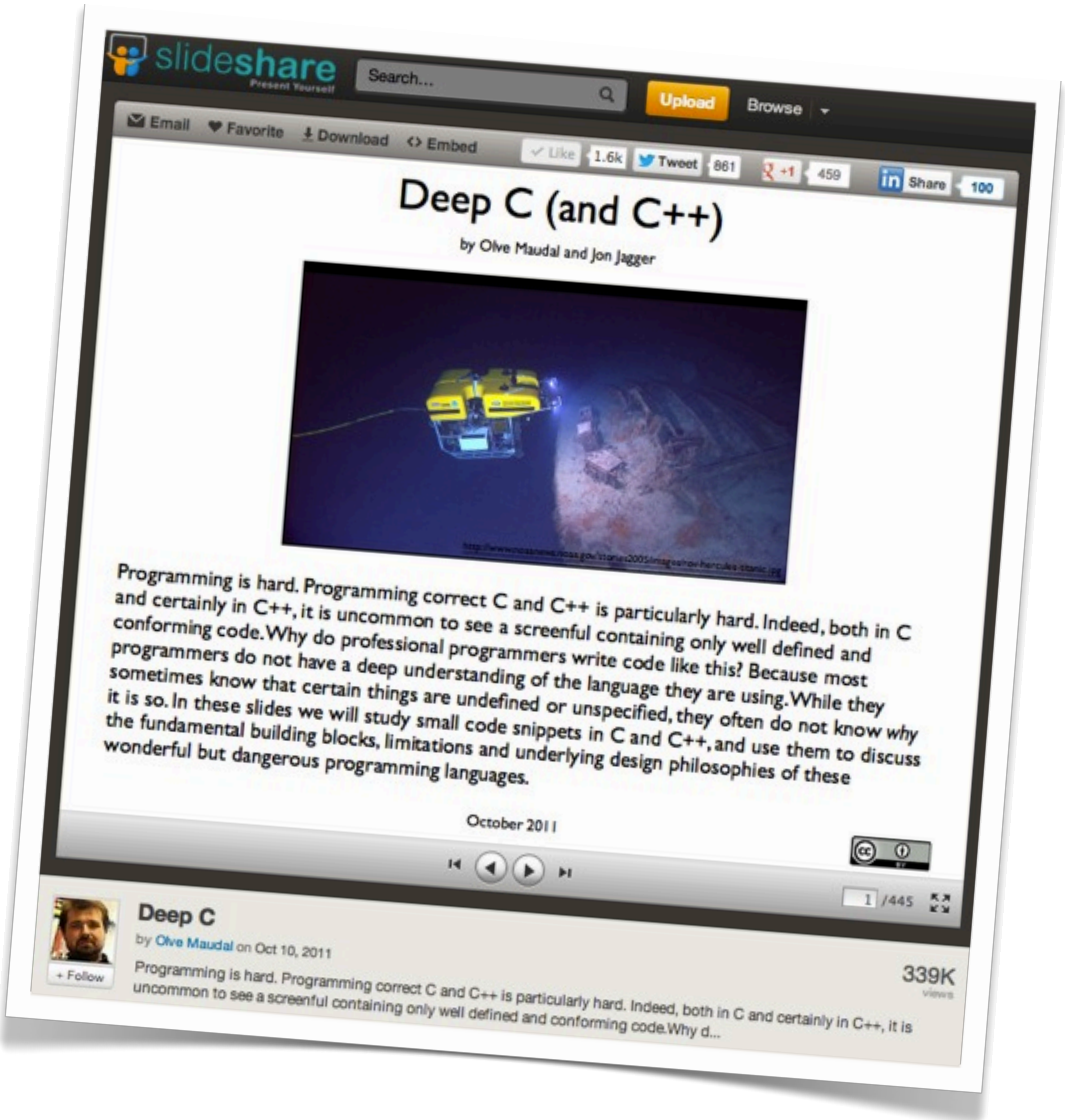
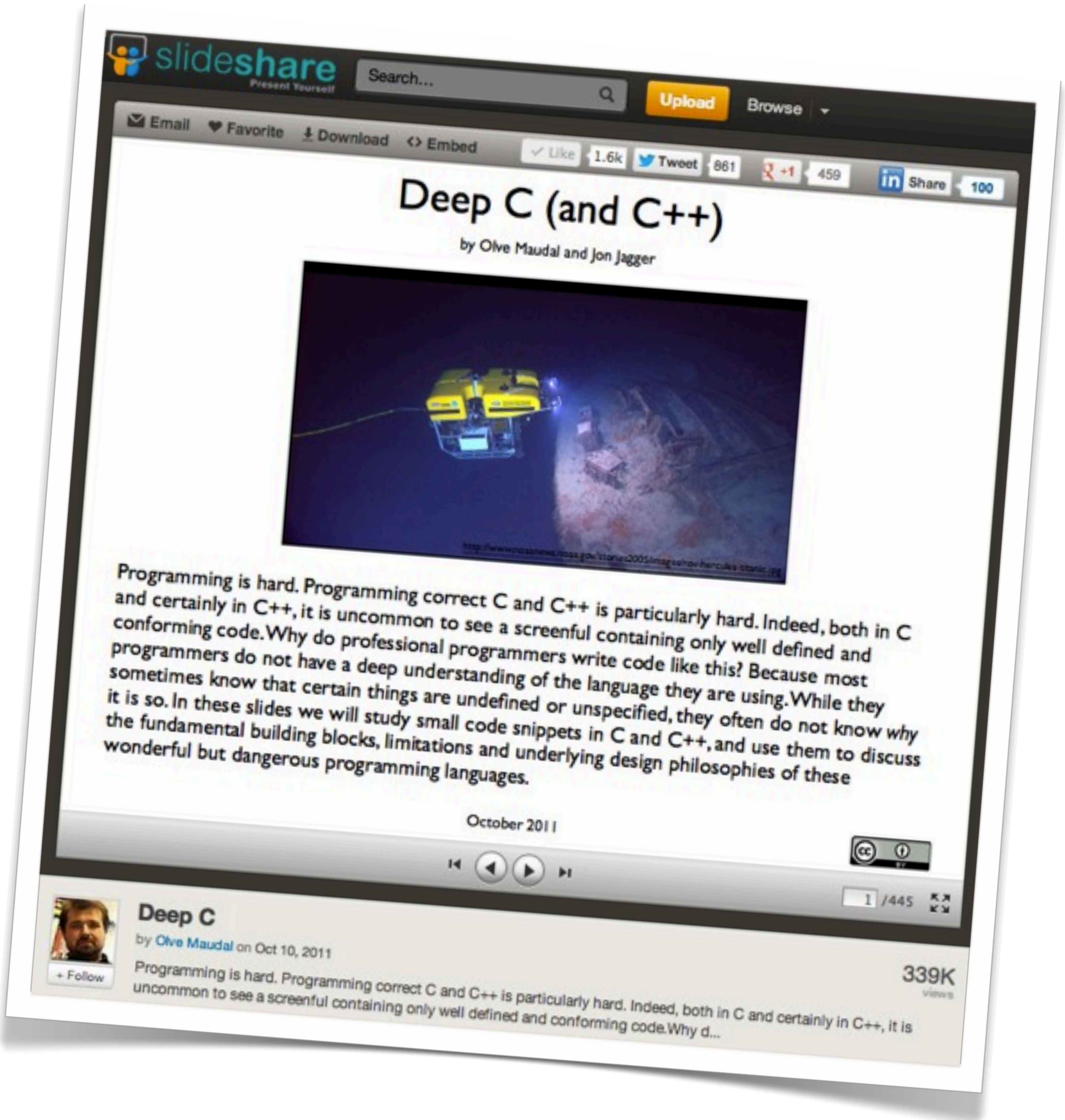
Deep C++

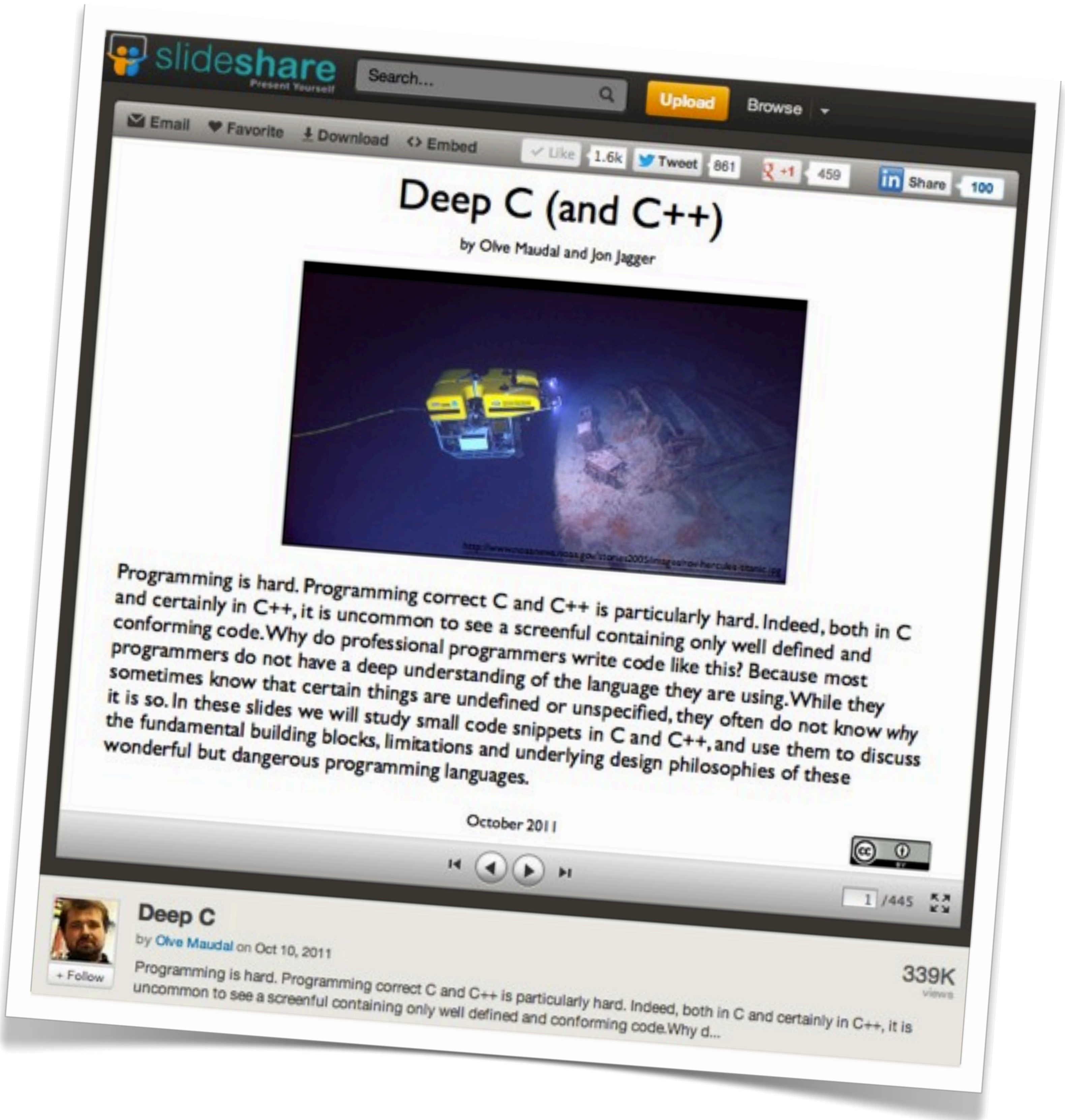
by Olve Maudal

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

A 45 minute session at TNG Big Techday 5, Friday, June 15th, 2012

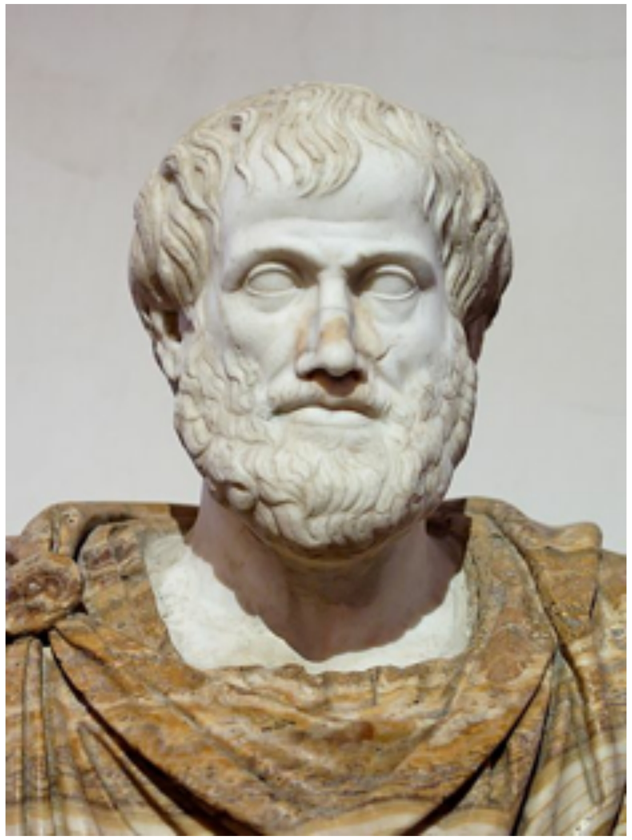

slideshare
Present Yourself
Search...
Upload
Browse
Email
Favorite
Download
Embed
Like
1.6k
Tweet
861
+1
459
Share
100
Deep C (and C++)
by Olve Maudal and Jon Jagger
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, it is uncommon to see 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 these slides we will study small code snippets in C and C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of these wonderful but dangerous programming languages.
October 2011
1 /445
Deep C
by Olve Maudal on Oct 10, 2011
+ Follow
339K
views
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, it is uncommon to see a screenful containing only well defined and conforming code. Why d...

