Deep C (and C++)

by Olve Maudal

http://www.noaanews.noaa.gov/stories2005/images/rov-hercules-titanic.jpg

Programming is hard. Programming correct C and 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 talk we will study small code snippets of C and C++, and use them to discuss and explore the fundamental building blocks, limitations and underlying design philosophies of these wonderful but dangerous programming languages.

A 90 minute session at Geekup.in, Bangalore
Wednesday, October 23, 2013

JAWS

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
12
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
```

Exercise

What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
13
```

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Let's add some flags for better diagnostics.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Let's add some flags for better diagnostics.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

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

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

It is important to understand that C (and C++) are not really high-level languages compared to most other common programming languages.

They are more like just portable assemblers where you have to appreciate the underlying architecture to program correctly. This is reflected in the language definition and in how compiler deals with "incorrect" code.

Without a deep understanding of the language, its history, and its design goals, you are doomed to fail.

http://www.slideshare.net/olvemaudal/deep-c

```
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
4
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
4
4
```

```
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
```

```c
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
5
```

```c
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
5
6
```

```c
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
4
5
6
```

```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

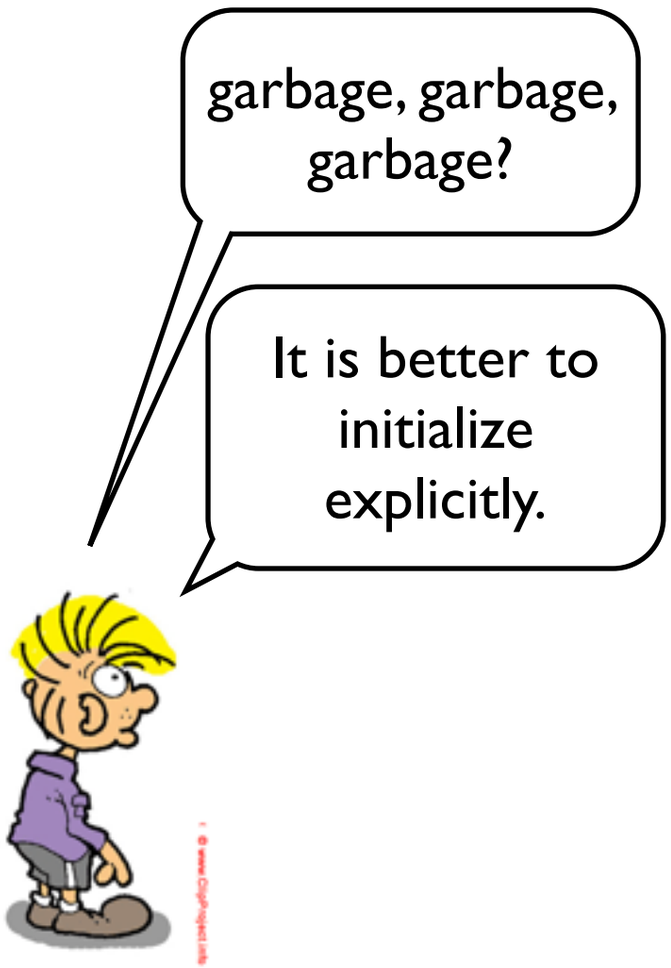
garbage, garbage, garbage?

```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

garbage, garbage, garbage?

```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0




```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0

I agree, in this case. But, as a professional programmer, you sometimes have to read code written by other people.

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0

I agree, in this case. But, as a professional programmer, you sometimes have to read code written by other people.

```
1
```

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0

I agree, in this case. But, as a professional programmer, you sometimes have to read code written by other people.

```
1
2
```

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

No. Variables with static storage duration are initialized to 0

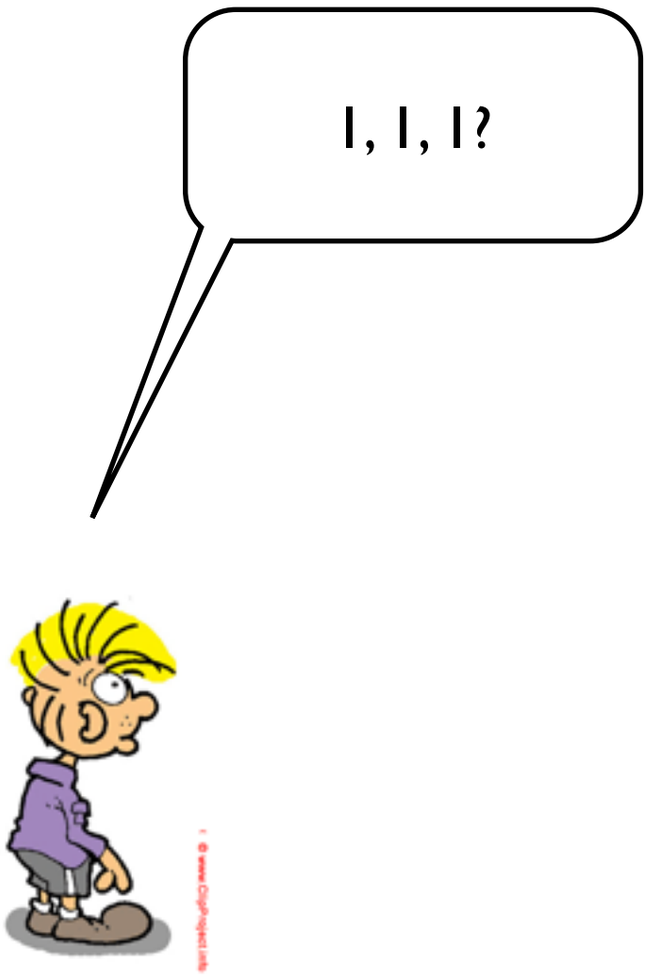
I agree, in this case. But, as a professional programmer, you sometimes have to read code written by other people.

```
1
2
3
```

```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

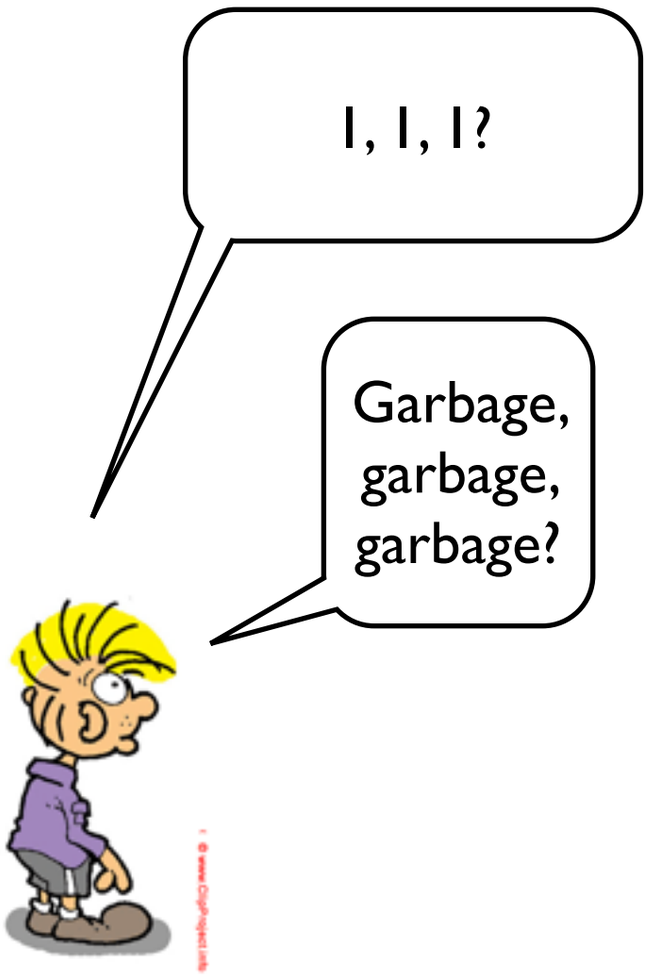



```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

1, 1, 1?

