << std::endl;

    int j = bar(foo(3), foo(4));
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
347
437
```

but you might also get

```
437
347
```

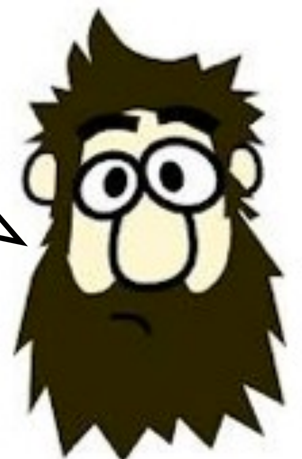
or

```
437
437
```

or

```
347
347
```

C and C++ are among the few programming languages where evaluation order is *mostly* unspecified. This is an example of **unspecified behaviour.**



In C++. Why is the evaluation order mostly unspecified?

In C++. Why is the evaluation order mostly unspecified?

In C++. Why is the evaluation order mostly unspecified?

Because C++ is a braindead programming language?

In C++. Why is the evaluation order mostly unspecified?

Because C++ is a braindead programming language?

Because there is a design goal to allow optimal execution speed on a wide range of architectures. In C++ the compiler can choose to evaluate expressions in the order that is most optimal for a particular platform. This allows for better optimization.

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
```

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
42
```

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

What? Inconceivable!

```
$ c++ foo.cpp && ./a.out
42
```

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

What? Inconceivable!

```
$ c++ foo.cpp && ./a.out
42
```

This is a typical example of **undefined behaviour**. Anything can happen! Nasal demons can start flying out of your nose!

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

What? Inconceivable!

```
$ c++ foo.cpp && ./a.out
42
```

I agree this is crap code, but why is it wrong?

This is a typical example of **undefined behaviour**. Anything can happen! Nasal demons can start flying out of your nose!

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
42
```

This is a typical example of **undefined behaviour**. Anything can happen! Nasal demons can start flying out of your nose!

I agree this is crap code, but why is it wrong?

In this case? Line 6. What is i*3? Is it 2*3 or 3*3 or something else? In C++ you can not assume anything about a variable with side-effects (here i++) before there is a **sequence point**.

```cpp
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
42
```

I don't care, I never write code like that.

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

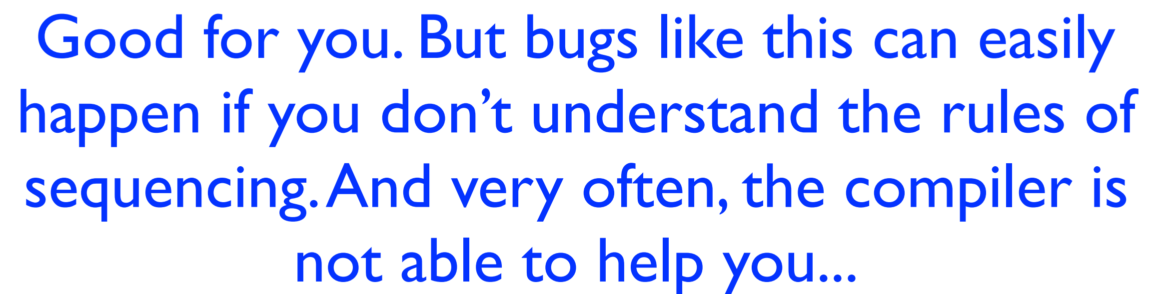
```
$ c++ foo.cpp && ./a.out
42
```

I don't care, I never write code like that.

Good for you. But bugs like this can easily happen if you don't understand the rules of sequencing. And very often, the compiler is not able to help you...

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
42
```



But why do we not get warning on this by default?

```
#include <iostream>

int main() {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    std::cout << j << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
42
```

Good for you. But bugs like this can easily happen if you don't understand the rules of sequencing. And very often, the compiler is not able to help you...

But why do we not get warning on this by default?

At least two reasons. First of all it is often very hard to detect such sequencing violations. Secondly, there is so much existing code out there that breaks these rules, so issuing warnings here might cause other problems.

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
12
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
12
$ clang++ foo.cpp && ./a.out
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
12
$ clang++ foo.cpp && ./a.out
11
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
12
$ clang++ foo.cpp && ./a.out
11
$ icc foo.cpp && ./a.out
```

Exercise

This code is **undefined behavior**, but what do you think actually happens if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ foo.cpp && ./a.out
12
$ clang++ foo.cpp && ./a.out
11
$ icc foo.cpp && ./a.out
13
```

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
12
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang++ -O -Wall -Wextra -pedantic foo.c && ./a.out
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang++ -O -Wall -Wextra -pedantic foo.c && ./a.out
11
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang++ -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -O -Wall -Wextra -pedantic foo.c && ./a.out
```

It is always a good idea to add some flags for better diagnostics.

foo.c

```
#include <iostream>

int main()
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    std::cout << n << std::endl;
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ g++ -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang++ -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

What do these code snippets print?

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

2

```
int a=41; a++ & printf("%d\n", a);
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

2

```
int a=41; a++ & printf("%d\n", a);
```

3

```
int a=41; a++ && printf("%d\n", a);
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

2

```
int a=41; a++ & printf("%d\n", a);
```

3

```
int a=41; a++ && printf("%d\n", a);
```

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

2

```
int a=41; a++ & printf("%d\n", a);
```

3

```
int a=41; a++ && printf("%d\n", a);
```

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

5

```
int a=41; a = a++; printf("%d\n", a);
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

2

```
int a=41; a++ & printf("%d\n", a);
```

3

```
int a=41; a++ && printf("%d\n", a);
```

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

5

```
int a=41; a = a++; printf("%d\n", a);
```

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

3

```
int a=41; a++ && printf("%d\n", a);
```

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

5

```
int a=41; a = a++; printf("%d\n", a);
```

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

5

```
int a=41; a = a++; printf("%d\n", a);
```

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

5

```
int a=41; a = a++; printf("%d\n", a);
```

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

42

5

```
int a=41; a = a++; printf("%d\n", a);
```

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

42

5

```
int a=41; a = a++; printf("%d\n", a);
```

undefined

6

```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

42

5

```
int a=41; a = a++; printf("%d\n", a);
```

undefined

6

```
int a=41; a = foo(a++); printf("42\n");
```

?

What do these code snippets print?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

undefined

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

42

5

```
int a=41; a = a++; printf("%d\n", a);
```

undefined

6

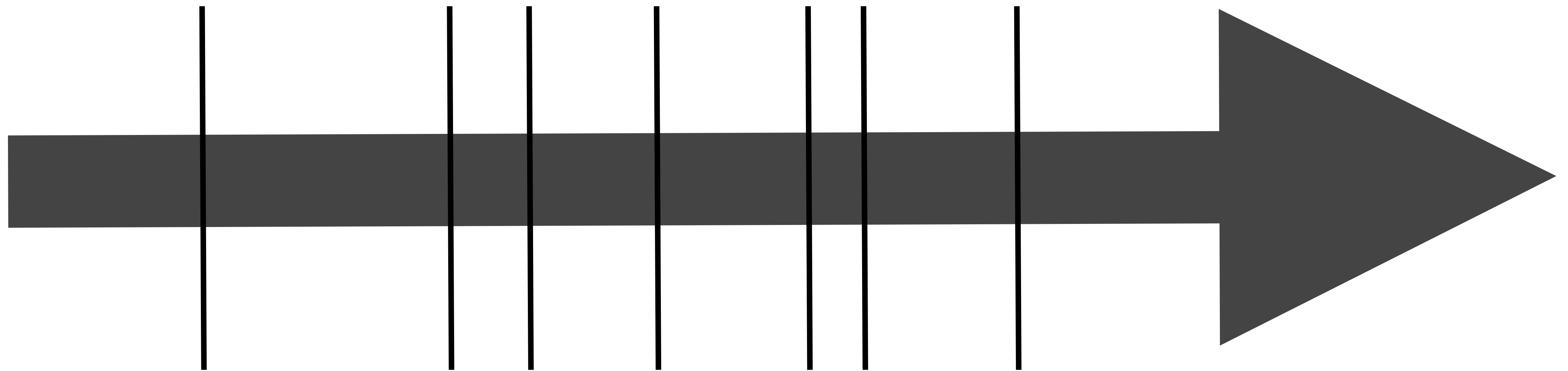
```
int a=41; a = foo(a++); printf("42\n");
```

?

__When__ exactly do side-effects take place in C and C++?

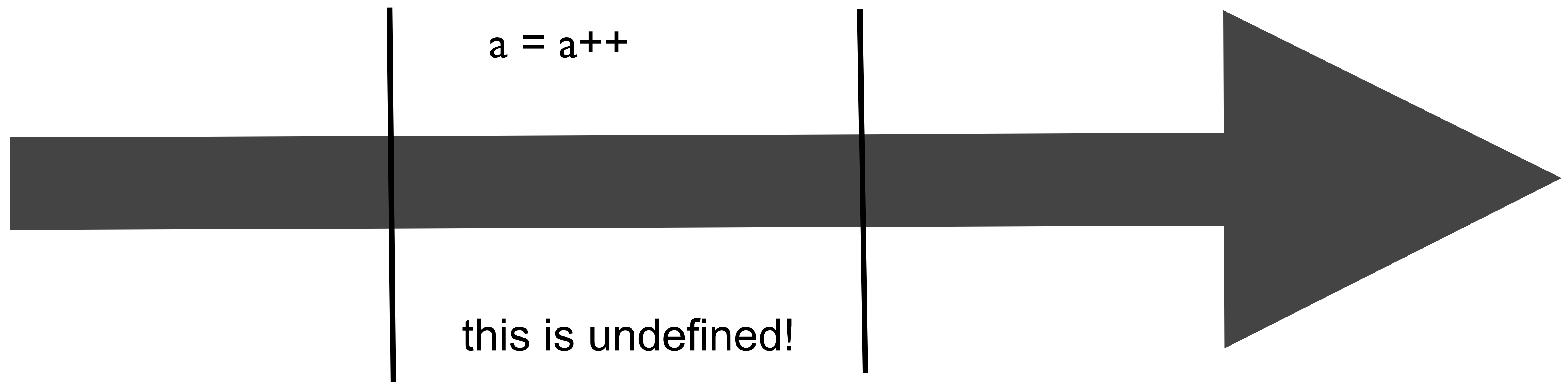
Sequence Points

A sequence point is a point in the program's execution sequence where all previous side-effects *shall* have taken place and where all subsequent side-effects *shall not* have taken place



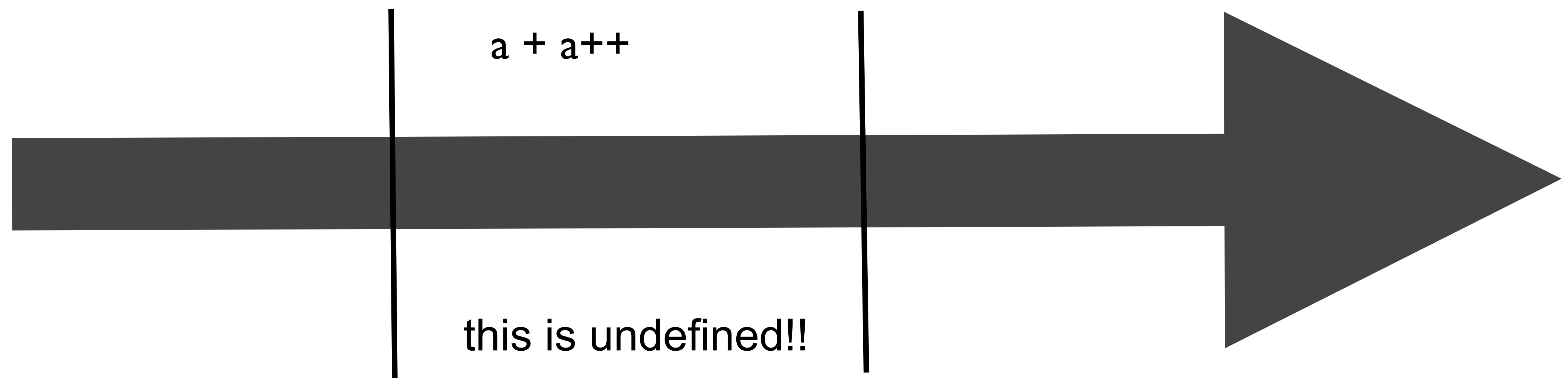
Sequence Points - Rule 1

Between the previous and next sequence point an object *shall* have its stored value modified at most once by the evaluation of an expression.



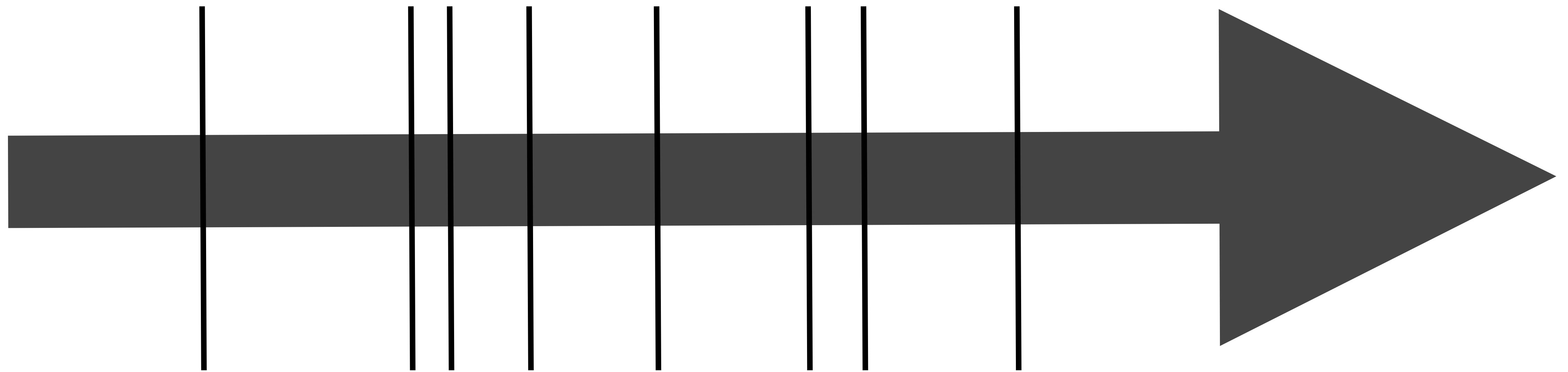
Sequence Points - Rule 2

Furthermore, the prior value *shall* be read only to determine the value to be stored.



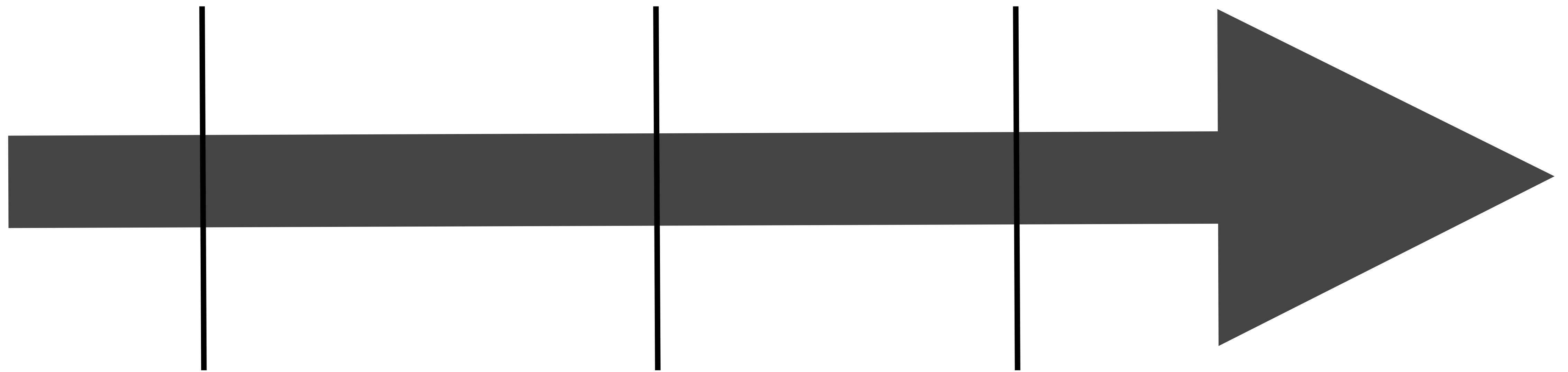
Sequence Points

A lot of developers think C has *many* sequence points



Sequence Points

The reality is that C has very *few* sequence points.



This helps to maximize optimization opportunities for the compiler.

Sequence points in C++

1) At the end of a full expression there is a sequence point.

```
a = i++;
++i;
if (++i == 42) { ... }
```

2) In a function call, there is a sequence point after the evaluation of the arguments, but before the actual call.

```
foo(++i)
```

3) The logical and (&&) and logical or (| |) guarantees a left-to-right evaluation, and if the second operand is evaluated, there is a sequence point between the evaluation of the first and second operands.

```
if (p && *p++ == 42) { ... }
```

4) The comma operator (,) guarantees left-to-right evaluation and there is a sequence point between evaluating the left operand and the right operand.

```
i = 39; a = (i++, i++, ++i);
```

5) For the conditional operator (? :), the first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated)

```
a++ > 42 ? --a : ++a;
```

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    int a = 3;
    ++a;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
```

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
```

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
```

```cpp
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

They are all morons!

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

They are all morons!

Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

Did you know about sequence points? Do you have a deep understanding of when side-effects really take place in C++?

ehh...

```
#include <iostream>

void foo()
{
    int a = 3;
    a++;
    std::cout << a << std::endl;
}

int main()
{
    foo();
    foo();
    foo();
}
```

```
$ c++ foo.cpp
$ ./a.out
4
4
4
```

Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

Did you know about sequence points? Do you have a deep understanding of when side-effects really take place in C++?

Behavior

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

int main()
{
    // implementation-defined
    int i = ~0;
    i >>= 1;
    printf("%d\n", i);

    // unspecified output
    printf("4") + printf("2");
    printf("\n");

    // undefined
    int k = INT_MAX;
    k += 1;
    printf("%d\n", k);
}
```

implementation-defined behavior: the construct is not incorrect; the code must compile; the compiler must document the behavior

unspecified behavior: the same as implementation-defined except the behavior need not be documented

undefined behavior: the standard imposes no requirements ; anything at all can happen, all bets are off, nasal demons might fly out of your nose.

Note that many compilers will not give you any warnings when compiling this code, and due to the undefined behavior caused by signed integer overflow above, the whole program is in theory undefined.

Behavior

... and, locale-specific behavior

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

int main()
{
    // implementation-defined
    int i = ~0;
    i >>= 1;
    printf("%d\n", i);

    // unspecified output
    printf("4") + printf("2");
    printf("\n");

    // undefined
    int k = INT_MAX;
    k += 1;
    printf("%d\n", k);
}
```

implementation-defined behavior: the construct is not incorrect; the code must compile; the compiler must document the behavior

unspecified behavior: the same as implementation-defined except the behavior need not be documented

undefined behavior: the standard imposes no requirements ; anything at all can happen, all bets are off, nasal demons might fly out of your nose.

Note that many compilers will not give you any warnings when compiling this code, and due to the undefined behavior caused by signed integer overflow above, the whole program is in theory undefined.

the C++ standard defines the expected behaviour, but says very little about ***how*** it should be implemented.

the C++ standard defines the expected behaviour, but says very little about ***how*** it should be implemented.

this is a key feature of C++, and one of the reason why C++ is such a successful programming language on a wide range of hardware!

A real story of "anything can happen" if your program is undefined behavior.

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior http://markshroyer.com/2012/06/c-both-true-and-false/)

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (-02) I get:

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior http://markshroyer.com/2012/06/c-both-true-and-false/)

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (-O2) I get:

```
false
```

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior http://markshroyer.com/2012/06/c-both-true-and-false/)

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (`-O2`) I get:

```
false
```

```
false
```

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior http://markshroyer.com/2012/06/c-both-true-and-false/)

Exercise

This code is **undefined behavior** because `b` is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.cpp

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

bar.cpp

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

This is what I get on my computer with no optimization (`-O0 -m32 -mtune=i386`) :

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (-O2) I get:

```
false
```

```
false
```

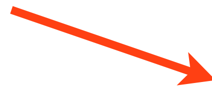
```
false
```

(thanks to Mark Shroyer for blogging about this very interesting gcc behavior http://markshroyer.com/2012/06/c-both-true-and-false/)

It is looking at assembler time!

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```c
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



icc 13.0.1 with no optimization (-O0)

```
0x00001f5c   push   ebp
0x00001f5d   mov    ebp,esp
0x00001f5f   sub    esp,0x8
0x00001f62   call   0x1f67 <foo+11>
0x00001f67   pop    eax
0x00001f68   mov    DWORD PTR [ebp-0x4],eax
0x00001f6b   movzx  eax,BYTE PTR [ebp-0x8]
0x00001f6f   movzx  eax,al
0x00001f72   test   eax,eax
0x00001f74   je     0x1f8d <foo+49>
0x00001f76   add    esp,0xfffffff0
0x00001f79   mov    eax,DWORD PTR [ebp-0x4]
0x00001f7c   lea    eax,[eax+0x89]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1fca <dyld_stub_printf>
0x00001f8a   add    esp,0x10
0x00001f8d   movzx  eax,BYTE PTR [ebp-0x8]
0x00001f91   movzx  eax,al
0x00001f94   test   eax,eax
0x00001f96   jne    0x1faf <foo+83>
0x00001f98   add    esp,0xfffffff0
0x00001f9b   mov    eax,DWORD PTR [ebp-0x4]
0x00001f9e   lea    eax,[eax+0x91]
0x00001fa4   mov    DWORD PTR [esp],eax
0x00001fa7   call   0x1fca <dyld_stub_printf>
0x00001fac   add    esp,0x10
0x00001faf   leave
0x00001fb0   ret
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

icc 13.0.1 with no optimization (–O0)

```
0x00001f5c    push    ebp
0x00001f5d    mov     ebp,esp
0x00001f5f    sub     esp,0x8
0x00001f62    call    0x1f67 <foo+11>
0x00001f67    pop     eax
0x00001f68    mov     DWORD PTR [ebp-0x4],eax
0x00001f6b    movzx   eax,BYTE PTR [ebp-0x8]
0x00001f6f    movzx   eax,al
0x00001f72    test    eax,eax
0x00001f74    je      0x1f8d <foo+49>
0x00001f76    add     esp,0xfffffff0
0x00001f79    mov     eax,DWORD PTR [ebp-0x4]
0x00001f7c    lea     eax,[eax+0x89]
0x00001f82    mov     DWORD PTR [esp],eax
0x00001f85    call    0x1fca <dyld_stub_printf>
0x00001f8a    add     esp,0x10
0x00001f8d    movzx   eax,BYTE PTR [ebp-0x8]
0x00001f91    movzx   eax,al
0x00001f94    test    eax,eax
0x00001f96    jne     0x1faf <foo+83>
0x00001f98    add     esp,0xfffffff0
0x00001f9b    mov     eax,DWORD PTR [ebp-0x4]
0x00001f9e    lea     eax,[eax+0x91]
0x00001fa4    mov     DWORD PTR [esp],eax
0x00001fa7    call    0x1fca <dyld_stub_printf>
0x00001fac    add     esp,0x10
0x00001faf    leave
0x00001fb0    ret
```

icc 13.0.1 with no optimization (–O0)

```
void foo(void) {
    char b; // “random” value
    reg.a = b;
    if (b == 0)
        goto label1;
    printf("true\n");
  label1:
    if (b != 0)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

icc 13.0.1 with no optimization (–O0)

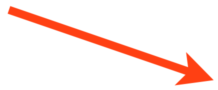
```
0x00001f5c    push    ebp
0x00001f5d    mov     ebp,esp
0x00001f5f    sub     esp,0x8
0x00001f62    call    0x1f67 <foo+11>
0x00001f67    pop     eax
0x00001f68    mov     DWORD PTR [ebp-0x4],eax
0x00001f6b    movzx   eax,BYTE PTR [ebp-0x8]
0x00001f6f    movzx   eax,al
0x00001f72    test    eax,eax
0x00001f74    je      0x1f8d <foo+49>
0x00001f76    add     esp,0xfffffff0
0x00001f79    mov     eax,DWORD PTR [ebp-0x4]
0x00001f7c    lea     eax,[eax+0x89]
0x00001f82    mov     DWORD PTR [esp],eax
0x00001f85    call    0x1fca <dyld_stub_printf>
0x00001f8a    add     esp,0x10
0x00001f8d    movzx   eax,BYTE PTR [ebp-0x8]
0x00001f91    movzx   eax,al
0x00001f94    test    eax,eax
0x00001f96    jne     0x1faf <foo+83>
0x00001f98    add     esp,0xfffffff0
0x00001f9b    mov     eax,DWORD PTR [ebp-0x4]
0x00001f9e    lea     eax,[eax+0x91]
0x00001fa4    mov     DWORD PTR [esp],eax
0x00001fa7    call    0x1fca <dyld_stub_printf>
0x00001fac    add     esp,0x10
0x00001faf    leave
0x00001fb0    ret
```

icc 13.0.1 with no optimization (–O0)

