Deep C++

by Olve Maudal

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

Programming is hard. Programming correct C++ is particularly hard. Indeed, it is uncommon to see a screenful containing only well defined and conforming code. Why do professional programmers write code like this? Because most programmers do not have a deep understanding of the language they are using. While they sometimes know that certain things are undefined or unspecified, they often do not know why it is so.

In this talk we will study small code snippets of C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of this wonderful but dangerous programming language.

A 60 minute session at Norwegian Developers Conference 2013
Friday, June 14, 2013

JAWS

Exercise

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

```
#include <iostream>

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

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

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

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

Exercise

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

```
#include <iostream>

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

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

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

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

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

Exercise

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

```
#include <iostream>

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

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

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

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

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

Exercise

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

```
#include <iostream>

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

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

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

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

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

Exercise

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

```
#include <iostream>

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

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

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

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

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

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

Exercise

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

Exercise

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

```
#include <iostream>

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

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

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

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

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

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

but you might also get

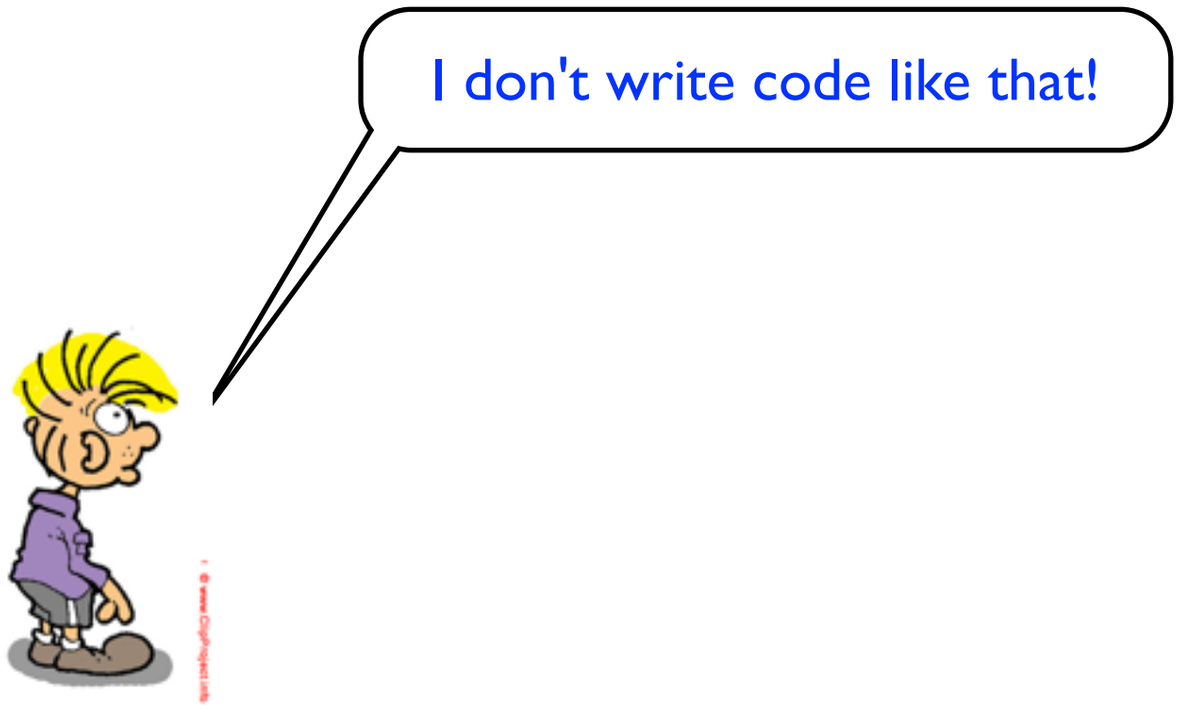
```
437
347
```

or

```
437
437
```

or

```
347
347
```


I don't write code like that!

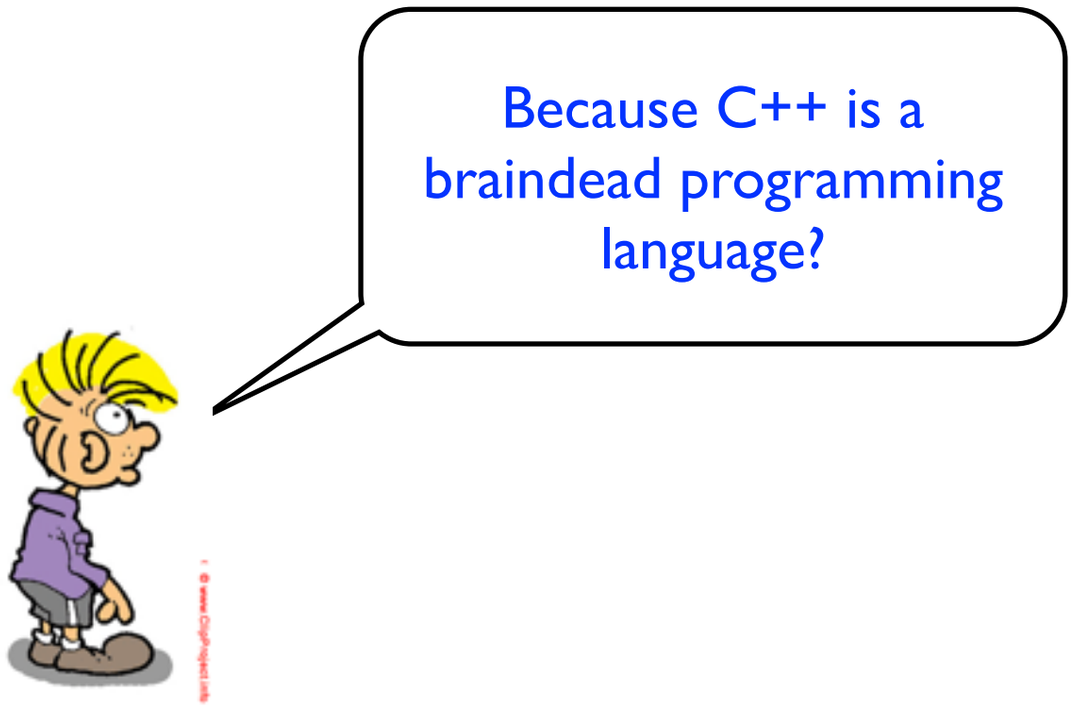
I don't write code like that!

