Deep C (and C++)

by Olve Maudal

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

Programming is hard. Programming correct C, and certainly 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 C++, and use them to discuss the fundamental building blocks, limitations and underlying design philosophies of these wonderful but dangerous programming languages.

A 3 hour course for students at NTNU
Monday, October 14, 2012

JAWS

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
```

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
$ ./foo
```

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
$ ./foo
11
```

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
$ ./foo
11
$ cc -Wall -Wextra -pedantic -o foo foo.c
```

Exercise

Type in this code. Compile and execute the program. What do you get?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
$ ./foo
11
$ cc -Wall -Wextra -pedantic -o foo foo.c
$ ./foo
```

Exercise

Type in this code. Compile and execute the program. What do you get?

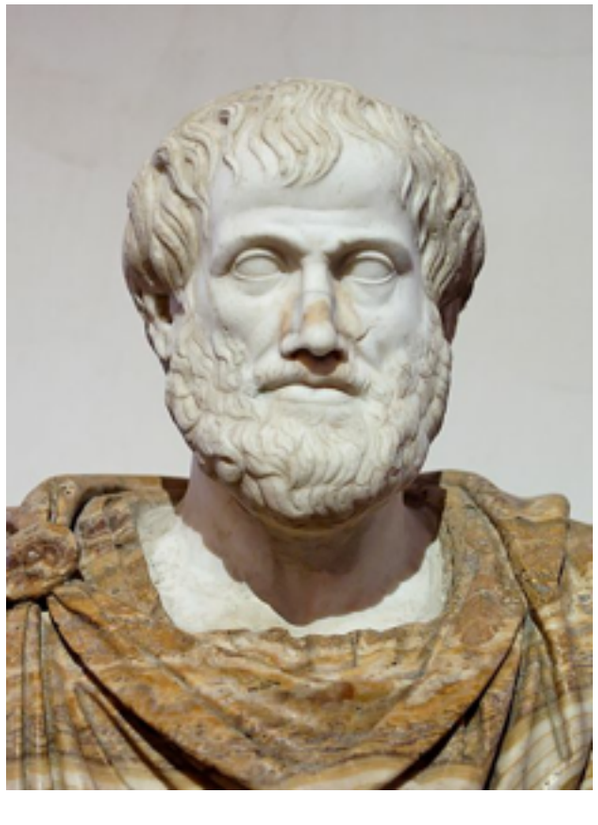
foo.c

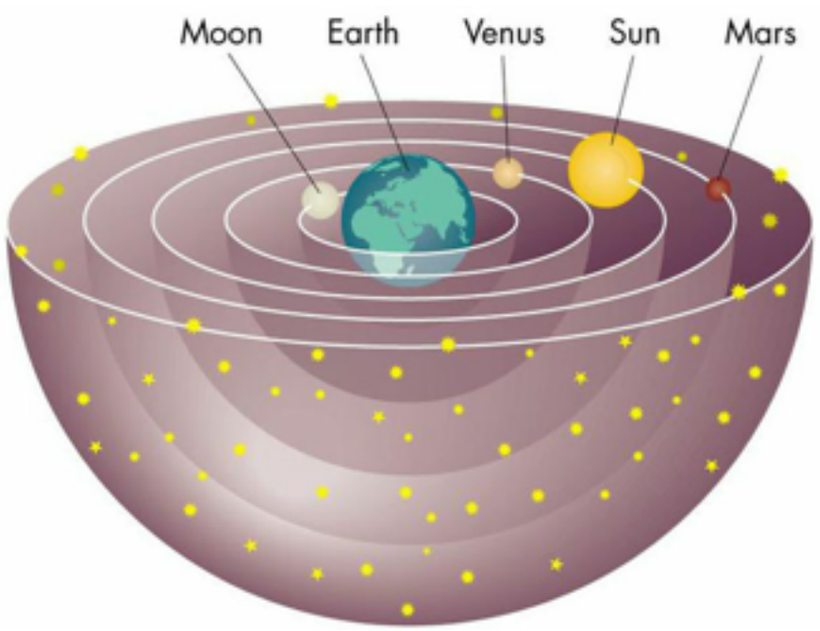
```
#include <stdio.h>

int main(void)
{
    int v[] = {2,4,6,8,10,12};
    int i = 1;
    int n = ++i + v[++i];
    printf("%d\n", n);
}
```