```
void foo(void) {
    char b; // “random” value
    reg.a = b;
    if (b == 0)
        goto label1;
    printf("true\n");
  label1:
    if (b != 0)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

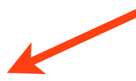
icc is doing what most programmers would expect might happen

```c
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



icc 13.0.1 with no optimization (–O0)

```
0x00001f5c   push    ebp
0x00001f5d   mov     ebp,esp
0x00001f5f   sub     esp,0x8
0x00001f62   call    0x1f67 <foo+11>
0x00001f67   pop     eax
0x00001f68   mov     DWORD PTR [ebp-0x4],eax
0x00001f6b   movzx   eax,BYTE PTR [ebp-0x8]
0x00001f6f   movzx   eax,al
0x00001f72   test    eax,eax
0x00001f74   je      0x1f8d <foo+49>
0x00001f76   add     esp,0xfffffff0
0x00001f79   mov     eax,DWORD PTR [ebp-0x4]
0x00001f7c   lea     eax,[eax+0x89]
0x00001f82   mov     DWORD PTR [esp],eax
0x00001f85   call    0x1fca <dyld_stub_printf>
0x00001f8a   add     esp,0x10
0x00001f8d   movzx   eax,BYTE PTR [ebp-0x8]
0x00001f91   movzx   eax,al
0x00001f94   test    eax,eax
0x00001f96   jne     0x1faf <foo+83>
0x00001f98   add     esp,0xfffffff0
0x00001f9b   mov     eax,DWORD PTR [ebp-0x4]
0x00001f9e   lea     eax,[eax+0x91]
0x00001fa4   mov     DWORD PTR [esp],eax
0x00001fa7   call    0x1fca <dyld_stub_printf>
0x00001fac   add     esp,0x10
0x00001faf   leave
0x00001fb0   ret
```



icc 13.0.1 with no optimization (–O0)

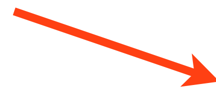
```c
void foo(void) {
    char b; // “random” value
    reg.a = b;
    if (b == 0)
        goto label1;
    printf("true\n");
  label1:
    if (b != 0)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

“Random” value	output
0	false
1	true
anything else	true

icc is doing what most programmers would expect might happen

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

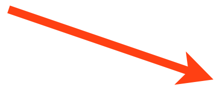
```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



icc 13.0.1 with optimization (–02)

```
0x00001e40   sub    esp,0x1c
0x00001e43   call   0x1e48 <foo+8>
0x00001e48   pop    edx
0x00001e49   test   al,al
0x00001e4b   jne    0x1e60 <foo+32>
0x00001e4d   add    esp,0x4
0x00001e50   lea    eax,[edx+0x1ac]
0x00001e56   push   eax
0x00001e57   call   0x1fc6 <dyld_stub_printf>
0x00001e5c   add    esp,0x1c
0x00001e5f   ret
0x00001e60   add    esp,0x4
0x00001e63   lea    eax,[edx+0x1a4]
0x00001e69   push   eax
0x00001e6a   call   0x1fc6 <dyld_stub_printf>
0x00001e6f   add    esp,0x1c
0x00001e72   ret
```

```c
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



icc 13.0.1 with optimization (–02)

```
0x00001e40    sub     esp,0x1c
0x00001e43    call    0x1e48 <foo+8>
0x00001e48    pop     edx
0x00001e49    test    al,al
0x00001e4b    jne     0x1e60 <foo+32>
0x00001e4d    add     esp,0x4
0x00001e50    lea     eax,[edx+0x1ac]
0x00001e56    push    eax
0x00001e57    call    0x1fc6 <dyld_stub_printf>
0x00001e5c    add     esp,0x1c
0x00001e5f    ret
0x00001e60    add     esp,0x4
0x00001e63    lea     eax,[edx+0x1a4]
0x00001e69    push    eax
0x00001e6a    call    0x1fc6 <dyld_stub_printf>
0x00001e6f    add     esp,0x1c
0x00001e72    ret
```

icc 13.0.1 with optimization (–02)

```c
void foo(void) {
    reg.a; // “random” value
    if (reg.a != 0)
        goto label1;
    printf("false\n");
    return;
  label1:
    printf("true\n");
    return;
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

icc 13.0.1 with optimization (–02)

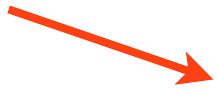
```
0x00001e40   sub    esp,0x1c
0x00001e43   call   0x1e48 <foo+8>
0x00001e48   pop    edx
0x00001e49   test   al,al
0x00001e4b   jne    0x1e60 <foo+32>
0x00001e4d   add    esp,0x4
0x00001e50   lea    eax,[edx+0x1ac]
0x00001e56   push   eax
0x00001e57   call   0x1fc6 <dyld_stub_printf>
0x00001e5c   add    esp,0x1c
0x00001e5f   ret
0x00001e60   add    esp,0x4
0x00001e63   lea    eax,[edx+0x1a4]
0x00001e69   push   eax
0x00001e6a   call   0x1fc6 <dyld_stub_printf>
0x00001e6f   add    esp,0x1c
0x00001e72   ret
```

icc 13.0.1 with optimization (–02)

```
void foo(void) {
    reg.a; // “random” value
    if (reg.a != 0)
        goto label1;
    printf("false\n");
    return;
  label1:
    printf("true\n");
    return;
}
```

Notice that icc does not even create space for the variable b. It is just using the random value stored in the eax register.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



icc 13.0.1 with optimization (–02)

```
0x00001e40    sub     esp,0x1c
0x00001e43    call    0x1e48 <foo+8>
0x00001e48    pop     edx
0x00001e49    test    al,al
0x00001e4b    jne     0x1e60 <foo+32>
0x00001e4d    add     esp,0x4
0x00001e50    lea     eax,[edx+0x1ac]
0x00001e56    push    eax
0x00001e57    call    0x1fc6 <dyld_stub_printf>
0x00001e5c    add     esp,0x1c
0x00001e5f    ret
0x00001e60    add     esp,0x4
0x00001e63    lea     eax,[edx+0x1a4]
0x00001e69    push    eax
0x00001e6a    call    0x1fc6 <dyld_stub_printf>
0x00001e6f    add     esp,0x1c
0x00001e72    ret
```

icc 13.0.1 with optimization (–02)

```
void foo(void) {
    reg.a; // “random” value
    if (reg.a != 0)
        goto label1;
    printf("false\n");
    return;
  label1:
    printf("true\n");
    return;
}
```

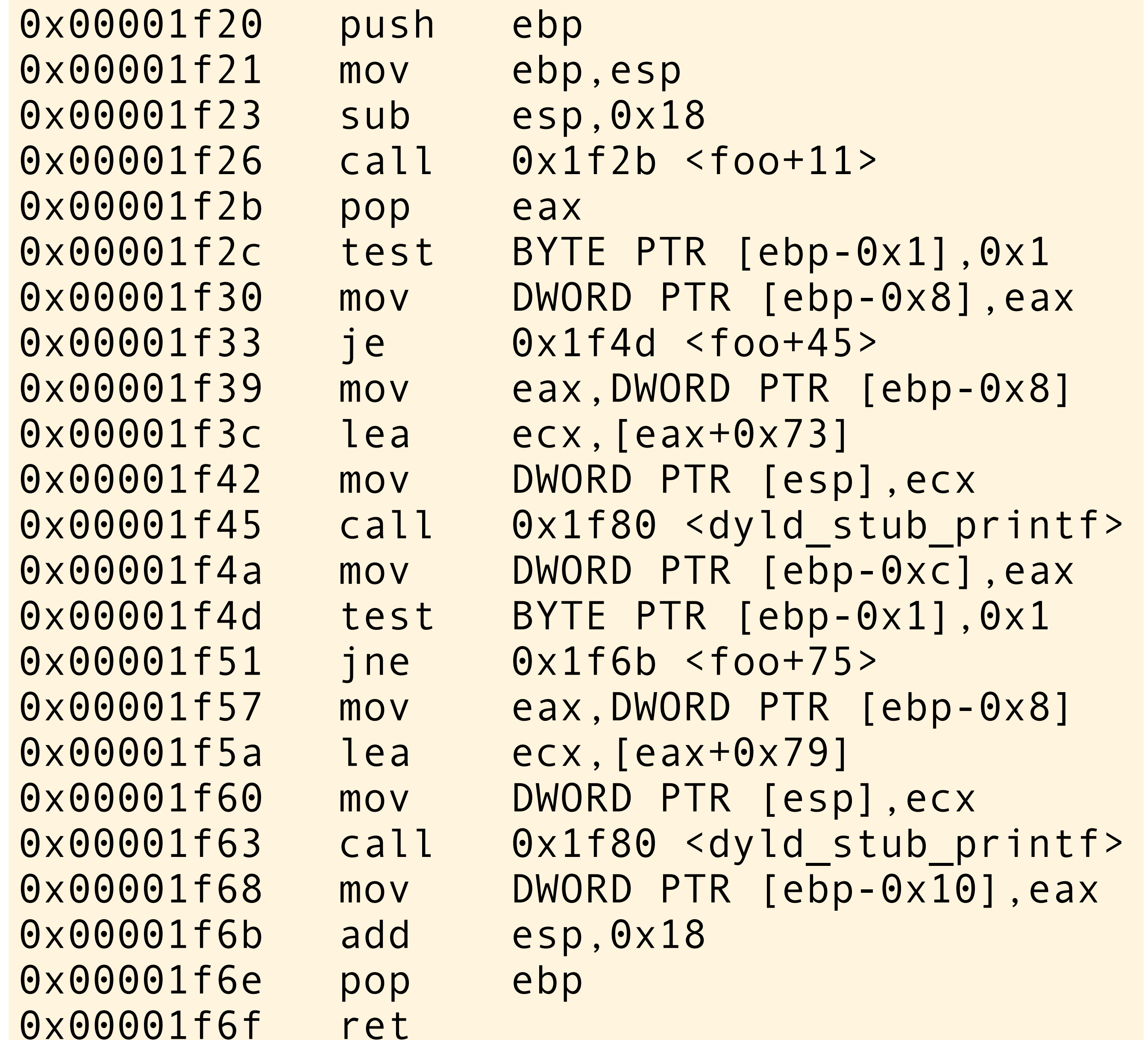
“Random” value	output
0	false
1	true
anything else	true

Notice that icc does not even create space for the variable b. It is just using the random value stored in the eax register.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

clang 4.1 with no optimization (–O0)



```
0x00001f20   push   ebp
0x00001f21   mov    ebp,esp
0x00001f23   sub    esp,0x18
0x00001f26   call   0x1f2b <foo+11>
0x00001f2b   pop    eax
0x00001f2c   test   BYTE PTR [ebp-0x1],0x1
0x00001f30   mov    DWORD PTR [ebp-0x8],eax
0x00001f33   je     0x1f4d <foo+45>
0x00001f39   mov    eax,DWORD PTR [ebp-0x8]
0x00001f3c   lea    ecx,[eax+0x73]
0x00001f42   mov    DWORD PTR [esp],ecx
0x00001f45   call   0x1f80 <dyld_stub_printf>
0x00001f4a   mov    DWORD PTR [ebp-0xc],eax
0x00001f4d   test   BYTE PTR [ebp-0x1],0x1
0x00001f51   jne    0x1f6b <foo+75>
0x00001f57   mov    eax,DWORD PTR [ebp-0x8]
0x00001f5a   lea    ecx,[eax+0x79]
0x00001f60   mov    DWORD PTR [esp],ecx
0x00001f63   call   0x1f80 <dyld_stub_printf>
0x00001f68   mov    DWORD PTR [ebp-0x10],eax
0x00001f6b   add    esp,0x18
0x00001f6e   pop    ebp
0x00001f6f   ret
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

clang 4.1 with no optimization (–O0)

```
0x00001f20    push    ebp
0x00001f21    mov     ebp,esp
0x00001f23    sub     esp,0x18
0x00001f26    call    0x1f2b <foo+11>
0x00001f2b    pop     eax
0x00001f2c    test    BYTE PTR [ebp-0x1],0x1
0x00001f30    mov     DWORD PTR [ebp-0x8],eax
0x00001f33    je      0x1f4d <foo+45>
0x00001f39    mov     eax,DWORD PTR [ebp-0x8]
0x00001f3c    lea     ecx,[eax+0x73]
0x00001f42    mov     DWORD PTR [esp],ecx
0x00001f45    call    0x1f80 <dyld_stub_printf>
0x00001f4a    mov     DWORD PTR [ebp-0xc],eax
0x00001f4d    test    BYTE PTR [ebp-0x1],0x1
0x00001f51    jne     0x1f6b <foo+75>
0x00001f57    mov     eax,DWORD PTR [ebp-0x8]
0x00001f5a    lea     ecx,[eax+0x79]
0x00001f60    mov     DWORD PTR [esp],ecx
0x00001f63    call    0x1f80 <dyld_stub_printf>
0x00001f68    mov     DWORD PTR [ebp-0x10],eax
0x00001f6b    add     esp,0x18
0x00001f6e    pop     ebp
0x00001f6f    ret
```

clang 4.1 with no optimization (–O0)

```
void foo(void) {
    char b; // “random” value
    if ((b & 1) != 1)
        goto label1;
    printf("true\n");
  label1:
    if ((b & 1) == 1)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

clang 4.1 with no optimization (–O0)

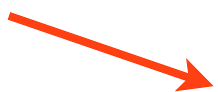
```
0x00001f20   push   ebp
0x00001f21   mov    ebp,esp
0x00001f23   sub    esp,0x18
0x00001f26   call   0x1f2b <foo+11>
0x00001f2b   pop    eax
0x00001f2c   test   BYTE PTR [ebp-0x1],0x1
0x00001f30   mov    DWORD PTR [ebp-0x8],eax
0x00001f33   je     0x1f4d <foo+45>
0x00001f39   mov    eax,DWORD PTR [ebp-0x8]
0x00001f3c   lea    ecx,[eax+0x73]
0x00001f42   mov    DWORD PTR [esp],ecx
0x00001f45   call   0x1f80 <dyld_stub_printf>
0x00001f4a   mov    DWORD PTR [ebp-0xc],eax
0x00001f4d   test   BYTE PTR [ebp-0x1],0x1
0x00001f51   jne    0x1f6b <foo+75>
0x00001f57   mov    eax,DWORD PTR [ebp-0x8]
0x00001f5a   lea    ecx,[eax+0x79]
0x00001f60   mov    DWORD PTR [esp],ecx
0x00001f63   call   0x1f80 <dyld_stub_printf>
0x00001f68   mov    DWORD PTR [ebp-0x10],eax
0x00001f6b   add    esp,0x18
0x00001f6e   pop    ebp
0x00001f6f   ret
```

clang 4.1 with no optimization (–O0)

```
void foo(void) {
    char b; // “random” value
    if ((b & 1) != 1)
        goto label1;
    printf("true\n");
  label1:
    if ((b & 1) == 1)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

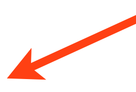
clang just tests the last bit of the byte it uses to represent the bool.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



clang 4.1 with no optimization (–O0)

```
0x00001f20    push    ebp
0x00001f21    mov     ebp,esp
0x00001f23    sub     esp,0x18
0x00001f26    call    0x1f2b <foo+11>
0x00001f2b    pop     eax
0x00001f2c    test    BYTE PTR [ebp-0x1],0x1
0x00001f30    mov     DWORD PTR [ebp-0x8],eax
0x00001f33    je      0x1f4d <foo+45>
0x00001f39    mov     eax,DWORD PTR [ebp-0x8]
0x00001f3c    lea     ecx,[eax+0x73]
0x00001f42    mov     DWORD PTR [esp],ecx
0x00001f45    call    0x1f80 <dyld_stub_printf>
0x00001f4a    mov     DWORD PTR [ebp-0xc],eax
0x00001f4d    test    BYTE PTR [ebp-0x1],0x1
0x00001f51    jne     0x1f6b <foo+75>
0x00001f57    mov     eax,DWORD PTR [ebp-0x8]
0x00001f5a    lea     ecx,[eax+0x79]
0x00001f60    mov     DWORD PTR [esp],ecx
0x00001f63    call    0x1f80 <dyld_stub_printf>
0x00001f68    mov     DWORD PTR [ebp-0x10],eax
0x00001f6b    add     esp,0x18
0x00001f6e    pop     ebp
0x00001f6f    ret
```



clang 4.1 with no optimization (–O0)

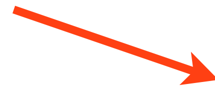
```
void foo(void) {
    char b; // “random” value
    if ((b & 1) != 1)
        goto label1;
    printf("true\n");
  label1:
    if ((b & 1) == 1)
        goto label2;
    printf("false\n");
  label2:
    ;
}
```

“Random” value	**output**
even number	false
odd number	true

clang just tests the last bit of the byte it uses to represent the bool.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

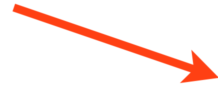
```c
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



clang 4.1 with optimization (–O2)

```
0x00001f70   push   ebp
0x00001f71   mov    ebp,esp
0x00001f73   sub    esp,0x8
0x00001f76   call   0x1f7b <foo+11>
0x00001f7b   pop    eax
0x00001f7c   lea    eax,[eax+0x37]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1f96 <dyld_stub_puts>
0x00001f8a   add    esp,0x8
0x00001f8d   pop    ebp
0x00001f8e   ret
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



clang 4.1 with optimization (–02)

```
0x00001f70   push   ebp
0x00001f71   mov    ebp,esp
0x00001f73   sub    esp,0x8
0x00001f76   call   0x1f7b <foo+11>
0x00001f7b   pop    eax
0x00001f7c   lea    eax,[eax+0x37]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1f96 <dyld_stub_puts>
0x00001f8a   add    esp,0x8
0x00001f8d   pop    ebp
0x00001f8e   ret
```

clang 4.1 with optimization (–02)

```
void foo(void) {
    puts("false");
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

clang 4.1 with optimization (–02)

```
0x00001f70   push   ebp
0x00001f71   mov    ebp,esp
0x00001f73   sub    esp,0x8
0x00001f76   call   0x1f7b <foo+11>
0x00001f7b   pop    eax
0x00001f7c   lea    eax,[eax+0x37]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1f96 <dyld_stub_puts>
0x00001f8a   add    esp,0x8
0x00001f8d   pop    ebp
0x00001f8e   ret
```

clang 4.1 with optimization (–02)

```
void foo(void) {
    puts("false");
}
```

clang just prints “false”. Simple and clean!

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

clang 4.1 with optimization (–02)

```
0x00001f70   push   ebp
0x00001f71   mov    ebp,esp
0x00001f73   sub    esp,0x8
0x00001f76   call   0x1f7b <foo+11>
0x00001f7b   pop    eax
0x00001f7c   lea    eax,[eax+0x37]
0x00001f82   mov    DWORD PTR [esp],eax
0x00001f85   call   0x1f96 <dyld_stub_puts>
0x00001f8a   add    esp,0x8
0x00001f8d   pop    ebp
0x00001f8e   ret
```

clang 4.1 with optimization (–02)

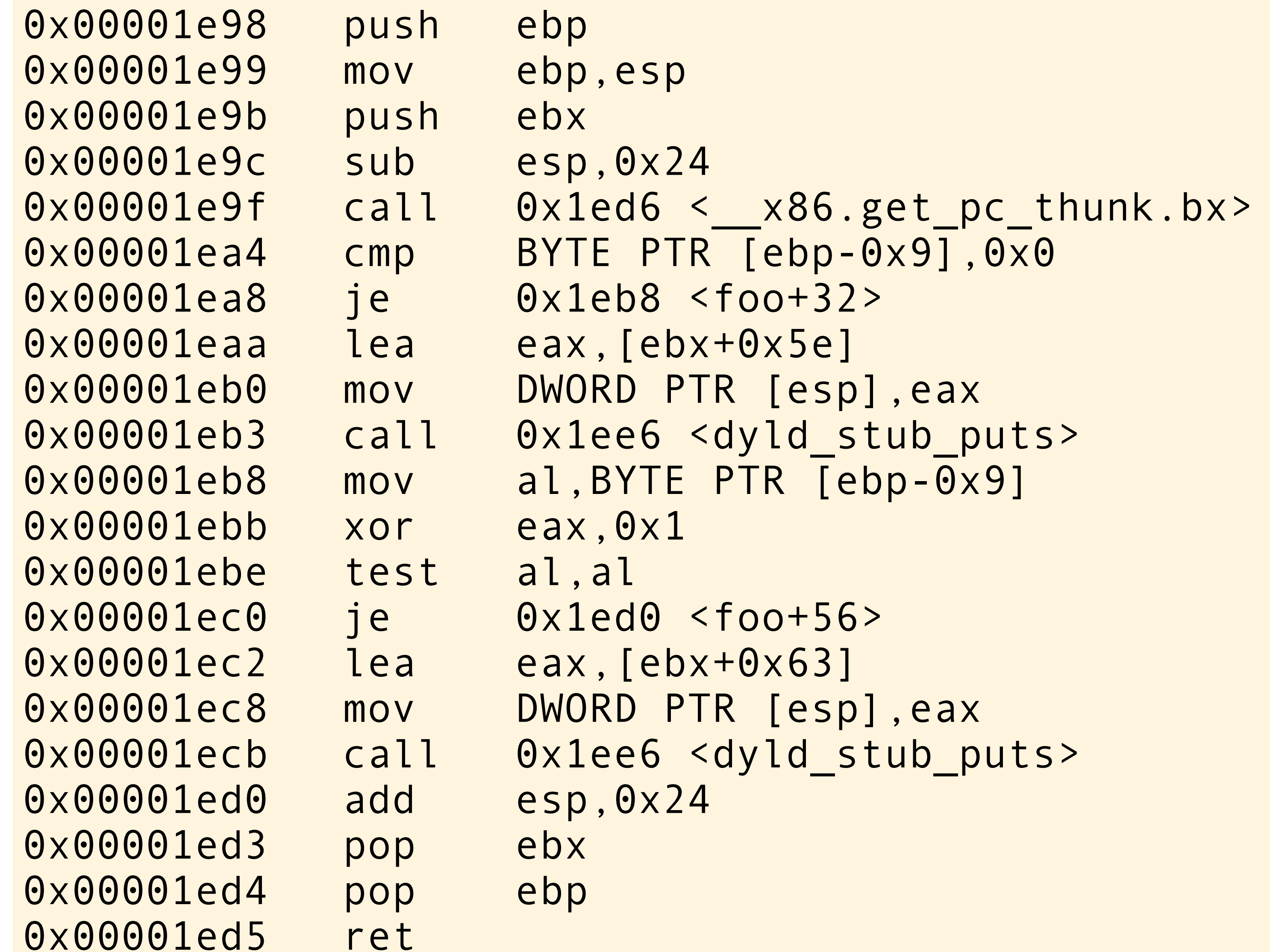
```
void foo(void) {
    puts("false");
}
```

clang just prints “false”. Simple and clean!

“Random” value	output
any number	false

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```



gcc 4.7.2 with no optimization (–O0)