Of course your don't!

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 better optimization.

Exercise

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

foo.cpp

```
#include <iostream>

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

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

```
$ clang++ foo.cpp && ./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.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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

Exercise

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

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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

Exercise

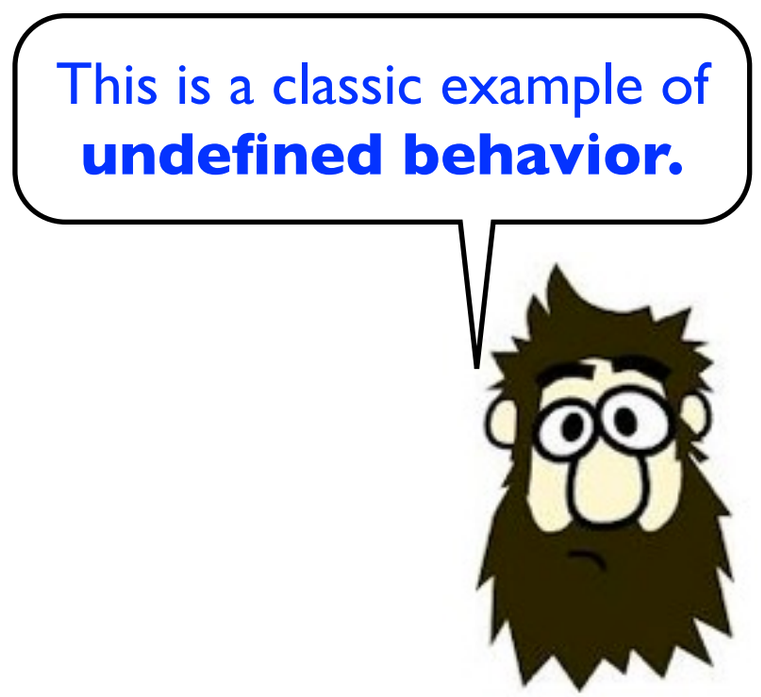
What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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



Exercise

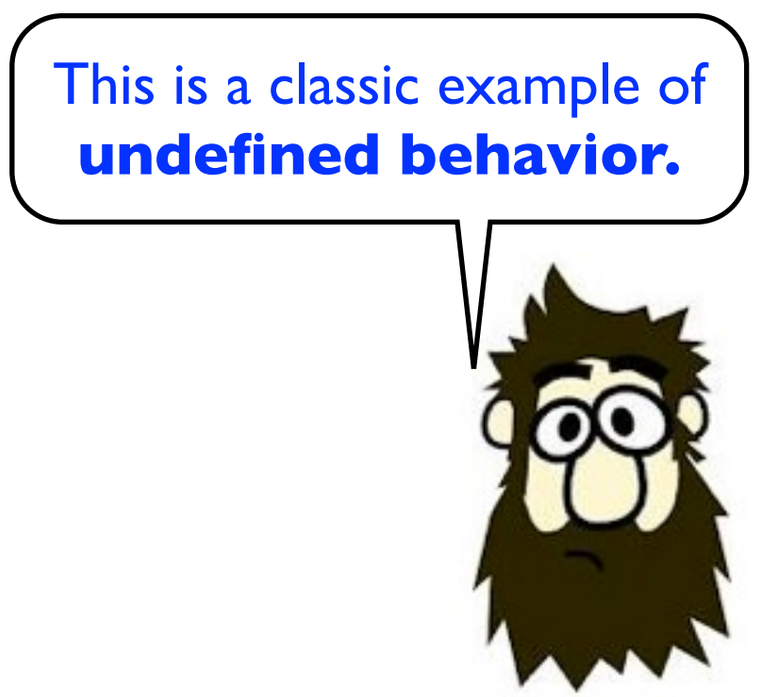
What do you think this code snippet might print if you compile, link and run it in your development environment?

foo.cpp

```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

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



foo.cpp

```
#include <iostream>

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

I don't write code like that!

Let's add some flags for better diagnostics.

foo.cpp

```
#include <iostream>

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

I don't write code like that!

Let's add some flags for better diagnostics.

foo.cpp

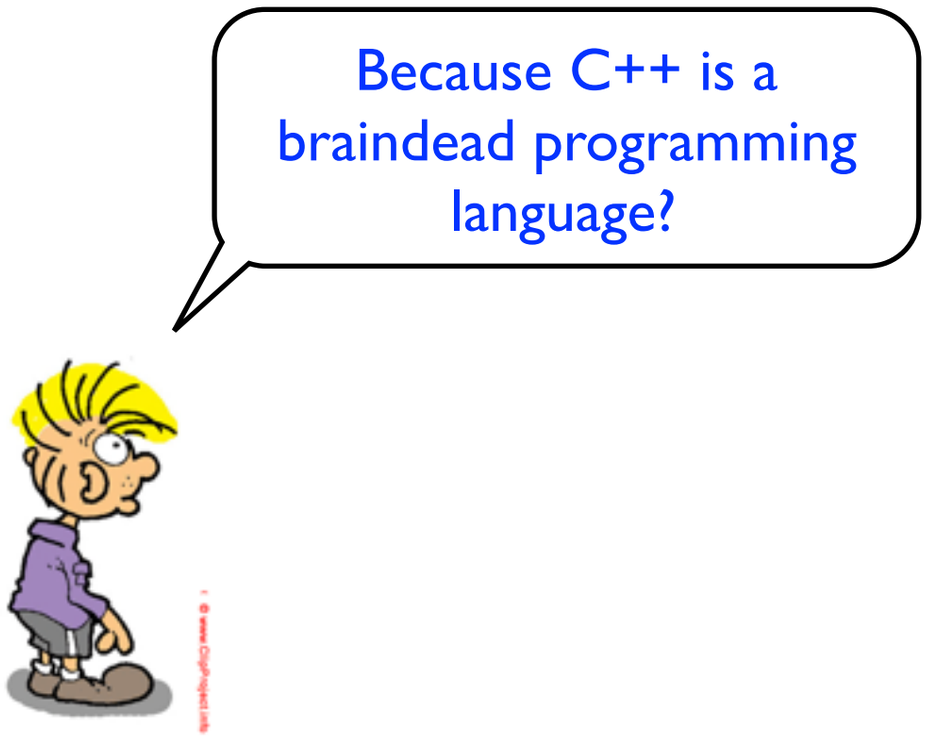
```
#include <iostream>

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

On my computer (Mac OS 10.8.3, clang 4.2, icc 13.0.2, gcc 4.2.1):

```
$ clang++ -O -Wall -Wextra -pedantic foo.cpp && ./a.out
11
$ icc -O -Wall -Wextra -pedantic foo.cpp && ./a.out
13
$ g++ -O -Wall -Wextra -pedantic foo.cpp && ./a.out
foo.cpp:7: warning: operation on 'i' may be undefined
12
```

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 primary design goals of C was that it should be relatively easy to write a compiler, which implies that the C standard could not add a requirement to detect and diagnose 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 primary design goals of C was that it should be relatively easy to write a compiler, which implies that the C standard could not add a requirement to detect and diagnose invalid code.