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

No, variables with automatic storage duration are not initialized implicitly




```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

No, variables with automatic storage duration are not initialized implicitly

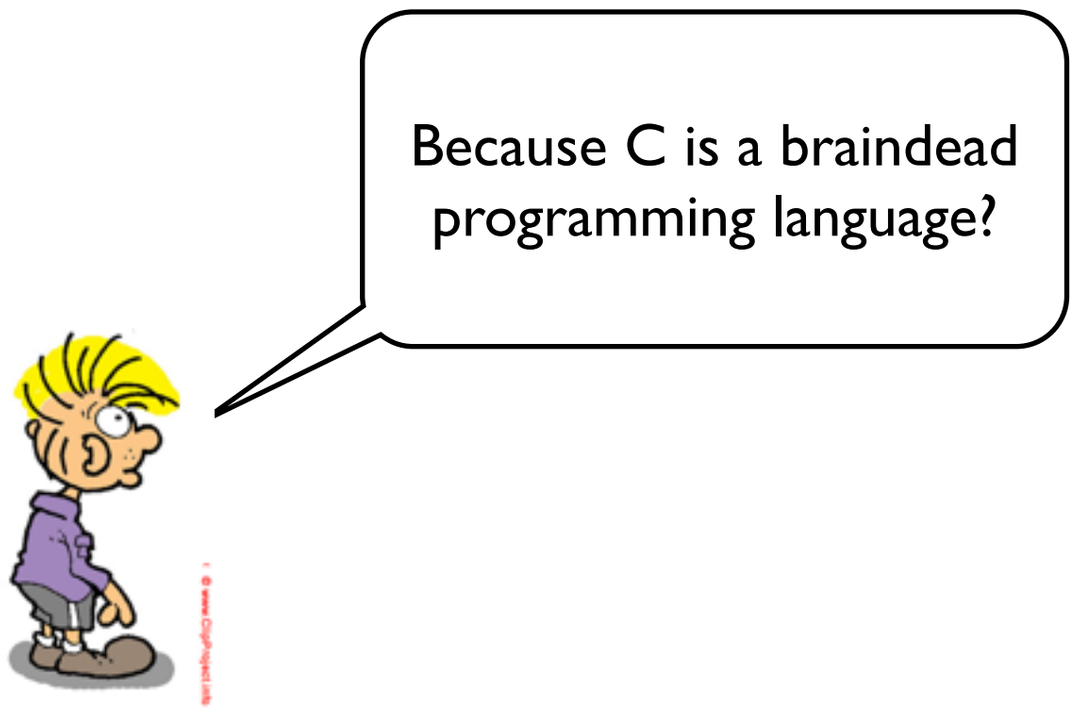

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

No, variables with automatic storage duration are not initialized implicitly

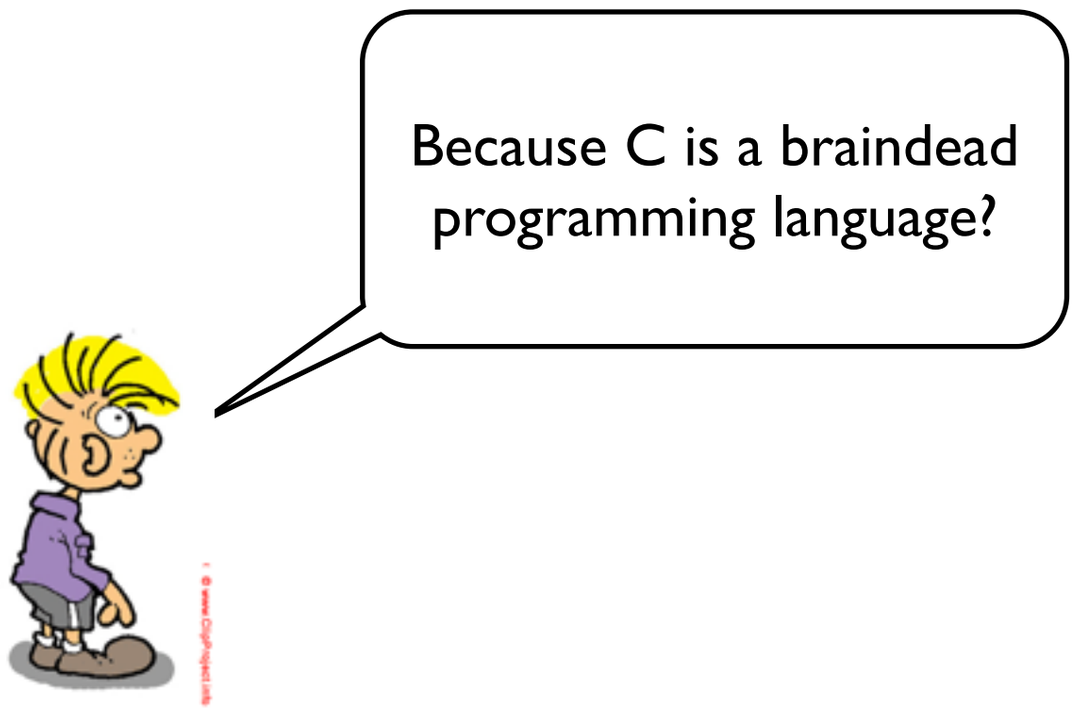
This is undefined behavior, so anything can happen

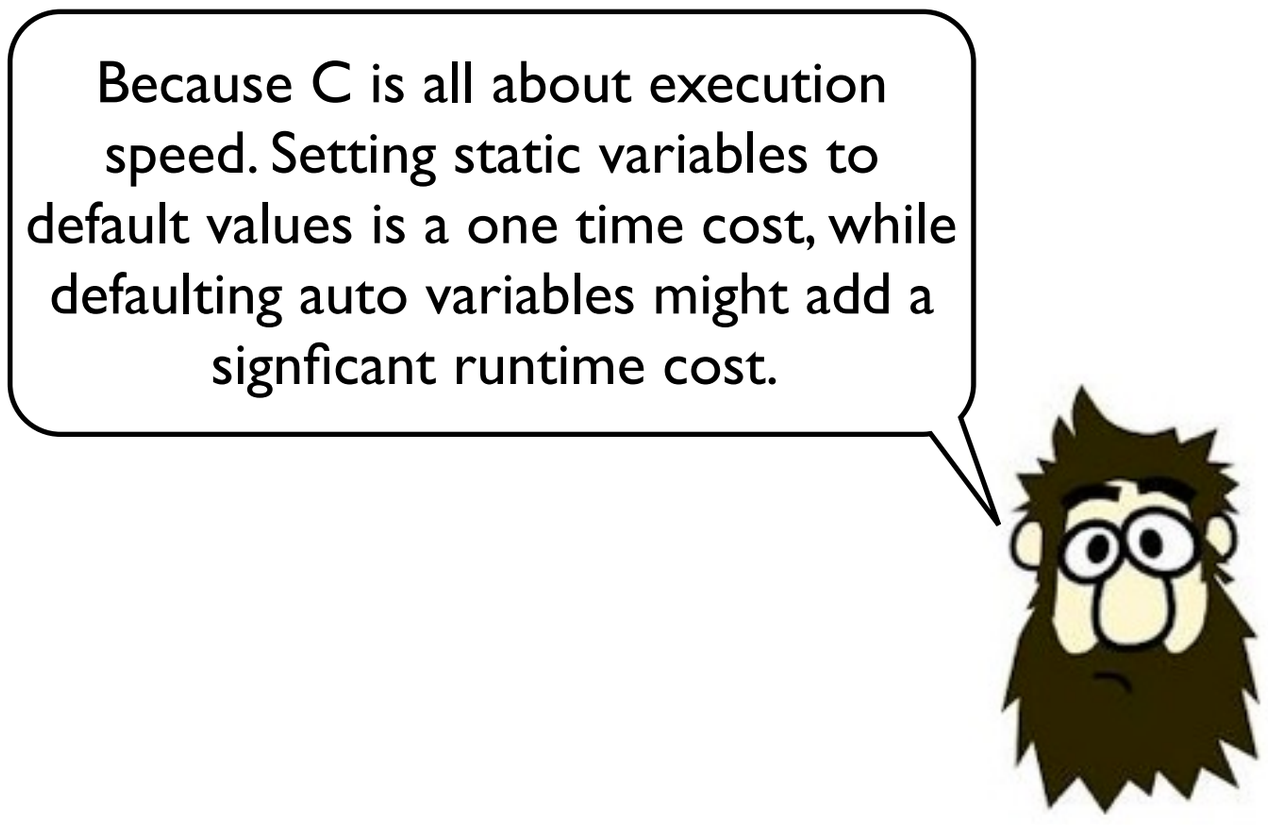
In C. Why do you think static variables gets a default value (usually 0), while auto variables does not get a default value?

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



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





```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

Let's try it on my machine

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
```

Let's try it on my machine

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
```

Let's try it on my machine

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Let's try it on my machine

Ehh...