```
0x00001e98   push   ebp
0x00001e99   mov    ebp,esp
0x00001e9b   push   ebx
0x00001e9c   sub    esp,0x24
0x00001e9f   call   0x1ed6 <__x86.get_pc_thunk.bx>
0x00001ea4   cmp    BYTE PTR [ebp-0x9],0x0
0x00001ea8   je     0x1eb8 <foo+32>
0x00001eaa   lea    eax,[ebx+0x5e]
0x00001eb0   mov    DWORD PTR [esp],eax
0x00001eb3   call   0x1ee6 <dyld_stub_puts>
0x00001eb8   mov    al,BYTE PTR [ebp-0x9]
0x00001ebb   xor    eax,0x1
0x00001ebe   test   al,al
0x00001ec0   je     0x1ed0 <foo+56>
0x00001ec2   lea    eax,[ebx+0x63]
0x00001ec8   mov    DWORD PTR [esp],eax
0x00001ecb   call   0x1ee6 <dyld_stub_puts>
0x00001ed0   add    esp,0x24
0x00001ed3   pop    ebx
0x00001ed4   pop    ebp
0x00001ed5   ret
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with no optimization (–O0)

```
0x00001e98   push   ebp
0x00001e99   mov    ebp,esp
0x00001e9b   push   ebx
0x00001e9c   sub    esp,0x24
0x00001e9f   call   0x1ed6 <__x86.get_pc_thunk.bx>
0x00001ea4   cmp    BYTE PTR [ebp-0x9],0x0
0x00001ea8   je     0x1eb8 <foo+32>
0x00001eaa   lea    eax,[ebx+0x5e]
0x00001eb0   mov    DWORD PTR [esp],eax
0x00001eb3   call   0x1ee6 <dyld_stub_puts>
0x00001eb8   mov    al,BYTE PTR [ebp-0x9]
0x00001ebb   xor    eax,0x1
0x00001ebe   test   al,al
0x00001ec0   je     0x1ed0 <foo+56>
0x00001ec2   lea    eax,[ebx+0x63]
0x00001ec8   mov    DWORD PTR [esp],eax
0x00001ecb   call   0x1ee6 <dyld_stub_puts>
0x00001ed0   add    esp,0x24
0x00001ed3   pop    ebx
0x00001ed4   pop    ebp
0x00001ed5   ret
```

gcc 4.7.2 with no optimization (–O0)

```
void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
  label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
  label2:
    ;
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with no optimization (–O0)

```
0x00001e98    push    ebp
0x00001e99    mov     ebp,esp
0x00001e9b    push    ebx
0x00001e9c    sub     esp,0x24
0x00001e9f    call    0x1ed6 <__x86.get_pc_thunk.bx>
0x00001ea4    cmp     BYTE PTR [ebp-0x9],0x0
0x00001ea8    je      0x1eb8 <foo+32>
0x00001eaa    lea     eax,[ebx+0x5e]
0x00001eb0    mov     DWORD PTR [esp],eax
0x00001eb3    call    0x1ee6 <dyld_stub_puts>
0x00001eb8    mov     al,BYTE PTR [ebp-0x9]
0x00001ebb    xor     eax,0x1
0x00001ebe    test    al,al
0x00001ec0    je      0x1ed0 <foo+56>
0x00001ec2    lea     eax,[ebx+0x63]
0x00001ec8    mov     DWORD PTR [esp],eax
0x00001ecb    call    0x1ee6 <dyld_stub_puts>
0x00001ed0    add     esp,0x24
0x00001ed3    pop     ebx
0x00001ed4    pop     ebp
0x00001ed5    ret
```

gcc 4.7.2 with no optimization (–O0)

```
void foo(void) {
    char b; // “random” value
    if (b == 0)
        goto label1;
    puts("true");
  label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
  label2:
    ;
}
```

gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with no optimization (-O0)

```
0x00001e98   push   ebp
0x00001e99   mov    ebp,esp
0x00001e9b   push   ebx
0x00001e9c   sub    esp,0x24
0x00001e9f   call   0x1ed6 <__x86.get_pc_thunk.bx>
0x00001ea4   cmp    BYTE PTR [ebp-0x9],0x0
0x00001ea8   je     0x1eb8 <foo+32>
0x00001eaa   lea    eax,[ebx+0x5e]
0x00001eb0   mov    DWORD PTR [esp],eax
0x00001eb3   call   0x1ee6 <dyld_stub_puts>
0x00001eb8   mov    al,BYTE PTR [ebp-0x9]
0x00001ebb   xor    eax,0x1
0x00001ebe   test   al,al
0x00001ec0   je     0x1ed0 <foo+56>
0x00001ec2   lea    eax,[ebx+0x63]
0x00001ec8   mov    DWORD PTR [esp],eax
0x00001ecb   call   0x1ee6 <dyld_stub_puts>
0x00001ed0   add    esp,0x24
0x00001ed3   pop    ebx
0x00001ed4   pop    ebp
0x00001ed5   ret
```

gcc 4.7.2 with no optimization (-O0)

```
void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
  label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
  label2:
    ;
}
```

"Random" value	output
0	false
1	true
anything else	true false

gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with no optimization (-O0)

```
0x00001e98   push   ebp
0x00001e99   mov    ebp,esp
0x00001e9b   push   ebx
0x00001e9c   sub    esp,0x24
0x00001e9f   call   0x1ed6 <__x86.get_pc_thunk.bx>
0x00001ea4   cmp    BYTE PTR [ebp-0x9],0x0
0x00001ea8   je     0x1eb8 <foo+32>
0x00001eaa   lea    eax,[ebx+0x5e]
0x00001eb0   mov    DWORD PTR [esp],eax
0x00001eb3   call   0x1ee6 <dyld_stub_puts>
0x00001eb8   mov    al,BYTE PTR [ebp-0x9]
0x00001ebb   xor    eax,0x1
0x00001ebe   test   al,al
0x00001ec0   je     0x1ed0 <foo+56>
0x00001ec2   lea    eax,[ebx+0x63]
0x00001ec8   mov    DWORD PTR [esp],eax
0x00001ecb   call   0x1ee6 <dyld_stub_puts>
0x00001ed0   add    esp,0x24
0x00001ed3   pop    ebx
0x00001ed4   pop    ebp
0x00001ed5   ret
```

gcc 4.7.2 with no optimization (-O0)

```
void foo(void) {
    char b; // "random" value
    if (b == 0)
        goto label1;
    puts("true");
  label1:
    reg.a = b;
    reg.a ^= 1;
    if (reg.a == 0)
        goto label2;
    puts("false");
  label2:
    ;
}
```

"Random" value	**output**
0	false
1	true
anything else	true false

gcc assumes that the bitpattern in the byte representing a bool is always 0 or 1, never anything else.

... and there is nothing wrong with that. We have broken the rules of the language by reading an unintialized object.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with optimization (–02)

```
0x00001edc   push   ebx
0x00001edd   sub    esp,0x18
0x00001ee0   call   0x1ef8 <__x86.get_pc_thunk.bx>
0x00001ee5   lea    eax,[ebx+0x52]
0x00001eeb   mov    DWORD PTR [esp],eax
0x00001eee   call   0x1f14 <dyld_stub_puts>
0x00001ef3   add    esp,0x18
0x00001ef6   pop    ebx
0x00001ef7   ret
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with optimization (–02)

```
0x00001edc    push    ebx
0x00001edd    sub     esp,0x18
0x00001ee0    call    0x1ef8 <__x86.get_pc_thunk.bx>
0x00001ee5    lea     eax,[ebx+0x52]
0x00001eeb    mov     DWORD PTR [esp],eax
0x00001eee    call    0x1f14 <dyld_stub_puts>
0x00001ef3    add     esp,0x18
0x00001ef6    pop     ebx
0x00001ef7    ret
```

gcc 4.7.2 with optimization (–02)

```
void foo(void) {
    puts("false");
}
```

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with optimization (–02)

```
0x00001edc    push    ebx
0x00001edd    sub     esp,0x18
0x00001ee0    call    0x1ef8 <__x86.get_pc_thunk.bx>
0x00001ee5    lea     eax,[ebx+0x52]
0x00001eeb    mov     DWORD PTR [esp],eax
0x00001eee    call    0x1f14 <dyld_stub_puts>
0x00001ef3    add     esp,0x18
0x00001ef6    pop     ebx
0x00001ef7    ret
```

gcc 4.7.2 with optimization (–02)

```
void foo(void) {
    puts("false");
}
```

gcc just prints “false”.

```
void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

gcc 4.7.2 with optimization (–02)

```
0x00001edc    push    ebx
0x00001edd    sub     esp,0x18
0x00001ee0    call    0x1ef8 <__x86.get_pc_thunk.bx>
0x00001ee5    lea     eax,[ebx+0x52]
0x00001eeb    mov     DWORD PTR [esp],eax
0x00001eee    call    0x1f14 <dyld_stub_puts>
0x00001ef3    add     esp,0x18
0x00001ef6    pop     ebx
0x00001ef7    ret
```

gcc 4.7.2 with optimization (–02)

```
void foo(void) {
    puts("false");
}
```

gcc just prints "false".

"Random" value	**output**
0	false
1	false
anything else	false

foo.cpp

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

"Random" value	icc -O0	icc -O2	clang -O0	clang -O2	gcc -O0	gcc -O2
0	false	false	false	false	false	false
1	true	true	true	false	true	false
anything else	true	true	true or false	false	true false	false

true if random value is odd
false if random value is even

It is common to think that undefined behavior is not such a big deal, and that it is possible to reason about what the compiler might do when encountering code that break the rules. I hope I have illustrated that really strange things can happen, and that it is not possible to generalize about what might happen.

While I don't show it in this presentation, it is also important to realize that undefined behavior is not only a local problem. The state of the runtime environment will be corrupted, but also the state of the compiler will be corrupted - meaning that UB might result in strange behavior in apparently unrelated parts of the codebase.

About Data Structures

About Data Structures

How structs and classes are
implemented in C++

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

So you get 9?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

So you get 9?

Could be, but this is what I get on my machine

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

So you get 9?

Could be, but this is what I get on my machine

```
$ c++ foo.cpp && ./a.out
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

So you get 9?

```
$ c++ foo.cpp && ./a.out
12
```

Could be, but this is what I get on my machine

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

I guess integers are 4 bytes and char is 1 byte.

Yes, on my machine it is

So you get 9?

Could be, but this is what I get on my machine

```
$ c++ foo.cpp && ./a.out
12
```

Yeah of course, I forgot about that, because in C++ the structs are padded so the size becomes multiple 4

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
12
```

```
struct X {
    int a;
    char b;
    int c;
};
```

packed struct

a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned

a	a	a	a
b	.	.	.
c	c	c	c

`sizeof(struct X) == 12`

```
struct X {
    int a;
    char b;
    int c;
};
```

packed struct

a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned

a	a	a	a
b	.	.	.
c	c	c	c

`sizeof(struct X) == 12`

Imagine how the assembly code for this snippet would look like:

```
void foo(struct X * x) {
  x->c += 42;
}
```

```
struct X {
    int a;
    char b;
    int c;
};
```

packed struct ?			
a	a	a	a
b	c	c	c
c			

```
sizeof(struct X) == 9
```

memory aligned ✔			
a	a	a	a
b	.	.	.
c	c	c	c

```
sizeof(struct X) == 12
```

Imagine how the assembly code for this snippet would look like:

```
void foo(struct X * x) {
  x->c += 42;
}
```

```
struct X {
    int a;
    char b;
    int c;
};
```

This is quite common.

packed struct ?

a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned ✔

a	a	a	a
b	.	.	.
c	c	c	c

`sizeof(struct X) == 12`

Imagine how the assembly code for this snippet would look like:

```
void foo(struct X * x) {
  x->c += 42;
}
```

```
struct X {
    int a;
    char b;
    int c;
};
```

This is quite common.

But please remember that this is **implementation defined**.

packed struct ?			
a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned ✔			
a	a	a	a
b	.	.	.
c	c	c	c

`sizeof(struct X) == 12`

Imagine how the assembly code for this snippet would look like:

```
void foo(struct X * x) {
  x->c += 42;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

So what if I add a member function?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

So what if I add a member function?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;


};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Now this code will print 16. Because there will be a pointer to the function.

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Now this code will print 16. Because there will be a pointer to the function.

ok?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Now this code will print 16. Because there will be a pointer to the function.

ok?

Lets add two more functions...

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }


};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Then it will print 24. Two more pointers.

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Then it will print 24. Two more pointers.

This is what I get on my machine

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Then it will print 24. Two more pointers.

This is what I get on my machine

```
$ c++ foo.cpp && ./a.out
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Then it will print 24. Two more pointers.

This is what I get on my machine

```
$ c++ foo.cpp && ./a.out
12
```

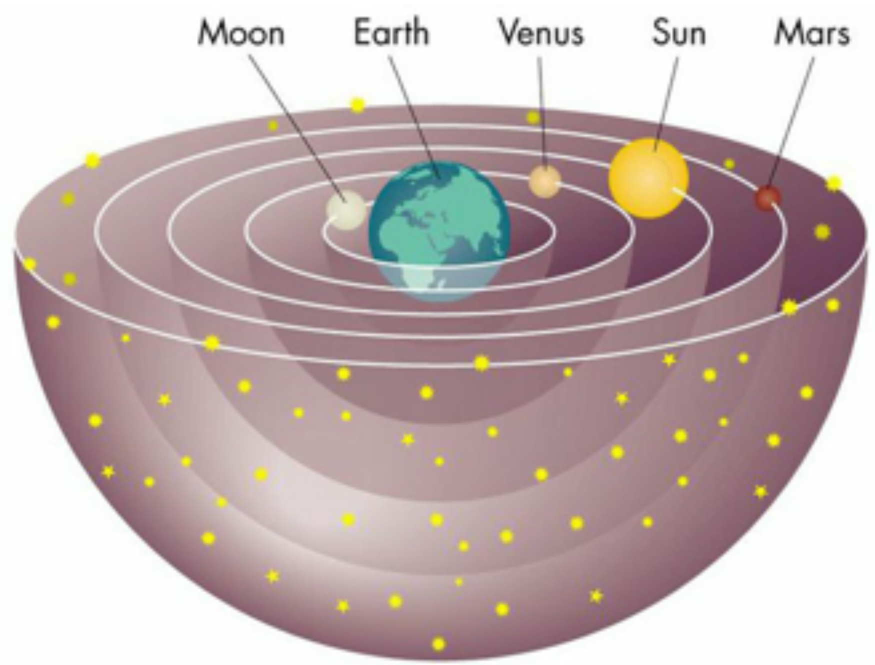
```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

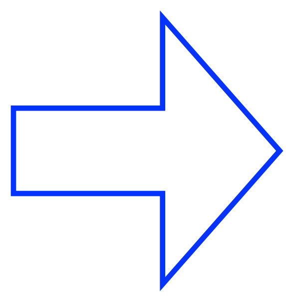
Then it will print 24. Two more pointers.

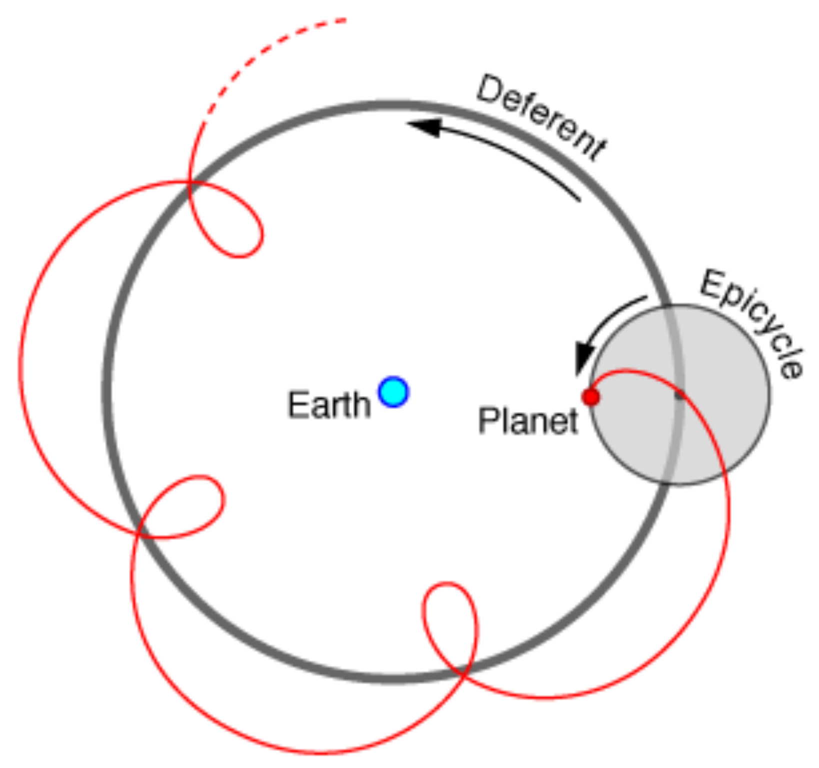
Huh? Probably some weird optimization going on, perhaps because the functions are never called.

This is what I get on my machine

```
$ c++ foo.cpp && ./a.out
12
```


Moon
Earth
Venus
Sun
Mars




Deferent
Epicycle
Earth
Planet

```cpp
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Because adding member functions like this does not change the size of the struct. In C++, the object does not know about it's functions, it is the functions that know about the object.

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Because adding member functions like this does not change the size of the struct. In C++, the object does not know about it's functions, it is the functions that know about the object.

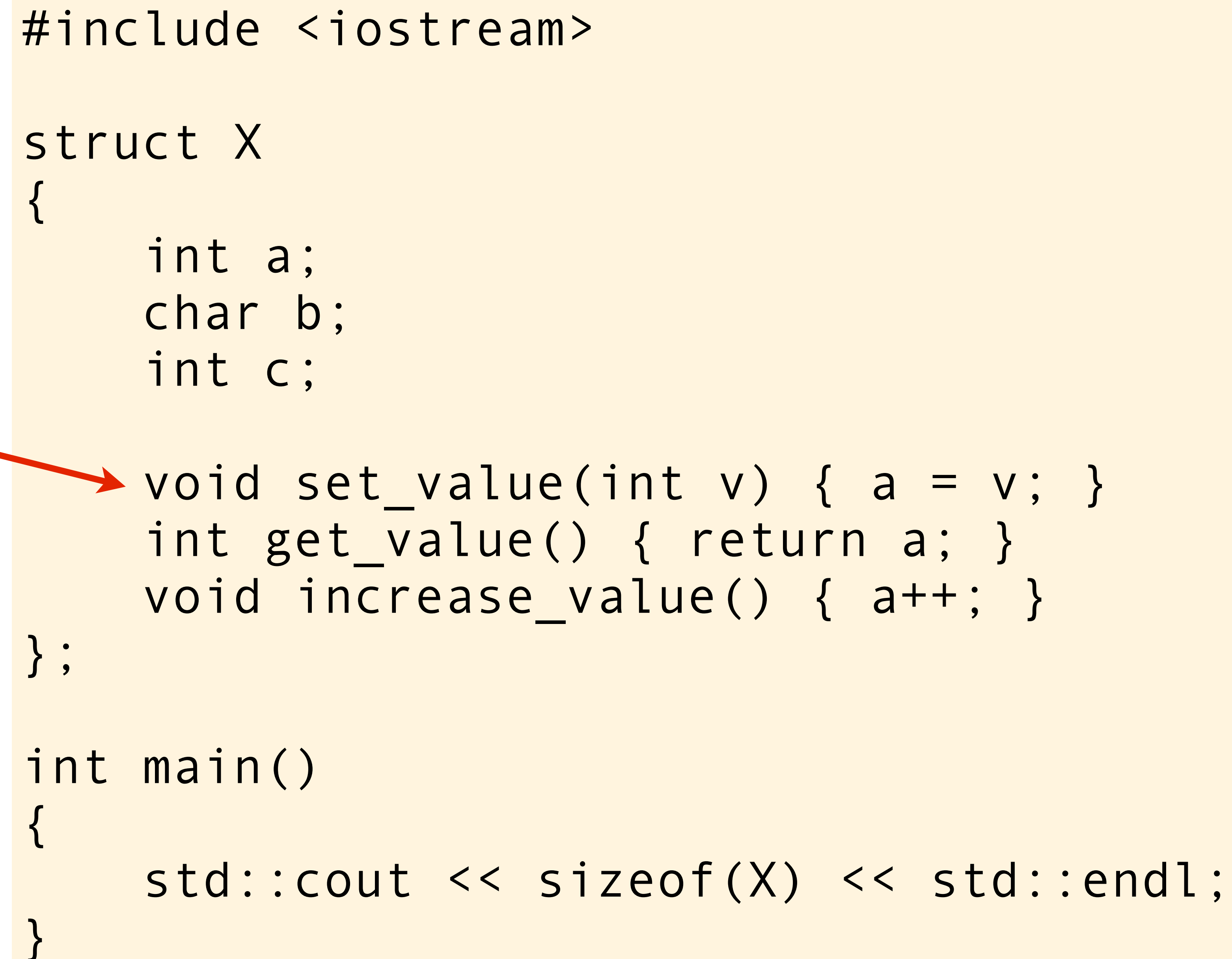
If you rewrite this into C it becomes obvious.

C++

```
struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};
```

C

```
struct X
{
    int a;
    char b;
    int c;
};

void set_value(struct X * this, int v) { this->a = v; }
int get_value(struct X * this) { return this->a; }
void increase_value(struct X * this) { this->a++; }
```


#include <iostream>
struct X
{
int a;
char b;
int c;
void set_value(int v) { a = v; }
int get_value() { return a; }
void increase_value() { a++; }
};
int main()
{
std::cout << sizeof(X) << std::endl;
}

```cpp
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

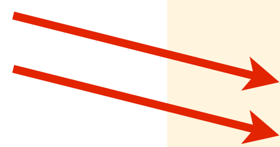
```
$ c++ foo.cpp && ./a.out
24
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Ehh...

```
$ c++ foo.cpp && ./a.out
24
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



Ehh...

```
$ c++ foo.cpp && ./a.out
24
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

48?

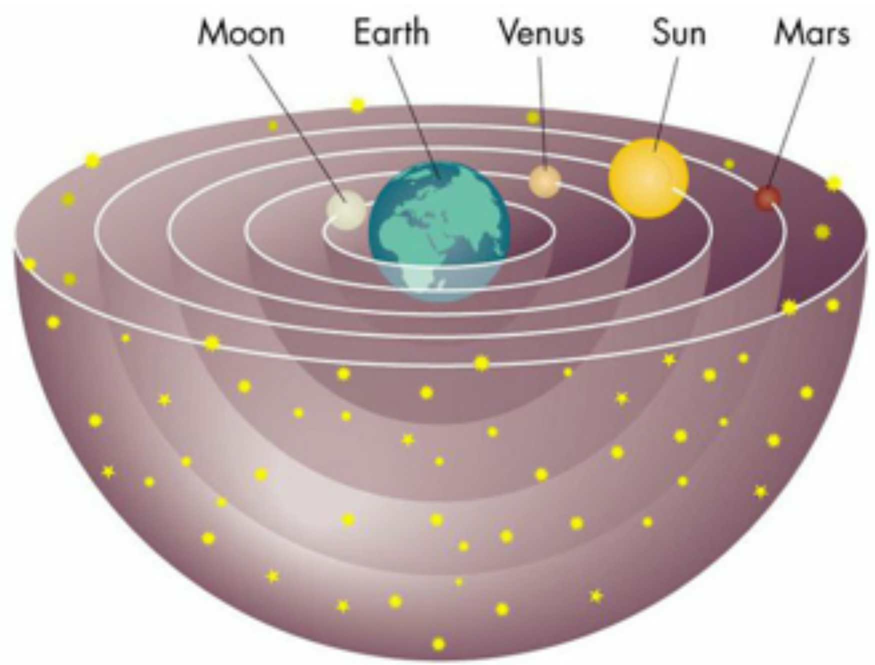
```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

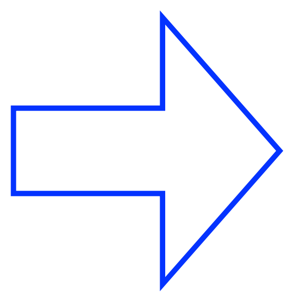
48?