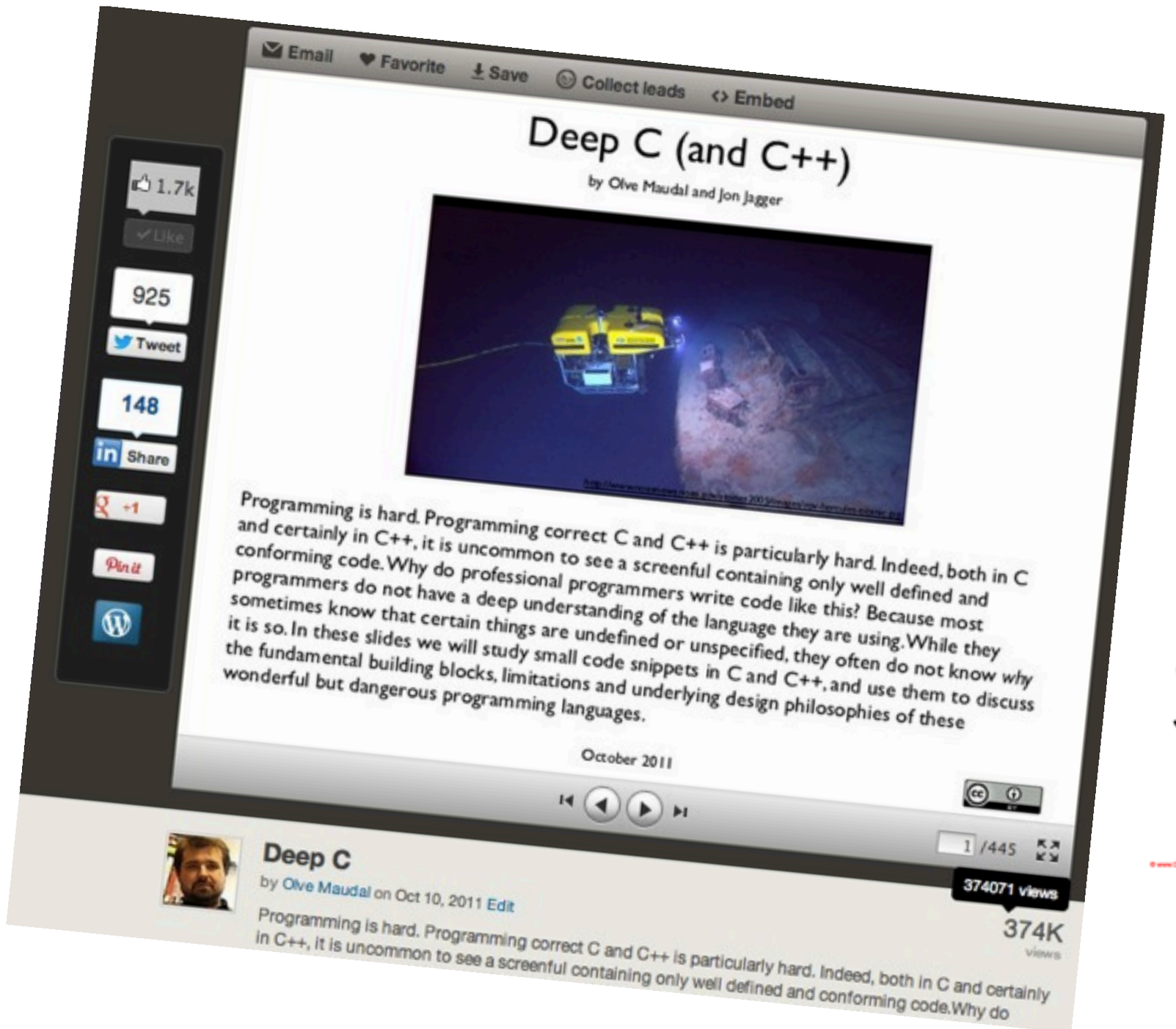
C++ has kind of adopted that attitude from C, and therefore the C++ standard does not say much about what should happen if the code is not well-formed.

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 and respect 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



Let's start with some basic stuff...

```
#include <iostream>

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```
#include <iostream>

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

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

```
#include <iostream>

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

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

```
4
```

```cpp
#include <iostream>

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

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

```
4
5
```

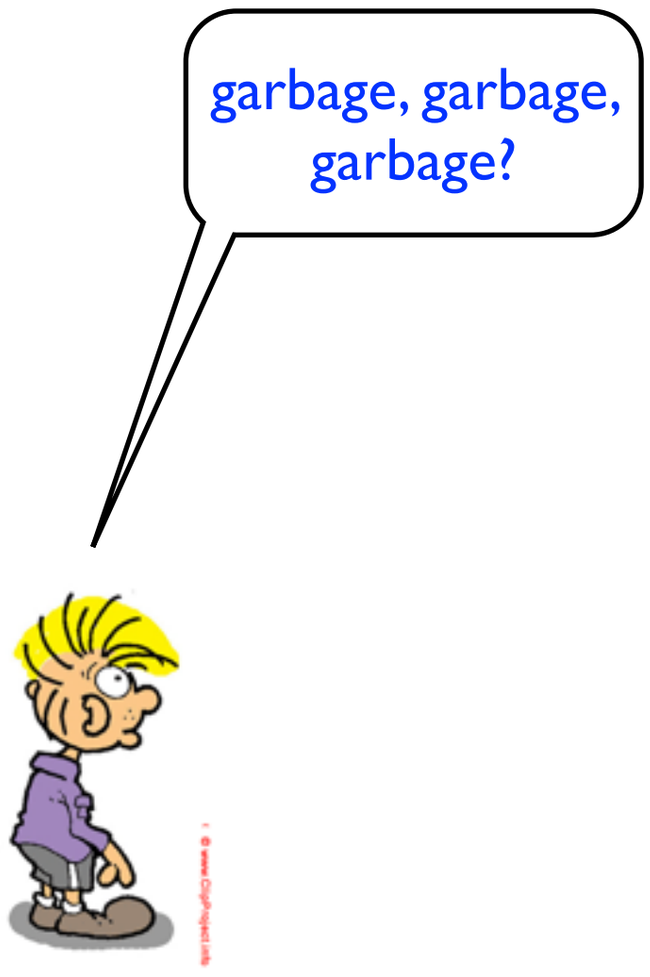
```cpp
#include <iostream>

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

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

```
4
5
6
```

```cpp
#include <iostream>

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

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

```
4
5
6
```

```
#include <iostream>

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

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




```
#include <iostream>

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

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

garbage, garbage, garbage?