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Let's try it on my machine

Ehh...

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Let's try it on my machine

any plausible explaination for this behavior?

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Ehh...

Let's try it on my machine

any plausible explaination for this behavior?

Is it because: "The value of an object with automatic storage duration is used while it is indeterminate"?

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Ehh...

Let's try it on my machine

any plausible explaination for this behavior?

Is it because: "The value of an object with automatic storage duration is used while it is indeterminate"?

That explains why this is **undefined behavior**, but it does not explain the phenomenon we just observed: 1,2,3

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Ehh...

Let's try it on my machine

any plausible explaination for this behavior?

Is it because: "The value of an object with automatic storage duration is used while it is indeterminate"?

That explains why this is **undefined behavior**, but it does not explain the phenomenon we just observed: 1,2,3

But if it is UB, do I need to care?

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
1
2
3
```

Ehh...

Let's try it on my machine

any plausible explaination for this behavior?

Is it because: "The value of an object with automatic storage duration is used while it is indeterminate"?

That explains why this is **undefined behavior**, but it does not explain the phenomenon we just observed: 1,2,3

But if it is UB, do I need to care?

You do, because everything becomes much easier if you can and are willing to reason about these things...

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
```

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
$ ./a.out
```

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
$ ./a.out
1
```

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
$ ./a.out
1
2
```

But seriously, I don't need to know, because I let the compiler find bugs like this

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
$ ./a.out
1
2
3
```

But seriously, I don't need to know, because I let the compiler find bugs like this

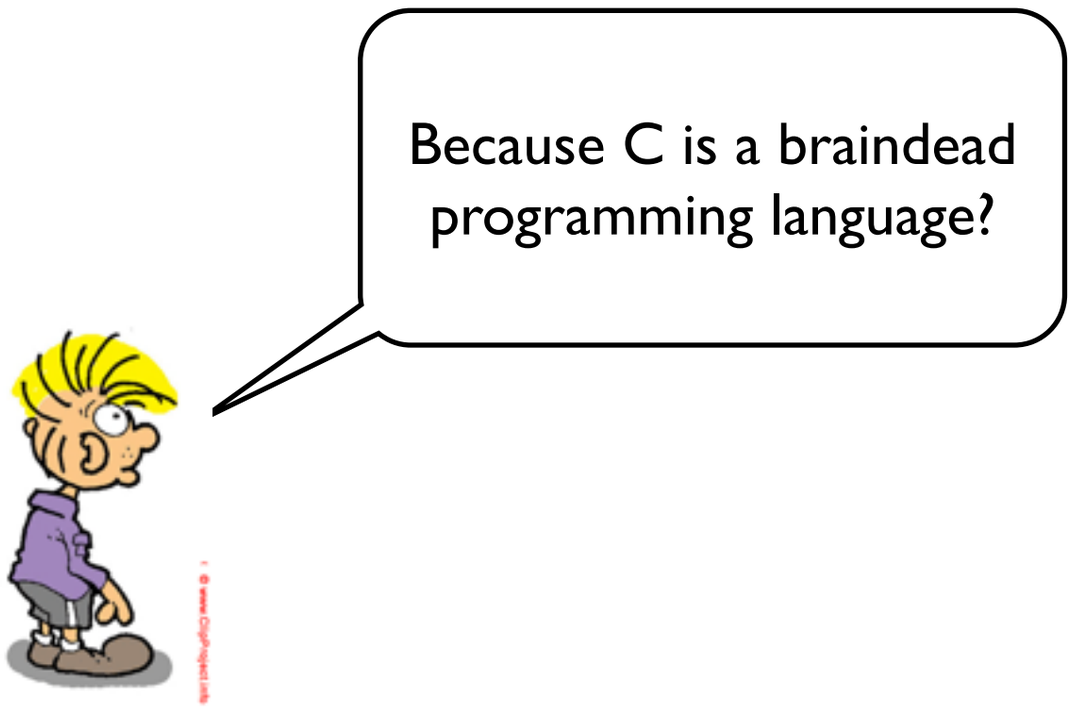
Lousy compiler!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -Wall -Wextra -pedantic foo.c
$ ./a.out
1
2
3
```

Why don’t the C standard require that you always get a warning or error on invalid code?

Why don't the C standard require that you always get a warning or error on invalid code?



Why don't the C standard require that you always get a warning or error on invalid code?

Because C is a braindead programming language?

One of the design goals of C is that it should be relatively easy to write a compiler. Adding a requirement that the compilers should refuse or warn about invalid code would add a huge burden on the compiler writers.

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

Pro tip:
Always
compile with
optimization!

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
```

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
```

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
```

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
```


```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
```

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
$ cc -O -Wall -Wextra foo.c
```


```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
```

Pro tip:
Always
compile with
optimization!

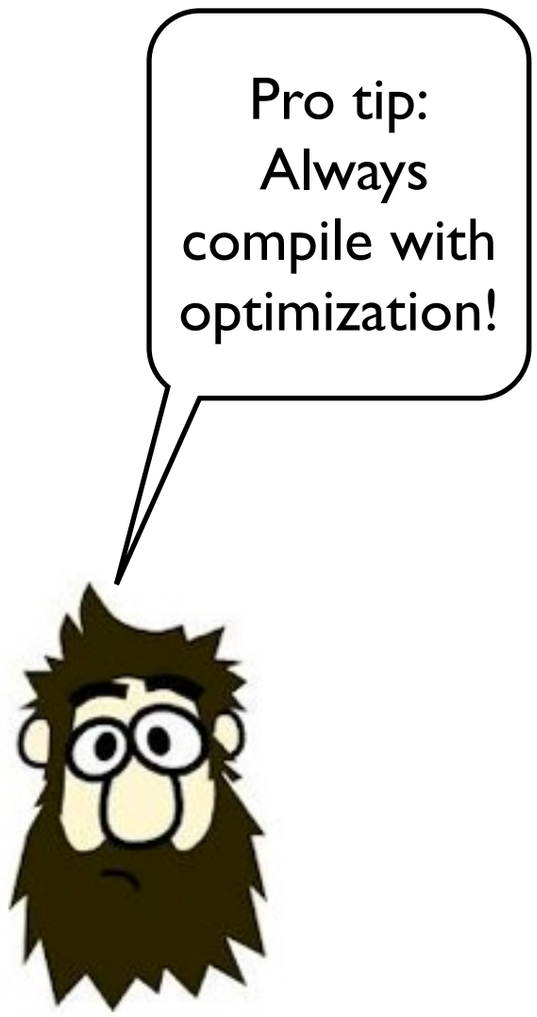
```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1450342656
```

Pro tip:
Always
compile with
optimization!

```c
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1450342656
1450340344
```




```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

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

```
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1494497536
1494495224
1494495224
$ cc -O -Wall -Wextra foo.c
foo.c:6: warning: 'a' is used uninitialized in this function
1450342656
1450340344
1450340344
```

I am now going to show you something cool!

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

I am now going to show you something cool!