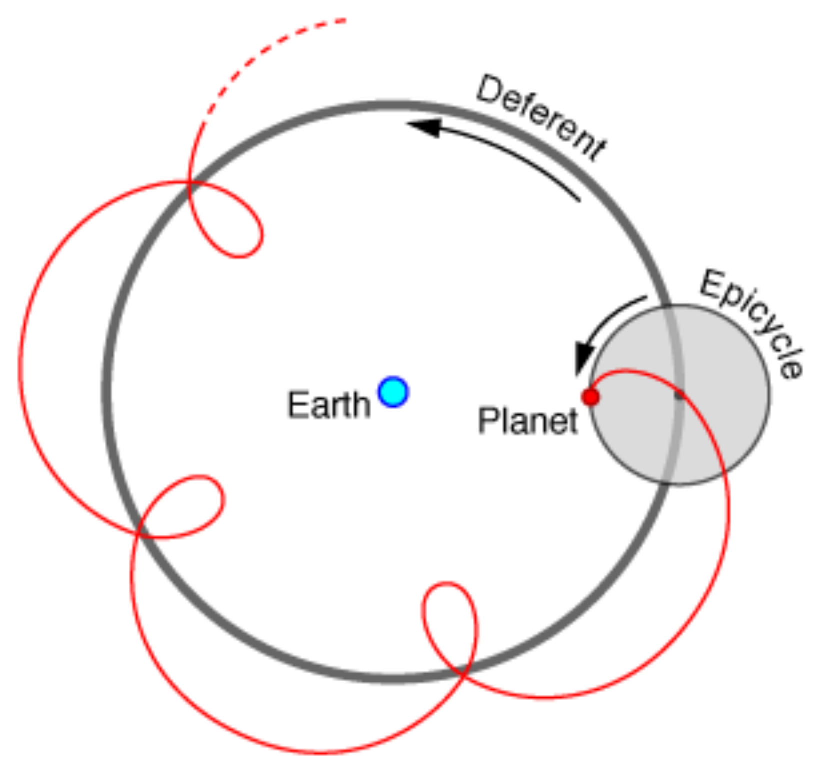
```
$ c++ foo.cpp && ./a.out
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
24
```


Moon
Earth
Venus
Sun
Mars



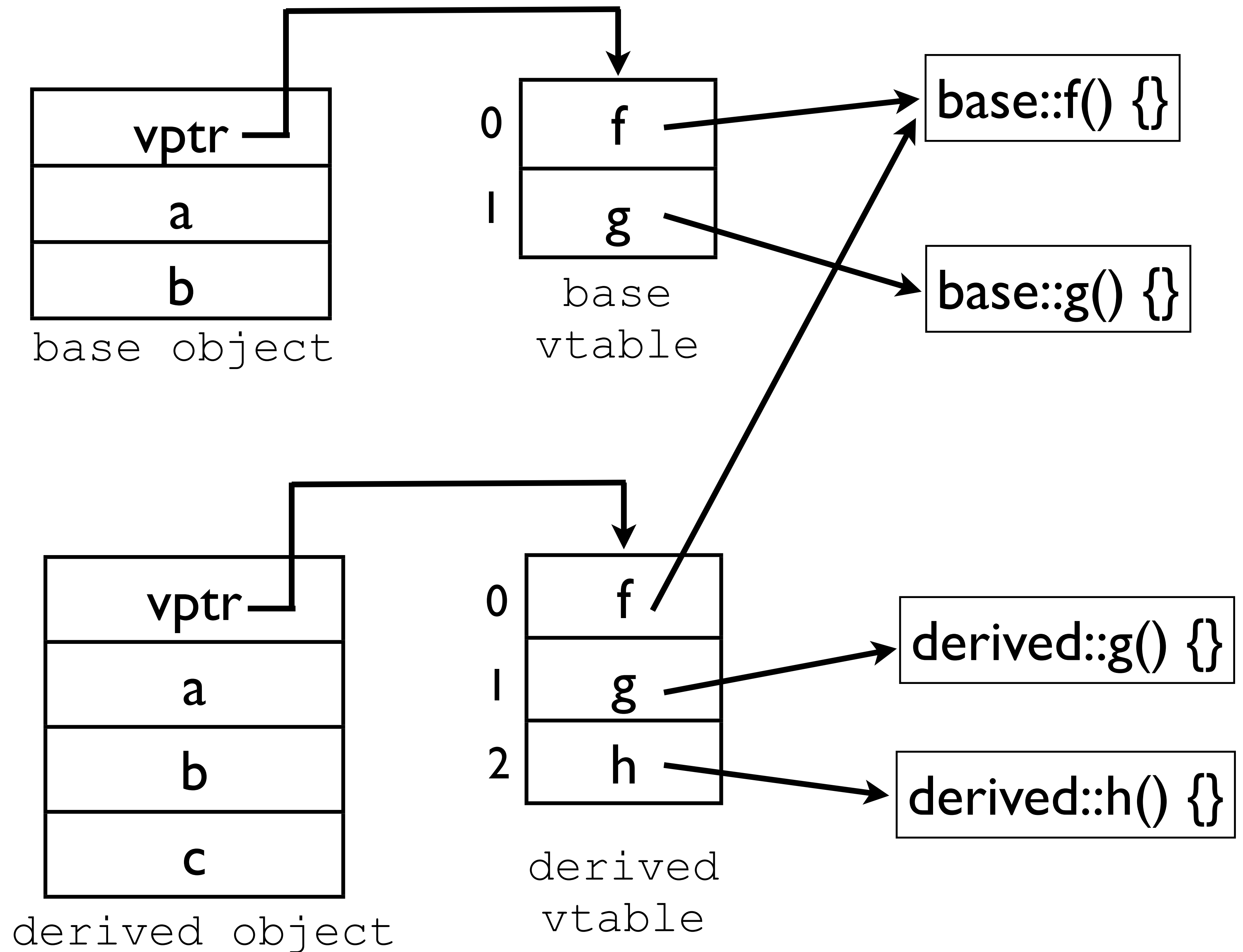

Deferent
Epicycle
Earth
Planet

```cpp
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
24
```

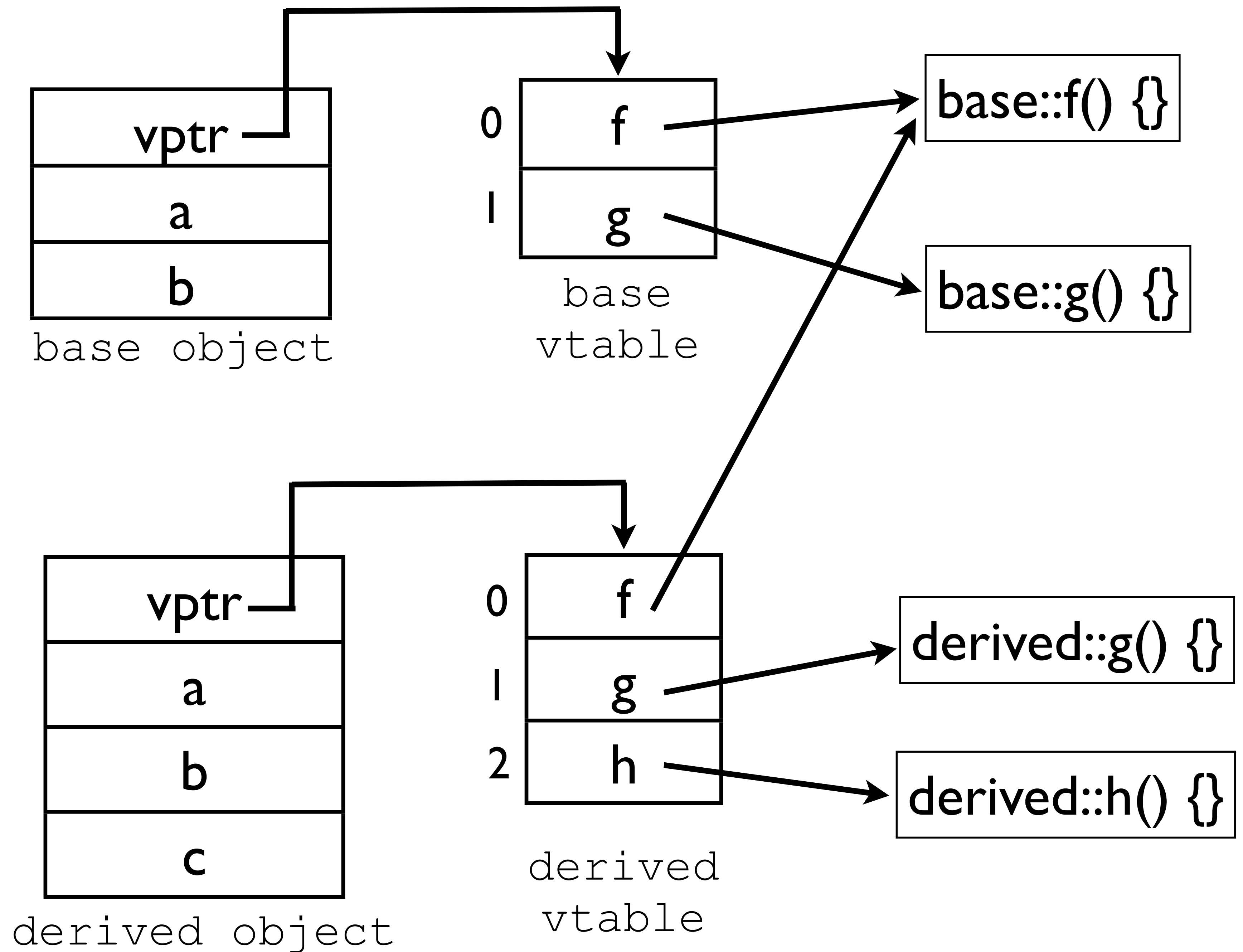
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



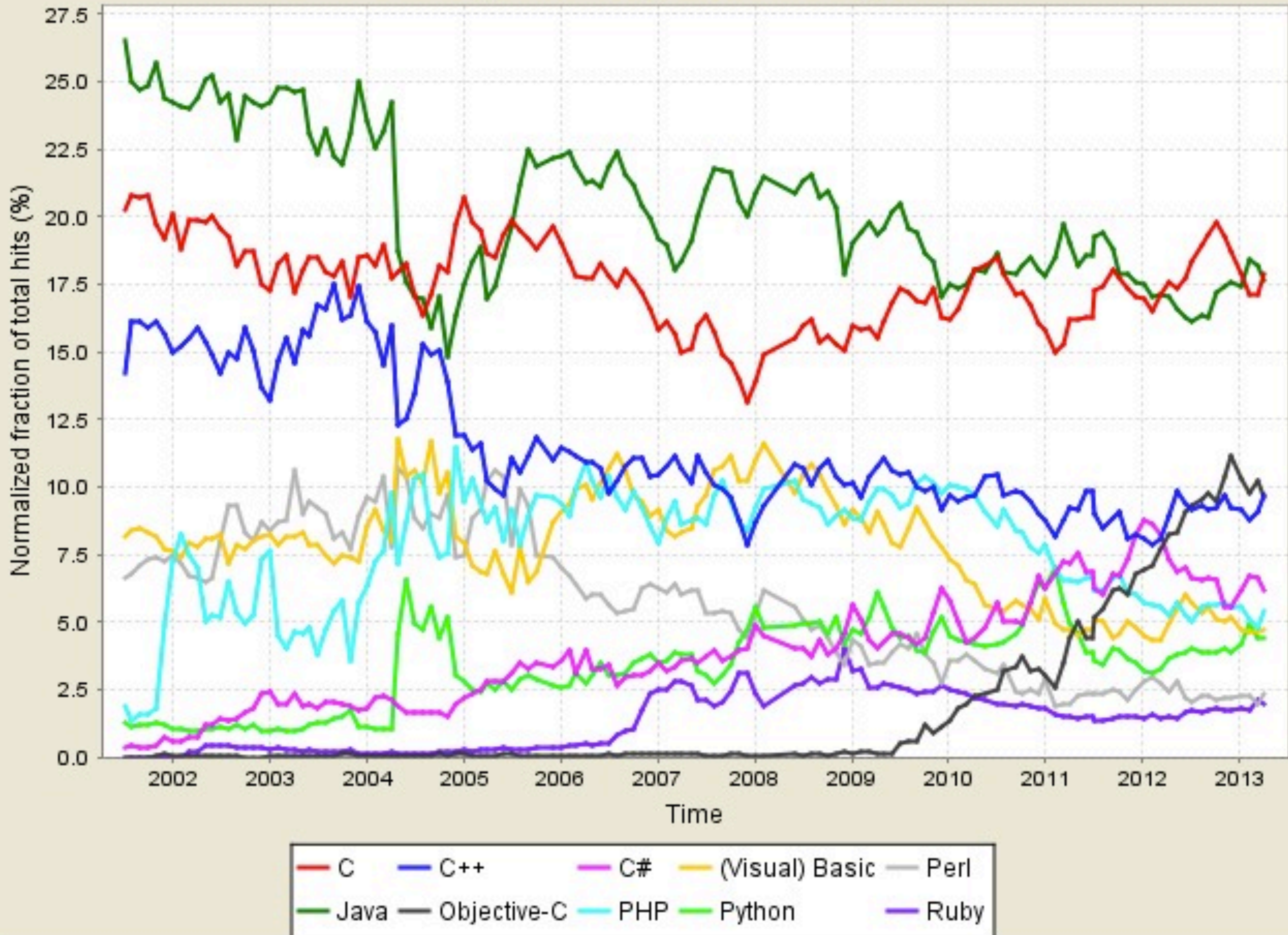
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



This is a common way of implementing virtual functions in C++

Why C++



www.tiobe.com

Apr 2013

Programming Language	Position Apr 2013	Position Apr 2008	Position Apr 1998	Position Apr 1988
C	1	2	1	1
Java	2	1	3	-
C++	3	3	2	5
Objective-C	4	42	-	-
C#	5	8	-	-
PHP	6	4	-	-
(Visual) Basic	7	5	4	8
Python	8	7	28	-
Perl	9	6	6	21
Ruby	10	10	-	-
Lisp	13	16	12	2
Ada	21	17	14	3

Where not to use C++ ?

Web portals?

Min meldingsboks - Altinn | Gmail - Compose Mail

ETEN I BRONNOYSUND [NO] https://www.altinn.no/Pages/ServiceEngine/MyMainPage/MyMainPage.aspx

gelsk ▾ Vil du ha den oversatt? Oversett Nei takk Alternativer ▾

Aktuelt og presse | Kontakt og hjelp | Driftsmeldinger | Altinn A-Å | Om Altinn | Søk

English

Nynorsk

representerer nå KENNETH Logg ut

boks | Skjema og tjenester | Starte og drive bedrift | Min profil | Tilgangsstyring

▾ Velg hva som skal vises i listen – ▾ Til min behandling

- Den jeg representerer nå
  KENNETH
- Alle jeg kan representere

Velg periode -

Søk på tittel

Oppdater

Element 1 - 21 av 21

Vis pr. side 50 Side 1 av 1

Tittel Dato↓ Fra/Til/Endret av	Status Frist/Ref.	Handlinger
Selvangivelse 2011 20.03.2012 00:00:00 Fra: Skatteetaten	Ulest	Slett Overstyr tilgang >
RF-1030 Selvangivelse for lønnstakere og pensjonister mv. 2011 20.03.2012 00:00:00 Endret av: Skatteetaten	Utfylling 30.04.2012 23:59:59	Utskrift Slett Om skjema
Skatteoppgjøret 2010 for lønnstakere/pensjonister 20.06.2011 20:08:54 Fra: Skatteetaten	Lest	Slett Overstyr tilgang >
Selvangivelse 2010 22.03.2011 20:57:51 Fra: Skatteetaten	Lest	Slett Overstyr tilgang >
Skatteoppgjøret 2009 for lønnstakere/pensjonister 24.06.2010 09:37:48 Fra: Skatteetaten	Lest	Slett Overstyr tilgang >
Selvangivelsen 2009 23.03.2010 11:01:20 Fra: Skatteetaten	Lest	Slett Overstyr tilgang >
Årsoppgave 22.03.2010 13:55:27 Fra: Husbanken	Lest	Slett Overstyr tilgang >

- Innføring – Min meldingsboks
- Samle og søke informasjon
- Hjelp til å finne skjema
- Hjelp til å rapportere for andre

Introduksjon og hjelp

Her i Min meldingsboks er alle skjema og tjenester du har under arbeid, eller har sendt og mottatt gjennom Altinn.

I menyen til venstre finner du noen valg som gjør det enklere å finne frem til de dokumentene du er ute etter. Bruk også søkefeltene øverst på siden for å finne fram.

Web portals?

Min meldingsboks - Altinn | Gmail - Compose Mail

https://www.altinn.no/Pages/ServiceEngine/MyMainPage/MyMainPage.aspx

Vil du ha den oversatt? Oversett | Nei takk | Alternativer

Aktuelt og presse | Kontakt og hjelp | Driftsmeldinger | Altinn A-Å | Om Altinn | Søk | English | Nynorsk

representerer nå KENNETH | Logg ut

Skjema og tjenester | Starte og drive bedrift | Min profil | Tilgangsstyring

Velg hva som skal vises i listen – Til min behandling

- Den jeg representerer nå: KENNETH
- Alle jeg kan representere

Velg periode: -
Søk på tittel:

Oppdater

Element 1 - 21 av 21 | Vis pr. side 50 | Side 1 av 1

Tittel / Dato / Fra/Til/Endret av	Status / Frist/Ref.	Handlinger
Selvangivelse 2011 20.03.2012 00:00:00 Fra: Skatteetaten	Ulest	Slett, Overstyr tilgang >
RF-1030 Selvangivelse for lønnstakere og pensjonister mv. 2011 20.03.2012 00:00:00 Endret av: Skatteetaten	Utfylling 30.04.2012 23:59:59	Utskrift, Slett, Om skjema
Skatteoppgjøret 2010 for lønnstakere/pensjonister 20.06.2011 20:08:54 Fra: Skatteetaten	Lest	Slett, Overstyr tilgang >
Selvangivelse 2010 22.03.2011 20:57:51 Fra: Skatteetaten	Lest	Slett, Overstyr tilgang >
Skatteoppgjøret 2009 for lønnstakere/pensjonister 24.06.2010 09:37:48 Fra: Skatteetaten	Lest	Slett, Overstyr tilgang >
Selvangivelsen 2009 23.03.2010 11:01:20 Fra: Skatteetaten	Lest	Slett, Overstyr tilgang >
Årsoppgave 22.03.2010 13:55:27 Fra: Husbanken	Lest	Slett, Overstyr tilgang >

- Innføring – Min meldingsboks
- Samle og søke informasjon
- Hjelp til å finne skjema
- Hjelp til å rapportere for andre

Introduksjon og hjelp

Her i Min meldingsboks er alle skjema og tjenester du har under arbeid, eller har sendt og mottatt gjennom Altinn.

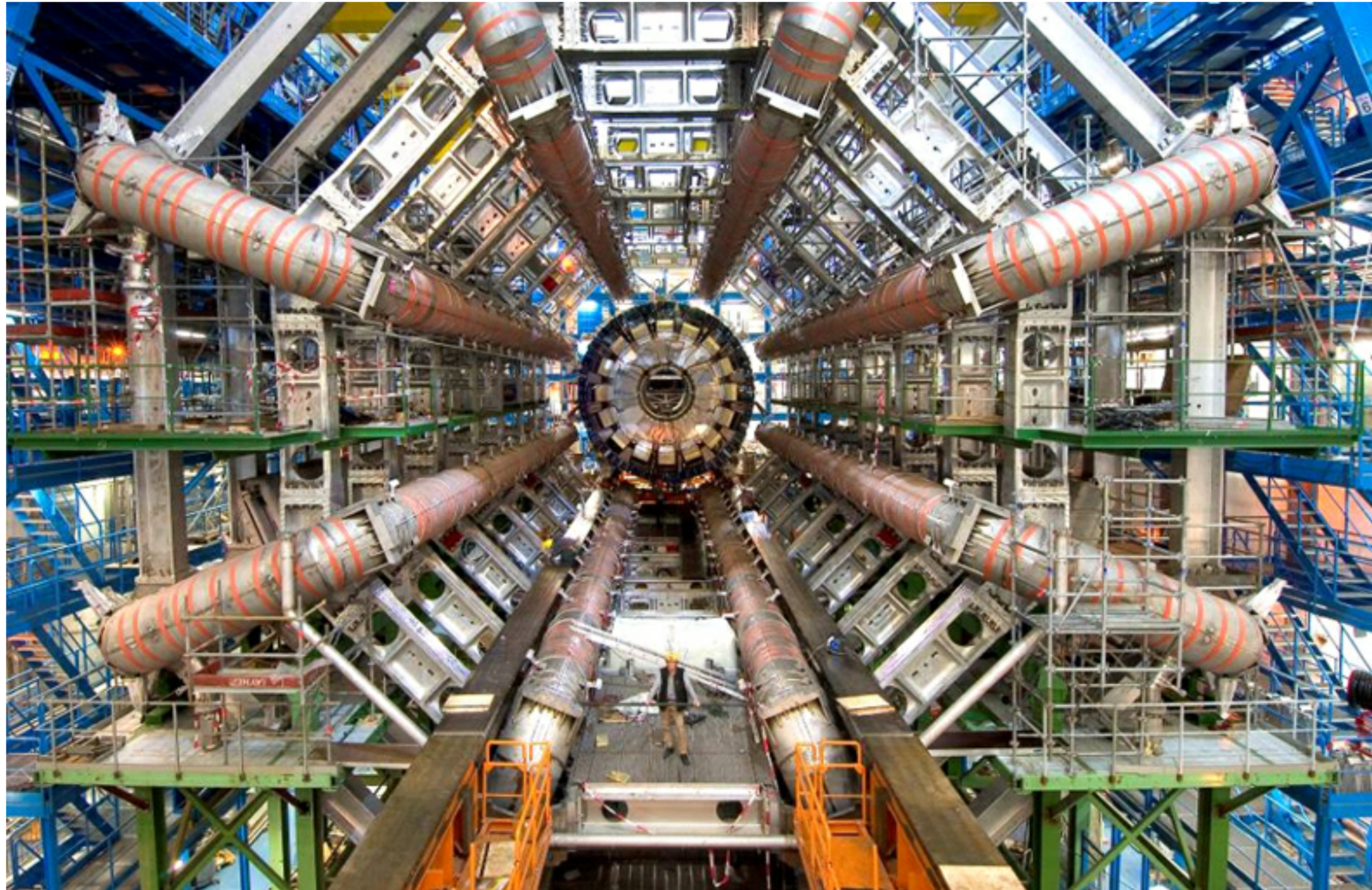
I menyen til venstre finner du noen valg som gjør det enklere å finne frem til de dokumentene du er ute etter. Bruk også søkefeltene øverst på siden for å finne fram.

Very simple stuff?

Where is C++ relevant?

embedded

supercomputing



games

realtime

datacenters

green computing

mobile computing

multimedia systems

competition programming

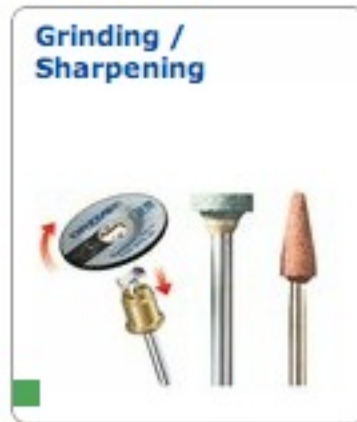
Where is C++ relevant?

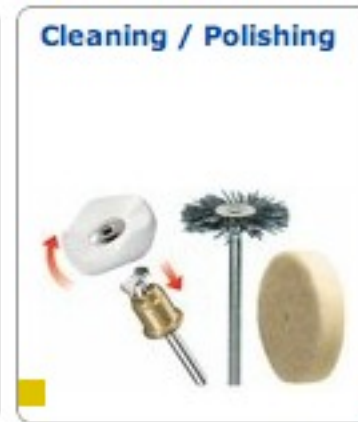
embedded
supercomputing
games
realtime
datacenters
green computing
mobile computing
multimedia systems
competition programming

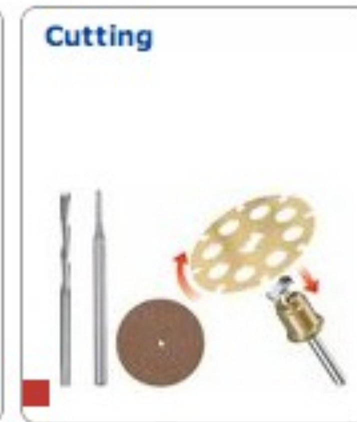
We are working on a new project...
Which language to use?

can a script language do the job?

is there support for a vm based language?







is it convenient to use just C?

if no to all previous questions,
then you may want to consider C++

C++

History of C++

- PhD, Simula, BCPL (Cambridge)
- C with Classes (Cpre, 1979)
- First external paper (1981)
- C++ named (1983)
- CFront 1.0 (1985)
- TC++PL, Ed1 (1985)
- ANSI X3J16 meeting (1989)
- The Annotated C++ Reference Manual (1990)
- First WG21 meeting (1991)
- The Design and Evolution of C++ (1994)
- ISO/IEC 14882:1998 (C++98)
- ISO/IEC 14882:2003 (C++03)
- ISO/IEC TR 19768:2007 (C++TR1)
- ISO/IEC 14882:2011 (C++11)

Why C++?