```
#include <iostream>

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

```
1
```

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

```
1
2
```

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

```
1
2
3
```

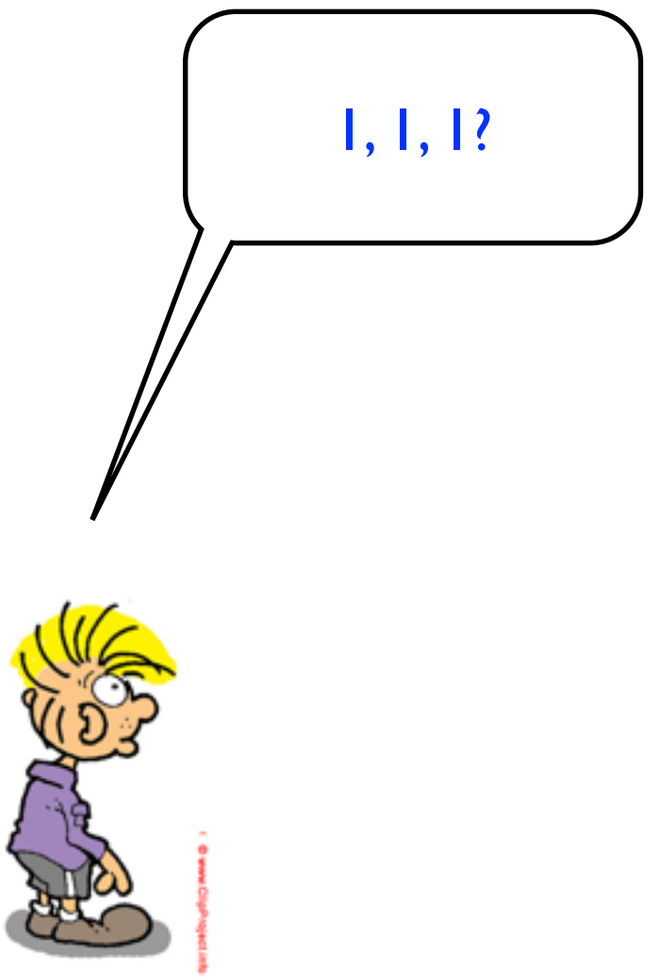
No. In C++, variables with static storage duration are initialized to their default value, in this case 0

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

```
#include <iostream>

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

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

```
#include <iostream>

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

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

```
#include <iostream>

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

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

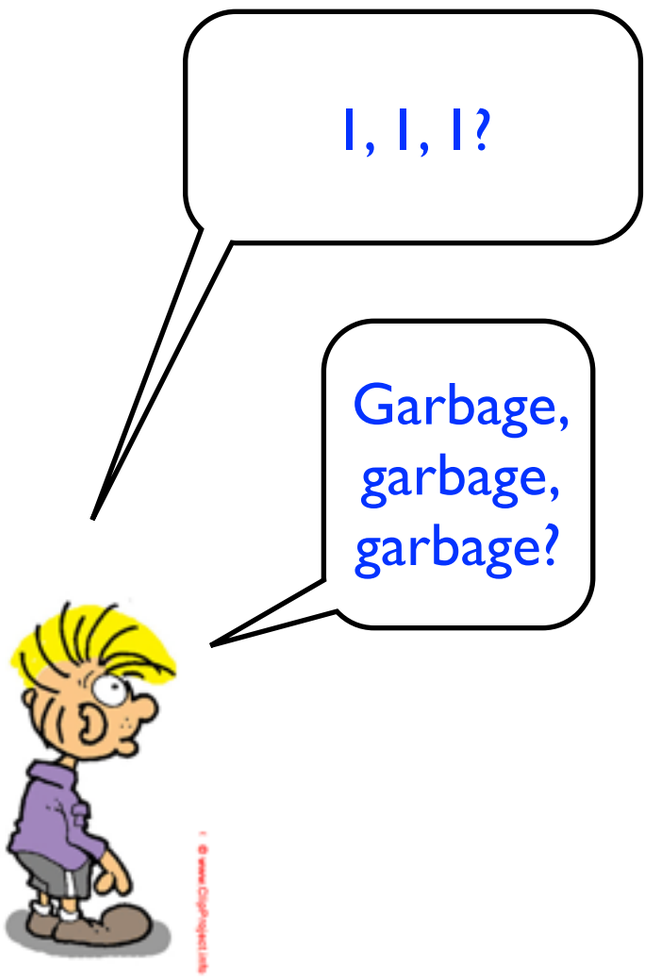



```
#include <iostream>

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

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

1, 1, 1?