```
$ cc foo.c && ./a.out
```

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

Can you explain this behavior?

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

Can you explain this behavior?

If you can give a plausible explanation for this behavior, you should feel both good and bad:

Bad because you obviously know something you are supposed to not know when programming in C. You make assumptions about the underlying implementation and architecture.

Good because being able to understand such phenomenons are essential for troubleshooting C programs and for avoiding falling into all the traps laid out for you.

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

Can you explain this behavior?

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

Can you explain this behavior?

eh?

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

Can you explain this behavior?

eh?

Perhaps this compiler has a pool of named variables that it reuses. Eg variable `a` was used and released in `bar()`, then when `foo()` needs an integer named `a` it will get the same variable for reuse. If you rename the variable in `bar()` to, say `b`, then I don't think you will get 42.

I am now going to show you something cool!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```
$ cc foo.c && ./a.out
42
```

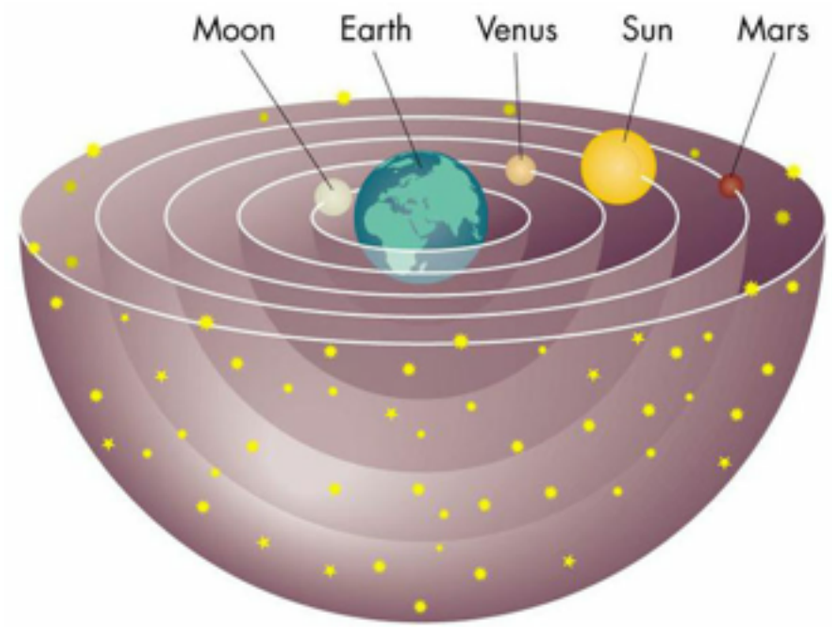
Can you explain this behavior?

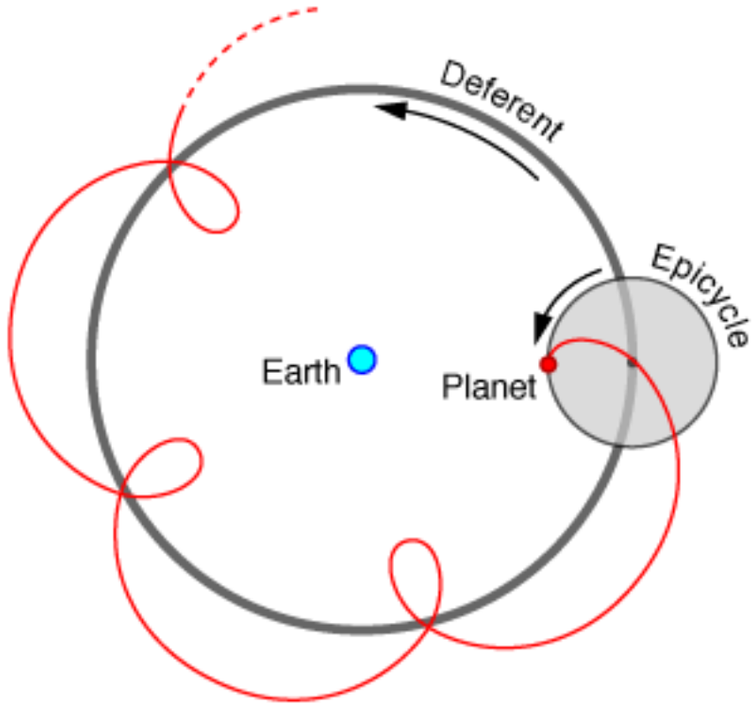
eh?

Perhaps this compiler has a pool of named variables that it reuses. Eg variable `a` was used and released in `bar()`, then when `foo()` needs an integer named `a` it will get the same variable for reuse. If you rename the variable in `bar()` to, say `b`, then I don't think you will get 42.

Yeah, sure...