```
$ cc -o foo foo.c
$ ./foo
11
$ cc -Wall -Wextra -pedantic -o foo foo.c
$ ./foo
11
```

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

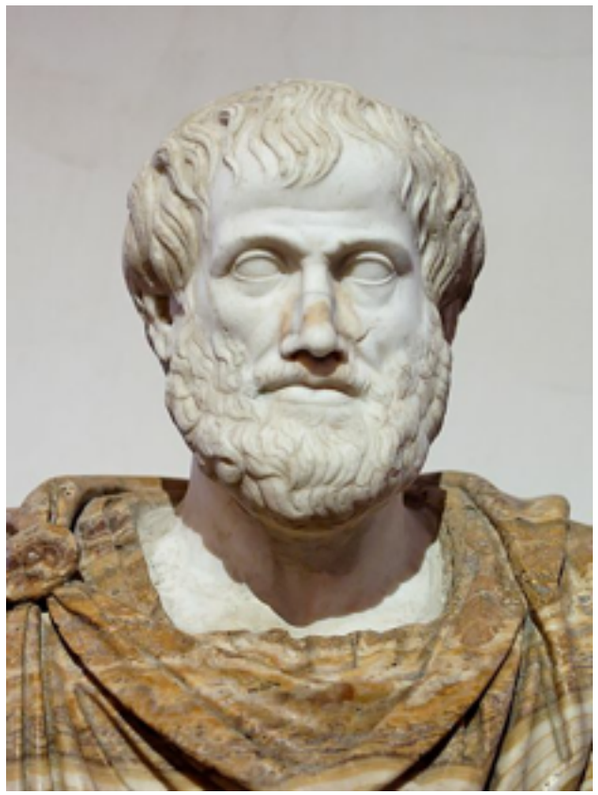
Aristotle (384 BC – 322 BC)



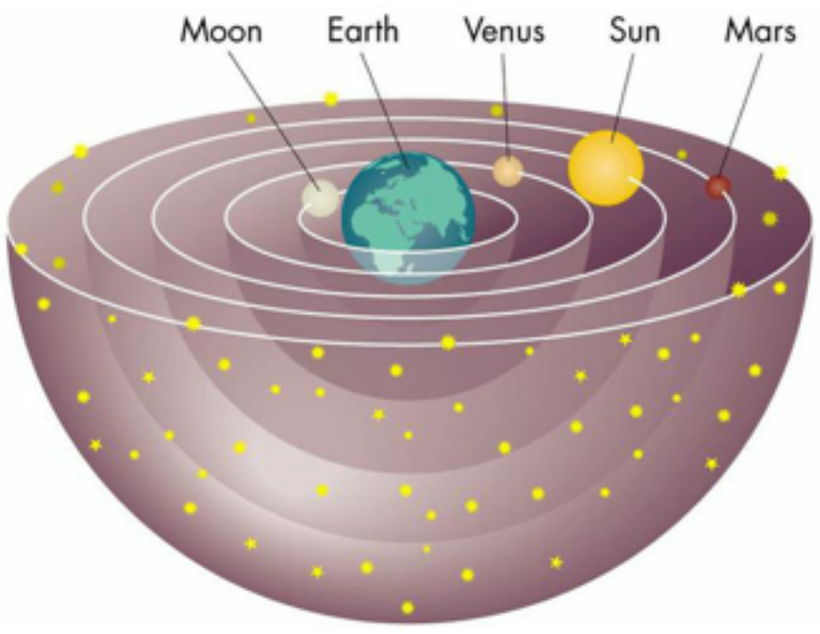