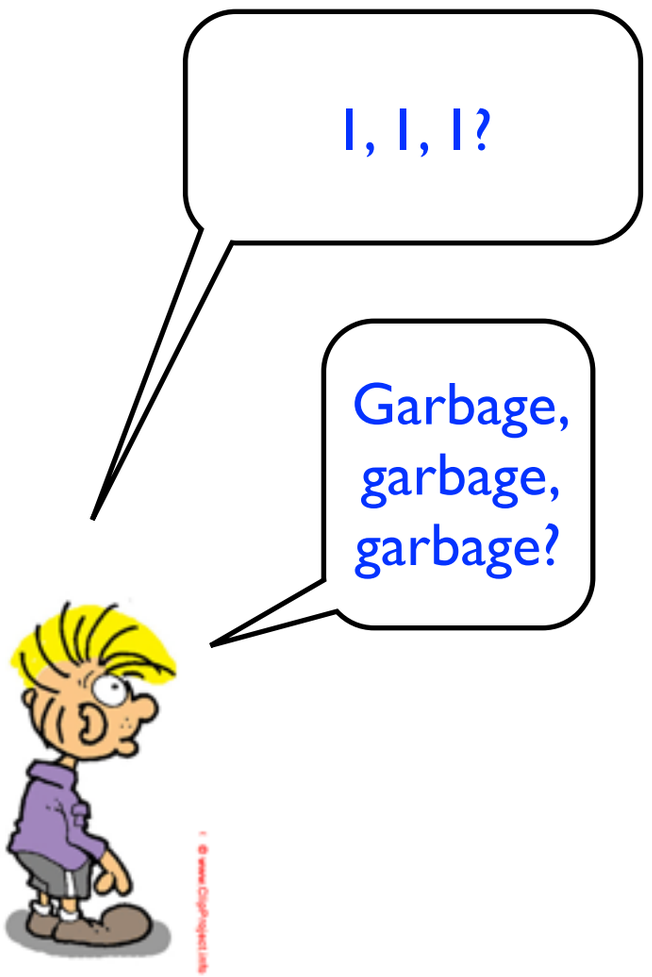
```
#include <iostream>

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

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

Variables with automatic storage duration are not initialized implicitly




```
#include <iostream>

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

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

Variables with automatic storage duration are not initialized implicitly



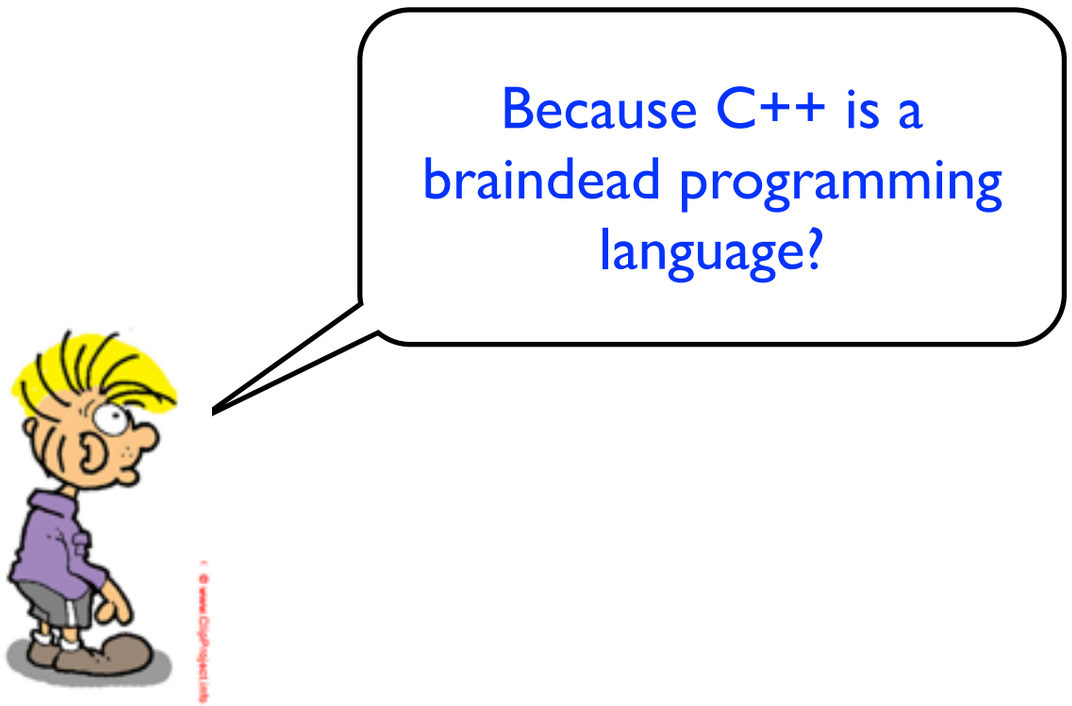

```
#include <iostream>

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

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

Variables with automatic storage duration are not initialized implicitly

Maybe.

In C++. Why do you think objects with static storage duration (eg, static variables) gets a default value (in this case 0), while objects with automatic storage duration (eg, local variables) does not get a default value?

In C++. Why do you think objects with static storage duration (eg, static variables) gets a default value (in this case 0), while objects with automatic storage duration (eg, local variables) does not get a default value?



In C++. Why do you think objects with static storage duration (eg, static variables) gets a default value (in this case 0), while objects with automatic storage duration (eg, local variables) does not get a default value?

```
#include <iostream>

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

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

```
#include <iostream>

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

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

So, let's try this code on my machine.

```
#include <iostream>

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

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

So, let's try this code on my machine.

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

```
#include <iostream>

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

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

So, let's try this code on my machine.

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

```cpp
#include <iostream>

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

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

So, let's try this code on my machine.

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

```cpp
#include <iostream>

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

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

So, let's try this code on my machine.

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

```cpp
#include <iostream>

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

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

So, let's try this code on my machine.

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

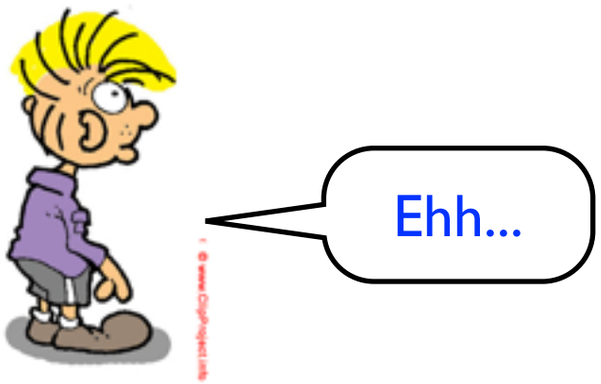
```cpp
#include <iostream>

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

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

So, let's try this code on my machine.

any plausible explaination for this behavior?

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

```
#include <iostream>

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

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

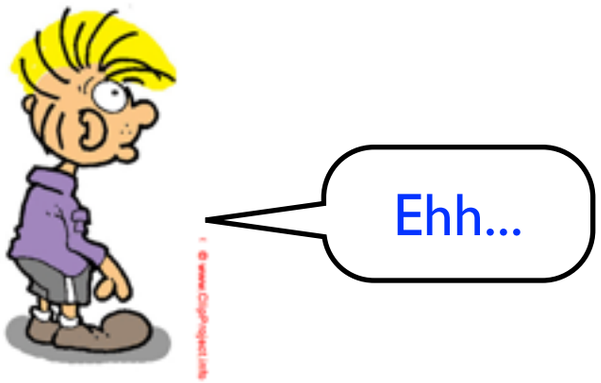


So, let's try this code on my machine.

any plausible explaination for this behavior?