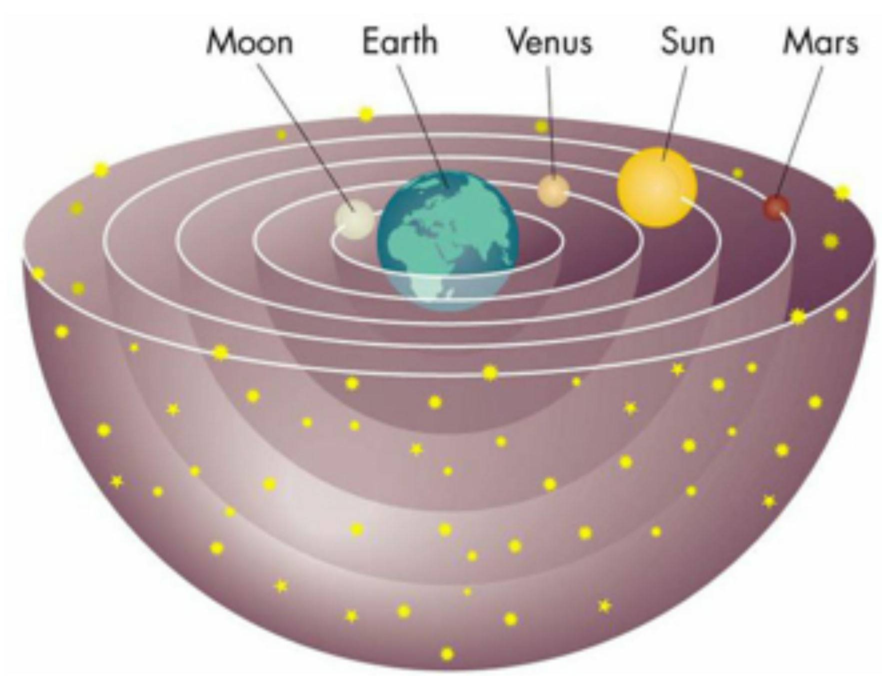

slideshare
Present Yourself
Search...
Upload
Browse
Email
Favorite
Download
Embed
Like
1.6k
Tweet
861
+1
459
Share
100
Deep C (and C++)
by Olve Maudal and Jon Jagger
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, it is uncommon to see 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 these slides we will study small code snippets in C and C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of these wonderful but dangerous programming languages.
October 2011
1 /445
Deep C
by Olve Maudal on Oct 10, 2011
+ Follow
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, it is uncommon to see a screenful containing only well defined and conforming code. Why d...
339K
views


slideshare
Present Yourself
Search...
Upload
Browse
Email
Favorite
Download
Embed
Like 1.6k
Tweet 861
+1 459
in Share 100
Deep C (and C++)
by Olve Maudal and Jon Jagger
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, 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 these slides we will study small code snippets in C and C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of these wonderful but dangerous programming languages.
October 2011
1 /445
Deep C
by Olve Maudal on Oct 10, 2011
+ Follow
Programming is hard. Programming correct C and C++ is particularly hard. Indeed, both in C and certainly in C++, it is uncommon to see a screenful containing only well defined and conforming code. Why d...
339K
views


JAWS

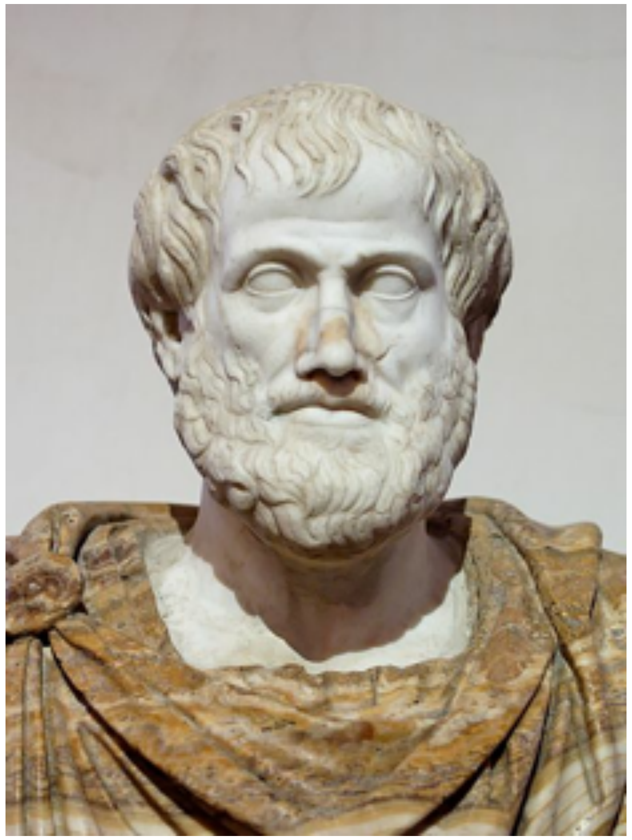
Aristotle (384 BC – 322 BC)



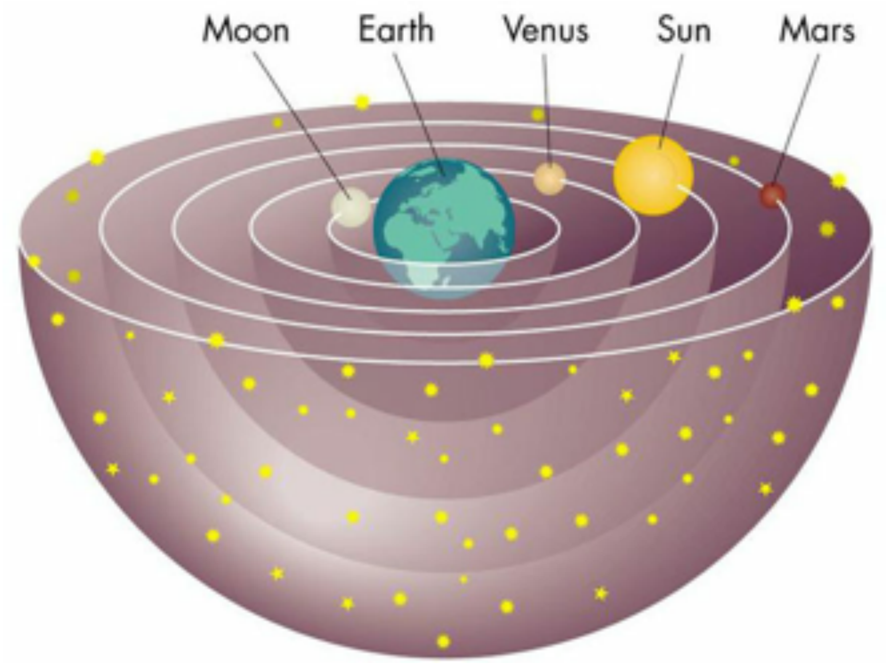


Aristotles Universe

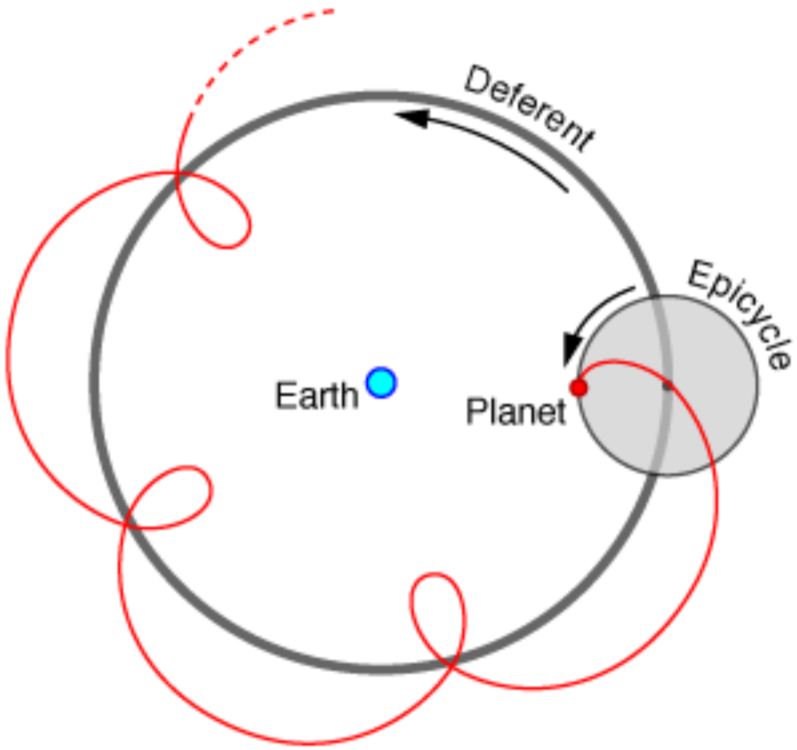
Aristotle (384 BC – 322 BC)



Ptolemy (90 AD – 168 AD)

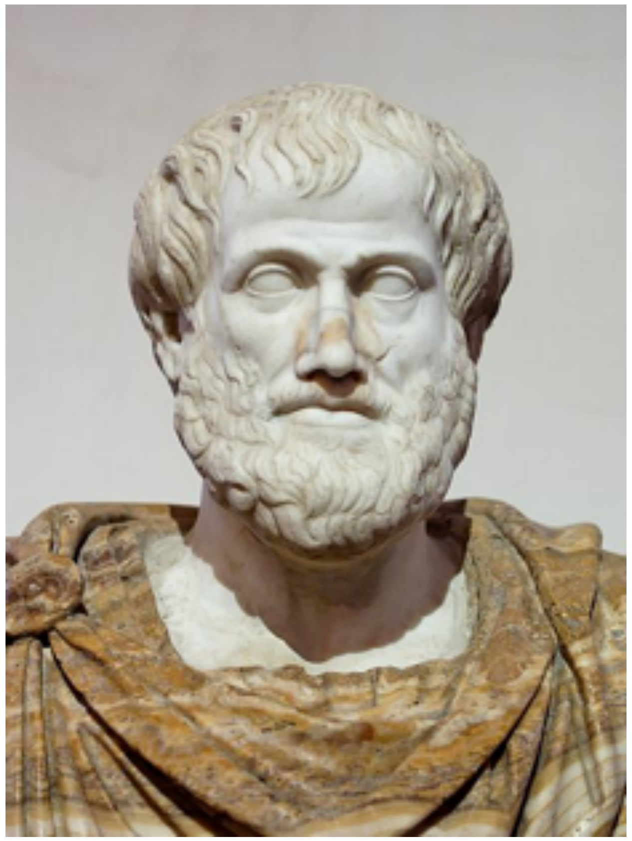


Aristotles Universe



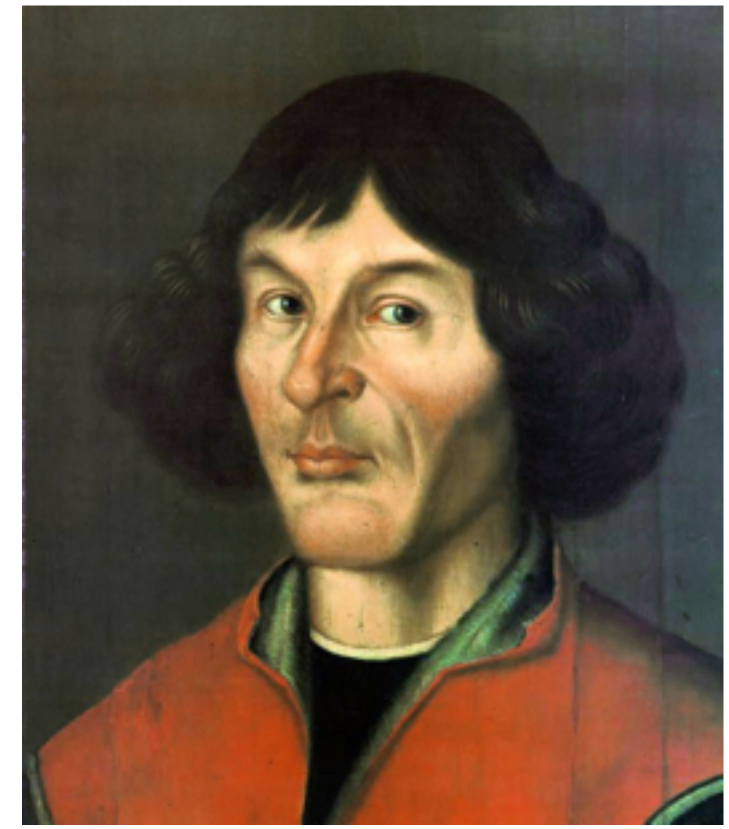
Ptolemy Universe

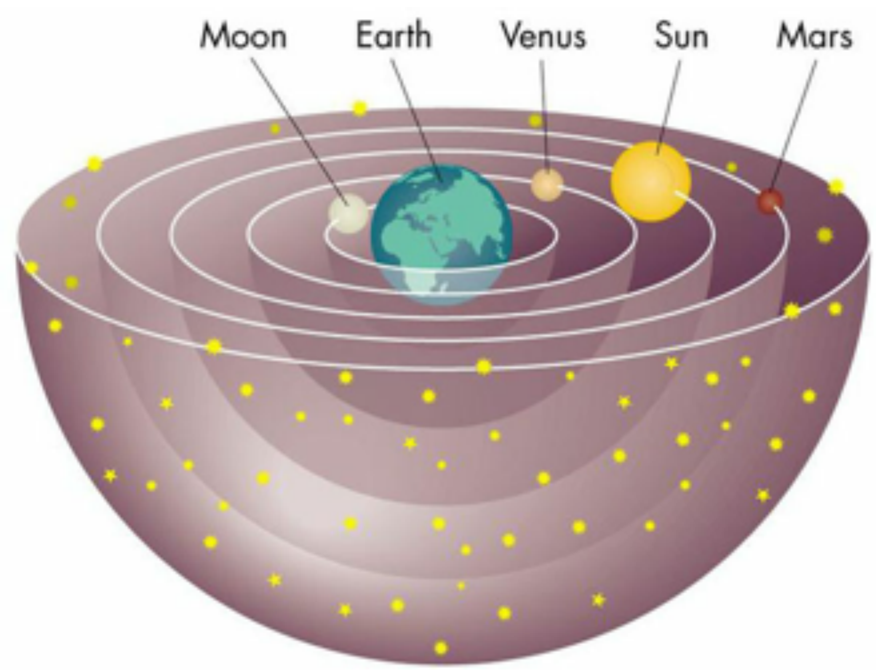
Aristotle (384 BC – 322 BC)



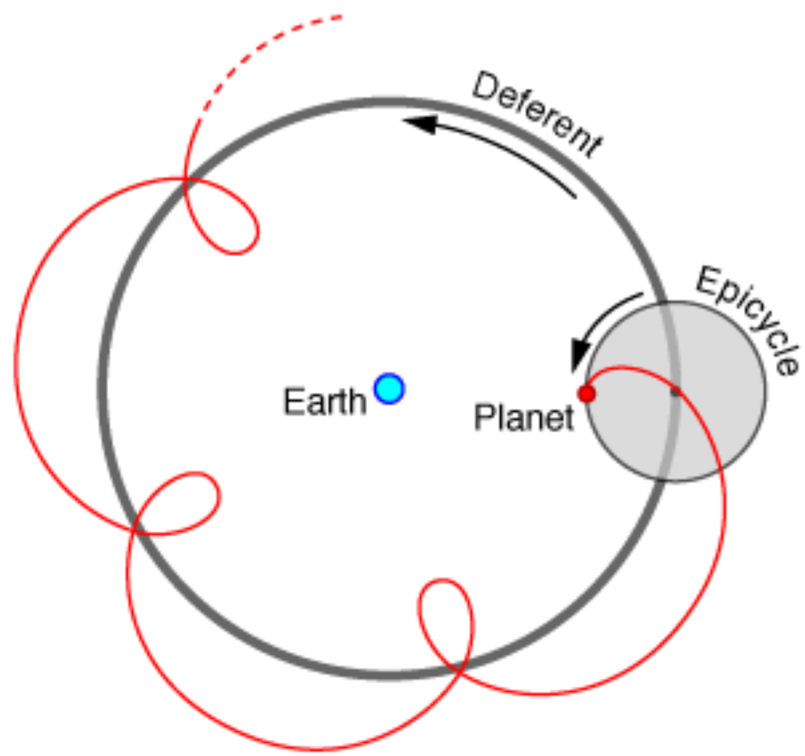
Ptolemy (90 AD – 168 AD)

Copernicus (1473 – 1543)