Aristotles Universe

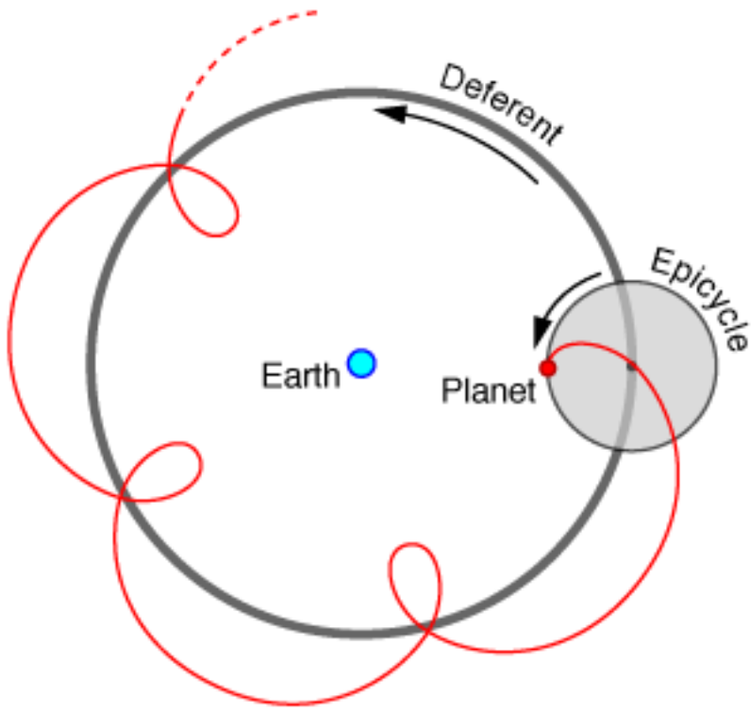
Aristotle (384 BC – 322 BC)



Ptolemy (90 AD – 168 AD)

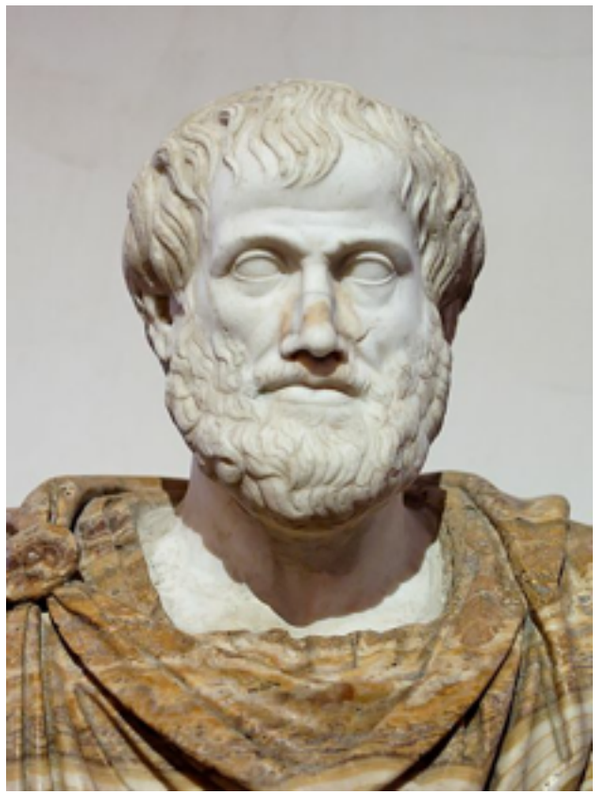


Aristotles Universe



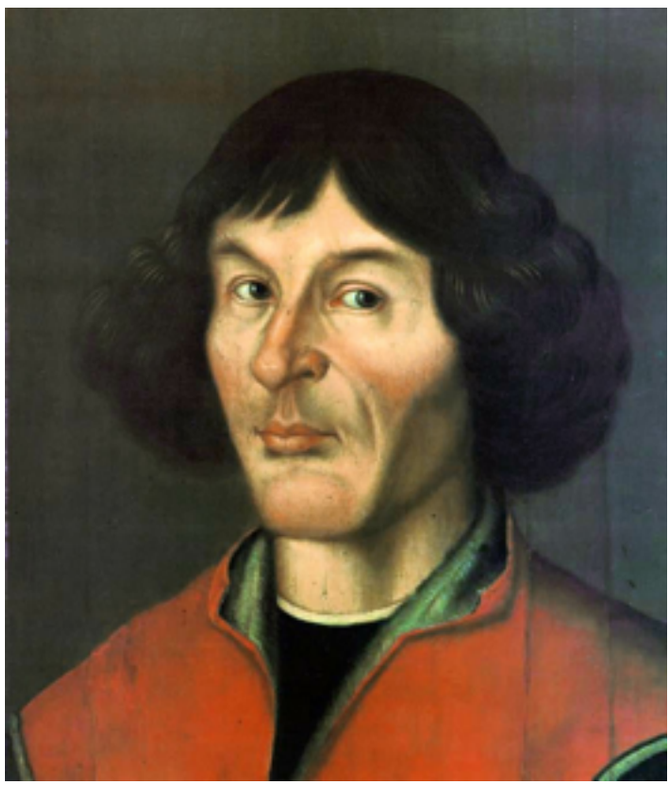
Ptolemy Universe