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

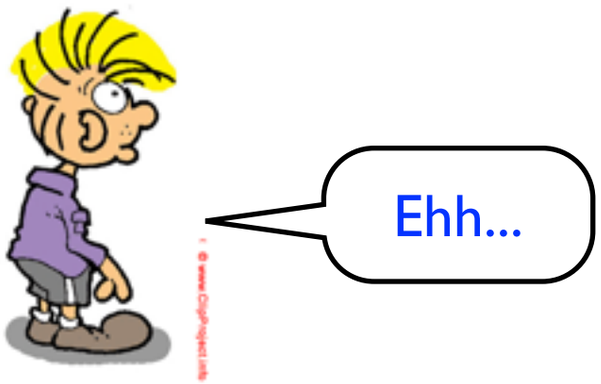
```cpp
#include <iostream>

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

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



When you compile without optimization, objects of automatic storage duration are often placed in an activation frame on a stack. In this case it seems like the "garbage value" is in the same memory location, and it is increased every time foo() is called.

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

```cpp
#include <iostream>

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

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

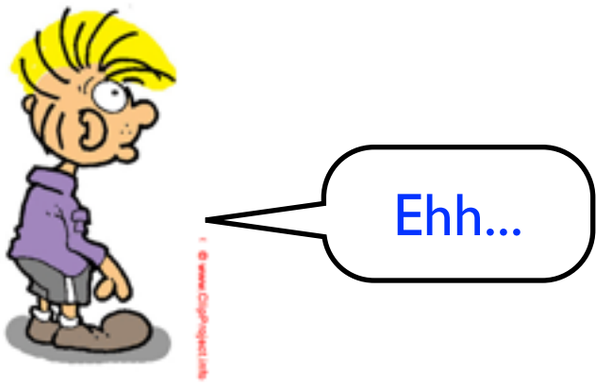


When you compile without optimization, objects of automatic storage duration are often placed in an activation frame on a stack. In this case it seems like the "garbage value" is in the same memory location, and it is increased every time foo() is called.

Here we compile without optimization. The compiler might try to be helpful and memset the whole stack to 0... just to make debugging simpler. That might be a reason why we get 1,2,3

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

```cpp
#include <iostream>

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

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



When you compile without optimization, objects of automatic storage duration are often placed in an activation frame on a stack. In this case it seems like the "garbage value" is in the same memory location, and it is increased every time foo() is called.

Here we compile without optimization. The compiler might try to be helpful and memset the whole stack to 0... just to make debugging simpler. That might be a reason why we get 1,2,3

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

Insight like this is very useful, but you should also know that...

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

It is important to understand that, at least in theory, you might as well get:

```
#include <iostream>

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

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

or even

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

or even

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

or even

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

Since we are using a value that is indeterminate (not initialized) this is Undefined Behavior!

```
#include <iostream>

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

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

It is important to understand that, at least in theory, you might as well get:

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

or

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

or even

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

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

```
#include <iostream>

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

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

OK, let's add some flags then

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

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

Lousy compiler!

```
#include <iostream>

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

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

OK, let's add some flags then

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

```
#include <iostream>

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

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

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

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

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

```
$ c++ -O -Wall -Wextra foo.cpp
warning: variable "a" is used before its value is set
```

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

```
$ c++ -O -Wall -Wextra foo.cpp
warning: variable "a" is used before its value is set
$ ./a.out
```

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

```
$ c++ -O -Wall -Wextra foo.cpp
warning: variable "a" is used before its value is set
$ ./a.out
1395100640
```

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

```
$ c++ -O -Wall -Wextra foo.cpp
warning: variable "a" is used before its value is set
$ ./a.out
1395100640
1543516672
```

Pro tip:
"always" compile
with optimization!

```
#include <iostream>

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

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

```
$ c++ -O -Wall -Wextra foo.cpp
warning: variable "a" is used before its value is set
$ ./a.out
1395100640
1543516672
1543516672
```

I am now going to show you something cool!

I am now going to show you something cool!

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

I am now going to show you something cool!

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

I am now going to show you something cool!

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

Can you explain this behavior?

I am now going to show you something cool!

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

Can you explain this behavior?

eh?

I am now going to show you something cool!

```
$ c++ -O0 foo.cpp && ./a.out
42
```

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

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 <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

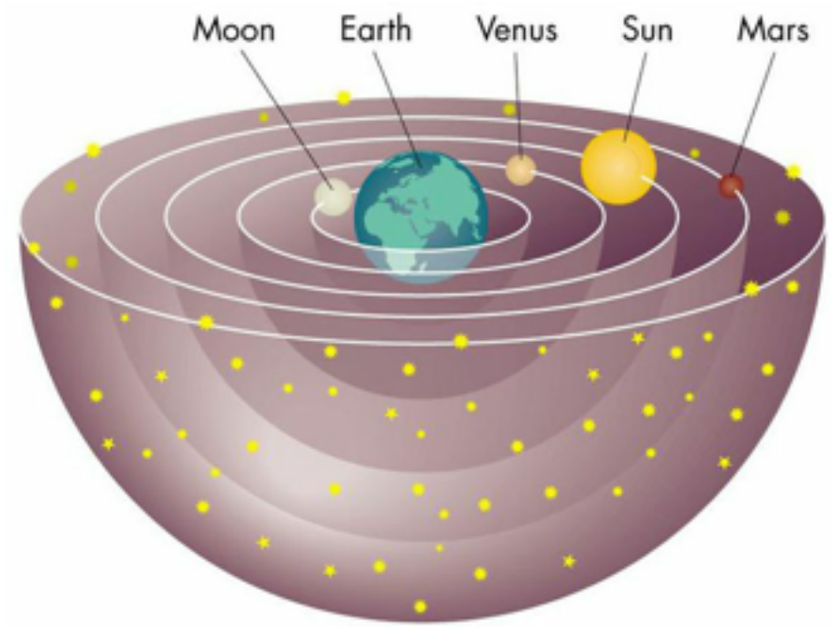
Can you explain this behavior?

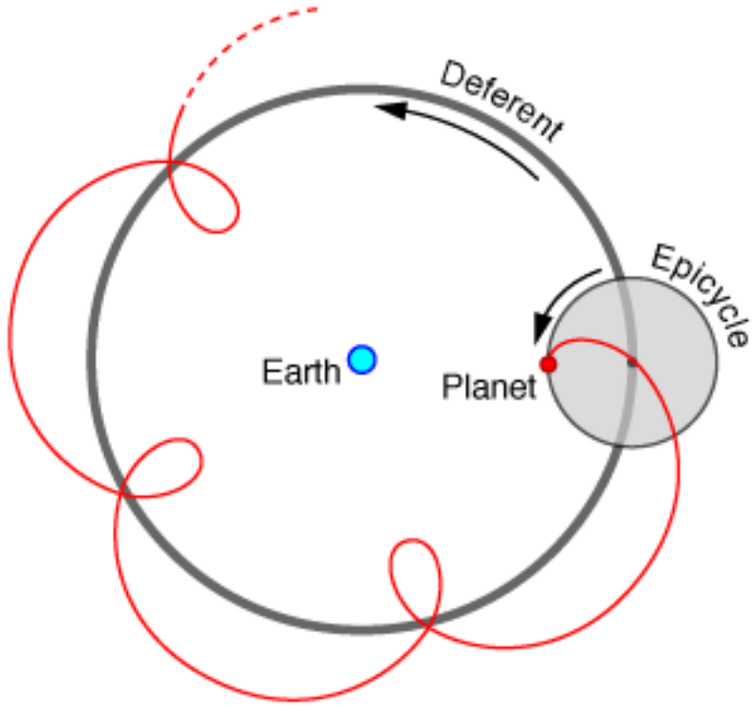
eh?

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!