Virtual vs **Native**
Productivity vs **Performance**
Effectiveness vs **Efficiency**

Hello, world!

Hello, world!

Everything is easier if you learn to use the proper terminology when discussing C++.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

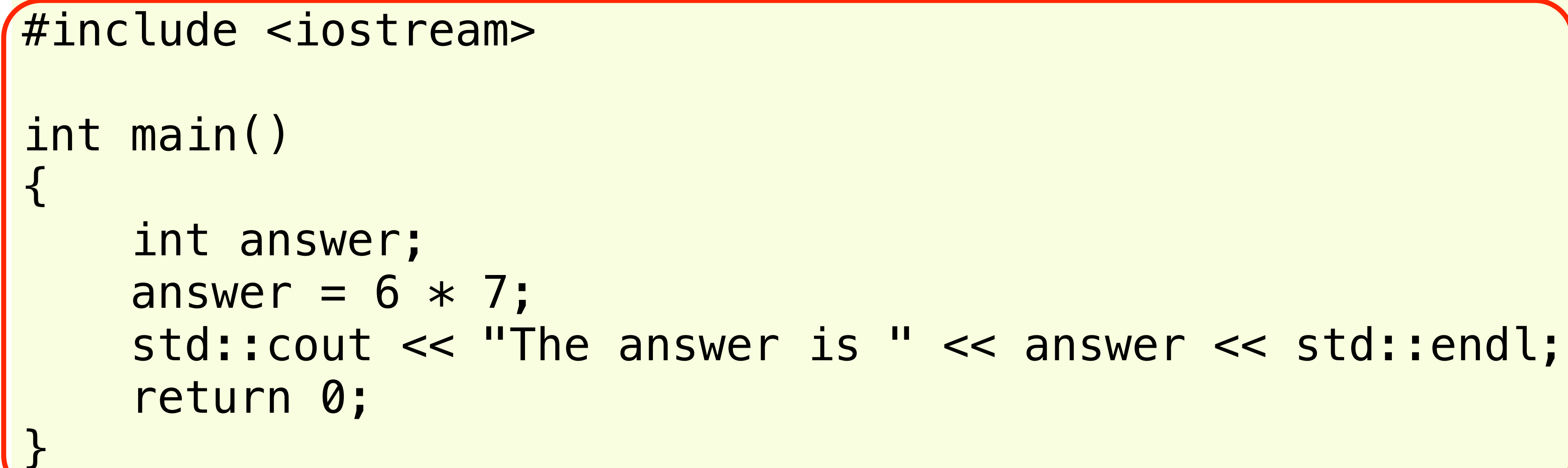
```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here is an example of how you might compile and execute this program in a Unix-like environment.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

Let's analyze what we see here.

the_answer.cpp



```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is the source code for our program.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

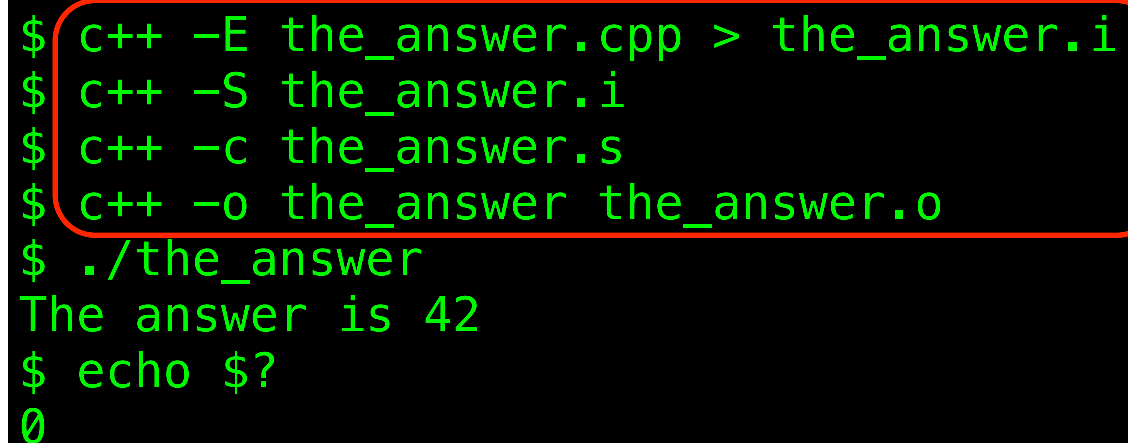
```cpp
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Our code is stored in a named file - the **source file**.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```



the_answer.cpp

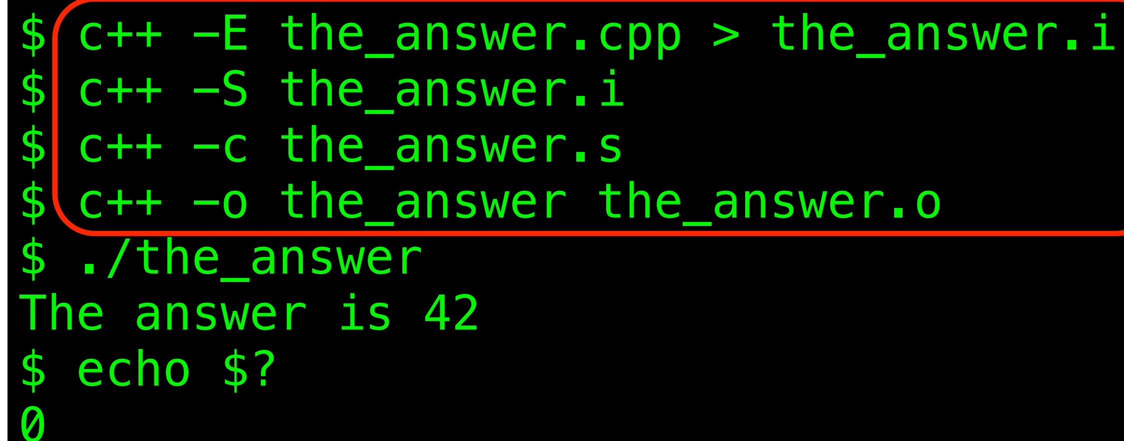
```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

We could have done this in one step instead, like:

```
c++ -o the_answer the_answer.cpp
```

This is often referred to as "compiling the program"

But it is useful to understand what is happening under the hood.



the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The source file is first preprocessed. The **preprocessor** imports other files specified by include directives and does macro processing. The result is a **translation unit**.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The **translation step** converts the C++ program into **assembler language.**

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The **assembler** converts the code into **machine code** and creates an **object file**.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The **linker** binds together object files and libraries. In this case our object file, the **standard c++ library**, and a **c++ run-time library** is linked together to create an **executable file**.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Here we tell the **execution environment** (operating system) to **load** the file into memory and **start** the program.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```cpp
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The expected result is printed to the **standard output stream**, which in this case is the console.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

When our program terminates it will return the value 0 to the execution environment to indicate success. A non-zero **return status** usually indicates failure.

```
$ c++ -E the_answer.cpp > the_answer.i
$ c++ -S the_answer.i
$ c++ -c the_answer.s
$ c++ -o the_answer the_answer.o
$ ./the_answer
The answer is 42
$ echo $?
0
```

```cpp
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```cpp
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is an **include directive** for the preprocessor. In this case the headers for the standard IO library of C++ will be expanded into our program so that they can be used. The <> brackets indicates that the file can be found in the standard include directories for the compiler.

```
#include <iostream>

int main()
{
     int answer;
     answer = 6 * 7;
     std::cout << "The answer is " << answer << std::endl;
     return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Every C++ program must have a **`main()`** function. This is the entry point for the program and it is usually called by the C++ run-time library after some preparations and initializations. By definition the main function should return an integer to indicate the **return status**.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

These curly braces encloses a **block** of **statements**.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is a statement, a **declaration statement**.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is a statement, a **declaration statement**.

Here we declare a **local variable** or, more precisely, an **automatic object** named `answer`. It has automatic storage duration, which means that it is created with an **indeterminate value** when we enter the block ({) and automatically destroyed when we exit the block (}).

All objects must have a particular **type**, in this case the variable is a **signed integer**.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is also a statement, an **assignment statement.**

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is also a statement, an **assignment statement.**

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

Without the semicolon, this is an **expression**. An expression will be computed and yield a result.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is also an **expression**, or actually, a **sub-expression** that will be computed. The highlighted expression has one **operator** (*) and two **operands** (6 and 7). Operands have a type, in this case two signed integers, and the result of applying the multiplication operator with these two operands gives a value that also is of type signed integer.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The equal sign is in fact also an operator, the **assignment operator**. After 6*7 is evaluated, the result together with the variable `answer` are operands to the assignment operator. In this case, both operands are of type signed integer so no **type conversions** are necessary. This is an example of a **full-expression**.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The equal sign is in fact also an operator, the **assignment operator**. After 6*7 is evaluated, the result together with the variable `answer` are operands to the assignment operator. In this case, both operands are of type signed integer so no **type conversions** are necessary. This is an example of a **full-expression**.

§1.9.10 A full-expression is an expression that is not a subexpression of another expression.

Understanding **full-expression** is important when reasoning about sequencing and temporary objects, we will discuss this more later.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The equal sign is in fact also an operator, the **assignment operator**. After 6*7 is evaluated, the result together with the variable `answer` are operands to the assignment operator. In this case, both operands are of type signed integer so no **type conversions** are necessary. This is an example of a **full-expression**.

§1.9.10 A full-expression is an expression that is not a subexpression of another expression.

Understanding **full-expression** is important when reasoning about sequencing and temporary objects, we will discuss this more later.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

With the semicolon this becomes an **expression statement**. In this case the result of computing the whole expression is discarded, but it has a significant **side-effect** in changing the stored value of variable `answer`.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

This is also a statement, an expression statement with three operators and while the result of the computation is discarded, it is the side-effects of computing the expression that is interesting.

Since the **output operator** (<<) is **left-associative** the whole statement could have been written like this:

```
(((std::cout << "The answer is ") << answer) << std::endl);
```

and now you see there are three sub-expressions to be computed to complete the statement.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

When analyzing a large expression, it is useful to consider the types involved.

std::ostream

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

When analyzing a large expression, it is useful to consider the types involved.

std::ostream

string literal

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

When analyzing a large expression, it is useful to consider the types involved.

std::ostream

string literal

signed int

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

When analyzing a large expression, it is useful to consider the types involved.

std::ostream
string literal
signed int
manipulator

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

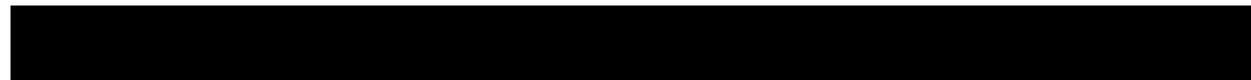
When analyzing a large expression, it is useful to consider the types involved.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

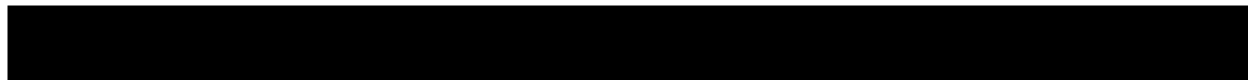

When analyzing a large expression, it is useful to consider the types involved.

In this case, when the << operator is applied to the operands the side-effect is that something is printed to the standard output stream and the result after evaluating the sub-expression is std::ostream. For example:

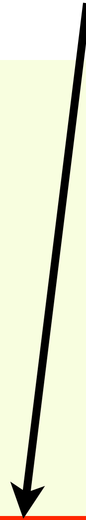
```
std::cout << "The answer is " << answer << std::endl;
```



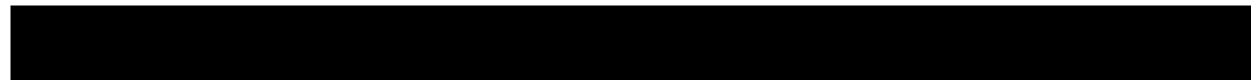
```
std::cout << "The answer is " << answer << std::endl;
```



std::ostream



```
std::cout << "The answer is " << answer << std::endl;
```



(object of type std::ostream) `<< answer << std::endl;`

```
The answer is
```

```
(object of type std::ostream)          << answer << std::endl;
```

```
The answer is
```

(object of type std::ostream) `<< std::endl;`

```
The answer is 42
```

(object of type std::ostream) `<< std::endl;`

```
The answer is 42
```

(object of type std::ostream)

```
The answer is 42
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

return is a keyword to give control back to the caller of a function. In this case the value 0 is returned to indicate success to the execution environment.

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

The block is finished. Local variables are implicitly destroyed (feel free to think garbage collection) and since this is the end of `main()` the program terminates and the control is given back to the executing environment.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

That concludes the initial analysis of our little Hello, world! program.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

That concludes the initial analysis of our little Hello, world! program.

However, there are a couple of things I should mention.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer;
    answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

It is common to declare and initialize an object at the same time. In C++ there are several ways of initializing an object.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer(6 * 7);

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer(6 * 7);

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

This is sometimes called a function-style initializer, as it is calling a constructor to create the object. Not so common to see this notation for simple data types.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

This is a more common notation. However, it is easy to confuse this with an assignment. It is *not* an assignment, but an initialization of the variable. This becomes a very important distinction when working with more complex objects.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

This is a more common notation. However, it is easy to confuse this with an assignment. It is *not* an assignment, but an initialization of the variable. This becomes a very important distinction when working with more complex objects.

Aside: In C++11 there is a third way of initializing objects, often called **uniform initialization**, we might discuss this later.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;
    return 0;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

And, finally, `main()` is special in many ways. One of them is that you don't really need to explicitly return a value. If you just drop out of the main function, then by default the value to indicate success is returned to the execution environment. Indeed, in C++ is common practice **not** to explicitly return from main.

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;

    std::cout << "The answer is " << answer << std::endl;

}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;

}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

Hello, world!

the_answer.cpp

```
#include <iostream>

int main()
{
    int answer = 6 * 7;
    std::cout << "The answer is " << answer << std::endl;
}
```

```
$ c++ -o the_answer the_answer.cpp
$ ./the_answer
The answer is 42
$ echo $?
0
```

This is a nice "Hello, world!" program.

Data, Code and Program Organization

Todo?:
include <cstddef> properly
include <string> properly
use const std::string &

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

Here we create an array of integer values and then print them.

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

Here we create an array of integer values and then print them.

```
$ c++ -Wall -Wextra -pedantic foo.cpp && ./a.out
1990
2001
2000
1988
1995
```

```cpp
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

This is a preprocessor directive. Before the C++ compiler kicks in, the preprocessor will do macro processing. In this case, replacing every token NELEMS with the number 5.

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

This is a preprocessor directive. Before the C++ compiler kicks in, the preprocessor will do macro processing. In this case, replacing every token NELEMS with the number 5.

```
... maybe 16000 lines of headers...

#define NELEMS 5

int main()
{
    int values[   5  ] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i <    5   ; ++i)
        std::cout << values[i] << std::endl;
}
```

```
... maybe 16000 lines of headers...

#define NELEMS 5

int main()
{
    int values[   5  ] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i <    5   ; ++i)
        std::cout << values[i] << std::endl;
}
```

... and now the “real” C++ compiler can do its job.

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

The preprocessor is not really a part of the proper language, and it often causes a lot of troubles. For example, the preprocessor is partly the reason for why it is so difficult to develop good integrated development environments for C and C++.

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

The preprocessor is not really a part of the proper language, and it often causes a lot of troubles. For example, the preprocessor is partly the reason for why it is so difficult to develop good integrated development environments for C and C++.

Sometimes we do not have a choice, but in C++ we tend to avoid using the preprocessor as much as possible.

There are often better ways of doing things in C++.

For example, we can define a constant.

```
#include <iostream>

#define NELEMS 5

int main()
{
    int values[NELEMS] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < NELEMS; ++i)
        std::cout << values[i] << std::endl;
}
```

The preprocessor is not really a part of the proper language, and it often causes a lot of troubles. For example, the preprocessor is partly the reason for why it is so difficult to develop good integrated development environments for C and C++.

Sometimes we do not have a choice, but in C++ we tend to avoid using the preprocessor as much as possible.

There are often better ways of doing things in C++.

For example, we can define a constant.

```
#include <iostream>

const int nelems = 5;

int main()
{
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

const int nelems = 5;

int main()
{
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

const int nelems = 5;

int main()
{
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

const int nelems = 5;

int main()
{
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

It is often a good idea to define variables close to use.

```
#include <iostream>

const int nelems = 5;
int main()
{

    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

const int nelems = 5;
int main()
{

    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>


int main()
{
    const int nelems = 5;
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>


int main()
{
    const int nelems = 5;
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const int nelems = 5;
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const int nelems = 5;
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

A nice rule of thumb is to use long and explicit names for symbols (eg, variable names and function names) with a large scope, and small names for small scope.

Since the constant is used in a very small context here, I would probably use an even shorter name for the constant.

```
#include <iostream>

int main()
{
    const int nelems = 5;
    int values[nelems] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < nelems; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const int n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const int n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

In code like this, I would probably also choose to calculate the size of the array, rather than specify it explicitly.

```
#include <iostream>

int main()
{
    const int n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };


    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const int n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{

    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

unary-expression:
  sizeof *unary-expression*
  sizeof *(type-name)*

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

`sizeof` returns the number of bytes of memory needed to store a particular object or type. Here we just take the size of the whole value array and divide it by the size of one element. This shows one common way of calculating the number of elements.

unary-expression:
  sizeof *unary-expression*
  sizeof *(type-name)*

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

`sizeof` returns the number of bytes of memory needed to store a particular object or type. Here we just take the size of the whole value array and divide it by the size of one element. This shows one common way of calculating the number of elements.

the value returned from sizeof is really an unsigned integer type, more precisely a `std::size_t` data type.

Using a signed int, instead of the proper `std::size_t` can cause big problems in larger projects, especially when porting code between different architectures, so better get used to the proper types when appropriate.

unary-expression:
  sizeof *unary-expression*
  sizeof (*type-name*)

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

`sizeof` returns the number of bytes of memory needed to store a particular object or type. Here we just take the size of the whole value array and divide it by the size of one element. This shows one common way of calculating the number of elements.

the value returned from sizeof is really an unsigned integer type, more precisely a `std::size_t` data type.

Using a signed int, instead of the proper `std::size_t` can cause big problems in larger projects, especially when porting code between different architectures, so better get used to the proper types when appropriate.

unary-expression:
    sizeof *unary-expression*
    sizeof (*type-name*)

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    int         n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

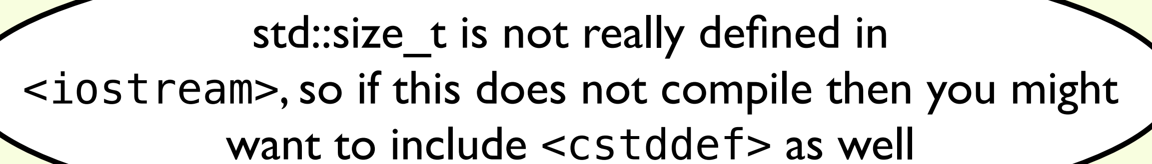
and while we are at it. When indexing into an array, you should also use `std::size_t`. Like this...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (int i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (int          i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```



```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

unary-expression:
  sizeof *unary-expression*
  sizeof (*type-name*)

Notice that when calculating the sizeof an expression you don't need to use parenthesis, but if you want to calculate the size of a type you must use parenthesis. Since it is considered a good practice to take the size of an object rather than the size of a type, there is often a preference to use the ***sizeof*** *unary-expression* version. And in this case it is easy to do...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof(int);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

unary-expression:
 sizeof *unary-expression*
 sizeof *(type-name)*

Notice that when calculating the sizeof an expression you don't need to use parenthesis, but if you want to calculate the size of a type you must use parenthesis. Since it is considered a good practice to take the size of an object rather than the size of a type, there is often a preference to use the ***sizeof*** *unary-expression* version. And in this case it is easy to do...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Here we just divide the size of the whole array, by the size of the first element.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Here we just divide the size of the whole array, by the size of the first element.

Another way of taking the size of the first element is just to “misuse” the fact that `values`, in this case, is treated as a pointer to the first element and therefore we can dereference it.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Here we just divide the size of the whole array, by the size of the first element.

Another way of taking the size of the first element is just to "misuse" the fact that `values`, in this case, is treated as a pointer to the first element and therefore we can dereference it.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof *values;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof *values;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Never eat at a place called "Mom's." Never play cards with a man called "Doc."

And never, ever, forget that C treats an l-value of type array-of-T in an expression as a pointer to the first element of the array.

- C programmers' saying (Traditional)

[source: Expert C programming, chapter 9, Peter van der Linden]

Aside: Unfortunately, a lot of C and C++ programmers believe that arrays and pointers are the same thing. They are not. But sometimes an array can be treated as a pointer to the first element.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof *values;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

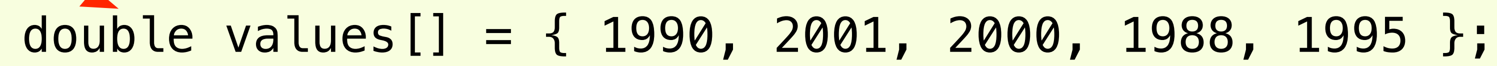
```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

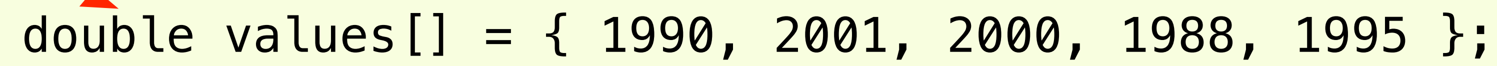
Notice now that the whole calculation required to determine `n` is type independent. This is a nice feature. We can change the type of the `values[]` array and the calculation will still be correct.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    double values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```



```
#include <iostream>

int main()
{
    double values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```



```
1990
2001
2000
1988
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```cpp
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

You can even write a macro to do the calculation and it will work for arrays of all types.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

You can even write a macro to do the calculation and it will work for arrays of all types.

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = COUNT_OF(values);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = COUNT_OF(values);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

This is a kind of “poor-man” generic programming technique.

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = COUNT_OF(values);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

This is a kind of "poor-man" generic programming technique.

But once again, the "rule of thumb" of avoiding macros kicks in... it is probably better to use the explicit and idiomatic way.

```
#include <iostream>

#define COUNT_OF(x) (sizeof(x) / sizeof(x[0]))

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = COUNT_OF(values);

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

This is a kind of "poor-man" generic programming technique.

But once again, the "rule of thumb" of avoiding macros kicks in... it is probably better to use the explicit and idiomatic way.

Aside: C++ have support for proper generic programming called templates. We will see more of that later. If you want to play around with the template system in C++, there are "clever" ways to create a template version of COUNT_OF (it's cool to know, but I am not sure how useful it is by itself though...)