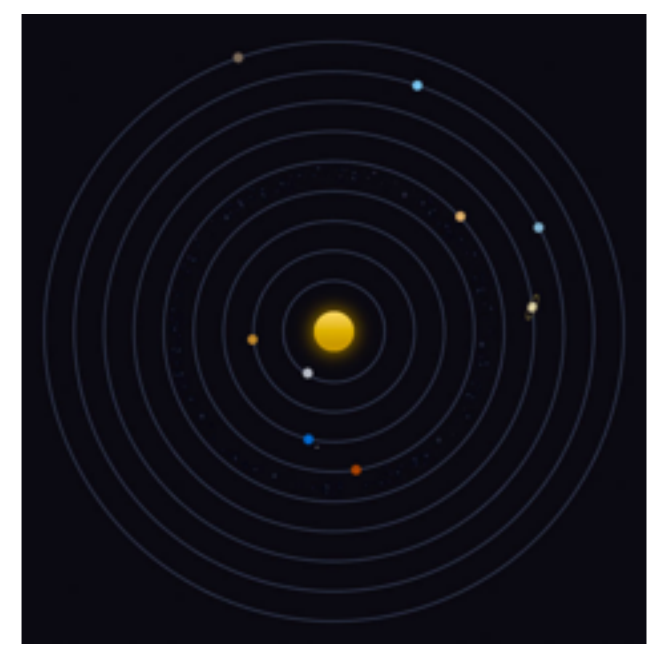




Aristotles Universe

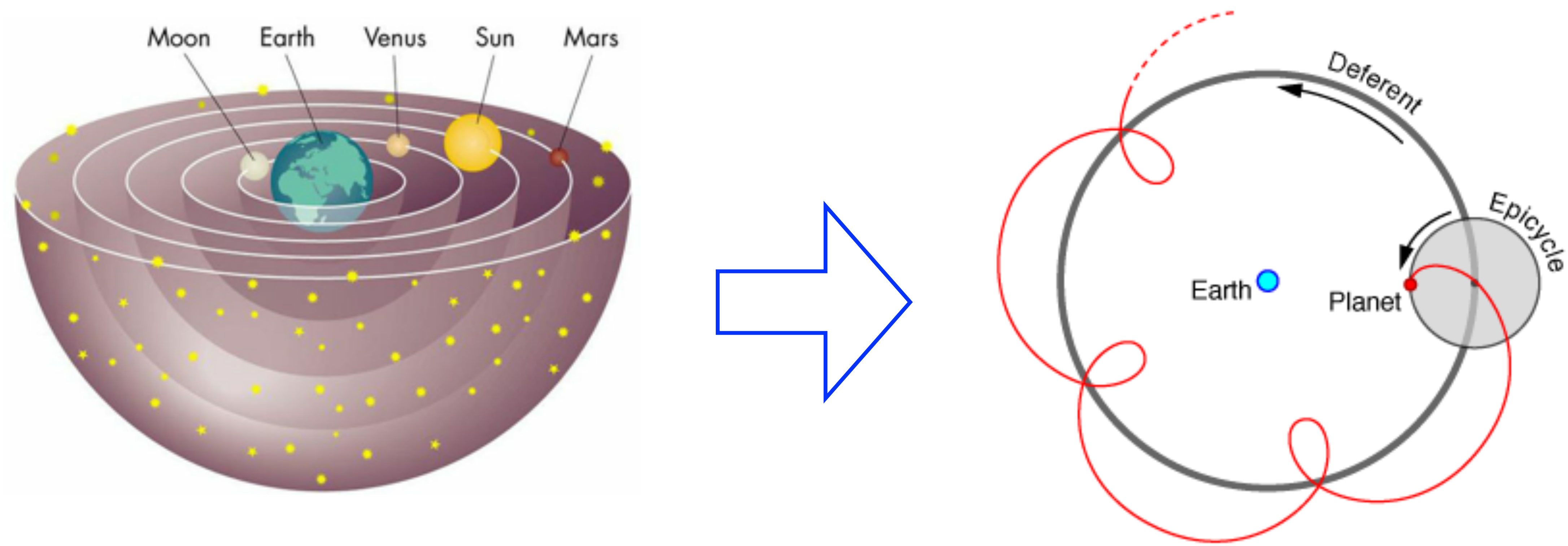


Ptolemy Universe



The Solar System

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



```
#include <iostream>

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

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

```
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```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
```

```
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

```
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```cpp
#include <iostream>

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

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

```
$ c++ foo.cpp
$ ./a.out
4
5
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 you still need to know that it is so. And it is very useful to know *why* it is so

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

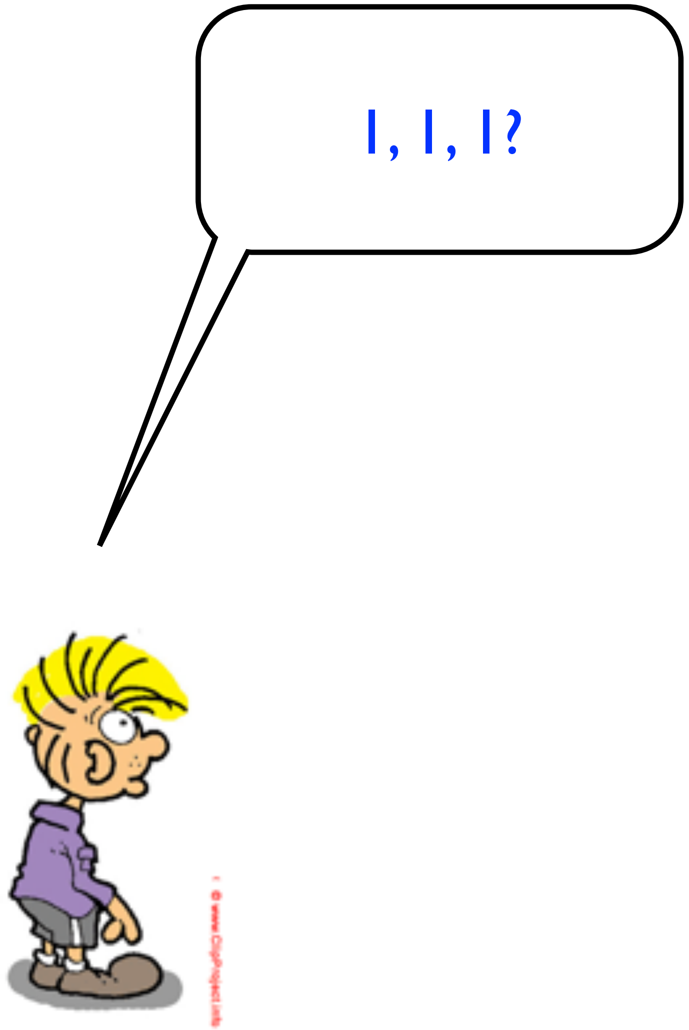
```
#include <iostream>

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

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

```
$ c++ foo.cpp
$ ./a.out
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 you still need to know that it is so. And it is very useful to know *why* it is so

```
#include <iostream>

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

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




```cpp
#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();
}
```

No, variables with automatic storage duration is not initialized implicitly

1, 1, 1?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

1, 1, 1?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

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

1, 1, 1?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

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

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

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

1, 1, 1?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let’s try it on my machine

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

I, I, I?

Garbage, garbage, garbage

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

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

I, I, I?

Garbage, garbage, garbage

Ehh...

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

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

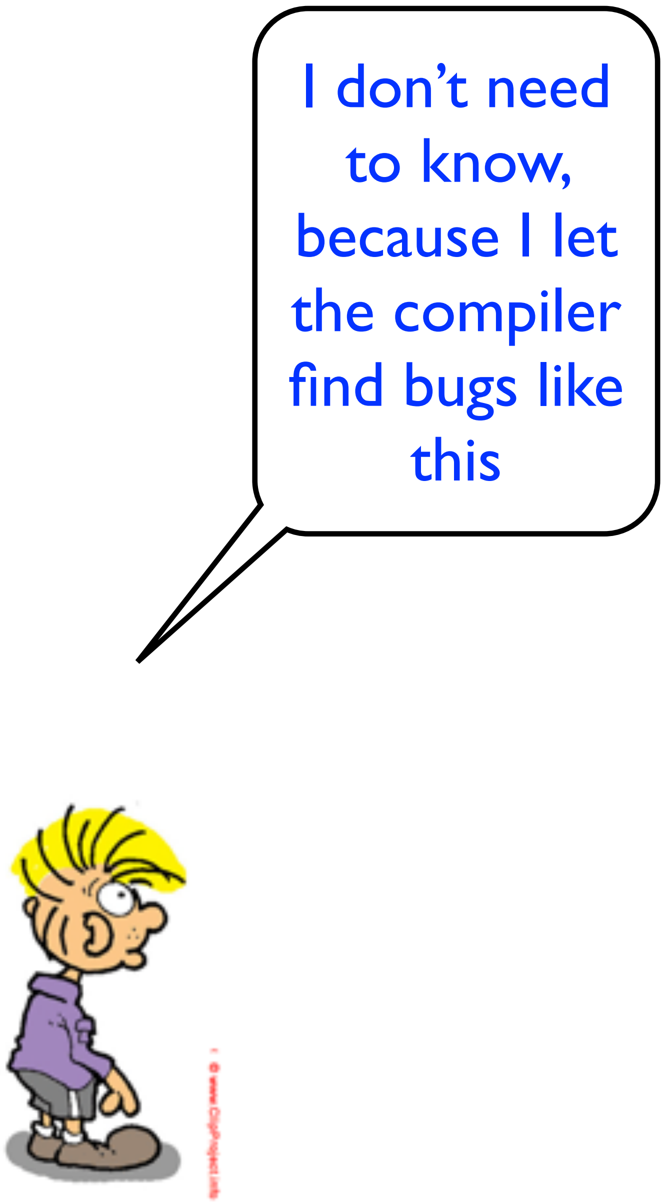

```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

Yes, in theory that is correct. Let's try it on my machine

any plausible explaination for this behaviour?

```
$ 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();
}
```

I don’t need to know, because I let the compiler find bugs like this

```
#include <iostream>

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

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

OK, let’s add some flags

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

```
#include <iostream>

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

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

OK, let's add some flags

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

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

```
#include <iostream>

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

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

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

OK, let's add some flags

I don’t need to know, because I let the compiler find bugs like this

```
#include <iostream>

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

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

OK, let’s add some flags

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

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

```
#include <iostream>

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

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

OK, let's add some flags

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

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

```
#include <iostream>

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

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

OK, let's add some flags

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

I don’t need to know, because I let the compiler find bugs like this

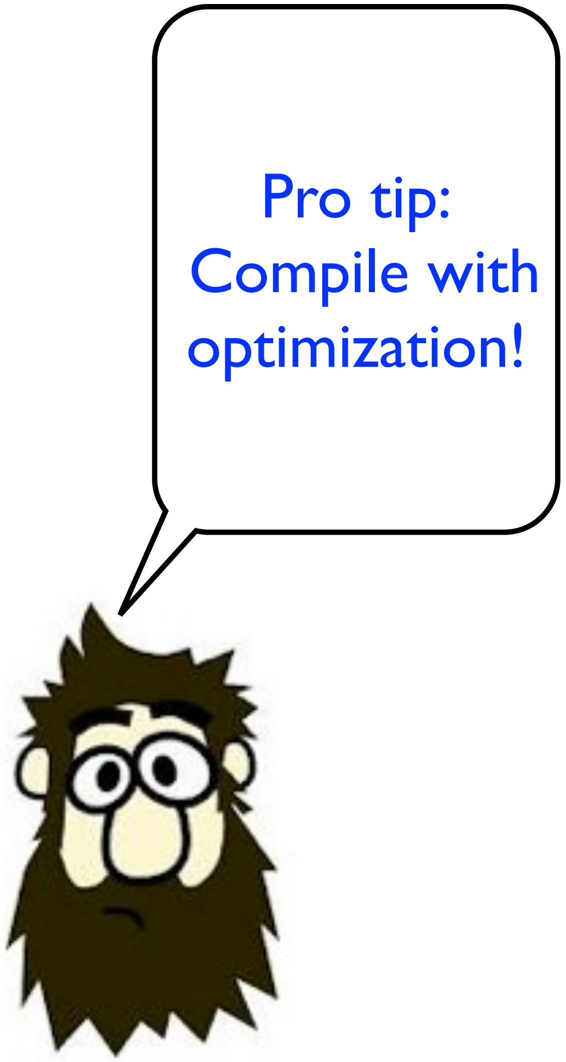
Lousy compiler!

```
#include <iostream>

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

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

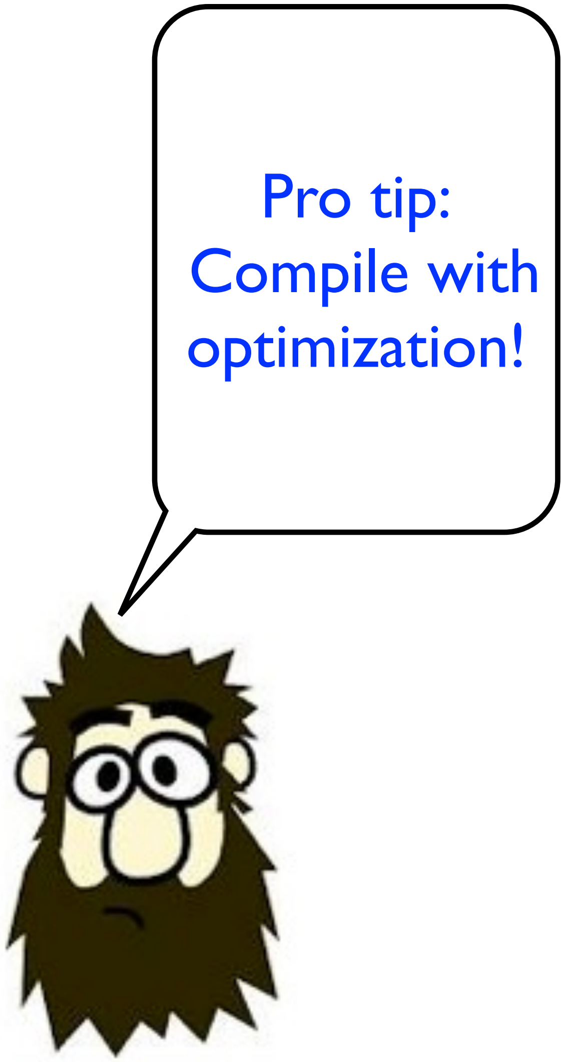
OK, let’s add some flags

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