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

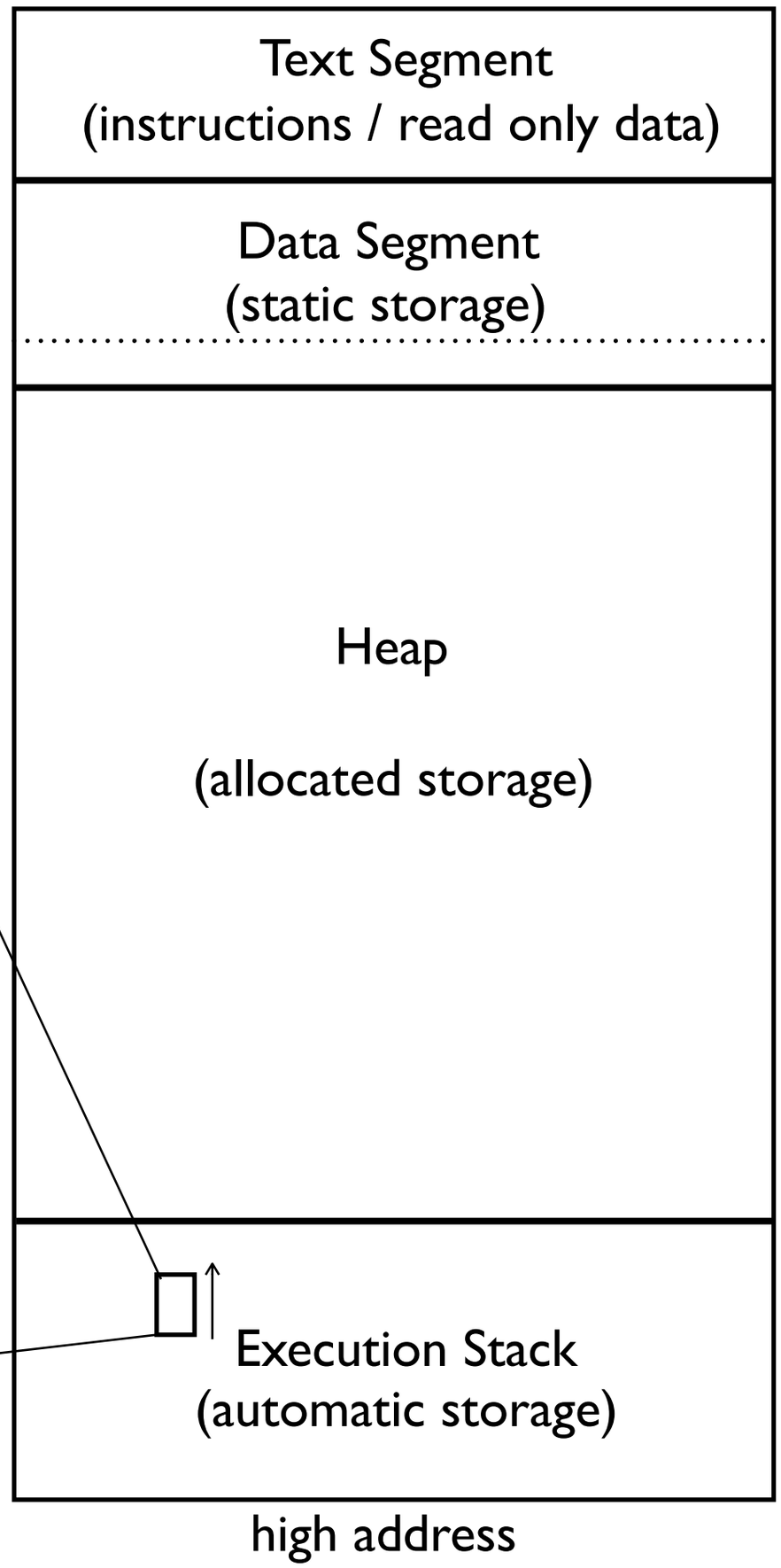




Memory Layout *

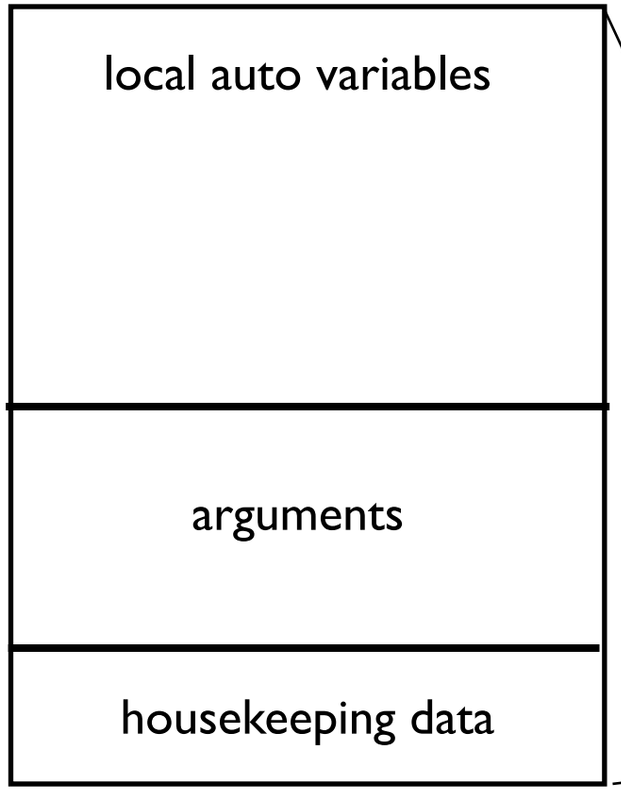
It is sometimes useful to assume that a C program uses a memory model where the instructions are stored in a **text segment**, and static variables are stored in a **data segment**. Automatic variables are allocated when needed together with housekeeping variables on an **execution stack** that is growing towards low address. The remaining memory, the **heap** is used for allocated storage.

The stack and the heap is typically not cleaned up in any way at startup, or during execution, so before objects are explicitly initialized they typically get garbage values based on whatever is left in memory from discarded objects and previous executions. In other words, the programmer must do all the housekeeping on variables with automatic storage and allocated storage.



Activation Record

And sometimes it is useful to assume that an **activation record** is created and pushed onto the execution stack every time a function is called. The activation record contains local auto variables, arguments to the functions, and housekeeping data such as pointer to the previous frame and the return address.



(*) The C standard does not dictate any particular memory layout, so what is presented here is just a useful conceptual example model that is similar to what some architecture and run-time enviornments look like

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

And since this is UB, using another compiler or just changing the optimization flag might change the behavior

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

And since this is UB, using another compiler or just changing the optimization flag might change the behavior

```
$ cc foo.c && ./a.out
42
```

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

And since this is UB, using another compiler or just changing the optimization flag might change the behavior

```
$ cc foo.c && ./a.out
42
```

```
$ cc -O foo.c && ./a.out
-294197238
```

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

And since this is UB, using another compiler or just changing the optimization flag might change the behavior

```
$ cc foo.c && ./a.out
42
```

```
$ cc -O foo.c && ./a.out
-294197238
```

Perhaps UB is the reason why our programs always crashes when we try to upgrade the compiler?

```c
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

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

And since this is UB, using another compiler or just changing the optimization flag might change the behavior

```
$ cc foo.c && ./a.out
42
```

```
$ cc -O foo.c && ./a.out
-294197238
```

Perhaps UB is the reason why our programs always crashes when we try to upgrade the compiler?

But whatever, anything wrong with gcc 3.0?

Behavior

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

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

    // unspecified
    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
    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 behavior, but says very little about ***how*** it should be implemented.

the C standard defines the expected behavior, 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!

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out


```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```
437
347
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

or

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

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

int main(void) {
    int i = foo(3) + foo(4);
    printf("%d\n", i);

    int j = bar(foo(3), foo(4));
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
347
437
```

but you might also get

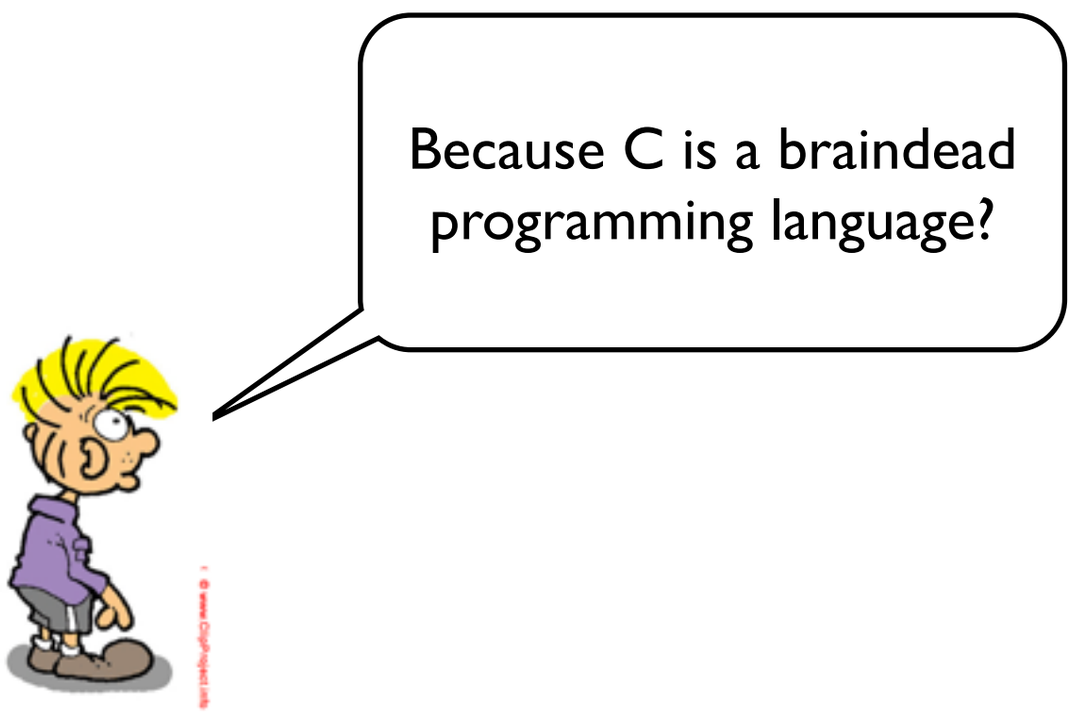
```
437
347
```
or
```
437
437
```
or
```
347
347
```

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

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?



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 great optimization opportunities.

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
```

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
42
```

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

What? Inconceivable!

```
$ cc foo.c && ./a.out
42
```

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

What? Inconceivable!

```
$ cc foo.c && ./a.out
42
```

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

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

What? Inconceivable!

```
$ cc foo.c && ./a.out
42
```

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

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

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

What? Inconceivable!

```
$ cc foo.c && ./a.out
42
```

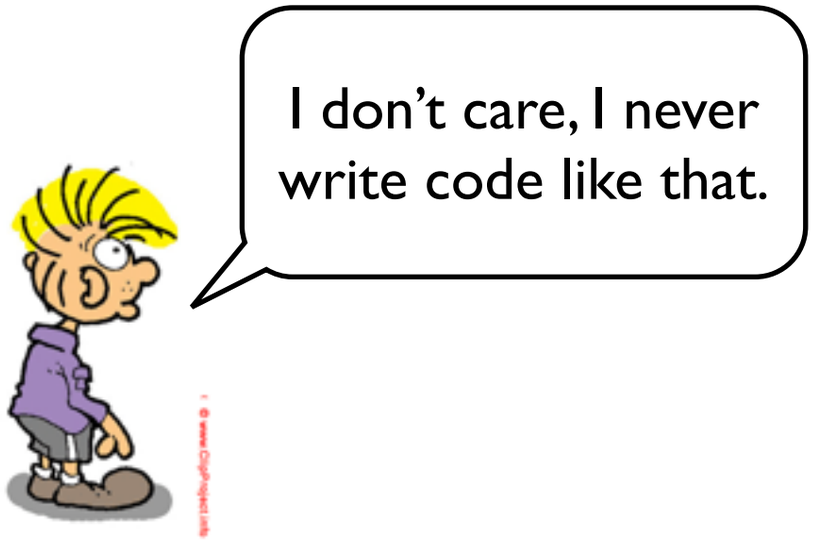
I agree this is crap code, but why is it wrong?

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

In this case? Line 7. As the evaluation order here is unspecified, the expression does not make sense when variable i is both read and updated in the expression. What is the value of i? When does the side-effect of i++ take place?