```cpp
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Enough about macros and templates for now... let’s talk about pointers and references instead.

```cpp
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

This is an example of a reference. We create an alias to the third object inside the array, and then we change the value through this alias reference.

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

This is an example of a pointer. We take the address of the fourth object (by using the & operator). The value of `fourth_value` is now an address into the program memory. In the second line we dereference the pointer (the address) so we end up with an 'alias' to the fourth element in the array and assign it the value 4.

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Of course, we don't really need to explicitly create a reference or a pointer to poke into memory. Here is a direct way of doing it instead.

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    third_value = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Of course, we don't really need to explicitly create a reference or a pointer to poke into memory. Here is a direct way of doing it instead.

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    values[2] = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int & third_value = values[2];
    values[2] = 3;

    int * fourth_value = &values[3];
    *fourth_value = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;

    int * fourth_value = &values[3];
    *(&values[3]) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```cpp
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;


    *(&values[3]) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;


    *(&values[3]) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;


    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;


    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];


    values[2] = 3;


    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

The [] is just an operator taking two arguments. For native types like this, it is just like a short hand notation for adding 2 to the base pointer of the array, and then dereference. Like this...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

The [] is just an operator taking two arguments. For native types like this, it is just like a short hand notation for adding 2 to the base pointer of the array, and then dereference. Like this...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(values + 3) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(values + 3) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Now, you would not normally do this, but it is very useful to understand these things.

I am not going to show you something cool. Since the operator + is commutative, we can now just write...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(values + 3) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Now, you would not normally do this, but it is very useful to understand these things.

I am not going to show you something cool. Since the operator + is commutative, we can now just write...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(3 + values) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(3 + values) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

and converting back to using the [] notation...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    *(3 + values) = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

and converting back to using the [] notation...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

the [] operator is commutative, so you can write...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << i[values] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << i[values] << std::endl;
}
```

```
1990
2001
3
4
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << i[values] << std::endl;
}
```

```
1990
2001
3
4
1995
```

but why would anyone do that? The only reason I show it here is because it is kind of cool trivia to know :-) Let’s fix it...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    *(values + 2) = 3;
    3[values] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << i[values] << std::endl;
}
```

```
1990
2001
3
4
1995
```

but why would anyone do that? The only reason I show it here is because it is kind of cool trivia to know :-) Let's fix it...

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    values[2] = 3;
    values[3] = 4;

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Now I will show an alternative way to iterate through an array.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = &values[0];
    int * values_end = &values[n];

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;
}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = &values[0];
    int * values_end = &values[n];

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;
}
```

```
1990
2001
2000
1988
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = &values[0];
    int * values_end = &values[n];

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;
}
```

Notice that we create two pointers, the first, `values_begin` pointing at the first element in the array, while `values_end` is actually pointing *one* element after the array. As long as you are not dereferencing the `values_end` pointer, this is perfectly legal C++. The whole example illustrates a very common way of iterating through a collection of objects in C++ as we will see later.

```
1990
2001
2000
1988
1995
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = &values[0];
    int * values_end = &values[n];

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;
}
```

```
1990
2001
2000
1988
1995
```

Notice that we create two pointers, the first, `values_begin` pointing at the first element in the array, while `values_end` is actually pointing *one* element after the array. As long as you are not dereferencing the `values_end` pointer, this is perfectly legal C++. The whole example illustrates a very common way of iterating through a collection of objects in C++ as we will see later.

of course, you can just as well write it like this.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = &values[0];
    int * values_end = &values[n];

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;
}
```

```
1990
2001
2000
1988
1995
```

Notice that we create two pointers, the first, `values_begin` pointing at the first element in the array, while `values_end` is actually pointing *one* element after the array. As long as you are not dereferencing the `values_end` pointer, this is perfectly legal C++. The whole example illustrates a very common way of iterating through a collection of objects in C++ as we will see later.

of course, you can just as well write it like this.

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = values;
    int * values_end = values + n;

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;

}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = values;
    int * values_end = values + n;

    for (int * at = values_begin; at < values_end; ++at)
        std::cout << *at << std::endl;

}
```

Compare iterators using != instead of <

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = values;
    int * values_end = values + n;

    for (int * at = values_begin; at != values_end; ++at)
        std::cout << *at << std::endl;

}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = values;
    int * values_end = values + n;

    for (int * at = values_begin; at != values_end; ++at)
        std::cout << *at << std::endl;

}
```

```
#include <iostream>

int main()
{
    int values[] = { 1990, 2001, 2000, 1988, 1995 };
    std::size_t n = sizeof values / sizeof values[0];

    int * values_begin = values;
    int * values_end = values + n;

    for (int * at = values_begin; at != values_end; ++at)
        std::cout << *at << std::endl;

}
```

Now you have seen the iterator idiom, we will use it a lot later. For now, let's go back to the simpler way of iterating through the array.

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

Suppose these values were not just some random values, but the debut years for some important authors of C++ books.

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int values[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << values[i] << std::endl;
}
```

```cpp
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int debut_years[n] = { 1990, 2001, 2000, 1988, 1995 };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << debut_years[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int debut_years[n] = { 1990, 2001, 2000, 1988, 1995 };
    std::string author_names[n] = { "Bjarne", "Andrei", "Herb", "Andrew", "Bob" };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << author_names[i] << " debuted " << debut_years[i] << std::endl;
}
```

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int debut_years[n] = { 1990, 2001, 2000, 1988, 1995 };
    std::string author_names[n] = { "Bjarne", "Andrei", "Herb", "Andrew", "Bob" };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << author_names[i] << " debuted " << debut_years[i] << std::endl;
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```
#include <iostream>

int main()
{
    const std::size_t n = 5;
    int debut_years[n] = { 1990, 2001, 2000, 1988, 1995 };
    std::string author_names[n] = { "Bjarne", "Andrei", "Herb", "Andrew", "Bob" };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << author_names[i] << " debuted " << debut_years[i] << std::endl;
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

It would be nice to group the `debut_year` and the `author_name` together into a new datatype, a structure.

Here is an example of how to do it in C++.

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

We have created a new data type called author.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

and this is one way of initializing an array of authors.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

Remember we said that << is an operator. One very powerful feature in C++ is operator overloading. You can decide yourself what should happen when applying an operator on certain types.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};


int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};




int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

We have defined what should happen if the output operator << is applied to an object of type `std::ostream` and an object of type `author`.

Let's give it a try...

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a.name << " debuted " << a.year << std::endl;
    }
}
```

We have defined what should happen if the output operator << is applied to an object of type `std::ostream` and an object of type `author`.

Let's give it a try...

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a << std::endl;
    }
}
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

You can overload most operators in C++. Overloaded operators are just like a function, only with a fancy way of calling it.

While unusual, you can actually write it like this...

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        std::cout << a << std::endl;
    }
}
```

You can overload most operators in C++. Overloaded operators are just like a function, only with a fancy way of calling it.

While unusual, you can actually write it like this...

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

Thinking about an operator like this make it easier to figure out what the function signature should look like.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, author a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

Non primitive types should generally be passed by const reference rather than by copy.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

That's better

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

Now we have briefly introduced operator overloading. Used correctly and carefully, it is a very useful feature, but you can also mess up your program completely. For example, if you overload the + operator: does it behave as you would expect? Is it commutative, have you thought about the += operator, and so on.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
Bjarne debuted 1990
Andrei debuted 2001
Herb debuted 2000
Andrew debuted 1988
Bob debuted 1995
```

Now we have briefly introduced operator overloading. Used correctly and carefully, it is a very useful feature, but you can also mess up your program completely. For example, if you overload the + operator: does it behave as you would expect? Is it commutative, have you thought about the += operator, and so on.

Since I need the screen estate... let's get rid of the `operator<<` overload for now so I have room to show something else.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

std::ostream & operator<<(std::ostream & out, const author & a) {
    return out << a.name << " debuted " << a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        author a = cppbook_authors[i];
        operator<<(std::cout, a) << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;

    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i) {
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
    }
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;

}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
}
```

```
1990
2001
2000
1988
1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
}
```

Here we use a function to hide the details of how to get the year attribute out of an `author` object.

As you see, we take a const reference to the object. But we could have taken a pointer to the object instead to achieve the same effect.

```
1990
2001
2000
1988
1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
}
```

Here we use a function to hide the details of how to get the year attribute out of an `author` object.

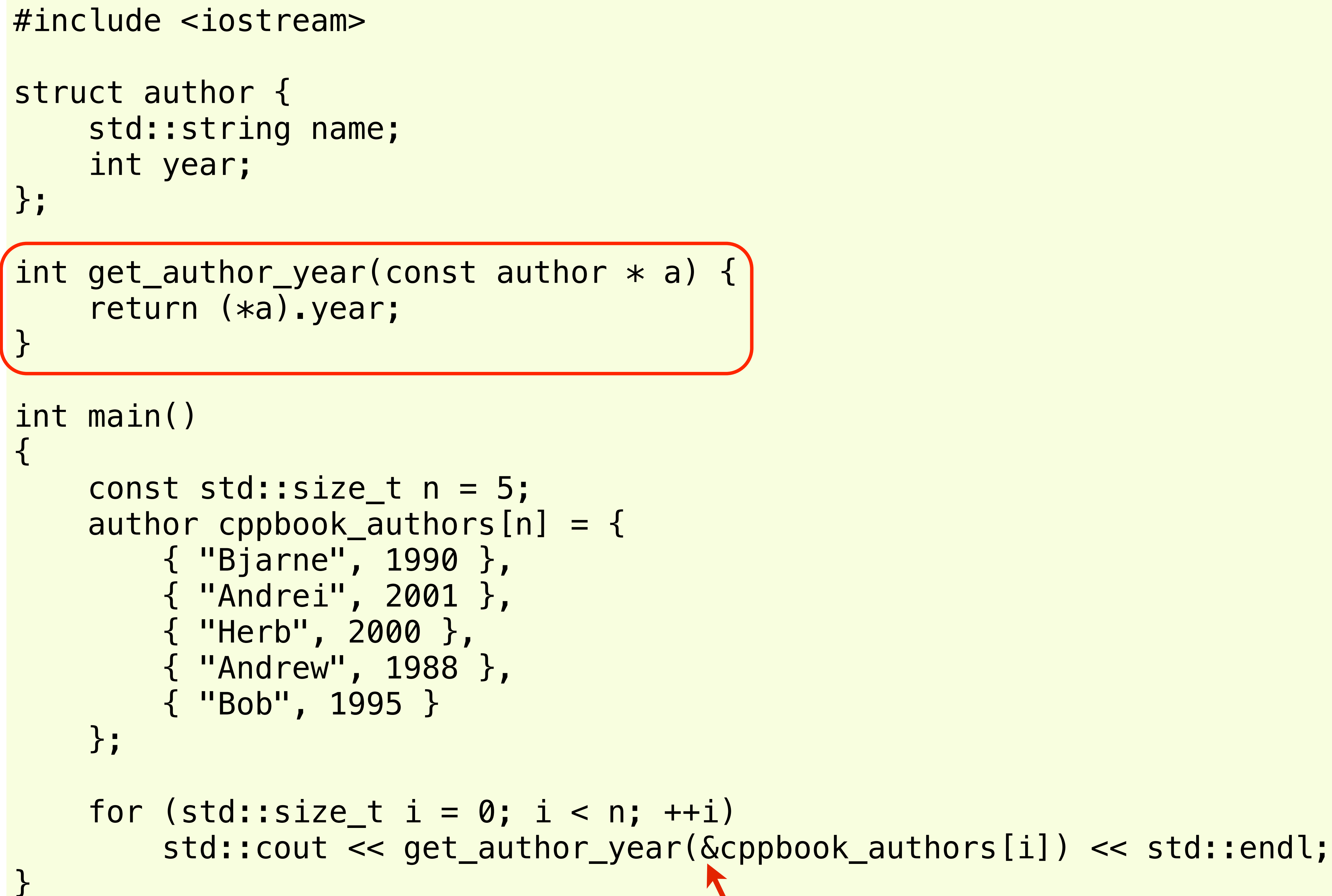
As you see, we take a const reference to the object. But we could have taken a pointer to the object instead to achieve the same effect.

```
1990
2001
2000
1988
1995
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author & a) {
    return a.year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(cppbook_authors[i]) << std::endl;
}
```

Here we use a function to hide the details of how to get the year attribute out of an `author` object.

As you see, we take a const reference to the object. But we could have taken a pointer to the object instead to achieve the same effect.



```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return (*a).year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return (*a).year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return (*a).year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

Here we dereference the a pointer and then access the year attribute of the `author` object.

There is a convenient short hand for doing this. We can use the -> operator which will first dereference the LHS operand, and then give a reference to the specified member.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return (*a).year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

Here we dereference the a pointer and then access the year attribute of the `author` object.

There is a convenient short hand for doing this. We can use the -> operator which will first dereference the LHS operand, and then give a reference to the specified member.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

One of the original ideas of C++, is to define accessor function like this *inside* the struct.

```
#include <iostream>

struct author {
    std::string name;
    int year;
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

One of the original ideas of C++, is to define accessor function like this *inside* the struct.

```
#include <iostream>

struct author {
    std::string name;
    int year;



};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << get_author_year(&cppbook_authors[i]) << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

And now we have two competing ways of reading the year attribute out of an author object.

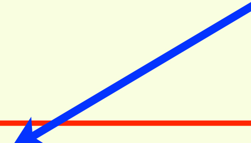
```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

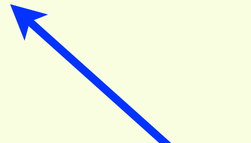
And now we have two competing ways of reading the year attribute out of an author object.

However, they are not exactly the same. Do you see the difference?

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

not const here





const here

And now we have two competing ways of reading the year attribute out of an author object.

However, they are not exactly the same. Do you see the difference?

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

The syntax for specifying that we want a const pointer to the `this` object is to add `const` after the argument list.

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* author * this */) {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

The syntax for specifying that we want a const pointer to the `this` object is to add `const` after the argument list.

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

const here

const here

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

const here

const here

Now they are equivalent. And we can delete the one we don't need.

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int get_author_year(const author * a) {
    return a->year;
}

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year(/* const author * this */) const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

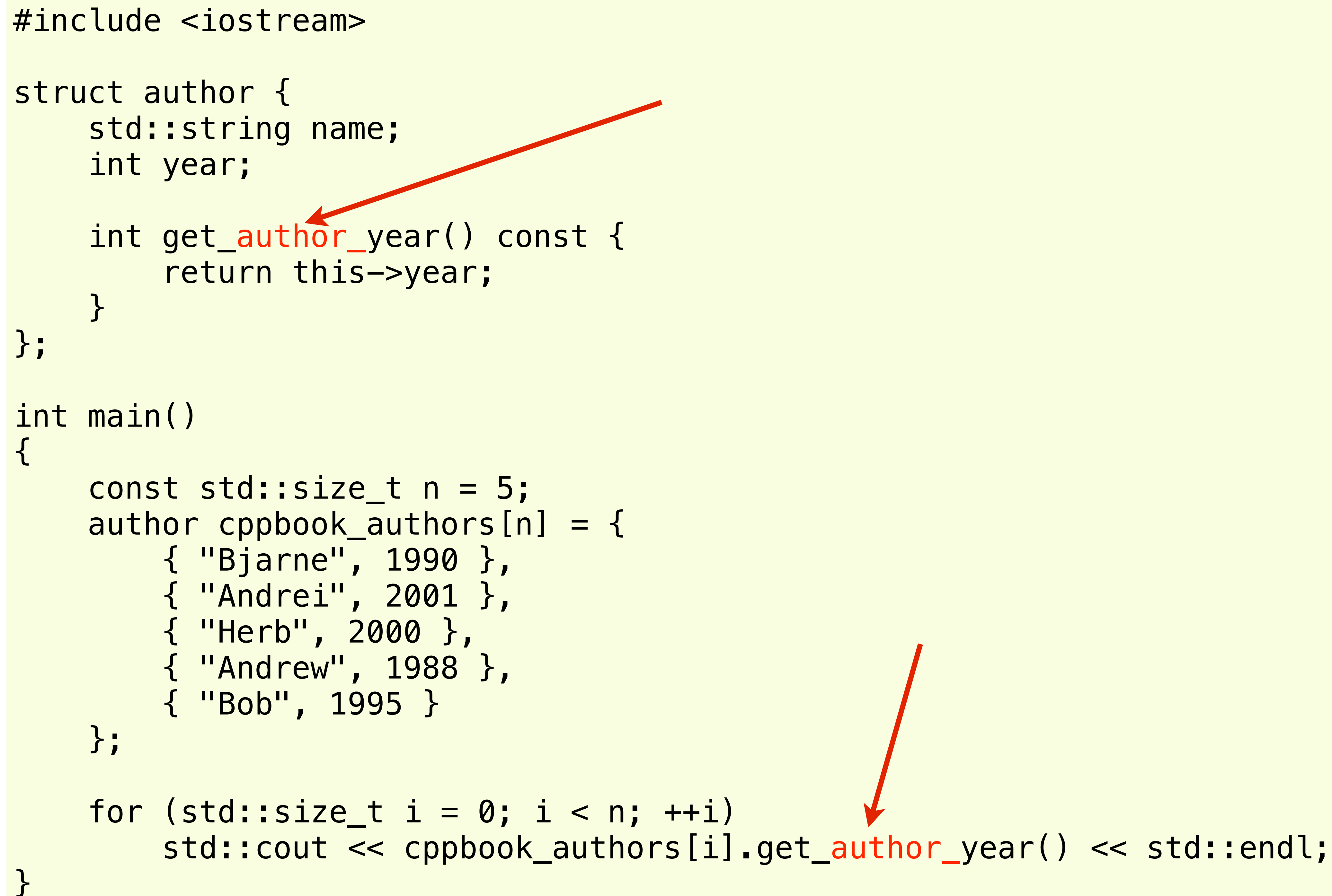
```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

And since the scope is smaller, it make sense to reduce the length of the function name as well.

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

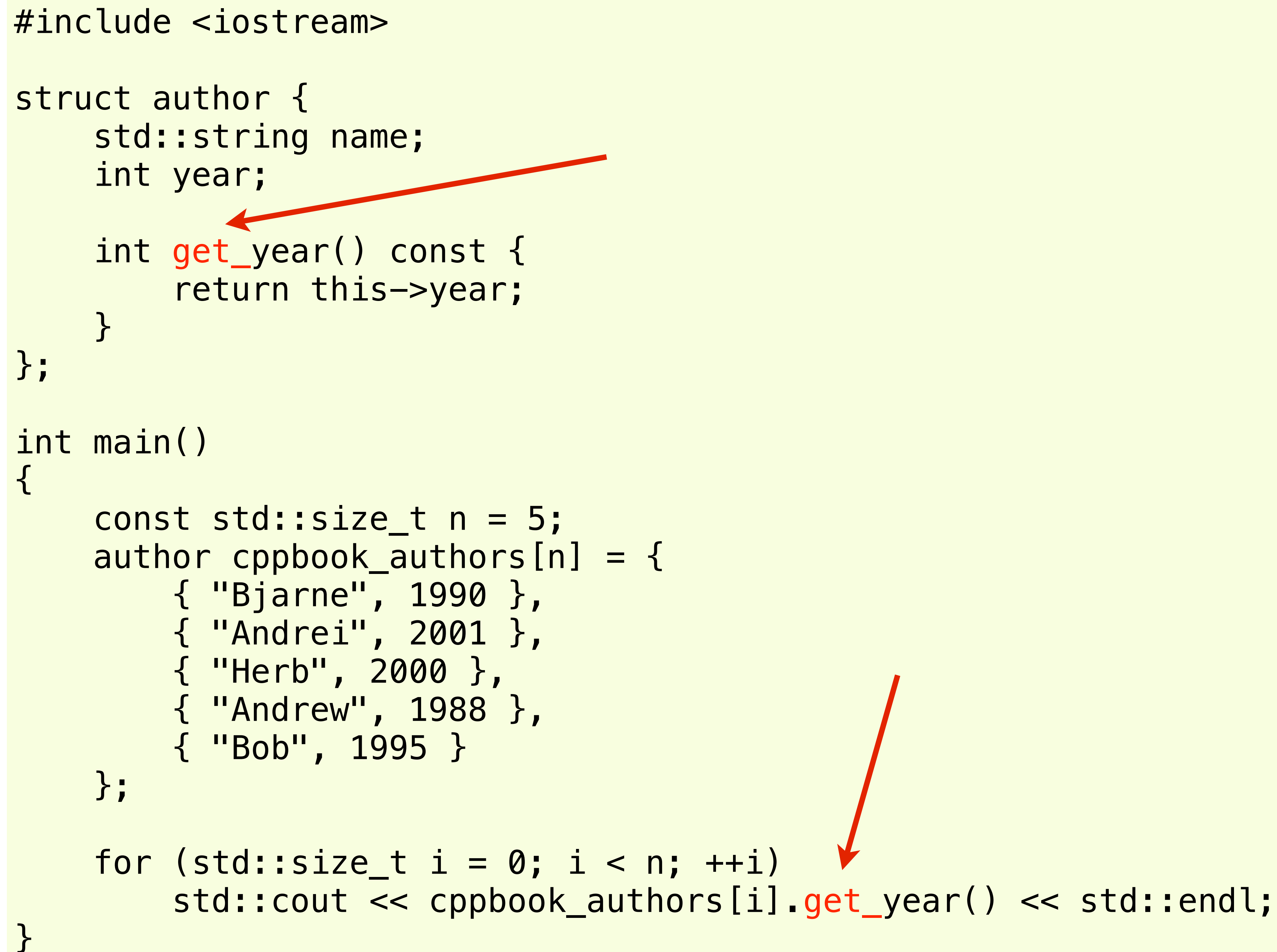


```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_author_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_author_year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_year() << std::endl;
}
```



```cpp
#include <iostream>

struct author {
    std::string name;
    int year;

    int get_year() const {
        return this->year;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].get_year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return this->year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return this->year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return this->year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

Notice that we have now added a suffix to each attribute name, this is to avoid name conflicts inside the struct.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return this->year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return this->year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

We do not need to explicitly dereference the `this` pointer, so you will typically just write.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

And now we have a pattern for how to write query functions inside the struct.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

And now we have a pattern for how to write query functions inside the struct.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }



};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].name() << " debuted "
                  << cppbook_authors[i].year() << std::endl;
}
```

```cpp
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].name() << " debuted "
                  << cppbook_authors[i].year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].name() << " debuted "
                  << cppbook_authors[i].year() << std::endl;
}
```

But wait, we have forgotten something, or actually *someone*, very important in this code snippet.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].name() << " debuted "
                  << cppbook_authors[i].year() << std::endl;
}
```

But wait, we have forgotten something, or actually *someone*, very important in this code snippet.

Scott wrote a seminal book about C++ in 1992. Let's change the code, and just use Scott as an example from now.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    const std::size_t n = 5;
    author cppbook_authors[n] = {
        { "Bjarne", 1990 },
        { "Andrei", 2001 },
        { "Herb", 2000 },
        { "Andrew", 1988 },
        { "Bob", 1995 }
    };

    for (std::size_t i = 0; i < n; ++i)
        std::cout << cppbook_authors[i].name() << " debuted "
                  << cppbook_authors[i].year() << std::endl;
}
```

But wait, we have forgotten something, or actually *someone*, very important in this code snippet.

Scott wrote a seminal book about C++ in 1992. Let's change the code, and just use Scott as an example from now.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Notice how we "initialize" the object here. We use a so called **compound literal** to create a **temporary object** and the use **implicit copy assignment operator** to set the attributes of the a object.

It would be nice to call a function so that we can also check the initialization values.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Notice how we "initialize" the object here. We use a so called **compound literal** to create a **temporary object** and the use **implicit copy assignment operator** to set the attributes of the a object.

It would be nice to call a function so that we can also check the initialization values.

Aside: Using a compound literal here is like a short hand for:

```
{ author tmp = {"Scott", 1992}; a = tmp; }
```

It came is as a feature in C99, and is often supported in C++ compilers although not strictly C++98 compliant.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};




int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    a = (author){"Scott", 1992};
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Now we can check the initialization values.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Now we can check the initialization values.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Throwing an exception is a useful way of indicating an “exceptional” situation. Here we refuse to accept debut years before C++ was invented.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw "year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Throwing an exception is a useful way of indicating an "exceptional" situation. Here we refuse to accept debut years before C++ was invented.

Here we throw a string object. The standard library also define a lot of useful exception classes that we can use in our program.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw "year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Throwing an exception is a useful way of indicating an "exceptional" situation. Here we refuse to accept debut years before C++ was invented.

Here we throw a string object. The standard library also define a lot of useful exception classes that we can use in our program.

```
#include <iostream>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}


int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw “year < 1979 does not make sense";
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw std::invalid_argument("year < 1979 does not make sense");
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw std::invalid_argument("year < 1979 does not make sense");
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw std::invalid_argument("year < 1979 does not make sense");
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

And just like we could move the query functions into the struct, we can also move the initialization function into the struct where it becomes the constructor for objects of type author.

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

void init_author_object(author * a, const std::string & name, int year)
{
    a->name_ = name;
    a->year_ = year;
    if (year < 1979)
        throw std::invalid_argument("year < 1979 does not make sense");
}

int main()
{
    author a;
    init_author_object(&a, "Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

and now we can declare and initialize the object in one step.

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

and now we can declare and initialize the object in one step.

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

The name of the constructor must be the same as the name of the struct. Also, note that the constructor does not have a return type.

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

This is called a **member initialization list** and it is the proper way of initializing the object.

Note: the members of an object are initialized in the order they appear in the declaration, so it is a good idea to make sure the member initialization list is in the same order as in the declaration. It is also a good idea to always explicitly initialize all members.