```
#include <iostream>

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

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

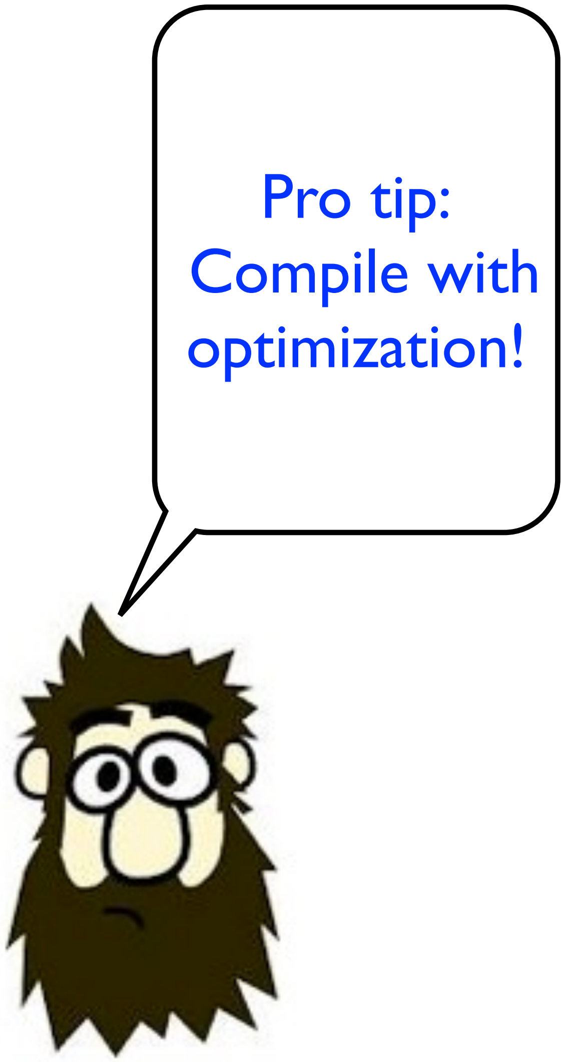



```cpp
#include <iostream>

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

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



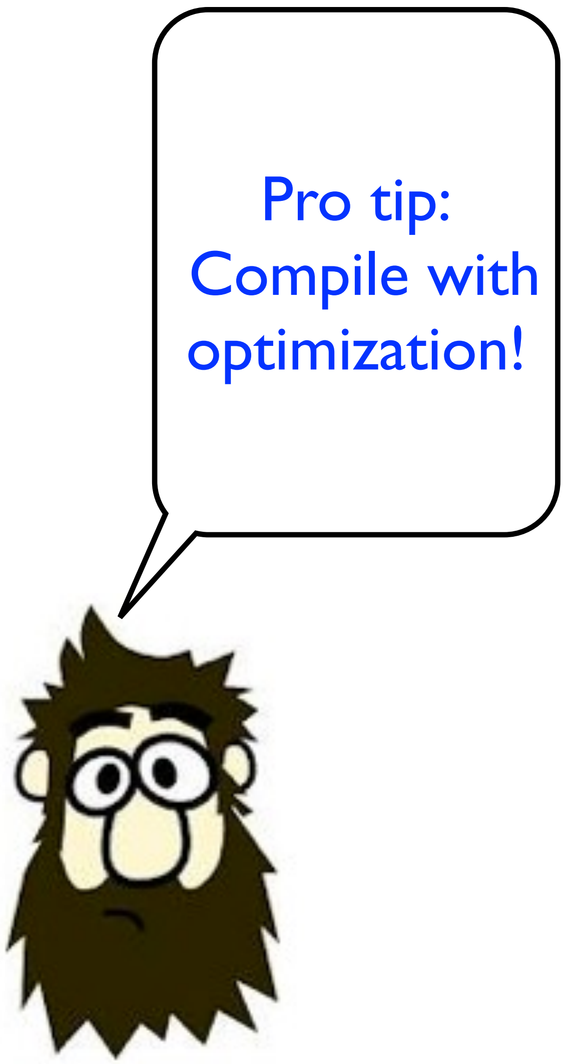

```
#include <iostream>

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

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

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




```
#include <iostream>

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

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

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




```
#include <iostream>

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

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

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

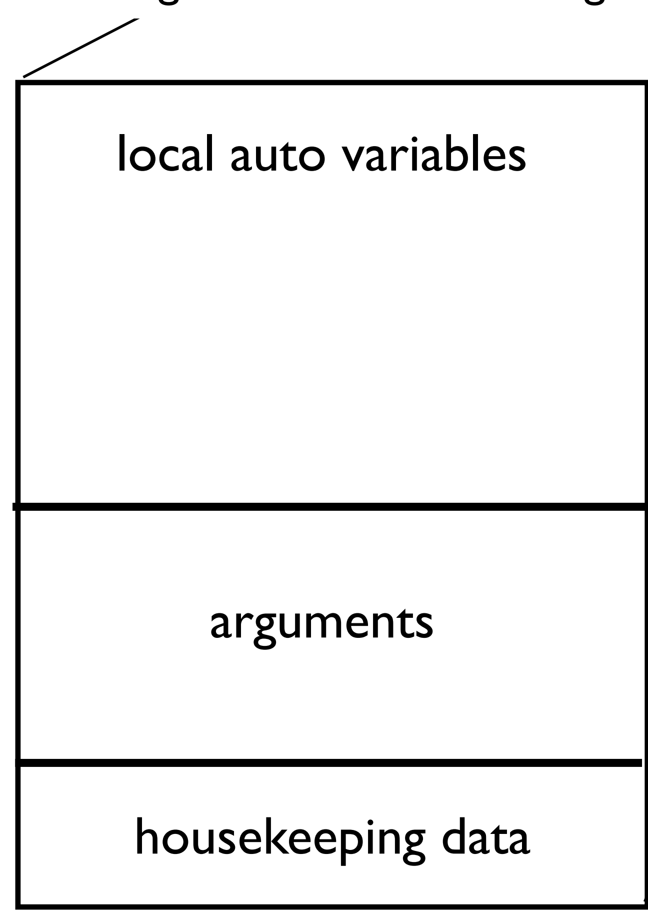
Memory Layout *

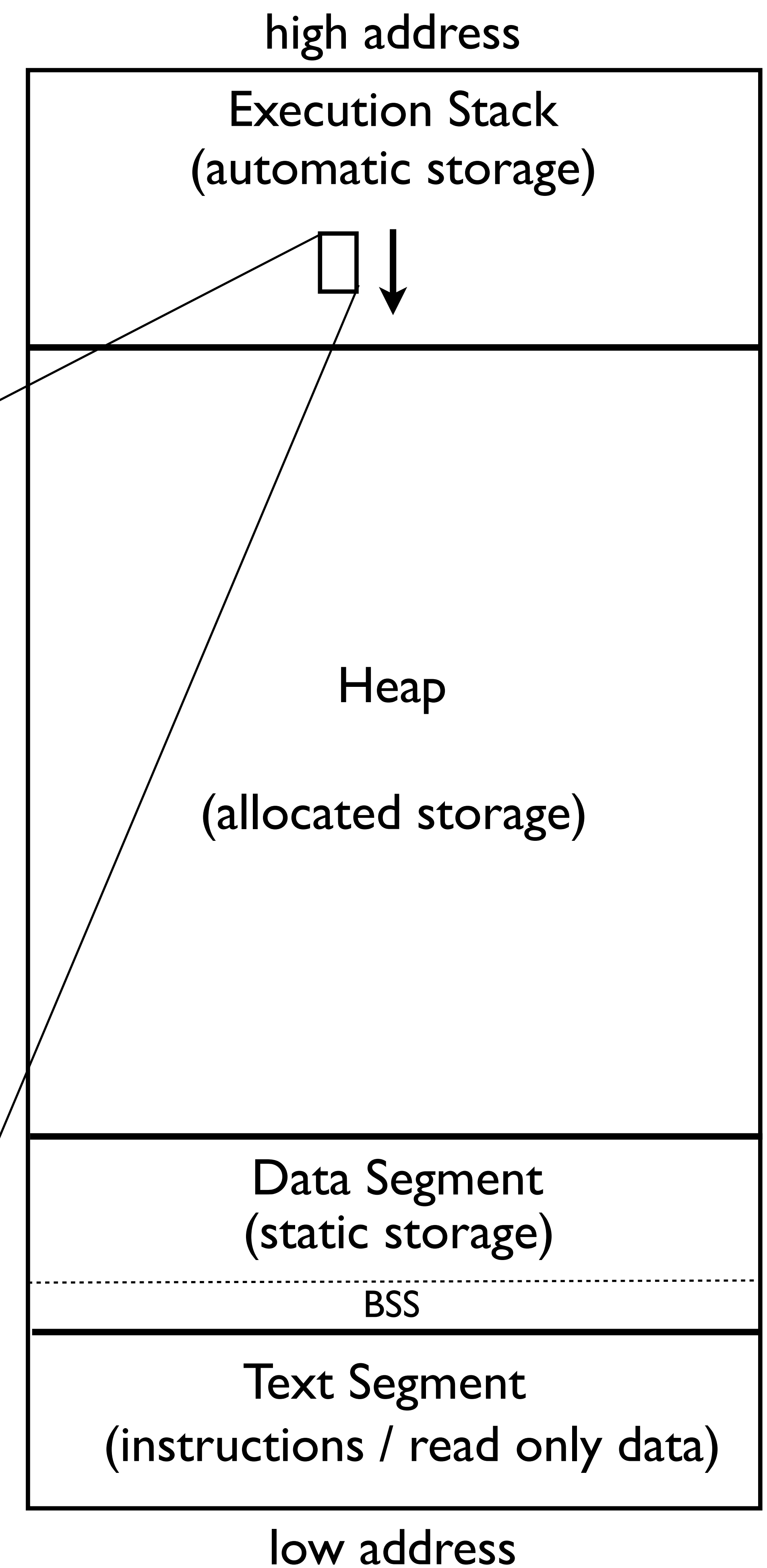
It is sometimes useful to assume that a C program uses a memory model where the instructions are stored in a **text segment**, and static variables are stored in a **data segment**. Automatic variables are allocated when needed together with housekeeping variables on an **execution stack** that is growing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

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

I am now going to show you something cool!

```cpp
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

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

Can you explain this behaviour?

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

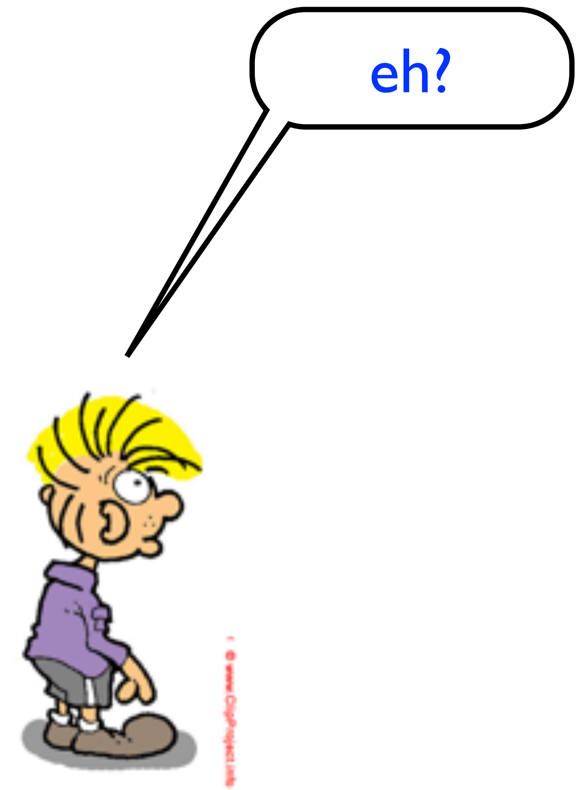
Can you explain this behaviour?

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

Can you explain this behaviour?



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

Can you explain this behaviour?

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 names a it will get the variable will get the same memory location. 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!

```cpp
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

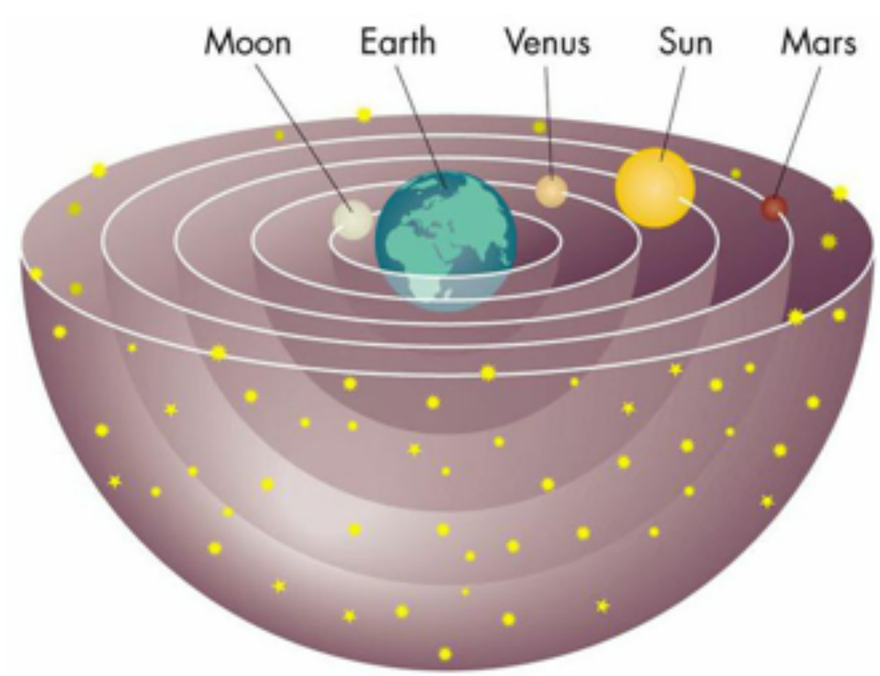
```
$ c++ foo.cpp && ./a.out
42
```

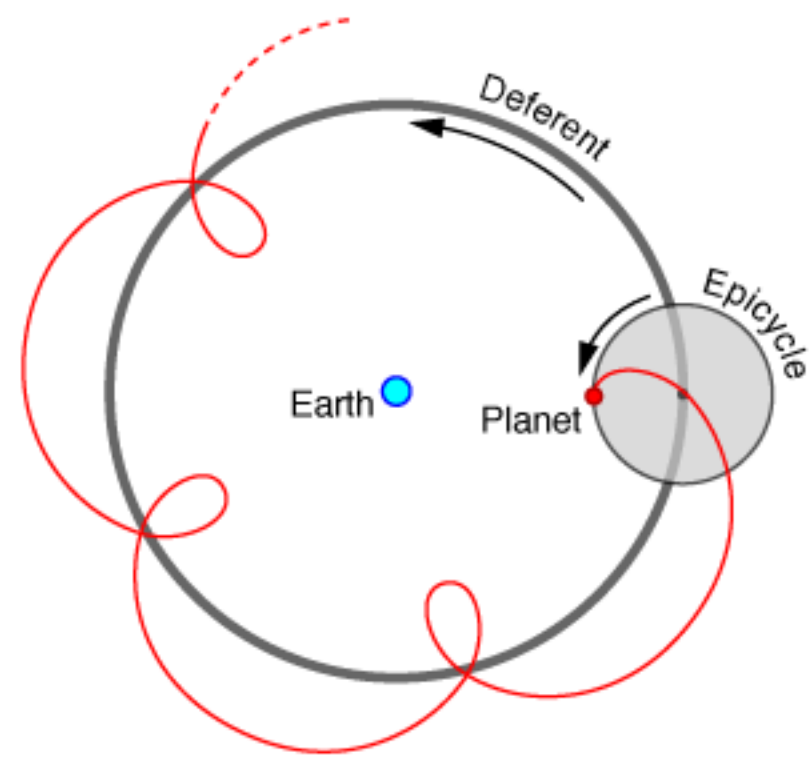
Can you explain this behaviour?

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 names a it will get the variable will get the same memory location. If you rename the variable in bar() to, say b, then I don't think you will get 42.