```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./a.out
42
```



```c
#include <stdio.h>

int v[3] = {1,2,3};

int main(void) {
    int i = 1;
    int j = i + v[i++];
    printf("%d\n", j);
}
```

```
$ cc foo.c && ./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...

Sequence points and sequencing

What do these code snippets print?

Sequence points and sequencing

What do these code snippets print?

1

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

Sequence points and sequencing

What do these code snippets print?

1

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

2

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

Sequence points and sequencing

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);
```

Sequence points and sequencing

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);
```

Sequence points and sequencing

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);
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

Sequence points and sequencing

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");
```

?

Sequence points and sequencing

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++?

C99

6.5 Expressions

An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

C11

6.5 Expressions

An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

C99

6.5 Expressions

An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression.
Furthermore, the prior value shall be read only to determine the value to be stored.

C11

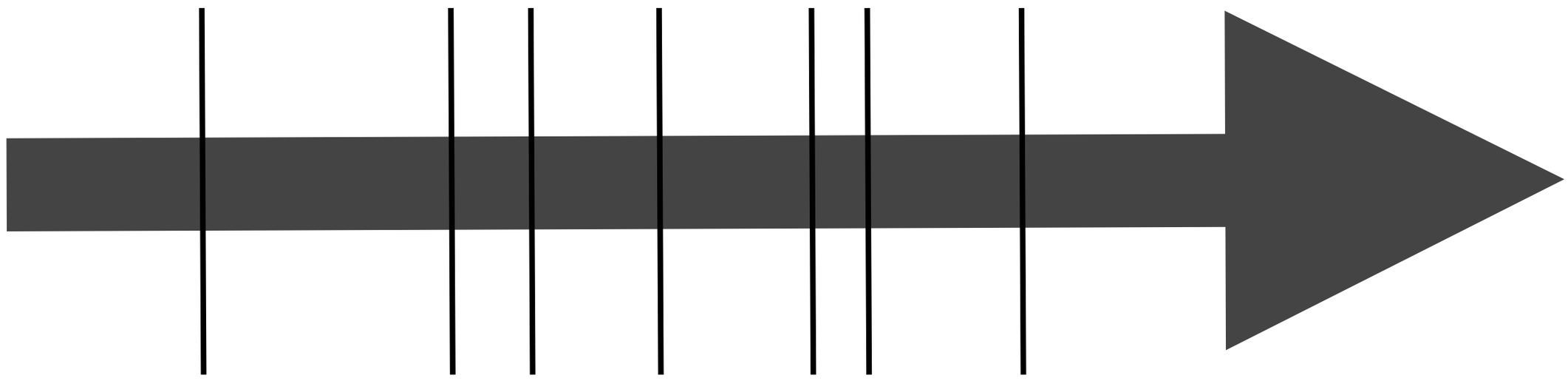
6.5 Expressions

An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. **The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.**

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

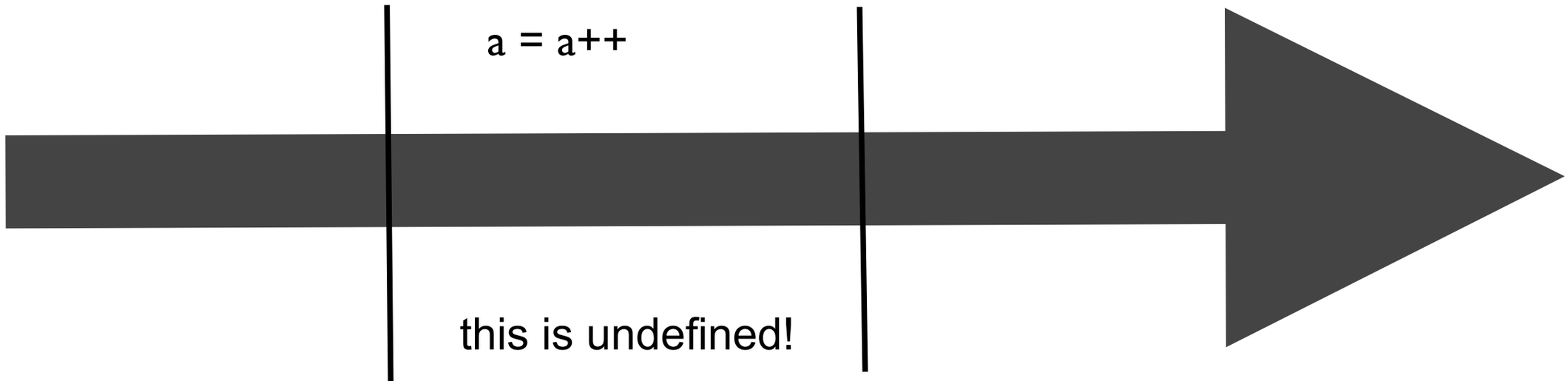
Sequence Points

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



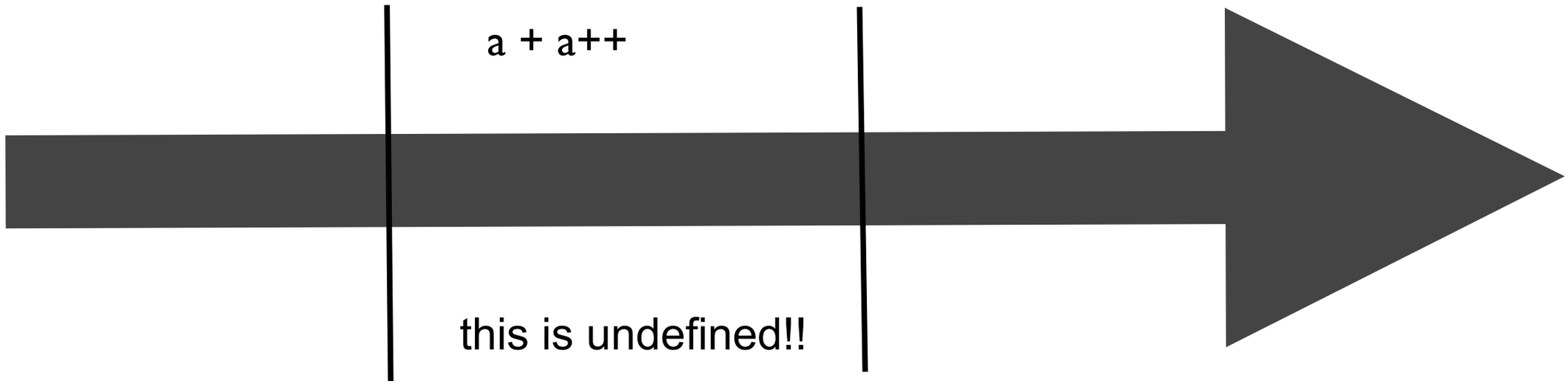
Sequence Points - Rule 1

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



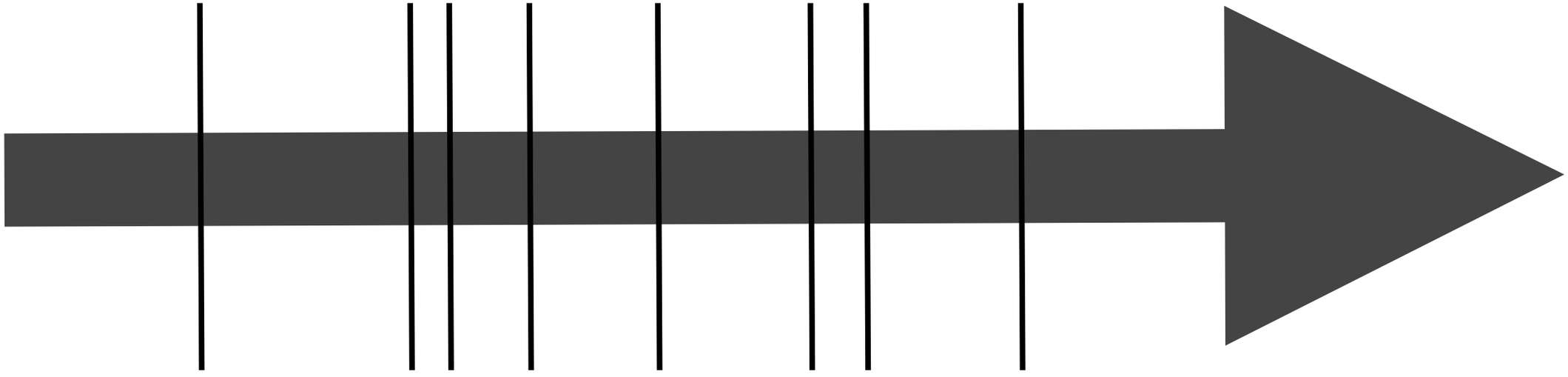
Sequence Points - Rule 2

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



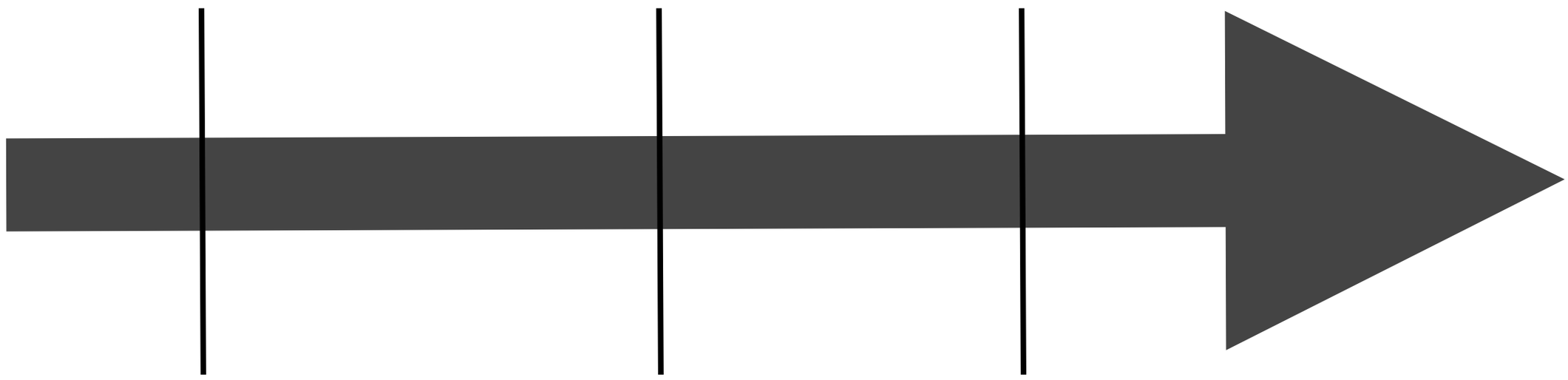
Sequence Points

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
(C99 & C++98)

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 <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```
$ cc foo.c
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```
$ cc foo.c
$ ./a.out
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```
$ cc foo.c
$ ./a.out
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```
$ cc foo.c
$ ./a.out
4
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