```cpp
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

```cpp
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

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 not supposed to know when programming in a high level language. 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.

```cpp
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

```
$ c++ -O0 foo.cpp && ./a.out
42
```

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 not supposed to know when programming in a high level language. 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.

```
$ c++ -O2 foo.cpp && ./a.out
1462303832
```

```
#include <iostream>

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

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

```
#include <iostream>

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

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

```
4
```

```
#include <iostream>

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

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

```
4
4
```

```
#include <iostream>

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

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

```
4
4
4
```

```
#include <iostream>

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

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

```
4
4
4
```

```
#include <iostream>

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

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

```
#include <iostream>

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

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

```
4
```

```
#include <iostream>

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

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

```
4
4
```

```
#include <iostream>

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

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

```
4
4
4
```

```
#include <iostream>

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

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

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

```
4
4
4
```

They are all morons!

```
#include <iostream>

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

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

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

```
4
4
4
```

They are all morons!

```
#include <iostream>

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

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

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

Do you really have a deep understanding of when side-effects take place in C++? Do you know the rules of **sequencing**?

```
4
4
4
```

They are all morons!

ehh...

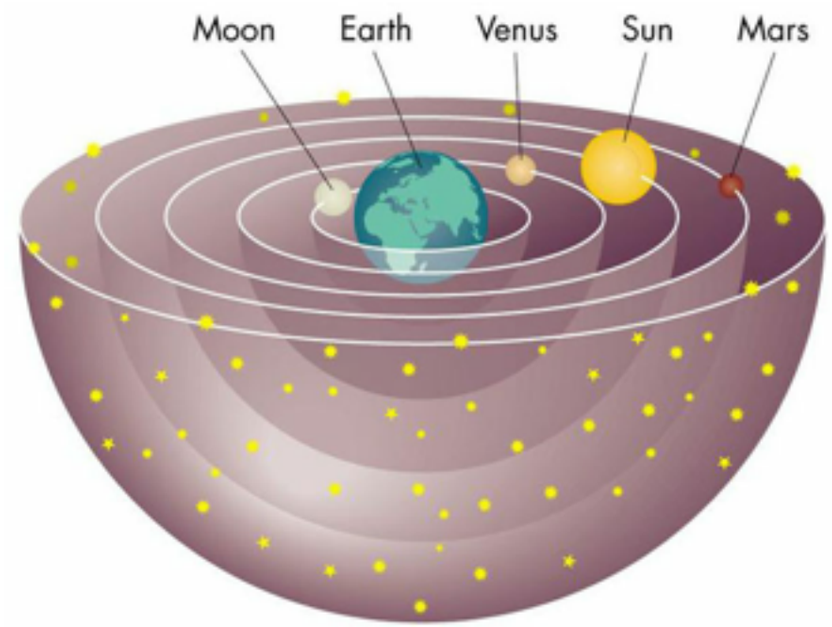
```
#include <iostream>

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

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

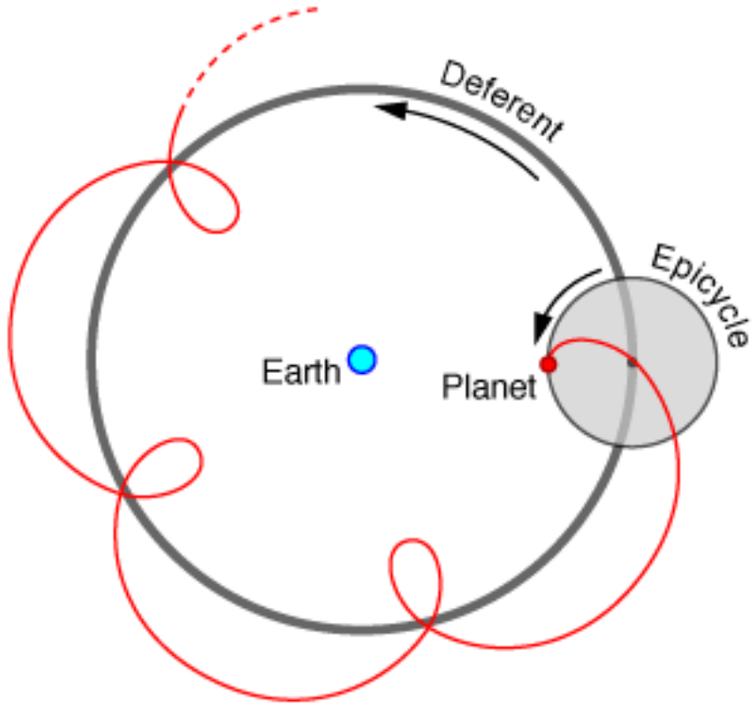
Believe it or not, I have met several programmers who thought this snippet would print 3,3,3.

Do you really have a deep understanding of when side-effects take place in C++? Do you know the rules of **sequencing**?