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

With the constructor and the member functions we can now use objects of type author without ever directly poking into the “internals” of the object. Thus we can hide them away by declaring them private, and then only expose a public interface into the object.

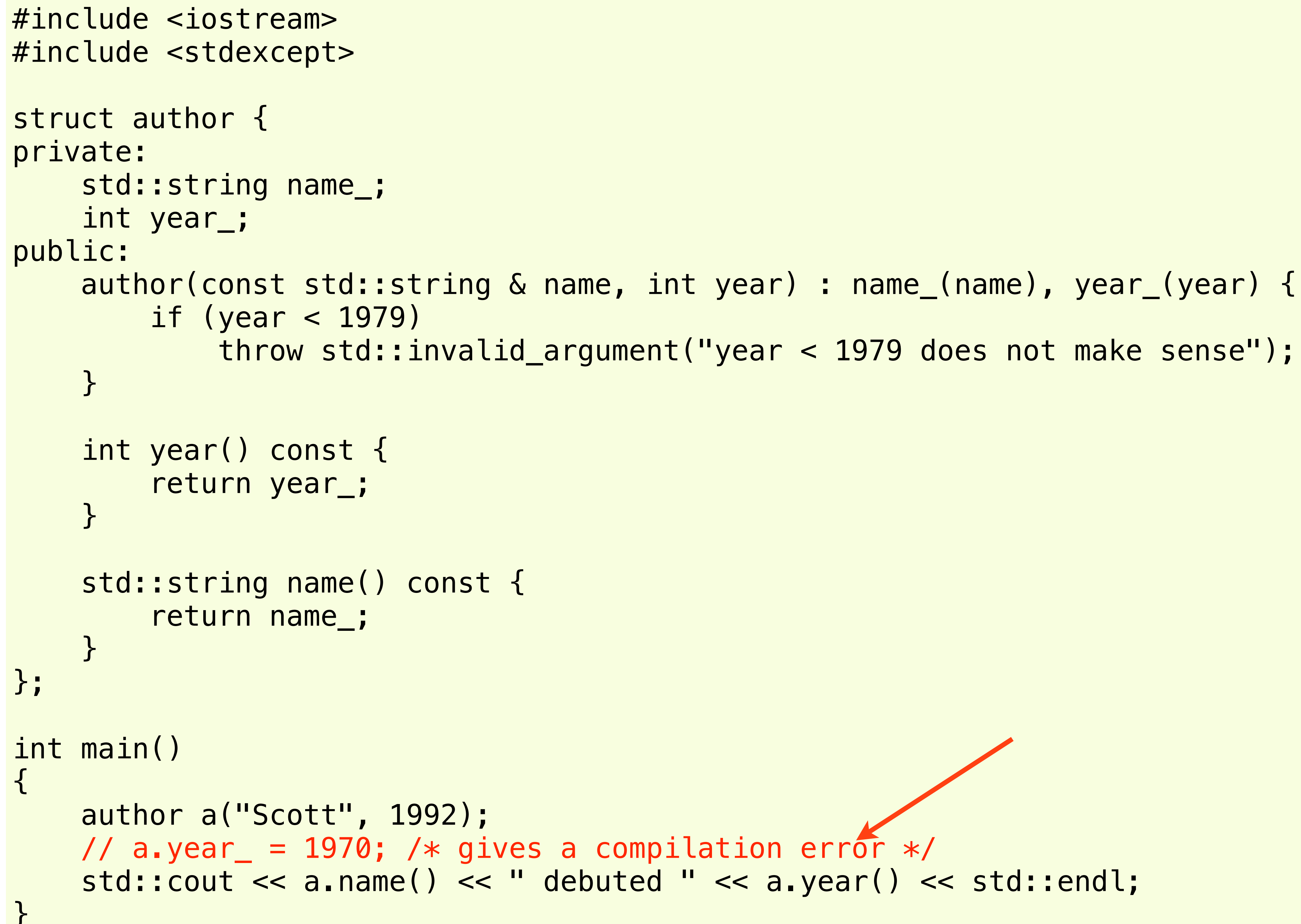
```
#include <iostream>
#include <stdexcept>

struct author {

    std::string name_;
    int year_;

    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

We have now added **protection labels**, also called **access specifiers**.

```cpp
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    // a.year_ = 1970; /* gives a compilation error */
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```



```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    // a.year_ = 1970; /* gives a compilation error */
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    // a.year_ = 1970; /* gives a compilation error */
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

if you now try to access a member variable directly you will get a compilation error. In this case we now have a guarantee that our `author` objects will be consistent and never have a year value less than 1979.

```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```cpp
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

In C++ you will often see the keyword `class` used instead of `struct` for types with member functions.

However, the only difference between struct and class is the default visibility. In struct things are public by default, in a class things are private by default.

```
#include <iostream>
#include <stdexcept>

struct author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

In C++ you will often see the keyword `class` used instead of `struct` for types with member functions.

However, the only difference between struct and class is the default visibility. In struct things are public by default, in a class things are private by default.

```
#include <iostream>
#include <stdexcept>

class author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```cpp
#include <iostream>
#include <stdexcept>

class author {
private:
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {

    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Now, just for fun. Let’s try to create an invalid author object.

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    try {
        author a("William", 1564);
        std::cout << a.name() << " debuted " << a.year() << std::endl;
    } catch(const std::invalid_argument & ex) {
        std::cerr << "Failed to create author object, because: "
                  << ex.what() << std::endl;
    }
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    try {
        author a("William", 1564);
        std::cout << a.name() << " debuted " << a.year() << std::endl;
    } catch(const std::invalid_argument & ex) {
        std::cerr << "Failed to create author object, because: "
                  << ex.what() << std::endl;
    }
}
```

```
Failed to create author object, because: year < 1979 does not make sense
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    try {
        author a("William", 1564);
        std::cout << a.name() << " debuted " << a.year() << std::endl;
    } catch(const std::invalid_argument & ex) {
        std::cerr << "Failed to create author object, because: "
                  << ex.what() << std::endl;
    }
}
```

```
Failed to create author object, because: year < 1979 does not make sense
```

Here is an example of how to catch an exception. We will not go into further details on exception handling in this course, but a nice rule of thumb is to only catch an exception at places in your code where you know exactly what to do with it.

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("William", 1564);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("William", 1564);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("William", 1564);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

The C++ run-time will take care of unhandled exceptions...

```
terminate called after throwing an instance of
'std::invalid_argument'
  what():  year < 1979 does not make sense
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```cpp
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
Scott debuted 1992
```

```cpp
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Have we discussed the scope resolution operator yet?

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Have we discussed the scope resolution operator yet?

The std:: prefix indicates that this is part of the C++ standard library. You can also create your own namespace, and this is very useful when organizing large programs.

```
#include <iostream>
#include <stdexcept>

class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>


class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};


int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>


class author {
    std::string name_;
    int year_;
public:
    author(const std::string & name, int year) : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int year() const {
        return year_;
    }

    std::string name() const {
        return name_;
    }
};


int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

You can import particular names from a namespace. Eg...

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{

    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    using mylib::author;
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    using mylib::author;
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

And now you don't need the scope resolution when referring to the `author` type inside main()

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    using mylib::author;
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    using mylib::author;
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

int main()
{
    using mylib::author;
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

You can also import the name in the outer scope.

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    using mylib::author;
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

using mylib::author;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```cpp
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

using mylib::author;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

It is even possible to import everything in a namespace, like this...

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

using mylib::author;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

It is even possible to import everything in a namespace, like this...

```cpp
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}

using namespace mylib;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const
            return name_;
        }
    };
}

using namespace mylib;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

And of course, you can import that whole `std` namespace if you want. This is a quite common thing to do... but,

importing whole namespaces might save some typing, but it does not always increase the readability of the code. Actually, I recommend to get used to mostly resolve the scope explicitly, even for things from the standard library.

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const
            return name_;
        }
    };
}

using namespace mylib;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

And of course, you can import that whole `std` namespace if you want. This is a quite common thing to do... but,

importing whole namespaces might save some typing, but it does not always increase the readability of the code. Actually, I recommend to get used to mostly resolve the scope explicitly, even for things from the standard library.

You can also create anonymous namespaces. This is useful for hiding stuff inside a translation unit, and it is an alternative to use the `static` linkage visibility specifier. Eg...

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const
            return name_;
        }
    };
}

using namespace mylib;

int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

And of course, you can import that whole `std` namespace if you want. This is a quite common thing to do... but,

importing whole namespaces might save some typing, but it does not always increase the readability of the code. Actually, I recommend to get used to mostly resolve the scope explicitly, even for things from the standard library.

You can also create anonymous namespaces. This is useful for hiding stuff inside a translation unit, and it is an alternative to use the `static` linkage visibility specifier. Eg...

```
#include <iostream>
#include <stdexcept>

namespace {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```cpp
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

When organizing a larger program, it is useful to break down in smaller modules that compile into separate object files and then linked together later.

In this case we can start by moving the whole mylib namespace into a header file.

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}



int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

#include "author.hpp"

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

```
#include <iostream>
#include <stdexcept>

#include "author.hpp"

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include <iostream>
#include <stdexcept>

#include "author.hpp"

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include <iostream>
#include <stdexcept>

#include "author.hpp"

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

It is often a good idea to include your own files first, and then include the standard libraries last, because then you will get a compilation error if you forget to include the right things in your header files.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include <iostream>
#include <stdexcept>

#include "author.hpp"

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

It is often a good idea to include your own files first, and then include the standard libraries last, because then you will get a compilation error if you forget to include the right things in your header files.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Here the member functions are implemented inline. For all but toy programs, it is better to separate them out into an implementation file.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year) : name_(name), year_(year) {
            if (year < 1979)
                throw std::invalid_argument("year < 1979 does not make sense");
        }

        int year() const {
            return year_;
        }

        std::string name() const {
            return name_;
        }
    };
}
```

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Now we can compile cppbooks.cpp and author.cpp separately, create two object files, and link together to create an executable.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Now we can compile cppbooks.cpp and author.cpp separately, create two object files, and link together to create an executable.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

```
$ c++ -c cppbooks.cpp
$ c++ -c author.cpp
$ c++ -o cppbooks cppbooks.o mylib.o
$ ./cppbooks
Scott debuted 1992
```

author.cpp

```
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```cpp
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```cpp
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```cpp
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

There are a few things we need to fix with the header files. First of all, neither cppbooks.cpp or author.hpp needs to know about the standard exceptions (they don't depend on stdexcept), only author.cpp needs to know.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>
#include <stdexcept>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

There are a few things we need to fix with the header files. First of all, neither cppbooks.cpp or author.hpp needs to know about the standard exceptions (they don't depend on stdexcept), only author.cpp needs to know.

author.hpp

```
#include <iostream>
#include <stdexcept>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>


int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>


namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <iostream>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Neither author.hpp nor author.cpp depends on iostream so no need to include that. However, they both needs string... but we have not included that. How come it still works? This is a typical example of namespace pollution. iostream is probably including string in a way that we implicitly get string included. It is a very common problem. Let's fix it here...

author.hpp

```
#include <iostream>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Neither author.hpp nor author.cpp depends on iostream so no need to include that. However, they both needs string... but we have not included that. How come it still works? This is a typical example of namespace pollution. iostream is probably including string in a way that we implicitly get string included. It is a very common problem. Let's fix it here...

author.hpp

```
#include <iostream>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <string>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

Finally... just a convention, but it is common to put the public part of a class first and then the private stuff. The argument is that you should focus on the api, and not the implementation details.

author.hpp

```
#include <string>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```cpp
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```cpp
#include <string>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```cpp
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <string>

namespace mylib {
    class author {
        std::string name_;
        int year_;
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <string>

namespace mylib {
    class author {
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    private:
        std::string name_;
        int year_;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <string>

namespace mylib {
    class author {
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    private:
        std::string name_;
        int year_;
    };
}
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

cppbooks.cpp

```
#include "author.hpp"

#include <iostream>

int main()
{
    mylib::author a("Scott", 1992);
    std::cout << a.name() << " debuted " << a.year() << std::endl;
}
```

author.hpp

```
#include <string>

namespace mylib {
    class author {
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    private:
        std::string name_;
        int year_;
    };
}
```

```
$ c++ -c cppbooks.cpp
$ c++ -c author.cpp
$ c++ -o cppbooks cppbooks.o mylib.o
$ ./cppbooks
Scott debuted 1992
```

author.cpp

```
#include "author.hpp"
#include <stdexcept>

namespace mylib {
    author::author(const std::string & name, int year)
        : name_(name), year_(year) {
        if (year < 1979)
            throw std::invalid_argument("year < 1979 does not make sense");
    }

    int author::year() const {
        return year_;
    }

    std::string author::name() const {
        return name_;
    }
}
```

Generic Programming

(about templates and how to use standard libraries)

Before we can start talking about the standard library, I must say a few words about templates and generic programming.

```
#include <iostream>

int min_int(int a, int b)
{
    return a < b ? a : b;
}

double min_double(double a, double b)
{
    return a < b ? a : b;
}

char min_char(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min_int(4,7) << std::endl;
    std::cout << min_double(7.3,4.1) << std::endl;
    std::cout << min_char('z', 'a') << std::endl;
}
```

```
#include <iostream>

int min_int(int a, int b)
{
    return a < b ? a : b;
}

double min_double(double a, double b)
{
    return a < b ? a : b;
}

char min_char(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min_int(4,7) << std::endl;
    std::cout << min_double(7.3,4.1) << std::endl;
    std::cout << min_char('z', 'a') << std::endl;
}
```

Here is an example of similar algorithms applied to different types.

In C++, instead of embedding the type in the function name you can write functions that only differ on the type of the arguments.

```
#include <iostream>

int min_int(int a, int b)
{
    return a < b ? a : b;
}

double min_double(double a, double b)
{
    return a < b ? a : b;
}

char min_char(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min_int(4,7) << std::endl;
    std::cout << min_double(7.3,4.1) << std::endl;
    std::cout << min_char('z', 'a') << std::endl;
}
```

```cpp
#include <iostream>

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

This is called **function overloading**. There is nothing magical happening here, it is just that the argument types are included in the **function signature** that the compiler and linker sees. This is also called **name mangling**, and this is something you need to be particularly aware of when trying to link C and C++ code together.

```
#include <iostream>

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

This is called **function overloading**. There is nothing magical happening here, it is just that the argument types are included in the **function signature** that the compiler and linker sees. This is also called **name mangling**, and this is something you need to be particularly aware of when trying to link C and C++ code together.

Function overloading is indeed very useful, but in this case it would be really nice if we could just "generate" code from a generic and type independent description of how an algorithm should look like. Indeed, you can do this in C++ by using **templates**.

```
#include <iostream>



int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

This is an example of a **function template**. If the compiler sees a call to a function, it will first look for an explicit function declaration with a signature corresponding to the arguments given. If it cannot find any, it will, in this case, use the function template to generate a function for you. You may think about it as something similar to a macro expansion, but an important distinction is that it is done by the proper compiler and not by a preprocessor. In C++, macros are bad, templates are good.

```cpp
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

So here, if we make a call to min() using two longs as argument. Eg like this:

```
min(42L, 256L)
```

then the function template will be used to create a function equivalent in your code:

```
long min(long a, long b)
{
    return a < b ? a : b;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

So here, if we make a call to min() using two longs as argument. Eg like this:

```
min(42L, 256L)
```

then the function template will be used to create a function equivalent in your code:

```
long min(long a, long b)
{
    return a < b ? a : b;
}
```

But this also means that we no longer need the explicit function declarations, we can just let the function template generate it for us.

```cpp
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int min(int a, int b)
{
    return a < b ? a : b;
}

double min(double a, double b)
{
    return a < b ? a : b;
}

char min(char a, char b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```cpp
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```cpp
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min<int>(4,7) << std::endl;
    std::cout << min<double>(7.3,4.1) << std::endl;
    std::cout << min<char>('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

Actually the object file produced by compiling this code snippet is identical to the object file created from the previous code snippet we had. But if you do not use the template, then no code will be generated.

```cpp
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
T min(T a, T b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

In this case we should pass and return T by const reference instead of by copy.

```cpp
#include <iostream>

template <typename T>
const T & min(const T & a, const T & b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
const T & min(const T & a, const T & b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```cpp
#include <iostream>

template <typename T>
const T & min(const T & a, const T & b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

```
#include <iostream>

template <typename T>
const T & min(const T & a, const T & b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << min(4,7) << std::endl;
    std::cout << min(7.3,4.1) << std::endl;
    std::cout << min('z', 'a') << std::endl;
}
```

You can also use templates to generate complete class declarations. Like this...

```
#include <iostream>
#include <list>


class stack_int {
    std::list<int> items;
public:
    class underflow {};
    stack() : items() {}
    void push(int item) {
        items.push_front(item);
    }
    int pop() {
        if (items.empty())
            throw underflow();
        int item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack_int s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>


class stack_int {
    std::list<int> items;
public:
    class underflow {};
    stack() : items() {}
    void push(int item) {
        items.push_front(item);
    }
    int pop() {
        if (items.empty())
            throw underflow();
        int item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack_int s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

Here is a regular, type-dependent version of a class implementing a simple stack of ints. Let's rewrite it into a template instead.

```
4
2
```

```
#include <iostream>
#include <list>


class stack_int {
    std::list<int> items;
public:
    class underflow {};
    stack() : items() {}
    void push(int item) {
        items.push_front(item);
    }
    int pop() {
        if (items.empty())
            throw underflow();
        int item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack_int s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

Here is a regular, type-dependent version of a class implementing a simple stack. Let's rewrite it into a template instead.

```
4
2
```

```
#include <iostream>
#include <list>


class stack_int {
    std::list<int> items;
public:
    class underflow {};
    stack() : items() {}
    void push(int item) {
        items.push_front(item);
    }
    int pop() {
        if (items.empty())
            throw underflow();
        int item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack_int s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

Here is a regular, type-dependent version of a class implementing a simple stack. Let's rewrite it into a template instead.

I have highlighted the type dependent stuff in this class. We replace it with a template and type parameter.

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(T item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(T item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(T item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

And now we have a **class template** for a stack of any type of objects!

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(T item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(T item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

But again, in this case, T should be passed by const reference rather than by copy.

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};



int main()
{
    stack<int> s;
    s.push(2);
    s.push(4);
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
4
2
```

```
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};

#include <string>

int main()
{
    stack<std::string> s;
    s.push("Bar");
    s.push("Foo");
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

```
Foo
Bar
```

```cpp
#include <iostream>
#include <list>

template <typename T>
class stack {
    std::list<T> items;
public:
    class underflow {};
    stack() : items() {}
    void push(const T & item) {
        items.push_front(item);
    }
    T pop() {
        if (items.empty())
            throw underflow();
        T item = items.front();
        items.pop_front();
        return item;
    }
};

#include "author.hpp"

int main()
{
    stack<mylib::author> s;
    s.push(mylib::author("Scott", 1992));
    s.push(mylib::author("Andrew", 1988));
    std::cout << s.pop() << std::endl;
    std::cout << s.pop() << std::endl;
}
```

author.hpp

```cpp
#include <iostream>
#include <stdexcept>
#include <string>

namespace mylib {
    class author {
    public:
        author(const std::string & name, int year);
        int year() const;
        std::string name() const;
    private:
        std::string name_;
        int year_;
    };
}

std::ostream & operator<<(
    std::ostream & out, const mylib::author & a);
```

```
Andrew debuted 1988
Scott debuted 1992
```

Now we can start looking into the standard library of C++. Most of the cool stuff is actually implemented as templates, type independent classes and algorithms for doing useful things. The **standard template library** is a very important part of C++. Learning how to use it well is essential knowledge for every serious C++ programmer.

Now we can start looking into the standard library of C++. Most of the cool stuff is actually implemented as templates, type independent classes and algorithms for doing useful things. The **standard template library** is a very important part of C++. Learning how to use it well is essential knowledge for every serious C++ programmer.

Even when the standard template library does not have exactly what you are looking for you can often customize what is already there by replacing default behaviour with your stuff, for example by "injecting" your own strategies into templated classes and algorithms.

I will just show a few example of things you can do with the standard library.

```
#include <iostream>
#include <vector>

void print_names(const std::vector<std::string> & names)
{
    std::cout << "* Names:" << std::endl;
    for (std::vector<std::string>::const_iterator it = names.begin();
         it != names.end(); ++it)
        std::cout << *it << std::endl;
}

int main()
{
    std::vector<std::string> names;
    names.insert(names.begin(), "Bjarne");
    names.insert(names.begin(), "Scott");
    names.insert(names.end(), "Andrei");
    names.push_back("Herb");
    names.insert(names.begin(),"Nico");
    print_names(names);

    names.pop_back();
    names.erase(names.begin());
    print_names(names);
}
```

```
* Names:
Nico
Scott
Bjarne
Andrei
Herb
* Names:
Scott
Bjarne
Andrei
```

```
#include <iostream>
#include <vector>

void print_names(const std::vector<std::string> & names)
{
    std::cout << "* Names:" << std::endl;
    for (std::vector<std::string>::const_iterator it = names.begin();
         it != names.end(); ++it)
        std::cout << *it << std::endl;
}

int main()
{
    std::vector<std::string> names;
    names.insert(names.begin(), "Bjarne");
    names.insert(names.begin(), "Scott");
    names.insert(names.end(), "Andrei");
    names.push_back("Herb");
    names.insert(names.begin(),"Nico");
    print_names(names);

    names.pop_back();
    names.erase(names.begin());
    print_names(names);
}
```

```
* Names:
Nico
Scott
Bjarne
Andrei
Herb
* Names:
Scott
Bjarne
Andrei
```

This is just an example of using the `std::vector` template. If you are not sure which container to use, the `std::vector` is often a good choice, but you should also consider `std::deque` and `std::list`

Notice that it is often easy to change from one collection to another collection later. In this case you can just replace every instance of `std::vector` with `std::deque` or `std::list` and it will still work.

```
#include <iostream>
#include <stack>

int main()
{
    std::stack<std::string> names;
    names.push("Bjarne");
    names.push("Scott");
    names.push("Andrei");

    std::cout << "size of stack is " << names.size() << std::endl;
    std::string name = names.top();
    std::cout << "popping " << name << std::endl;
    names.pop();
    std::cout << "size of stack is " << names.size() << std::endl;
}
```

```
size of stack is 3
popping Andrei
size of stack is 2
```

```cpp
#include <iostream>
#include <stack>

int main()
{
    std::stack<std::string> names;
    names.push("Bjarne");
    names.push("Scott");
    names.push("Andrei");

    std::cout << "size of stack is " << names.size() << std::endl;
    std::string name = names.top();
    std::cout << "popping " << name << std::endl;
    names.pop();
    std::cout << "size of stack is " << names.size() << std::endl;
}
```

Here is an example of using `std::stack`. There is also a `std::queue` implemented in the library.

```
size of stack is 3
popping Andrei
size of stack is 2
```

```
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```cpp
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```cpp
#include <iostream>
#include <set>
#include <algorithm>

void print_int(int i) {
    std::cout << i << ' ';
}

int main()
{
    std::set<int> values;
    values.insert(10);
    values.insert(20);
    values.insert(30);
    values.insert(20);
    values.insert(70);
    values.insert(50);

    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(20);
    std::cout << "size of set is " << values.size() << std::endl;
    values.erase(40);
    std::cout << "size of set is " << values.size() << std::endl;

    std::for_each(values.begin(), values.end(), print_int);
}
```

Here I demonstrate `std::set`, but also an example of how to use an algorithm that you can find in the standard library.

The std::for_each algorithm takes two iterators and a function. It then loops through every element in the given range and calls the given function.