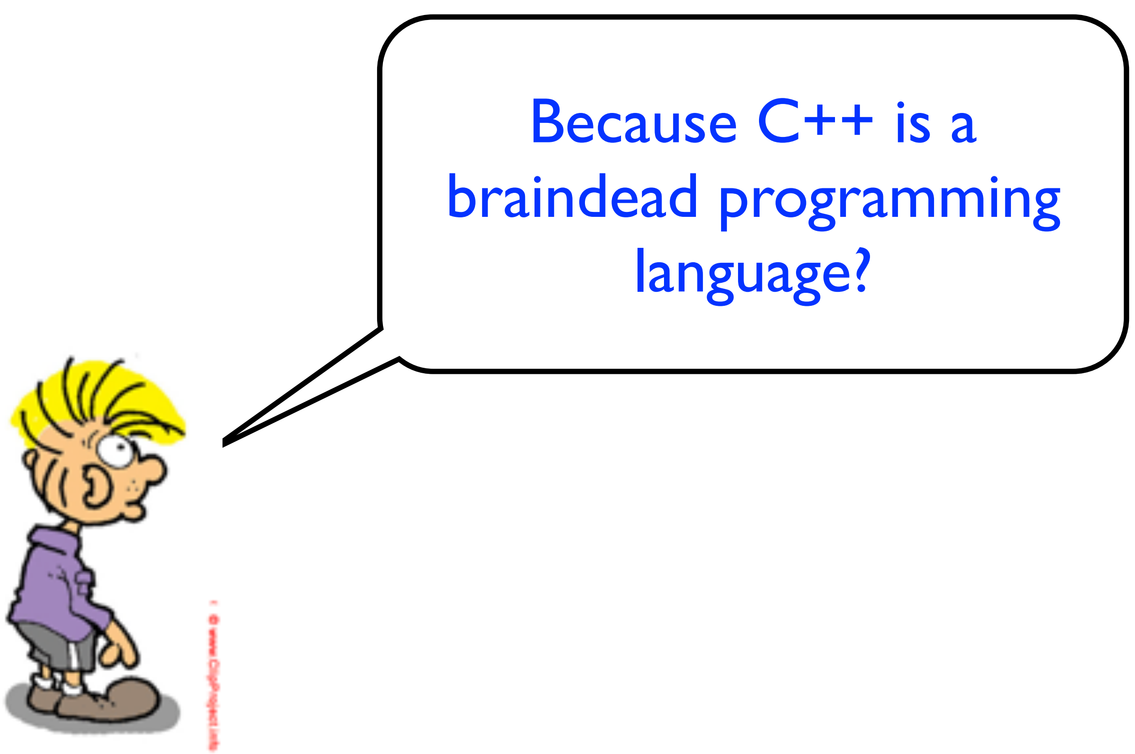
Yeah, sure...


Moon
Earth
Venus
Sun
Mars


Deferent
Epicycle
Earth
Planet

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?

Because C++ is a braindead programming language?

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

```cpp
#include <iostream>

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

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

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

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

```
#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
```

```
#include <iostream>

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

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

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

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

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

```cpp
#include <iostream>

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

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

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

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

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

```cpp
#include <iostream>

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

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

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

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

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

but you might also get

```
#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
```

```cpp
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```cpp
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```cpp
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

```
#include <iostream>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

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

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

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

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

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.

```
#include <iostream>

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

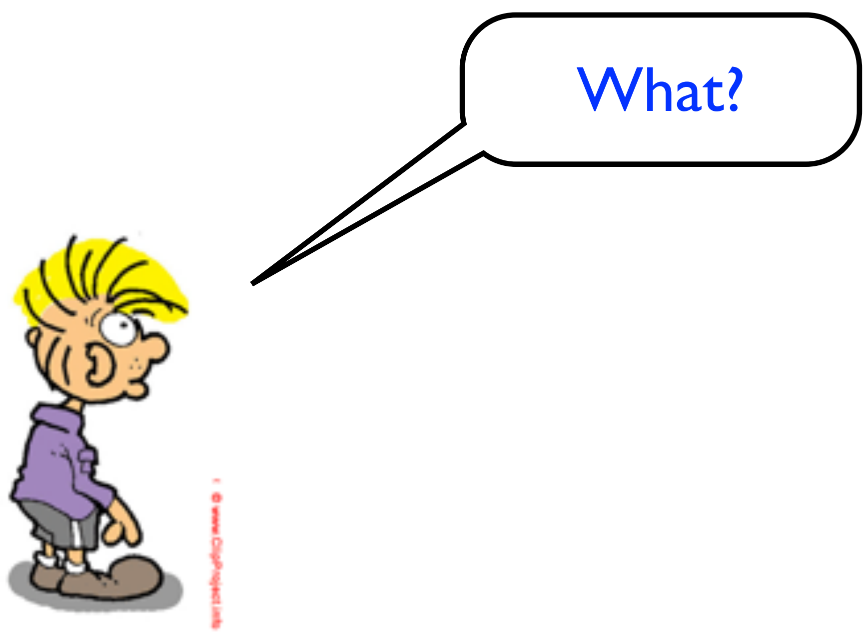
```
#include <iostream>

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

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

```cpp
#include <iostream>

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

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

```
#include <iostream>

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




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

```
#include <iostream>

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

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

What?

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

```
#include <iostream>

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

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

What?

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!

```
#include <iostream>

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

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

What?

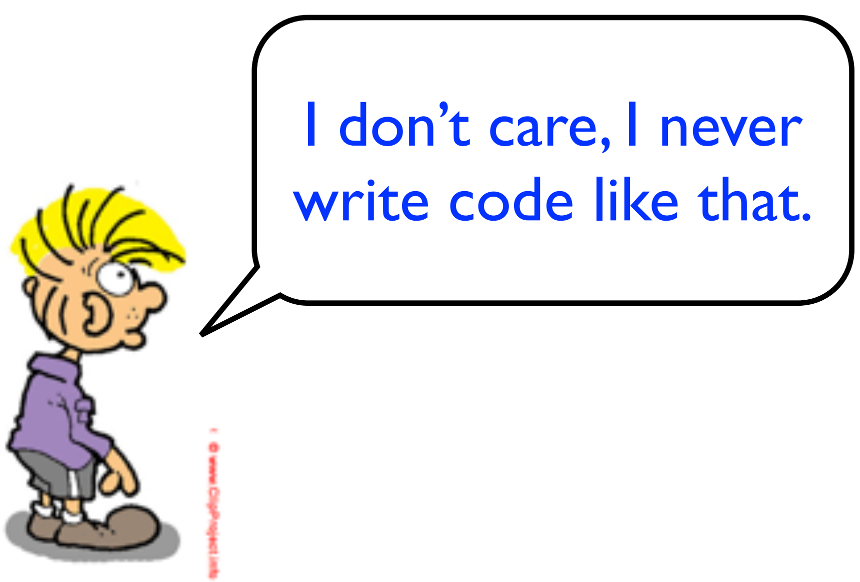
I agree this is crap code, but why is it wrong?

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

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

```
#include <iostream>

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

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



```
#include <iostream>

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

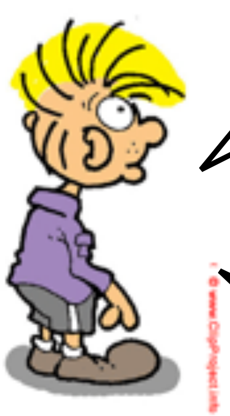
```
$ c++ foo.cpp && ./a.out
42
```

```
#include <iostream>

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

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

```
#include <iostream>

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

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



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

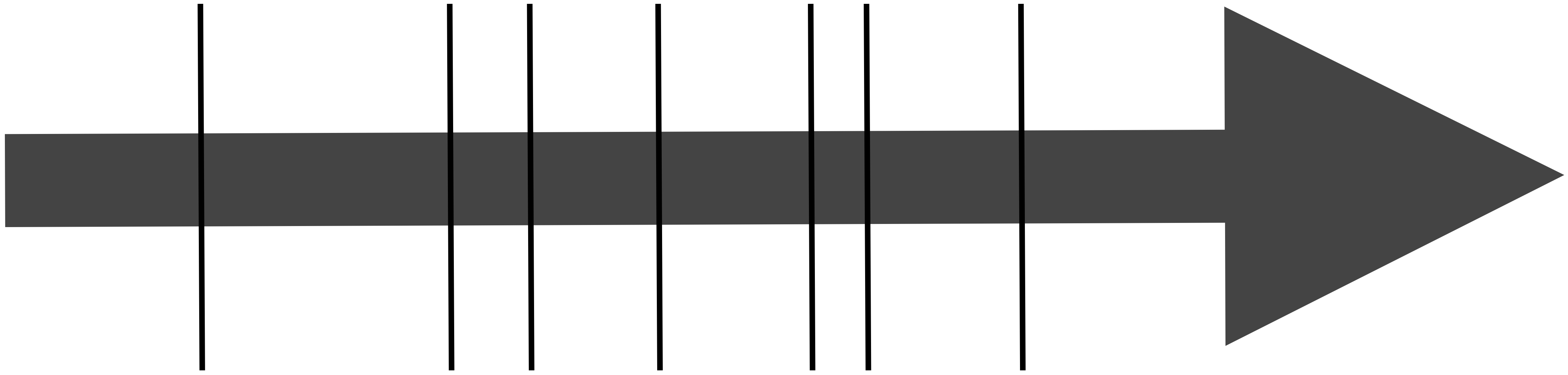
Good for you. But bugs like this can easily happen if you don’t understand the rules of sequencing. And very often, the compiler is not able to help you...

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

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

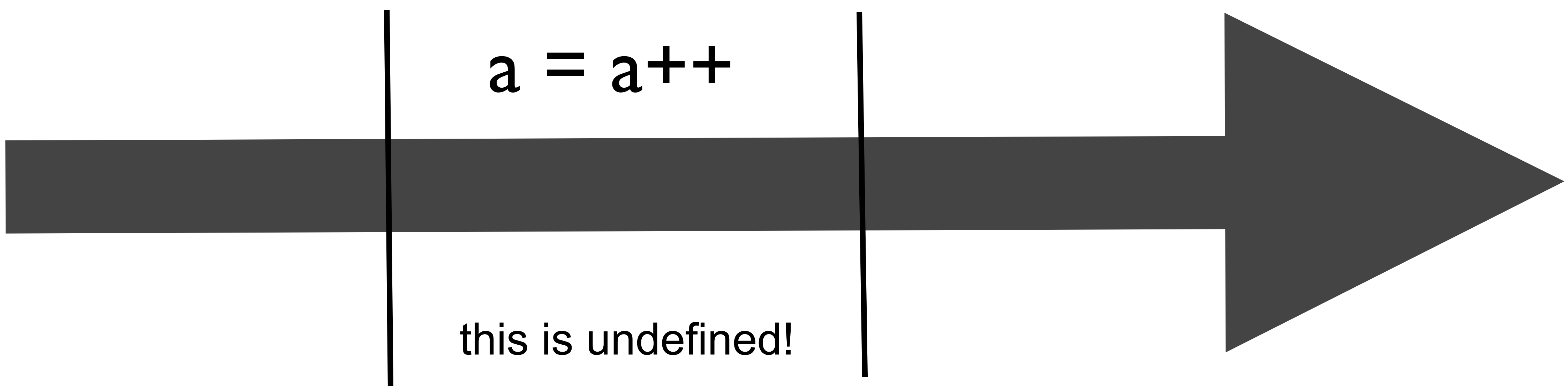
Sequence Points

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



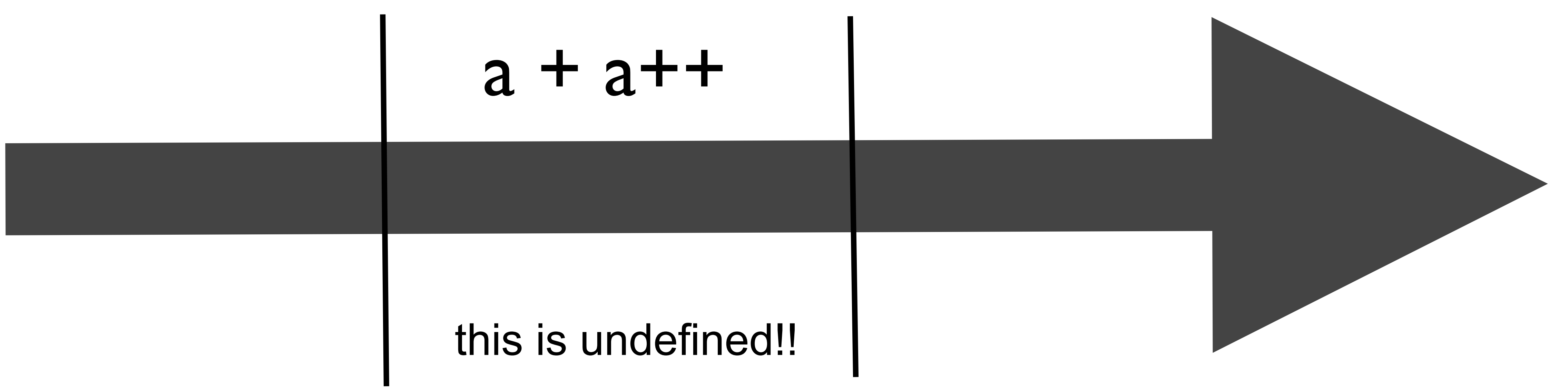
Sequence Points - Rule 1

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



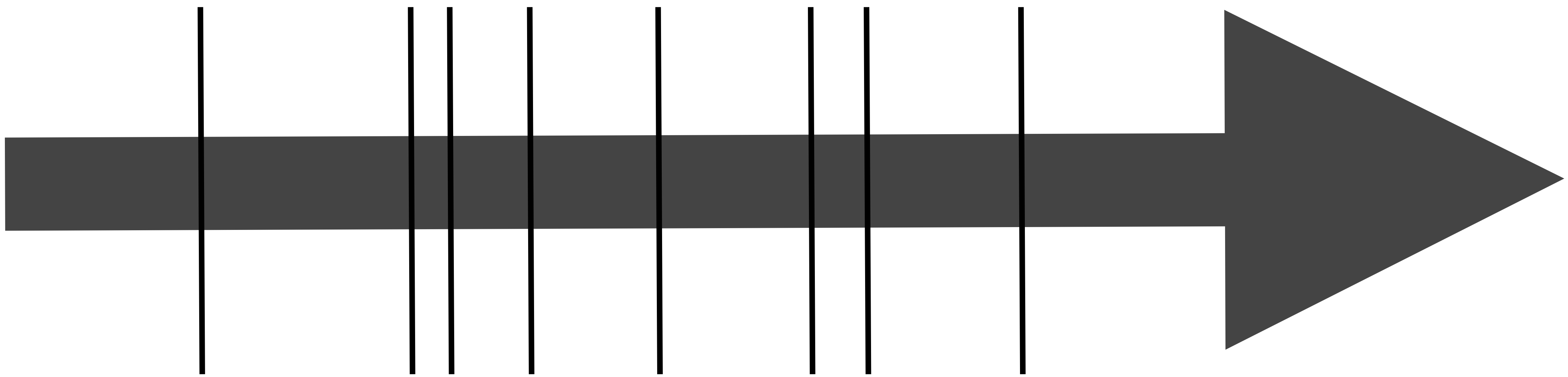
Sequence Points - Rule 2

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



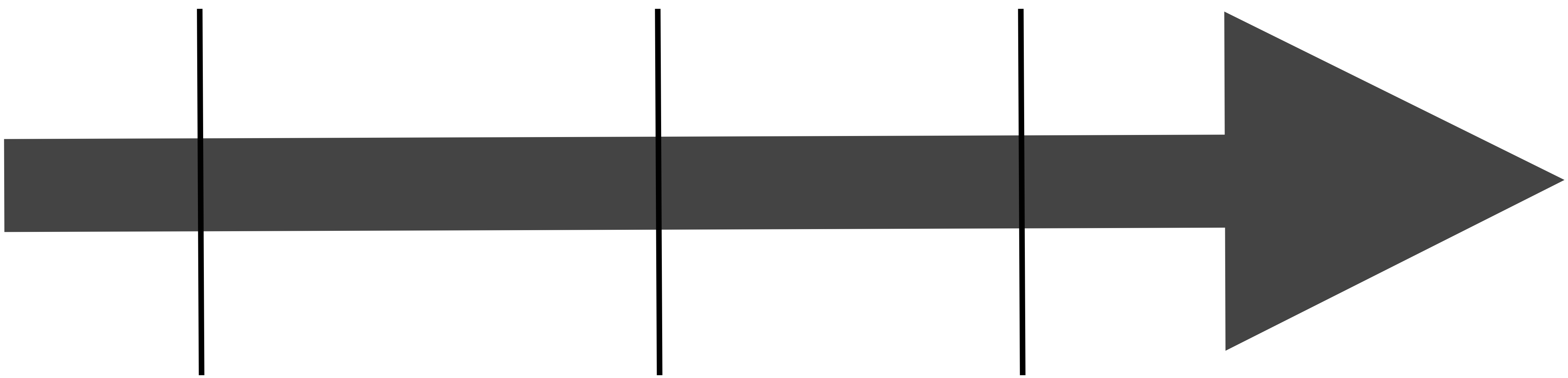
Sequence Points

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.

What do these code snippets print?

What do these code snippets print?

1

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

What do these code snippets print?

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

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

What do these code snippets print?

1

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

2

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

3

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

What do these code snippets print?

1

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

2

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

3

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

4

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

What do these code snippets print?

1

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

2

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

3

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

4

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

5

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

What do these code snippets print?

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

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);
```