```
$ cc foo.c
$ ./a.out
4
4
4
```

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

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

```
$ cc foo.c
$ ./a.out
4
4
4
```

They are all morons!

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

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

```
$ cc foo.c
$ ./a.out
4
4
4
```

They are all morons!

```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

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?

```
$ cc foo.c
$ ./a.out
4
4
4
```

They are all morons!

ehh...

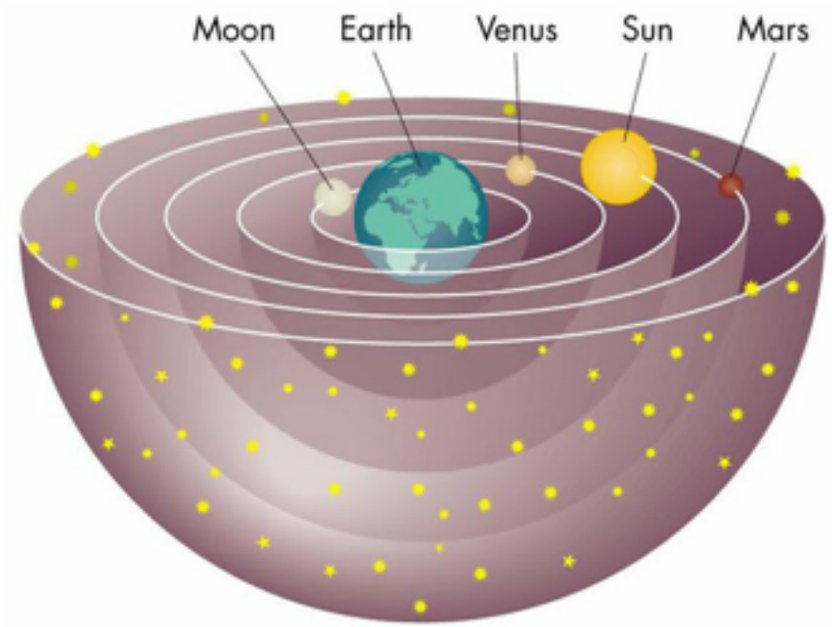
```c
#include <stdio.h>

void foo(void)
{
    int a = 3;
    a++;
    printf("%d\n", a);
}

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

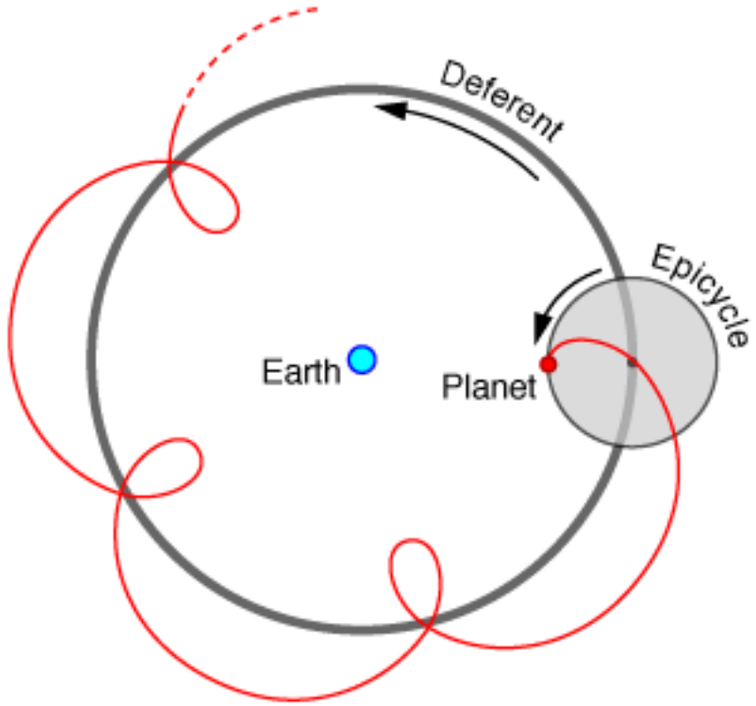
Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

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

```
$ cc foo.c
$ ./a.out
4
4
4
```

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





comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

comparing signed and unsigned ints

Before we continue... think carefully... what will this code print?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?

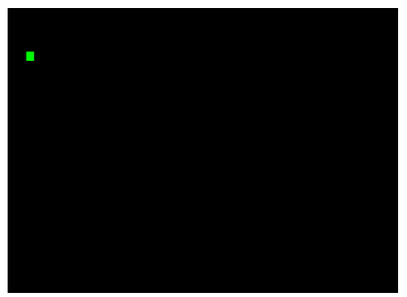
comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?



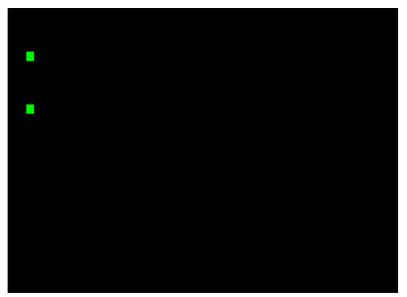
comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?



comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?

```
.
.
3
```

comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?

```
.
.
3
4
```

comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?