```
4
4
4
```

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





Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

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

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

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

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

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

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

6) `int a=41; printf("%d\n", (a++ < 42) ? a : a);` `// 42`

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

6) `int a=41; printf("%d\n", (a++ < 42) ? a : a);` `// 42`

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2) `int a=41; (a++ < 42) & printf("%d\n", a);` `// undefined`

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

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

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

6) `int a=41; printf("%d\n", (a++ < 42) ? a : a);` `// 42`

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

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

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

Exercise

Which of these snippets prints 42?
(hint: `printf()` returns the number of characters printed)

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

2)  int a=41; (a++ < 42) & printf("%d\n", a);         // undefined

3)  int a=41; (a++ < 42) && printf("%d\n", a);        // 42

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

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

6)  int a=41; printf("%d\n", (a++ < 42) ? a : a);     // 42

7)  int a=41; a = ++a; printf("%d\n", a);             // undefined

8)  int a=41; a = printf("%d\n", ++a);                // 42

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

deep_thought.cpp

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

deep_thought.cpp

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

deep_thought.cpp

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

This looks like quite innocent code?

But what if someone, years later, write some code that call this function with a very high seed?

deep_thought.cpp

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

This looks like quite innocent code?

But what if someone, years later, write some code that call this function with a very high seed?

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
... and the anwser is:
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
... and the anwser is:
3.14169265358979 32
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
... and the anwser is:
3.14169265358979 32
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
... and the anwser is:
3.14169265358979 32
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

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

deep_thought.cpp

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

```
$ c++ main.cpp deep_thought.cpp
$ ./a.out
... and the anwser is:
3.14169265358979 32
```

main.cpp

```
#include <iostream>
#include <limits>

int the_answer(int);

int main()
{
    printf("The answer is:\n");
    int a = the_answer(2147483647);
    printf("%d\n", a);
}
```

Inconceivable!

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

Integer overflow is undefined behavior. If you want to prevent this to happen you must write the logic yourself. In C++ you seldom get code you have not asked for.

Exercise

This program is UB because `b` is used without being initialized. But in practice, what do you think might happen when this function is called?

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

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

Exercise

This program is UB because `b` is used without being initialized. But in practice, what do you think might happen when this function is called?

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

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

Exercise

This program is UB because `b` is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

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

Exercise

This program is UB because b is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

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

Exercise

This program is UB because `b` is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

```
$ g++ foo.cpp bar.cpp main.cpp
```

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

Exercise

This program is UB because `b` is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

```
$ g++ foo.cpp bar.cpp main.cpp
$ ./a.out
```

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

Exercise

This program is UB because b is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

```
$ g++ foo.cpp bar.cpp main.cpp
$ ./a.out
true
```

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

Exercise

This program is UB because b is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

```
$ g++ foo.cpp bar.cpp main.cpp
$ ./a.out
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 program is UB because b is used without being initialized. But in practice, what do you think might happen when this function is called?

bar.cpp

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

foo.cpp

```
#include <iostream>

void foo(void)
{
    bool b;
    if (b)
        std::cout << "true" << std::endl;
    if (!b)
        std::cout << "false" << std::endl;
}
```

main.cpp

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

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

This is what I get on my computer (Mac OS 10.8.3, gcc since 4.7)

```
$ g++ foo.cpp bar.cpp main.cpp
$ ./a.out
true
false
$
```

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

A real story of “anything can happen”

```
bool b;
if (b)
  printf("b is true\n");
if (!b)
  printf("b is false\n");
```

A real story of "anything can happen"

```
bool b;
if (b)
  printf("b is true\n");
if (!b)
  printf("b is false\n");
```

```
    ; the following code assumes that $b is either 0 or 1

    load_reg_a    $b
    compare_reg_a 0
    jump_equal    label1
    call_proc     print_b_is_true
label1:
    load_reg_a    $b
    xor_reg_a     1
    compare_reg_a 0
    jump_equal    label2
    call_proc     print_b_is_false
label2:
```

A real story of "anything can happen"

```
bool b;
if (b)
  printf("b is true\n");
if (!b)
  printf("b is false\n");
```

```
    ; the following code assumes that $b is either 0 or 1

    load_reg_a    $b
    compare_reg_a 0
    jump_equal    label1
    call_proc     print_b_is_true
label1:
    load_reg_a    $b
    xor_reg_a     1
    compare_reg_a 0
    jump_equal    label2
    call_proc     print_b_is_false
label2:
```

this is approximately the code generated by one actual version of gcc, try to imagine what will happen if the garbage value of b is 2

A real story of "anything can happen"

```
bool b;
if (b)
  printf("b is true\n");
if (!b)
  printf("b is false\n");
```

```
    ; the following code assumes that $b is either 0 or 1

    load_reg_a    $b
    compare_reg_a 0
    jump_equal    label1
    call_proc     print_b_is_true
label1:
    load_reg_a    $b
    xor_reg_a     1
    compare_reg_a 0
    jump_equal    label2
    call_proc     print_b_is_false
label2:
```

this is approximately the code generated by one actual version of gcc, try to imagine what will happen if the garbage value of b is 2

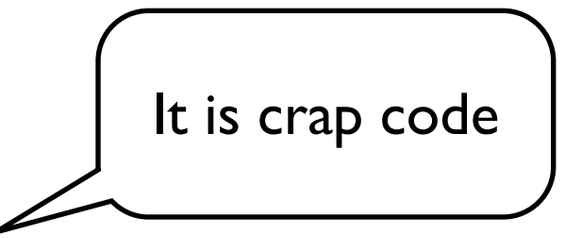
```
true
false
```

So what's wrong with this code?

```
#include <iostream>

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

So what's wrong with this code?

```
#include <iostream>

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



So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code

So what's wrong with this code?

```
#include <iostream>

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

It is 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?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code?

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code?

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code?

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

So what's wrong with this code?

```
#include <iostream>

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

It is crap code

The standard says that this is invalid code?

Update a variable multiple times between two semicolons

According to §1.9.15 in the standard: On line 7, the evaluations of the operands are unsequenced. The side effects on the scalar object `i` is unsequenced relative to the other side effect on `i`, as well as the value computation of `i`, so the behavior is undefined.

In C++ (and C), unlike most other languages, within a full expression the order in which subexpressions are evaluated is **mostly unspecified**. Therefore the expression

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

does not make sense and yields **undefined behavior**.
When we have UB then **anything can happen**.

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);
....
```

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 conceptual 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