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);
```

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);
```

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

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

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

```cpp
#include <iostream>

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

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

```
#include <iostream>

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

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

```cpp
#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
```

```
#include <iostream>

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

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

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

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

They are all morons!

```
#include <iostream>

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

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

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

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

They are all morons!

```
#include <iostream>

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

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

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

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

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

```
#include <iostream>

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

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

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

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

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

They are all morons!

ehh...

Behavior

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

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

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

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

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

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

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

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

Behavior

... and, locale-specific behavior

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

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

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

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

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

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

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

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

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

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

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

```
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "Denmark - Germany 1-4" << std::endl;
    else
      std::cout << "Denmark - Germany 9-2" << std::endl;
}
```

```
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "Denmark - Germany 1-4" << std::endl;
    else
      std::cout << "Denmark - Germany 9-2" << std::endl;
}
```

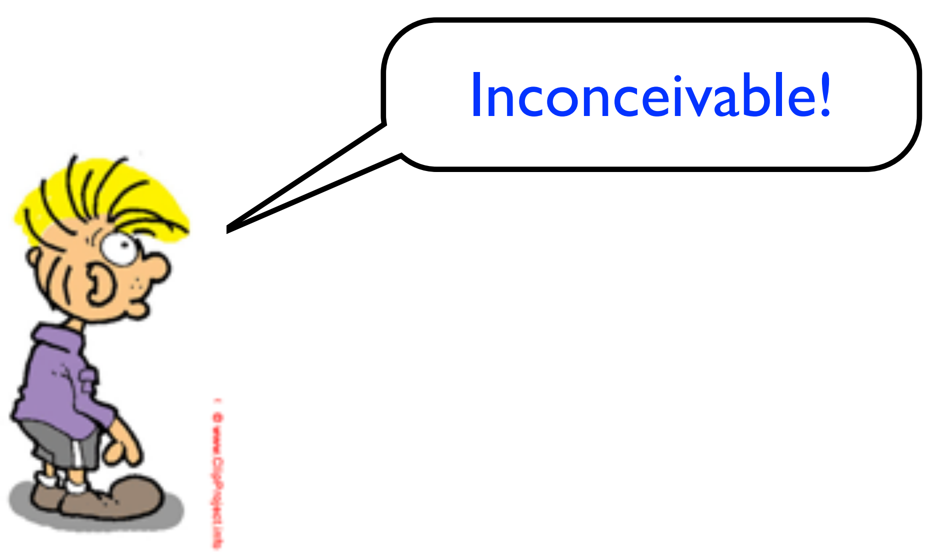
```
$ c++ foo.cpp && ./a.out
```

```cpp
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "Denmark - Germany 1-4" << std::endl;
    else
      std::cout << "Denmark - Germany 9-2" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
Denmark - Germany 9-2
```

```
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "Denmark - Germany 1-4" << std::endl;
    else
      std::cout << "Denmark - Germany 9-2" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
Denmark - Germany 9-2
```



```
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "Denmark - Germany 1-4" << std::endl;
    else
      std::cout << "Denmark - Germany 9-2" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
Denmark - Germany 9-2
```

```
#include <iostream>

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

Yes, on my machine it is

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

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

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

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

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

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

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

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

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

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

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

Yeah of course, I forgot about that, because in C++ the structs are padded to so size becomes multiple 4

```
#include <iostream>

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

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

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

Yes, on my machine it is

So you get 9?

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

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

Yeah of course, I forgot about that, because in C++ the structs are padded to so size becomes multiple 4

Kind of...

```
#include <iostream>

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

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

So what if I add a member function?

```cpp
#include <iostream>

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

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

So what if I add a member function?

```
#include <iostream>

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

};

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

```
#include <iostream>

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


};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
};

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

Now this code will print 16. Because there will be a pointer to the function.

```
#include <iostream>

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

    void set_value(int v) { a = v; }
};

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

Now this code will print 16. Because there will be a pointer to the function.

ok?

```
#include <iostream>

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

    void set_value(int v) { a = v; }
};

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

Now this code will print 16. Because there will be a pointer to the function.

ok?

Lets add two more functions...

```
#include <iostream>

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

    void set_value(int v) { a = v; }
};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }


};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Then it will print 24. Two more pointers.

This is what I get on my machine

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Then it will print 24. Two more pointers.

This is what I get on my machine

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Then it will print 24. Two more pointers.

This is what I get on my machine

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

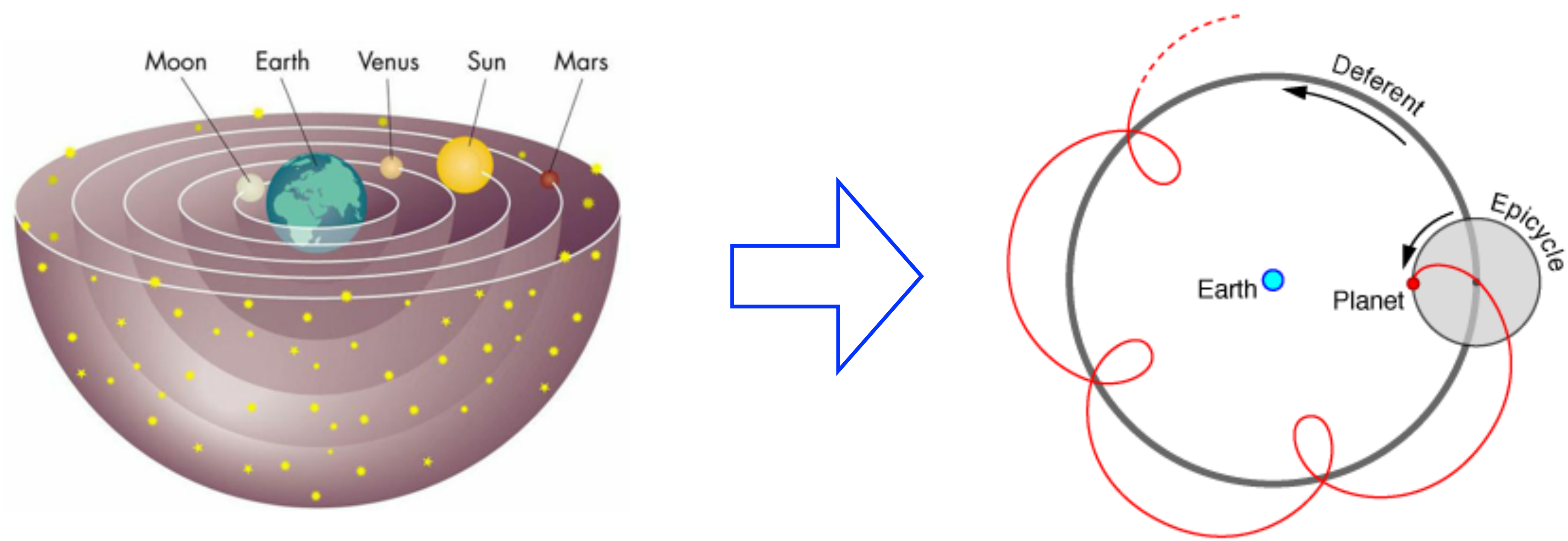
```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Then it will print 24. Two more pointers.

Huh? Probably some weird optimization going on, perhaps because the functions are never called.

This is what I get on my machine

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


Moon
Earth
Venus
Sun
Mars
Deferent
Epicycle
Earth
Planet

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Because adding member functions like this does not change the size of the struct. In C++, the object does not know about it's functions, it is the functions that know about the object.

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

Because adding member functions like this does not change the size of the struct. In C++, the object does not know about it's functions, it is the functions that know about the object.

If you rewrite this into C it becomes obvious.

C++

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};
```

C

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

void set_value(struct X * this, int v) { this->a = v; }
int get_value(struct X * this) { return this->a; }
void increase_value(struct X * this) { this->a++; }
```

```
#include <iostream>

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

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

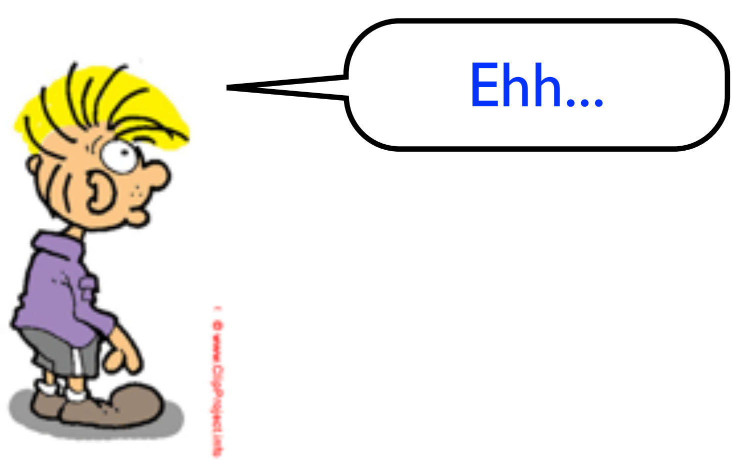
```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

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

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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

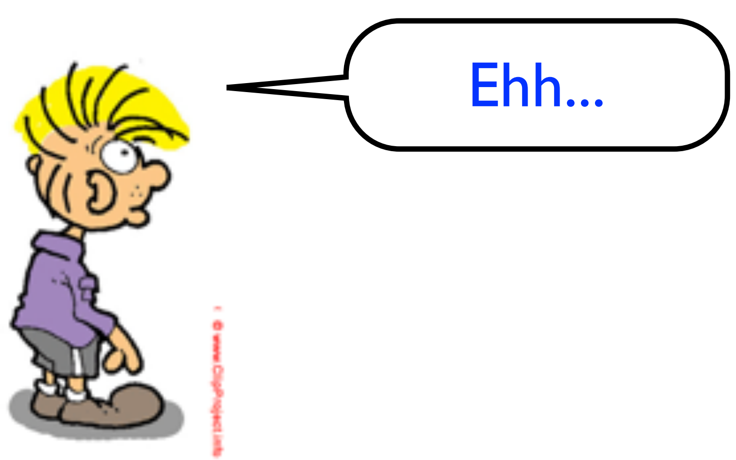
```
$ c++ foo.cpp && ./a.out
24
```

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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




```
$ c++ foo.cpp && ./a.out
24
```

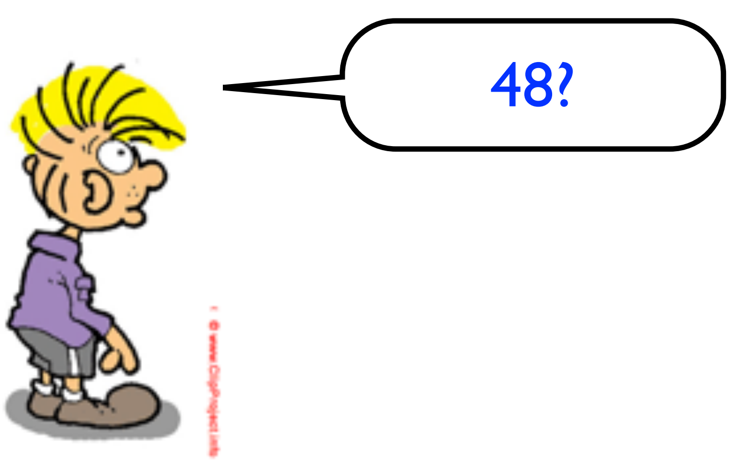
```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

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



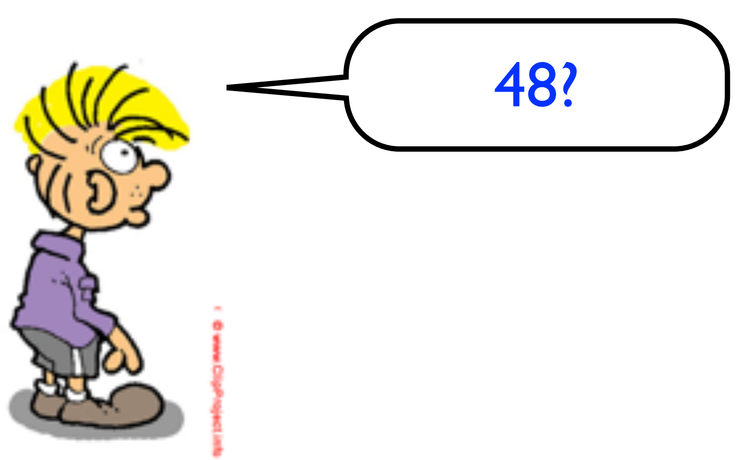

```
$ c++ foo.cpp && ./a.out
24
```

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

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

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

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

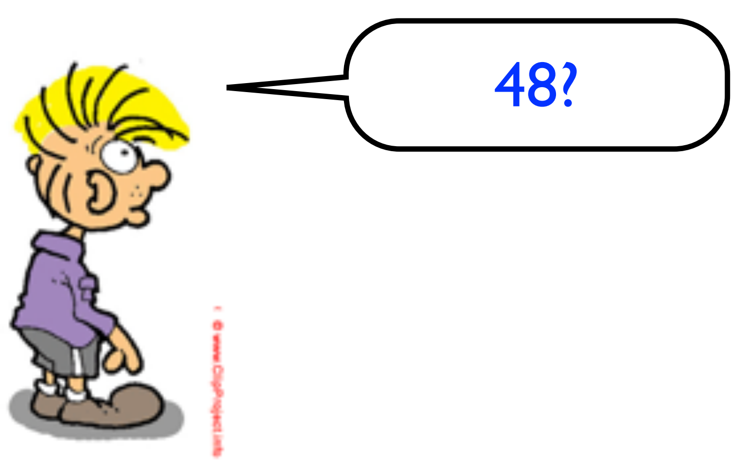