```
.
.
3
4
5
```

comparing signed and unsigned ints

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    int b = -1;
    if (a > b) puts("1"); else puts(".");

    unsigned long c = 1;
    long d = -1;
    if (c > d) puts("2"); else puts(".");

    unsigned short e = 1;
    short f = -1;
    if (e > f) puts("3"); else puts(".");

    unsigned short g = 1;
    int h = -1;
    if (g > h) puts("4"); else puts(".");

    unsigned int i = 1;
    long j = -1;
    if (i > j) puts("5"); else puts(".");
}
```

Before we continue... think carefully... what will this code print?

Don't rush it. Think first.

Are you ready for the answer?

```
.
.
3
4
5
```

Exactly as you expected?

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```

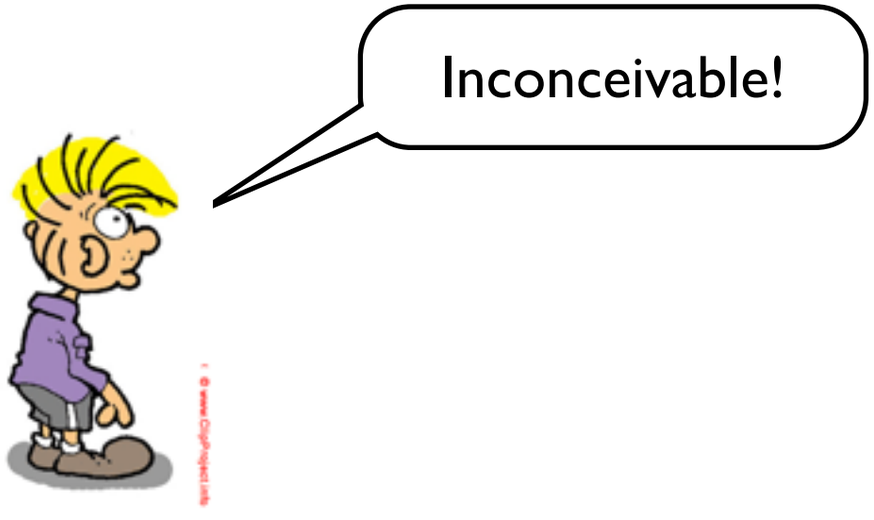
deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```



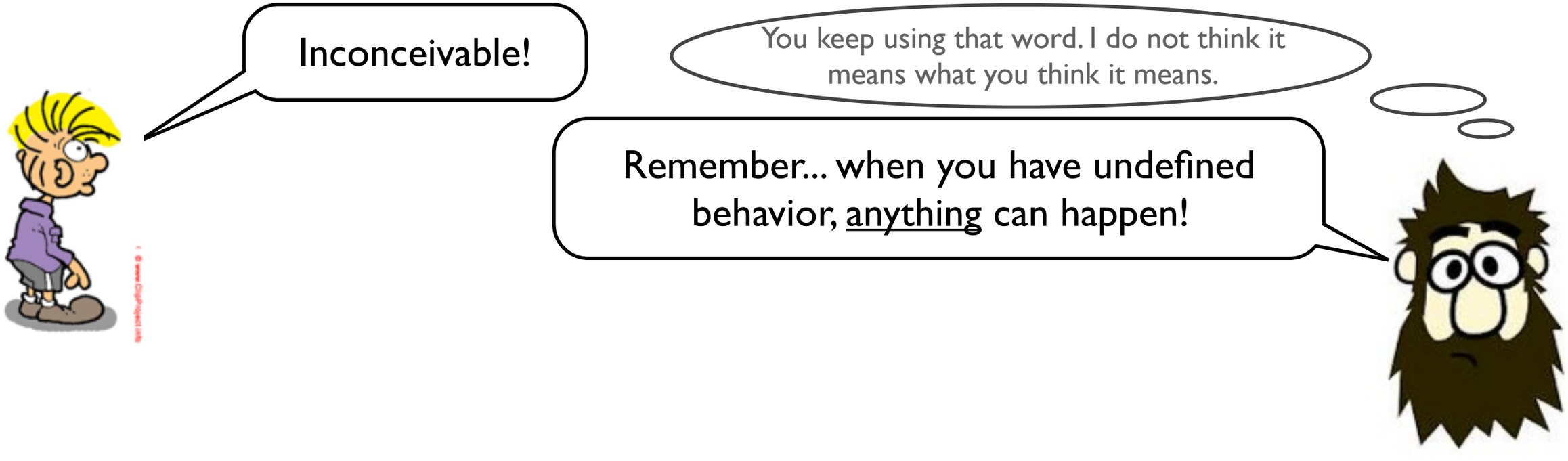
deep_thought.c

```c
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```



deep_thought.c

```c
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

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

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```

Inconceivable!

You keep using that word. I do not think it means what you think it means.

Remember... when you have undefined behavior, anything can happen!

Integer overflow gives undefined behavior. If you want to prevent this to happen you must write the logic yourself. This is the spirit of C, you don't get code you have not asked for.

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.c

```
#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.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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.c

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

foo.c

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

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

main.c

```
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?

bar.c

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

foo.c

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

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

main.c

```
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:

(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.c

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

foo.c

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

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

main.c

```
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?

bar.c

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

foo.c

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

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

main.c

```
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
```

```
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?

bar.c

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

foo.c

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

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

main.c

```
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
```

```
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");
}
```

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

icc 13.0.1 with no optimization (–00)

```
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 (–00)

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

icc 13.0.1 with no optimization (–00)

```
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 (–00)

```
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 (–00)

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

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");
}
```

icc 13.0.1 with no optimization (–00)

```
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 (–00)

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

```
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;
}
```

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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");
}
```

```
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
```

```
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 (–00)

```
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 (–00)

```
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 4.7.2 with no optimization (–00)

```
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 (–00)

```
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 4.7.2 with no optimization (–00)

```
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 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 (–00)

```
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 (–00)

```
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 (–00)

```
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 (–00)

```
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.c

```
#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

So what's wrong with this code?

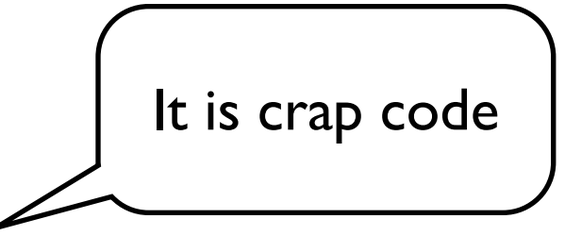
foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```



So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code ?

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

It is crap code

The standard says that this is invalid code ?

Update a variable multiple times between two semicolons

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

In C (and C++), unlike most other languages, the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators, is unspecified. Therefore the expression

```
i + v[++i] + v[++i]
```

does not make sense.

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 will happen. 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 can be corrupted - meaning that UB might result in strange behavior in apparently unrelated parts of the codebase.

But, seriously, who is releasing code with undefined behavior?

But, seriously, who is releasing code with undefined behavior?

But, seriously, who is releasing code with undefined behavior?

But, seriously, who is releasing code with undefined behavior?

But, seriously, who is releasing code with undefined behavior?

But, seriously, who is releasing code with undefined behavior?


cité
de
l'espace
esa
esa
cnes
cnes


AFP

But, seriously, who is releasing code with undefined behavior?

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

```
...
    /* if both are imag, store value, otherwise store 0.0 */
    if (!(li && ri)) {
        tfree(r);
        r = bcon(0);
    }
    p = buildtree(ASSIGN, l, r);
    p->n_type = p->n_type += (FIMAG-FLOAT);
....
```

But, seriously, who is releasing code with undefined behavior?

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

```
...
    /* if both are imag, store value, otherwise store 0.0 */
    if (!(li && ri)) {
        tfree(r);
        r = bcon(0);
    }
    p = buildtree(ASSIGN, l, r);
    p->n_type = p->n_type += (FIMAG-FLOAT);
....
```

But, seriously, who is releasing code with undefined behavior?

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

```
...
    /* if both are imag, store value, otherwise store 0.0 */
    if (!(li && ri)) {
        tfree(r);
        r = bcon(0);
    }
    p = buildtree(ASSIGN, l, r);
    p->n_type = p->n_type += (FIMAG-FLOAT);
....
```

It's undefined behavior because: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored
the you modify and use the value of a variable twice between sequence points"

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