```
size of set is 5
size of set is 4
size of set is 4
10 30 50 70
```

```cpp
#include <iostream>
#include <map>
#include <utility>
#include <algorithm>

void print(const std::pair<std::string,int> & a) {
    std::cout << a.first << " debuted " << a.second << std::endl;
}

int main()
{
    std::map<std::string, int> authors;
    authors["Bjarne"] = 1990;
    authors["Andrei"] = 2001;
    authors["Herb"] = 2000;
    authors["Andrew"] = 1988;
    authors["Bob"] = 1995;
    authors["Andrew"] = 1988;

    std::map<std::string, int>::iterator it = authors.find("Andrew");
    if (it != authors.end())
        authors.erase(it);

    authors.insert( std::pair<std::string, int>("Nico", 1995) );
    authors.find("Nico")->second = 1999;

    std::for_each(authors.begin(), authors.end(), print);
}
```

```
Andrei debuted 2001
Bjarne debuted 1990
Bob debuted 1995
Herb debuted 2000
Nico debuted 1999
```

```
#include <iostream>
#include <map>
#include <utility>
#include <algorithm>

void print(const std::pair<std::string,int> & a) {
    std::cout << a.first << " debuted " << a.second << std::endl;
}

int main()
{
    std::map<std::string, int> authors;
    authors["Bjarne"] = 1990;
    authors["Andrei"] = 2001;
    authors["Herb"] = 2000;
    authors["Andrew"] = 1988;
    authors["Bob"] = 1995;
    authors["Andrew"] = 1988;

    std::map<std::string, int>::iterator it = authors.find("Andrew");
    if (it != authors.end())
        authors.erase(it);

    authors.insert( std::pair<std::string, int>("Nico", 1995) );
    authors.find("Nico")->second = 1999;

    std::for_each(authors.begin(), authors.end(), print);
}
```

`std::map` is an associative container, where you can use a key to look up a value. Internally map uses `std::pair` which is a nice little utility template for pairing two items of data together.

Notice how well you can do rather complicated stuff in just a few lines of code by combining things from the standard library. C++ has a very powerful toolbox!

```
Andrei debuted 2001
Bjarne debuted 1990
Bob debuted 1995
Herb debuted 2000
Nico debuted 1999
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

Here I show two things:

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

Here I show two things:

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

Here I show two things:

an alternative way to print out a collection by using std::copy.

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

Here I show two things:

an alternative way to print out a collection by using std::copy.

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<std::string> & a) {
    std::ostream_iterator<std::string> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int main()
{
    const std::string s[] = {"Bjarne","Scott","Herb","Nico"};
    std::vector<std::string> authors(s, s+4);

    std::sort(authors.begin(), authors.end());
    print_vector(authors);

    std::reverse(authors.begin(), authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);

    std::rotate(authors.begin(), authors.begin()+1, authors.end());
    print_vector(authors);
}
```

Here I show two things:

an alternative way to print out a collection by using std::copy.

And a few more cool algorithms

```
Bjarne Herb Nico Scott
Scott Nico Herb Bjarne
Nico Herb Bjarne Scott
Herb Bjarne Scott Nico
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

This is an example of how to define your own strategy into existing algorithms.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

This is an example of how to define your own strategy into existing algorithms.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

This is an example of how to define your own strategy into existing algorithms.

we create our own function for returning a random number in the range [10,20) (eg, {10,11...18,19})

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

Then we use an algorithm to generate 20 random numbers. Which we sort and print.

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

Here we count all values in the range that are in the range [12,17). Instead of using a function to define a strategy we use a class where we have overloaded the () operator.

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

This is an example of what we call a **functor** in C++. You may think of it as a function with state. In this case we can set the low and high values when creating an instance of this class, and if someone "calls" the class (by using the () operator) we return true if a value is within the boundaries.

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

This is an example of what we call a **functor** in C++. You may think of it as a function with state. In this case we can set the low and high values when creating an instance of this class, and if someone "calls" the class (by using the () operator) we return true if a value is within the boundaries.

Instead of hardcoding an in_range_12_to_17() function we can use functors to parameterize a function. You can also do similar things with templates of course.

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

void print_vector(const std::vector<int> & a) {
    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy(a.begin(), a.end(), out_it);
    std::cout << std::endl;
}

int my_rand() { return std::rand() % 10 + 10; }

class in_range {
public:
    in_range(int low, int high) : low_(low), high_(high) {}
    bool operator()(int value) const { return (value >= low_) && (value < high_); }
private:
    int low_;
    int high_;
};

int main()
{
    std::vector<int> values(20);

    std::generate(values.begin(), values.end(), my_rand);
    std::sort(values.begin(), values.end());
    print_vector(values);

    int c = std::count_if(values.begin(), values.end(), in_range(12,17));
    std::cout << "there are " << c << " elements in range [12,17)" << std::endl;

    std::vector<int>::iterator it = std::unique(values.begin(), values.end());
    values.resize(it - values.begin());
    print_vector(values);
}
```

Here we remove all duplicate values. Be careful to read the manual when using certain things in the standard library. For example, `std::unique` is difficult to use correctly until you understand *why* the collection must be sorted first, and *why* the collection must be resized afterwards.

```
10 10 10 12 12 12 12 13 13 13 14 15 17 17 17 18 18 19 19 19
there are 9 elements in range [12,17)
10 12 13 14 15 17 18 19
```

Misc topics

prefix vs postfix increment

```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

prefix vs postfix increment

```
int main()
{
    for (foo f; f<4; ++f)
        std::cout << "f = " << f << std::endl;
}
```

```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

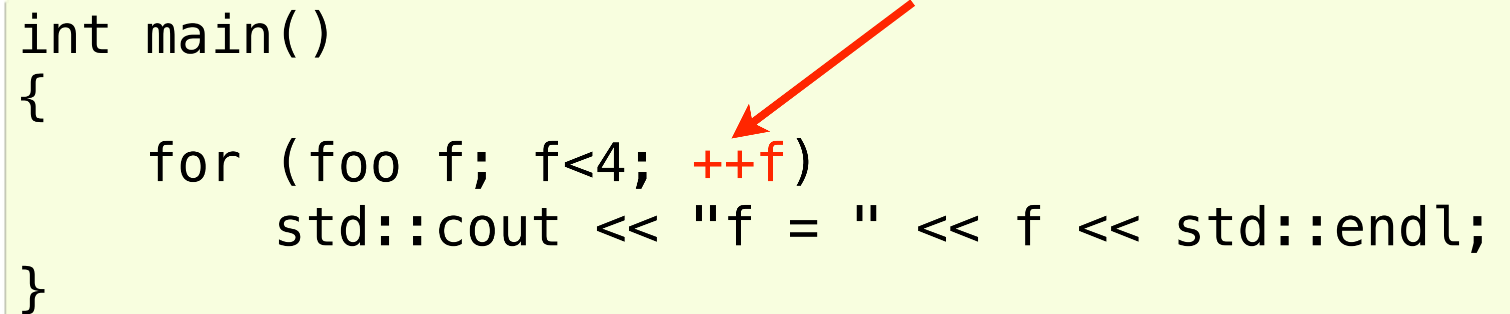
prefix vs postfix increment

```
int main()
{
    for (foo f; f<4; ++f)
        std::cout << "f = " << f << std::endl;
}
```

```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

```
* foo()
f = 0
* ++prefix on foo
f = 1
* ++prefix on foo
f = 2
* ++prefix on foo
f = 3
* ++prefix on foo
* ~foo()
```

prefix vs postfix increment




```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

```
* foo()
f = 0
* ++prefix on foo
f = 1
* ++prefix on foo
f = 2
* ++prefix on foo
f = 3
* ++prefix on foo
* ~foo()
```

prefix vs postfix increment

```
int main()
{
    for (foo f; f<4; f++)
        std::cout << "f = " << f << std::endl;
}
```

```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

prefix vs postfix increment

```
int main()
{
    for (foo f; f<4; f++)
        std::cout << "f = " << f << std::endl;
}
```

```
#include <iostream>

class foo {
public:
    foo() : counter_(0) {
        std::cout << "* foo()" << std::endl;
    }
    foo(const foo & other) : counter_(other.counter_) {
        std::cout << "* foo(const foo &)" << std::endl;
    }
    ~foo() {
        std::cout << "* ~foo()" << std::endl;
    }
    foo & operator++() {
        std::cout << "* ++prefix on foo" << std::endl;
        ++counter_;
        return *this;
    }
    foo operator++(int) {
        std::cout << "* postfix++ on foo" << std::endl;
        foo tmp(*this);
        ++counter_;
        return tmp;
    }
    operator int() const { return counter_; };
private:
    int counter_;
};
```

```
* foo()
f = 0
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 1
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 2
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 3
* postfix++ on foo
* foo(const foo &)
* ~foo()
* ~foo()
```

prefix vs postfix increment

++f

```
* foo()
f = 0
* ++prefix on foo
f = 1
* ++prefix on foo
f = 2
* ++prefix on foo
f = 3
* ++prefix on foo
* ~foo()
```

vs

f++

```
* foo()
f = 0
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 1
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 2
* postfix++ on foo
* foo(const foo &)
* ~foo()
f = 3
* postfix++ on foo
* foo(const foo &)
* ~foo()
* ~foo()
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

Suppose you have a resource that you should open to use and close when finished. Here is a way to do it. But it is not a very good way of doing it. Not only is it easy to forget to release a resource, but also it is very difficult to write exception safe code using this “naive” approach.

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

Suppose you have a resource that you should open to use and close when finished. Here is a way to do it. But it is not a very good way of doing it. Not only is it easy to forget to release a resource, but also it is very difficult to write exception safe code using this "naive" approach.

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

Suppose you have a resource that you should open to use and close when finished. Here is a way to do it. But it is not a very good way of doing it. Not only is it easy to forget to release a resource, but also it is very difficult to write exception safe code using this “naive” approach.

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

We can introduce a simple object that can own the resource and automatically release the resource when it goes out of scope.

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

We can introduce a simple object that can own the resource and automatically release the resource when it goes out of scope.

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

We can introduce a simple object that can own the resource and automatically release the resource when it goes out of scope.

And now we can rewrite into.

```
void foo()
{
    int h = open c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
    close(h);
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

```
void foo()
{
    connector c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

Since our connector object is allocated on stack, the destructor will be called when it goes out of scope, and we have a guarantee that we will close an open connection.

```
void foo()
{
    connector c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
}
```

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

grabbing and releasing resources

```
int open(const std::string & service) {
    std::cout << "* open connection to " << service << std::endl;
    // ...
    return 42;
}

void close(int connection) {
    std::cout << "* close connection " << connection << std::endl;
    // ...
}
```

```
class connector {
public:
    explicit connector(const std::string & service)
        : connection_(open(service)) {}
    ~connector() { close(connection_); }
private:
    int connection_;
};
```

Since our connector object is allocated on stack, the destructor will be called when it goes out of scope, and we have a guarantee that we will close an open connection.

```
void foo()
{
    connector c("cisco.com:4242/server");
    std::cout << "using service" << std::endl;
    // ...
}
```

This is an example of a so called RAII pattern - resource acquisition is initialization.

```
* open connection to cisco.com:8899/presence_server
using service
* close connection 42
```

a smart pointer (RAII)

```
template <typename T>
class smart_ptr {
public:
    explicit smart_ptr(T * p) : p_(p) {}
    ~smart_ptr() { delete p_; }
    T * operator->() { return p_; }
    T * operator*() { return *p_; }
private:
    T * p_;
    smart_ptr(const smart_ptr &);
    smart_ptr & operator=(const smart_ptr &);
};
```

a smart pointer (RAII)

```
template <typename T>
class smart_ptr {
public:
    explicit smart_ptr(T * p) : p_(p) {}
    ~smart_ptr() { delete p_; }
    T * operator->() { return p_; }
    T * operator*() { return *p_; }
private:
    T * p_;
    smart_ptr(const smart_ptr &);
    smart_ptr & operator=(const smart_ptr &);
};
```

You can generalize the idea into a class template. Here I have a silly implementation of a smart pointer to illustrate the idea.

a smart pointer (RAII)

```
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include “smart_ptr.hpp”

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    smart_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

```
template <typename T>
class smart_ptr {
public:
    explicit smart_ptr(T * p) : p_(p) {}
    ~smart_ptr() { delete p_; }
    T * operator->() { return p_; }
    T * operator*() { return *p_; }
private:
    T * p_;
    smart_ptr(const smart_ptr &);
    smart_ptr & operator=(const smart_ptr &);
};
```

You can generalize the idea into a class template. Here I have a silly implementation of a smart pointer to illustrate the idea.

a smart pointer (RAII)

```
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include “smart_ptr.hpp”

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    smart_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

```
template <typename T>
class smart_ptr {
public:
    explicit smart_ptr(T * p) : p_(p) {}
    ~smart_ptr() { delete p_; }
    T * operator->() { return p_; }
    T * operator*() { return *p_; }
private:
    T * p_;
    smart_ptr(const smart_ptr &);
    smart_ptr & operator=(const smart_ptr &);
};
```

You can generalize the idea into a class template. Here I have a silly implementation of a smart pointer to illustrate the idea.

```
allocating 38911 bytes
kaboom!
deleting 38911 bytes
bar failed
```

```cpp
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include “smart_ptr.hpp”

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    smart_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

```
allocating 38911 bytes
kaboom!
deleting 38911 bytes
bar failed
```

```
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include "smart_ptr.hpp"

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    smart_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

But of course, in this case you should use a smart pointer from the standard libraries. Boost, TR1 and C++11 all have good smart pointer to use.

```
allocating 38911 bytes
kaboom!
deleting 38911 bytes
bar failed
```

use a proper smart pointer (RAII)

```
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include <tr1/memory>

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    std::tr1::shared_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

```
allocating 38911 bytes
kaboom!
deleting 38911 bytes
bar failed
```

use a proper smart pointer (RAII)

```
#include <iostream>
#include <stdexcept>
#include <stdint.h>
#include <tr1/memory>

struct bigblob {
    bigblob() {
        std::cout << "allocating 38911 bytes\n";
    }
    ~bigblob() {
        std::cout << "deleting 38911 bytes\n";
    }
    uint8_t data_[38911];
};

void bar() {
    std::cout << "kaboom!" << std::endl;
    throw std::logic_error("bar failed");
}

void foo() {
    std::tr1::shared_ptr<bigblob> blob(new bigblob);
    bar();
}

int main()
{
    try {
        foo();
    } catch( std::exception & x ) {
        std::cout << x.what() << std::endl;
    }
}
```

Choosing the right smart pointer is important.

Don't use `auto_ptr`, it's been deprecated.

If you have Boost then consider `scoped_ptr`, and in C++11 you should consider `unique_ptr`.

Here I only have TR1 so I have to choose `shared_ptr` unless I want to write my own.

```
allocating 38911 bytes
kaboom!
deleting 38911 bytes
bar failed
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't specify them explicitly, the compiler will be "helpful" and try to create three things: a **destructor**, a **copy constructor** and **copy assignment operator**. This can be nice, but often the stuff created by the compiler is not what you want. A useful guideline is **the rule of three**: if you feel a need to implement any of the three yourself, you should probably implement or disable the other two as well.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't specify them explicitly, the compiler will be "helpful" and try to create three things: a **destructor**, a **copy constructor** and **copy assignment operator**. This can be nice, but often the stuff created by the compiler is not what you want. A useful guideline is **the rule of three**: if you feel a need to implement any of the three yourself, you should probably implement or disable the other two as well.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't specify them explicitly, the compiler will be "helpful" and try to create three things: a **destructor**, a **copy constructor** and **copy assignment operator**. This can be nice, but often the stuff created by the compiler is not what you want. A useful guideline is **the rule of three**: if you feel a need to implement any of the three yourself, you should probably implement or disable the other two as well.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't specify them explicitly, the compiler will be "helpful" and try to create three things: a **destructor**, a **copy constructor** and **copy assignment operator**. This can be nice, but often the stuff created by the compiler is not what you want. A useful guideline is **the rule of three**: if you feel a need to implement any of the three yourself, you should probably implement or disable the other two as well.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't specify them explicitly, the compiler will be "helpful" and try to create three things: a **destructor**, a **copy constructor** and **copy assignment operator**. This can be nice, but often the stuff created by the compiler is not what you want. A useful guideline is **the rule of three**: if you feel a need to implement any of the three yourself, you should probably implement or disable the other two as well.

If you do it correctly, you can work with objects as if they were native types. If you do it incorrectly though, you will often leak memory, corrupt the execution state, run out of resources... and so on.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

```
#include <iostream>

int main()
{
    foo f(40);
    f++;
    ++f;
    std::cout << "the answer is "
              << f << std::endl;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

```
#include <iostream>

int main()
{
    foo f(40);
    f++;
    ++f;
    std::cout << "the answer is "
              << f << std::endl;
}
```

```
the answer is 42
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
#include <iostream>

int main()
{
    foo f1(42);
    foo f2 = f1;
    foo f3;
    f3 = f2;
    std::cout << f3 << std::endl;
}
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo(const foo & other);
    foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
#include <iostream>

int main()
{
    foo f1(42);
    foo f2 = f1;
    foo f3;
    f3 = f2;
    std::cout << f3 << std::endl;
}
```

```
42
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter_(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

If you don't write a copy constructor and a copy assignment operator (for example) the compiler will try to write them for you...

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

You might hope that the compiler writes them correctly (or at least safely), like this...

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

You might hope that the compiler writes them correctly (or at least safely), like this...

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other) : counter(new int) {
    *counter_ = *other.counter_;
}

foo & foo::operator=(const foo & rhs) {
    *counter_ = *rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

Alas no. The compiler written versions make shallow pointer copies resulting in multiple pointers pointing to the same object in memory.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other)
    : counter_(other.counter_) {
}

foo & foo::operator=(const foo & rhs) {
    counter_ = rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

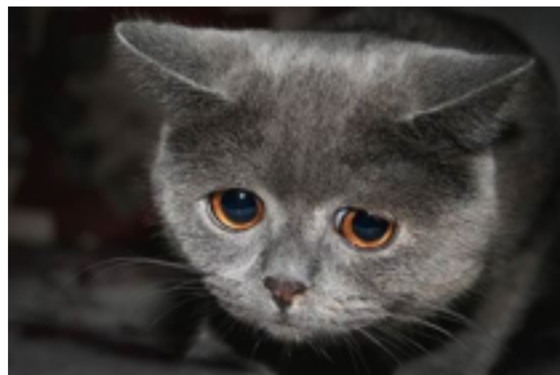
Bad things will happen.

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other)
    : counter_(other.counter_) {
}

foo & foo::operator=(const foo & rhs) {
    counter_ = rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

Bad things will happen.

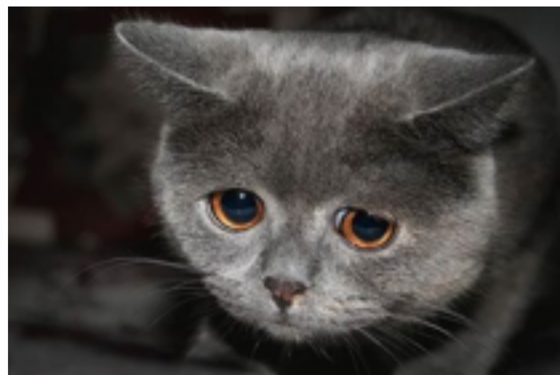


```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other)
    : counter_(other.counter_) {
}

foo & foo::operator=(const foo & rhs) {
    counter_ = rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    //foo(const foo & other);
    //foo & operator=(const foo & rhs);
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    int * counter_;
};
```

Bad things will happen.



```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo::foo(const foo & other)
    : counter_(other.counter_) {
}

foo & foo::operator=(const foo & rhs) {
    counter_ = rhs.counter_;
    return *this;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

the rule of three

```
class foo {
public:
    foo();
    explicit foo(int value);
    ~foo();
    foo & operator++();
    foo operator++(int);
    operator int() const;
private:
    foo & operator=(const foo & rhs);
    foo(const foo & other);
    int * counter_;
};
```

```
foo::foo() : counter_(new int) {
    *counter_ = 0;
}

foo::foo(int value) : counter_(new int) {
    *counter_ = value;
}

foo::~foo() {
    delete counter_;
}

foo & foo::operator++() {
    ++*counter_;
    return *this;
}

foo foo::operator++(int) {
    foo tmp(*this);
    operator++();
    return tmp;
}

foo::operator int() const {
    return *counter_;
}
```

In this case, if you don't want to support copying you can disable the copy constructor and copy assignment operator by hiding them in a private part of the class declaration and *not* writing implementations.

LSP, polymorphism, interfaces, ...

```
#include <iostream>
#include <memory>

class shape {
public:
    virtual ~shape() {}
    virtual void draw() = 0;
    virtual int area() const = 0;
};

class rectangle : public shape {
public:
    rectangle(int w, int h) : w_(w), h_(h) {}
    void draw() { std::cout << "* drawing a rectangle" << std::endl; }
    int area() const { return w_ * h_; }
private:
    int w_, h_;
};

class square : public shape {
public:
    square(int length) : length_(length) {}
    void draw() { std::cout << "* drawing a square" << std::endl; }
    int area() const { return length_ * length_; }
private:
    int length_;
};

int main()
{
    std::shared_ptr<shape> s1 = new rectangle(4,4);
    s1->draw();
    std::cout << "area(s1) = " << s1->area() << std::endl;

    std::shared_ptr<shape> s2 = new square(4);
    s2->draw();
    std::cout << "area(s2) = " << s2->area() << std::endl;
}
```

LSP, polymorphism, interfaces, ...

```
#include <iostream>
#include <memory>

class shape {
public:
    virtual ~shape() {}
    virtual void draw() = 0;
    virtual int area() const = 0;
};

class rectangle : public shape {
public:
    rectangle(int w, int h) : w_(w), h_(h) {}
    void draw() { std::cout << "* drawing a rectangle" << std::endl; }
    int area() const { return w_ * h_; }
private:
    int w_, h_;
};

class square : public shape {
public:
    square(int length) : length_(length) {}
    void draw() { std::cout << "* drawing a square" << std::endl; }
    int area() const { return length_ * length_; }
private:
    int length_;
};

int main()
{
    std::shared_ptr<shape> s1 = new rectangle(4,4);
    s1->draw();
    std::cout << "area(s1) = " << s1->area() << std::endl;

    std::shared_ptr<shape> s2 = new square(4);
    s2->draw();
    std::cout << "area(s2) = " << s2->area() << std::endl;
}
```

This is a decent example of how to implement interfaces, how to use inheritance and support polymorphism, and it also respects Liskov's Substitution Principle.

Be careful though you can easily do this wrong. For example, forgetting the virtual destructor, breaking LSP by assuming that shape is a specialisation of rectangle, or end up deriving things from concrete classes, and so on. Study OO well before doing fancy stuff with the inheritance capabilities in C++.

Good OO programs in C++ often avoid deep class hierarchies. Max depth of three: an interface, an abstract implementation and a concrete implementation could be what you should aim for.

The most important takeaways

The most important takeaways

- invalid conceptual model gives strange explanations

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes
- respect the rule of 3

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes
- respect the rule of 3
- try to learn the proper terms of the language

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes
- respect the rule of 3
- try to learn the proper terms of the language
- focus on readability, not writeability

The most important takeaways

- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes
- respect the rule of 3
- try to learn the proper terms of the language
- focus on readability, not writeability
- good dependency management is essential in real projects

The most important takeaways

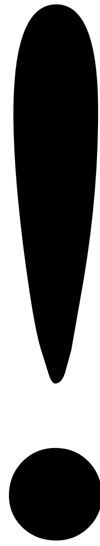
- invalid conceptual model gives strange explanations
- the evaluation order in C++ is mostly unspecified
- undefined behavior means **anything** can happen
- sequence point violation is undefined behavior
- understand object lifetimes
- respect the rule of 3
- try to learn the proper terms of the language
- focus on readability, not writeability
- good dependency management is essential in real projects
- C++ is a multi-paradigm language

C and C++ are not really high level languages, they are more like portable assemblers. When programming in C and C++ you *must* have a understanding of what happens under the hood! And if you don't have a decent understanding of it, then you are doomed to create lots of bugs...

C and C++ are not really high level languages, they are more like portable assemblers. When programming in C and C++ you *must* have a understanding of what happens under the hood! And if you don't have a decent understanding of it, then you are doomed to create lots of bugs...

But if you *do* have a useful mental model of what happens under the hood, then...

http://www.sharpshirter.com/assets/images/sharkpunchashgrey1.jpg



The spirit of C

trust the programmer

- let them do what needs to be done
- the programmer is in charge not the compiler

keep the language small and simple

- small amount of code → small amount of assembler
- provide only one way to do an operation
- new inventions are not entertained

make it fast, even if its not portable

- target efficient code generation
- int preference, int promotion rules
- sequence points, maximum leeway to compiler

rich expression support

- lots of operators
- expressions combine into larger expressions

Design principles for C++

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming)
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is designed to be as compatible with C as possible, therefore providing a smooth transition from C
- C++ avoids features that are platform specific or not general purpose
- C++ does not incur overhead for features that are not used (the "zero-overhead principle")
- C++ is designed to function without a sophisticated programming environment