```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

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

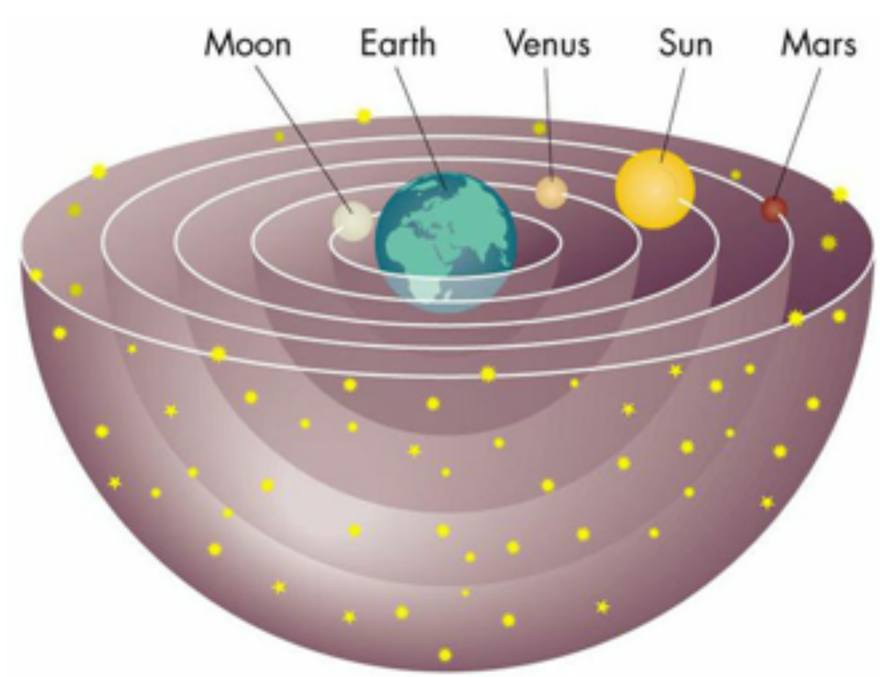


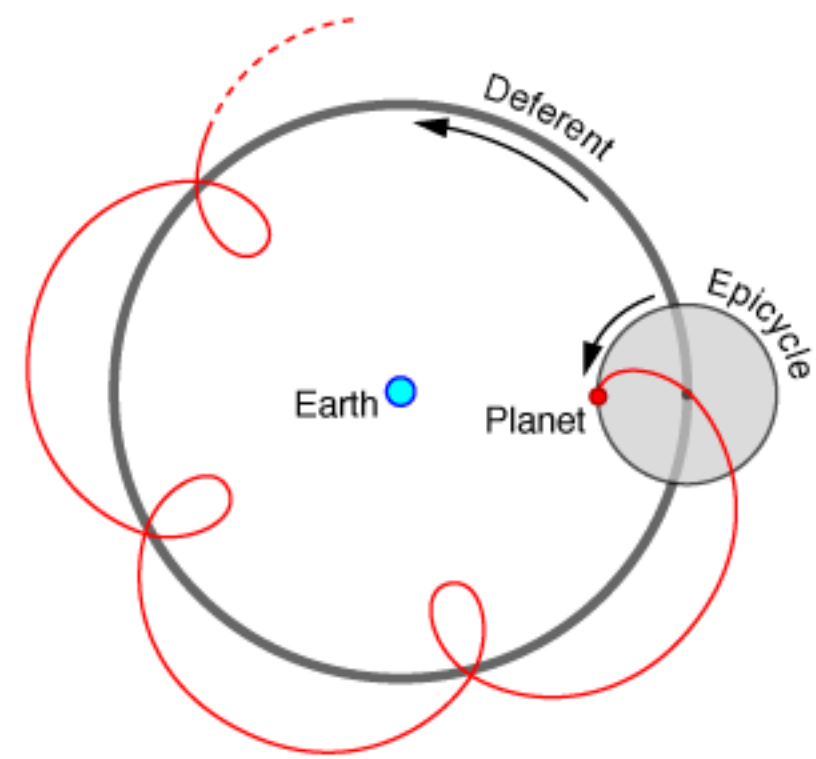

```
$ c++ foo.cpp && ./a.out
```

```cpp
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

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




```
$ c++ foo.cpp && ./a.out
24
```


Moon
Earth
Venus
Sun
Mars

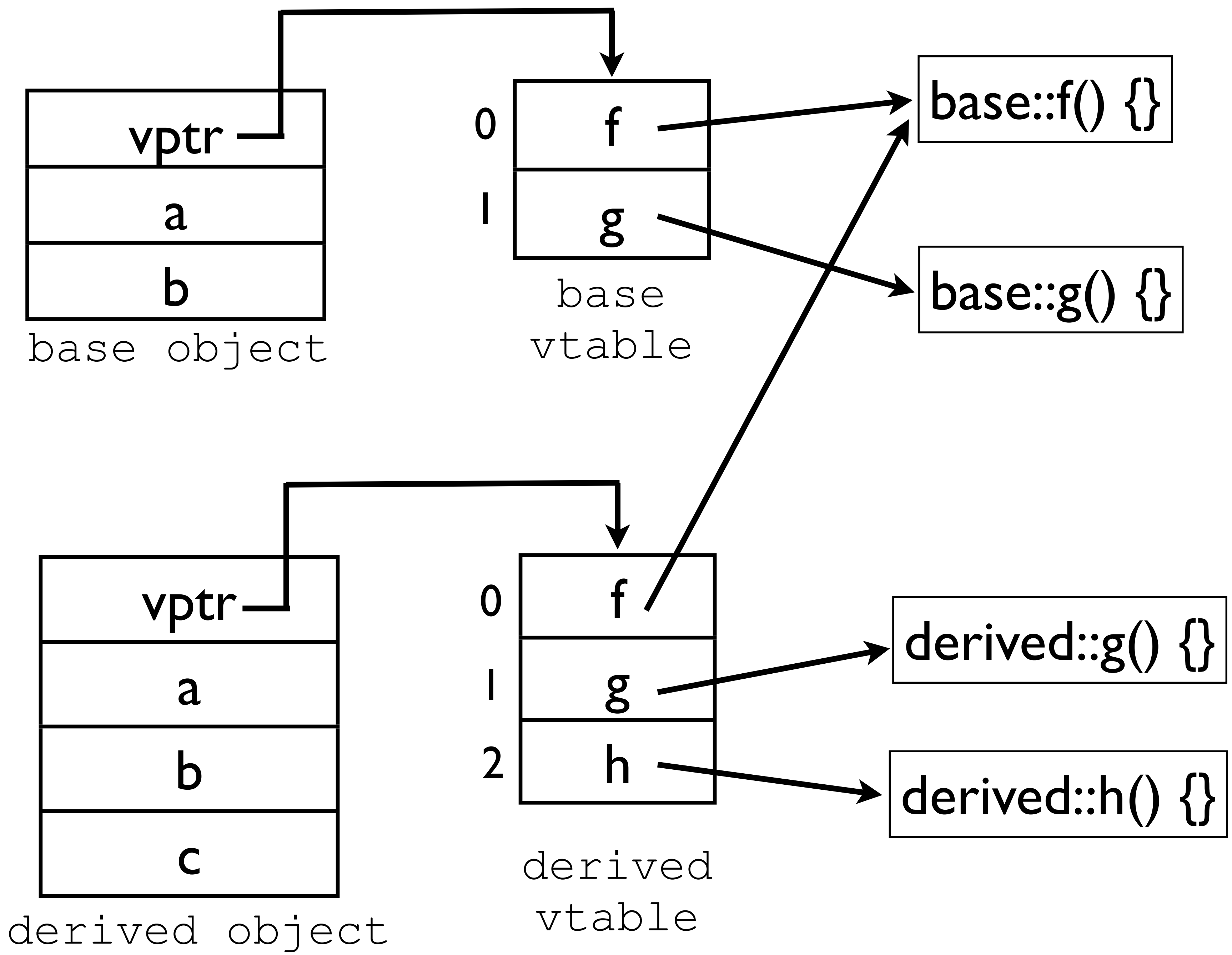

Deferent
Epicycle
Earth
Planet

```
#include <iostream>

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

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

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

```
$ c++ foo.cpp && ./a.out
24
```

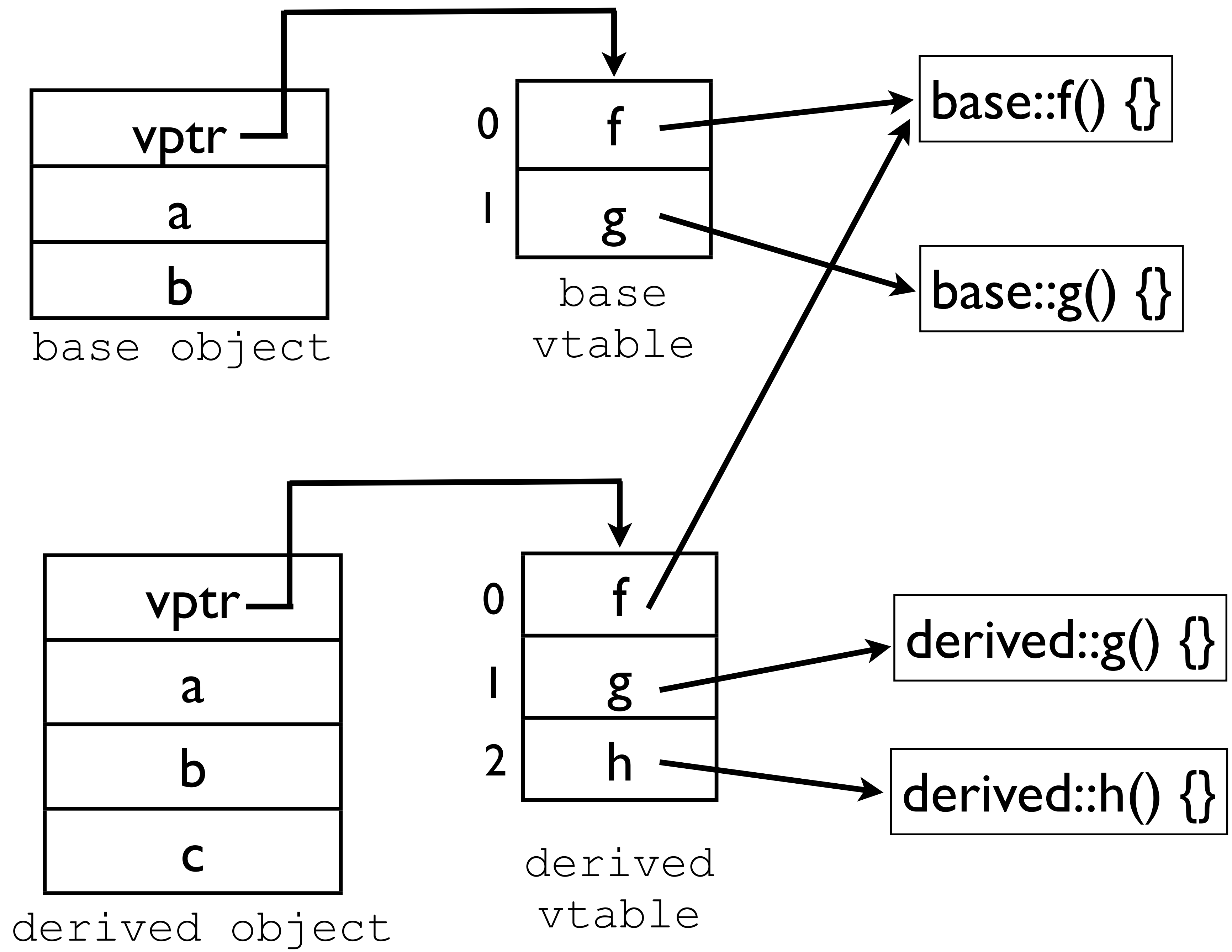
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



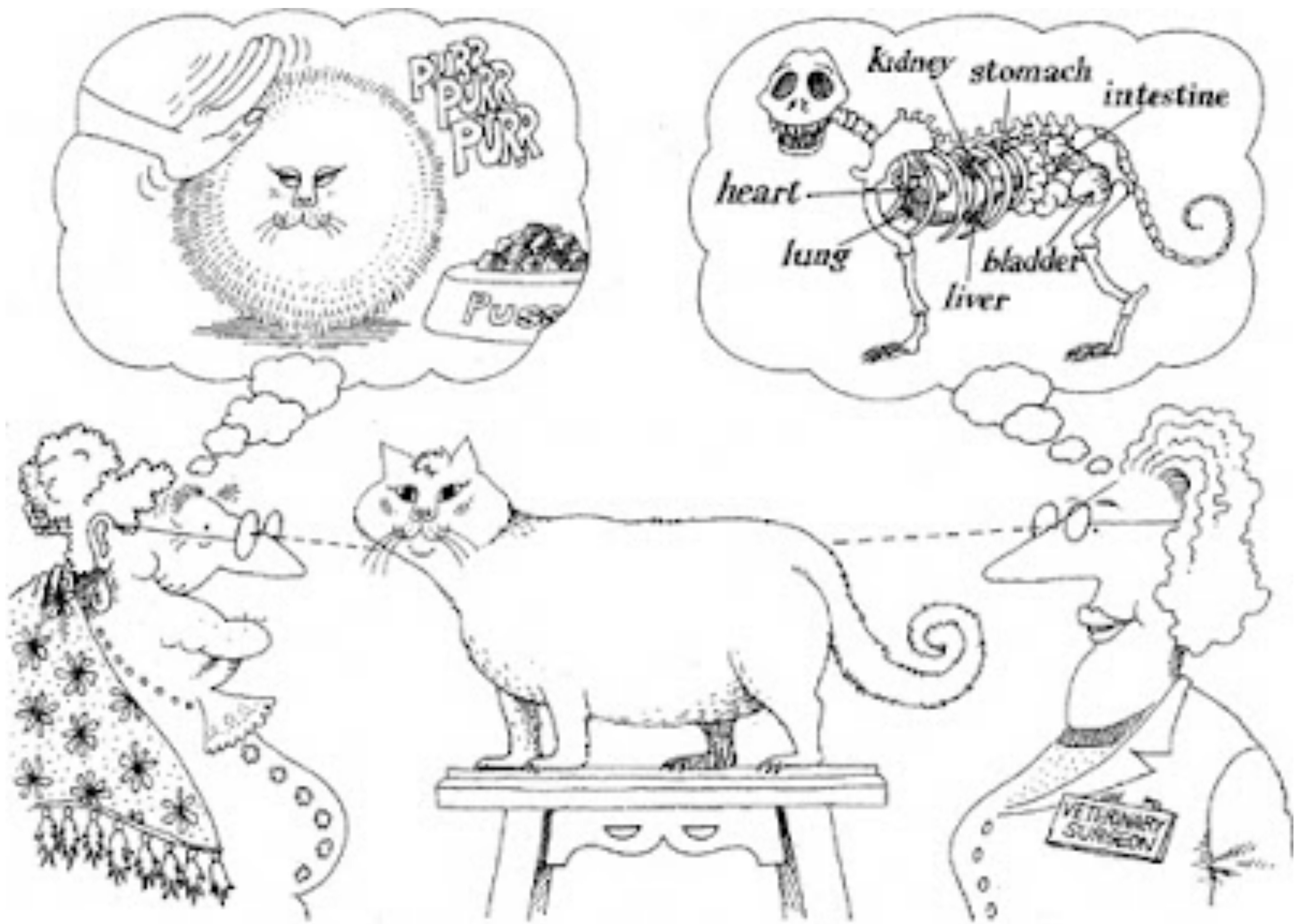
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