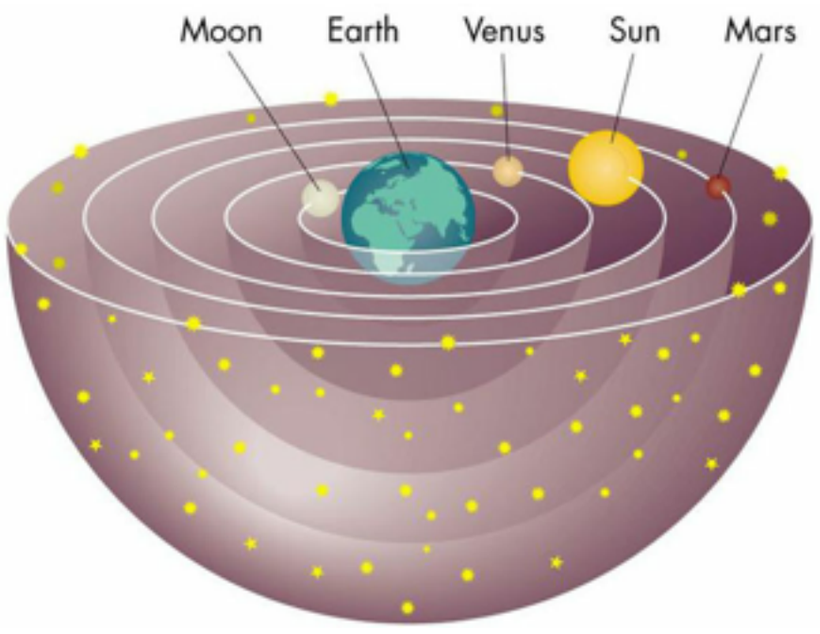
Aristotle (384 BC – 322 BC)



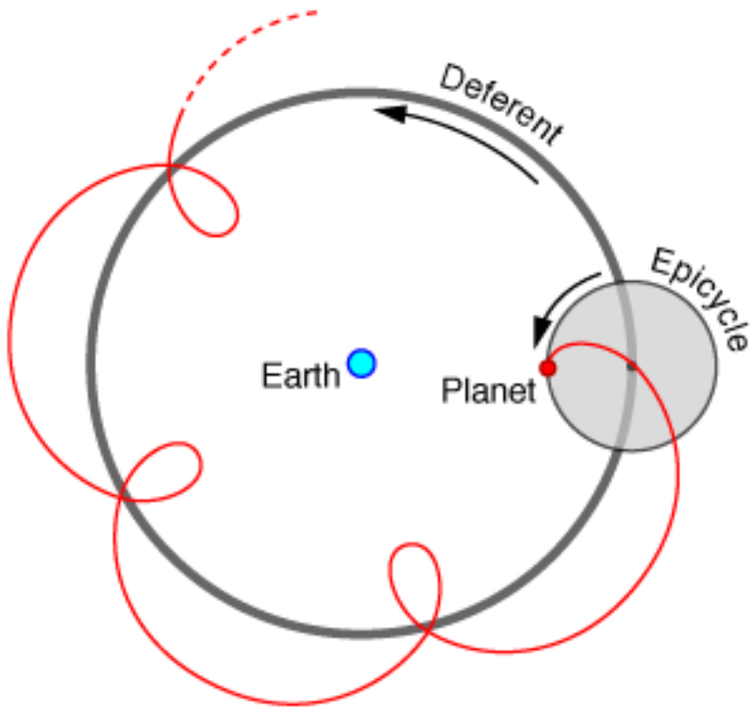
Ptolemy (90 AD – 168 AD)

Copernicus (1473 – 1543)

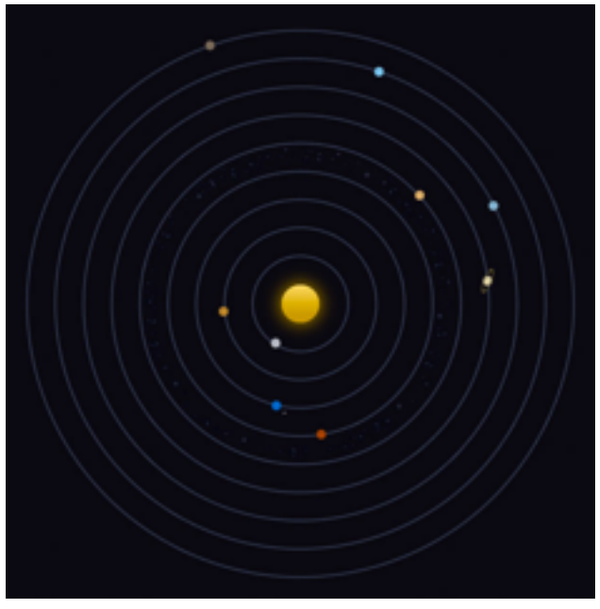




Aristotles Universe

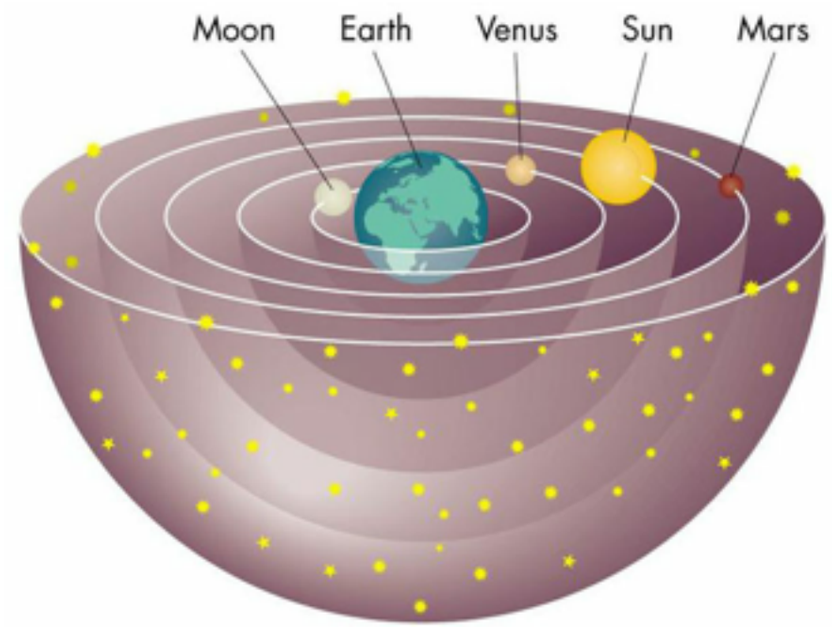