This is a common way of implementing virtual functions in C++

Let's move up one abstraction level



```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

This is a piece of shitty C++ code. Is this your code? First of all....

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

This is a piece of shitty C++ code. Is this your code? First of all....

never use 2 spaces for indentation.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

This is a piece of shitty C++ code. Is this your code? First of all....

never use 2 spaces for indentation.

The curly brace after class A should definitely start on a new line

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Take a look at this piece of code. Pretend like I am a junior C++ programmer joining your team. Here is a piece of code that I might present to you. Please be pedantic and try to gently introduce me to pitfalls of C++ and perhaps teach me something about the C++ way of doing things.

This is a piece of shitty C++ code. Is this your code? First of all....

never use 2 spaces for indentation.

The curly brace after class A should definitely start on a new line

sz_? I have never seen that naming convention, you should always use the GoF standard _sz or the Microsoft standard m_sz.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Do you see anything else?

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Do you see anything else?

eh?

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Do you see anything else?

eh?

Oh yes, I guess you know that in C++ all destructors should always be declared as virtual. I read it in some book and it is very important to avoid slicing when deleting objects of subtypes.

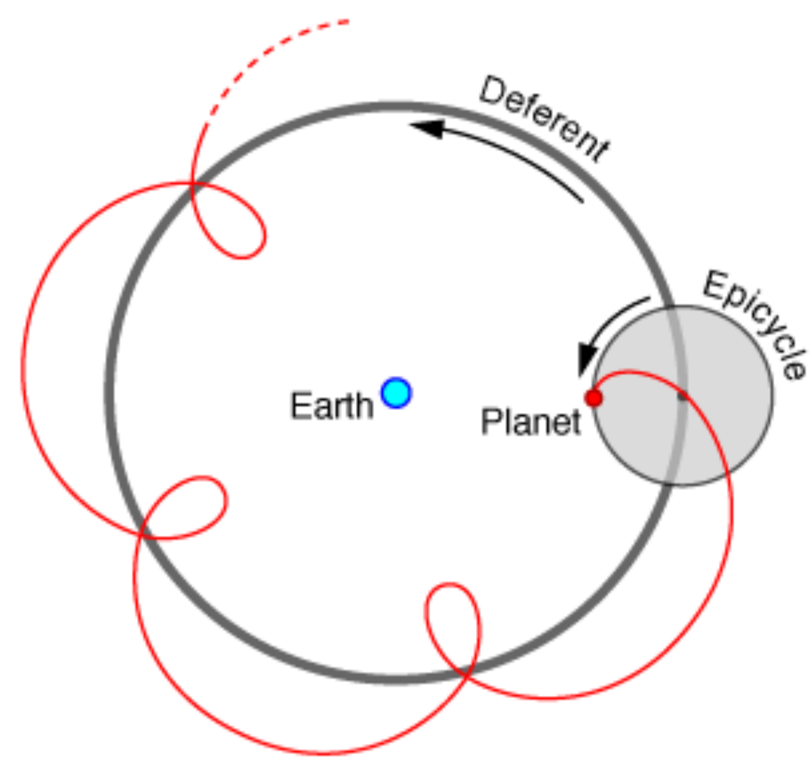
```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

Do you see anything else?

eh?

Oh yes, I guess you know that in C++ all destructors should always be declared as virtual. I read it in some book and it is very important to avoid slicing when deleting objects of subtypes.

or something like that...


Deferent
Epicycle
Earth
Planet

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

When you allocate an array, you must delete an array. Otherwise the destructors will not be called correctly.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

In this case, since you have a destructor like this, you **must** either implement or hide the copy constructor and assignment operator. This is called the rule of three, if you implement one of them, you must deal with them all.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:

  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:



  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

Not using the initializer list is usually a strong sign that the programmer does not really understand how to use C++. It does not make sense to first give member variables their default value, and *then* assign them a value.

```
#include "B.hpp"

class A {
public:
  A(int sz) { sz_ = sz; v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz) { v = new B[sz_]; }
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz), v(new B[sz_]) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz), v(new B[sz_]) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

oups! we just introduced a terrible bug!

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz), v(new B[sz_]) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

oups! we just introduced a terrible bug!

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz), v(new B[sz]) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : sz_(sz), v(new B[sz]) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : v(new B[sz]), sz_(sz) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : v(new B[sz]), sz_(sz) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;  ←—— ?
  int sz_;
};
```

```
#include "B.hpp"

class A {
public:
  A(int sz) : v(new B[sz]), sz_(sz) {}
  ~A() { delete[] v; }
  // ...
private:
  A(const A &);
  A & operator=(const A &);
  // ...
  B * v;
  int sz_;
};
```

Bald pointers are also often a sign of not using C++ correctly. When you see them, there are usually better ways of writing the code. In this case, perhaps a std::vector<B> is what you want?

Summary

Summary

- memory model

Summary

- memory model
- evaluation order

Summary

- memory model
- evaluation order
- sequence points

Summary

- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior

Summary

- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization

Summary

- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization
- vtables

Summary

- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization
- vtables
- object lifetimes

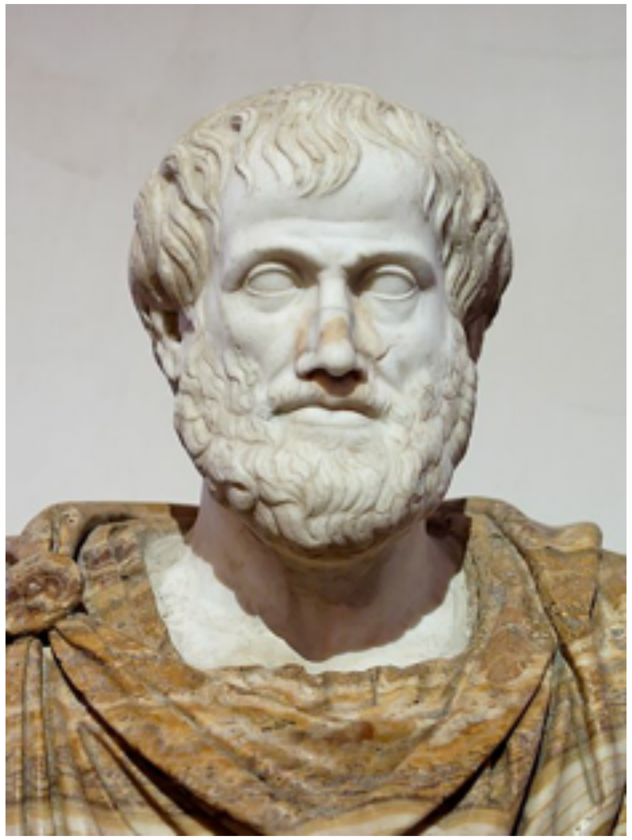
Summary

- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization
- vtables
- object lifetimes
- rule of 3

Summary

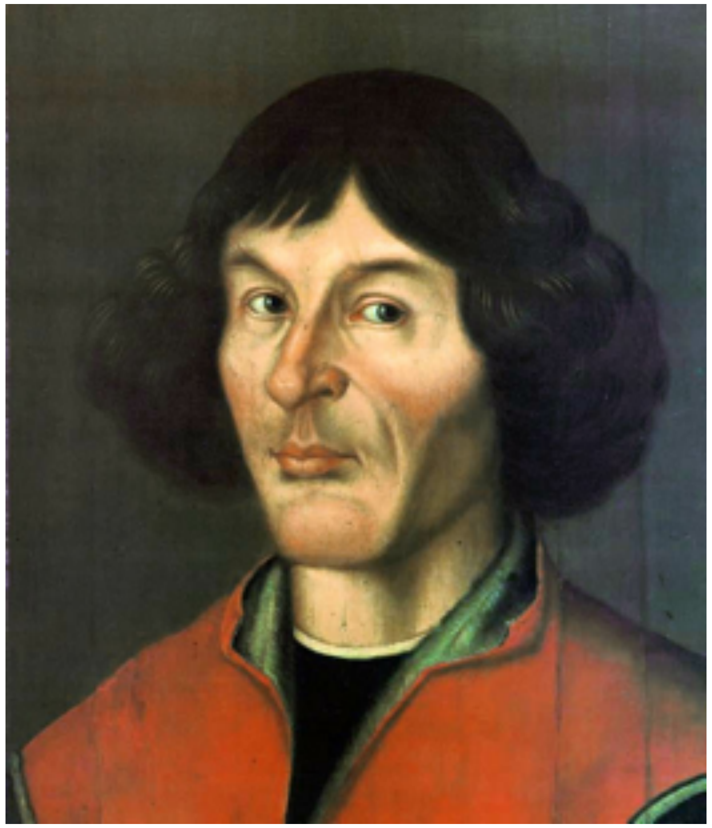
- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization
- vtables
- object lifetimes
- rule of 3
- initialization of objects

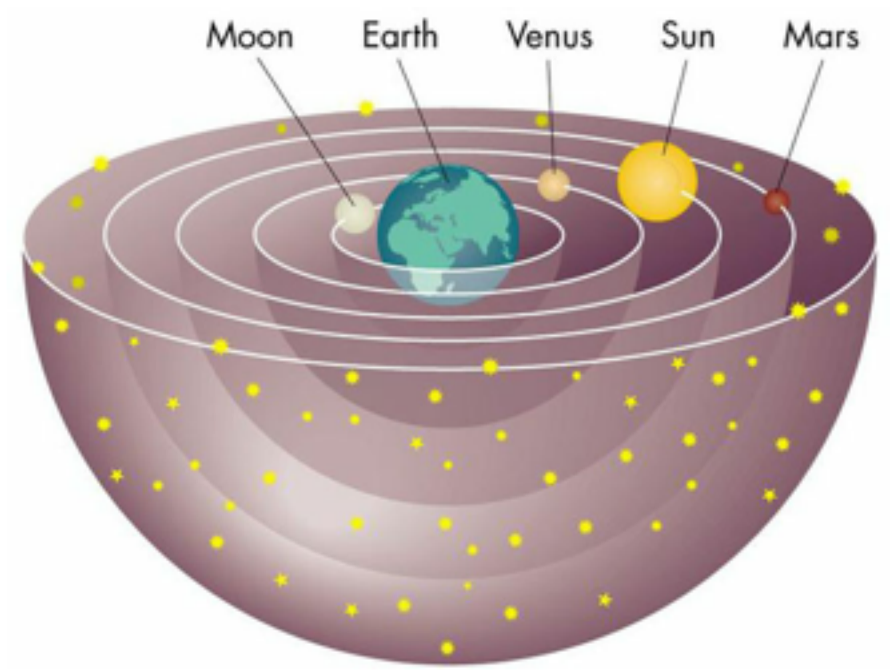
Aristotle (384 BC – 322 BC)



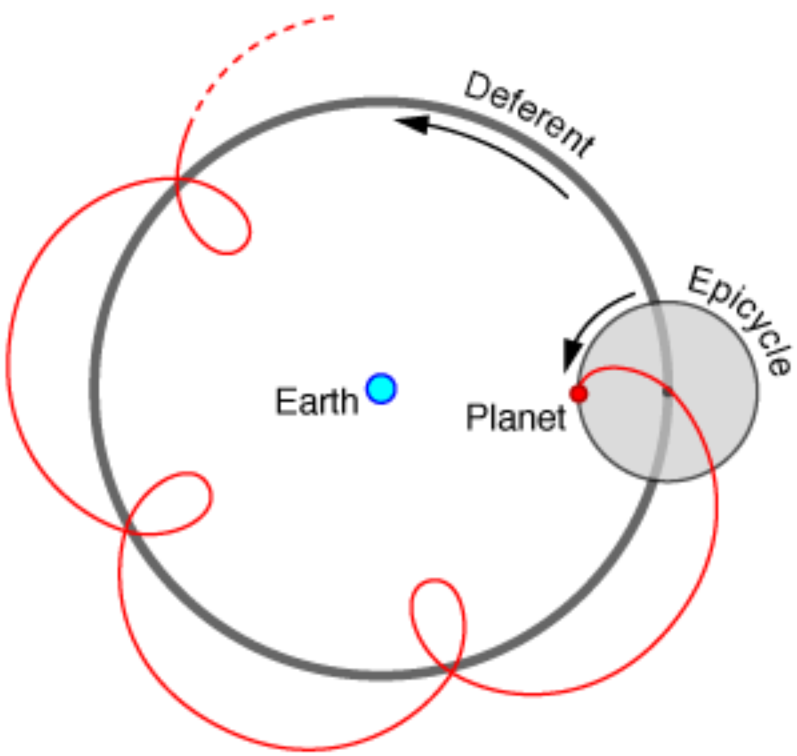
Ptolemy (90 AD – 168 AD)

Copernicus (1473 – 1543)

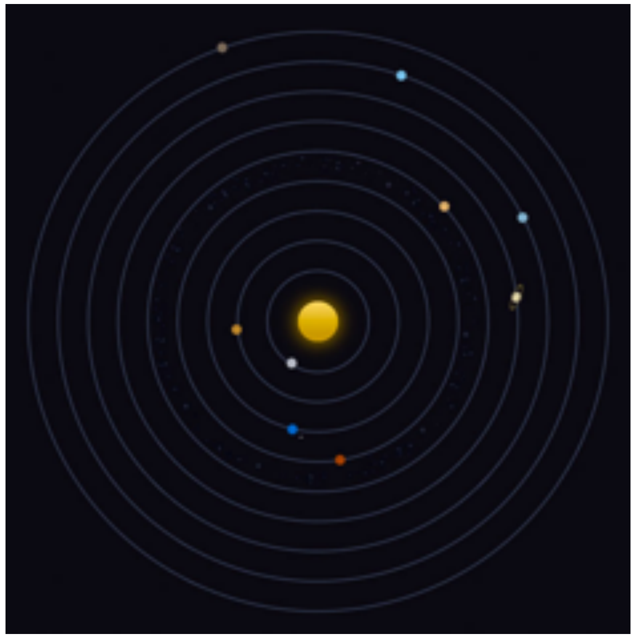




Aristotles Universe



Ptolemy Universe



The Solar System

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

!

Quick!

```
#include <iostream>

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

Quick!

```
#include <iostream>

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

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

Quick!

```
#include <iostream>

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

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

Quick!

```
#include <iostream>

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

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

Quick!

```
#include <iostream>

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

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

Quick!

```
#include <iostream>

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

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

Quick!

```
#include <iostream>

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

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

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