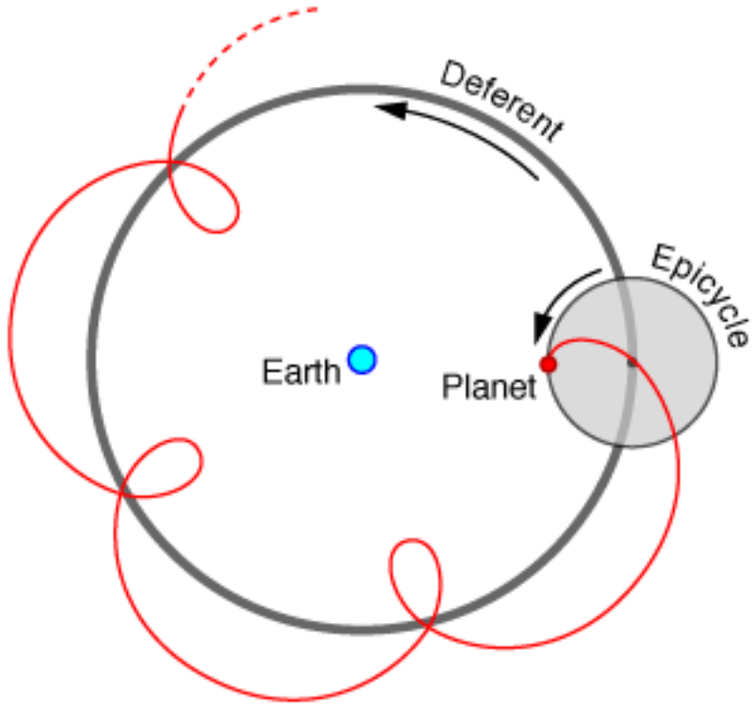
Ptolemy Universe



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();
}
```

```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();
}
```

```cpp
#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
```

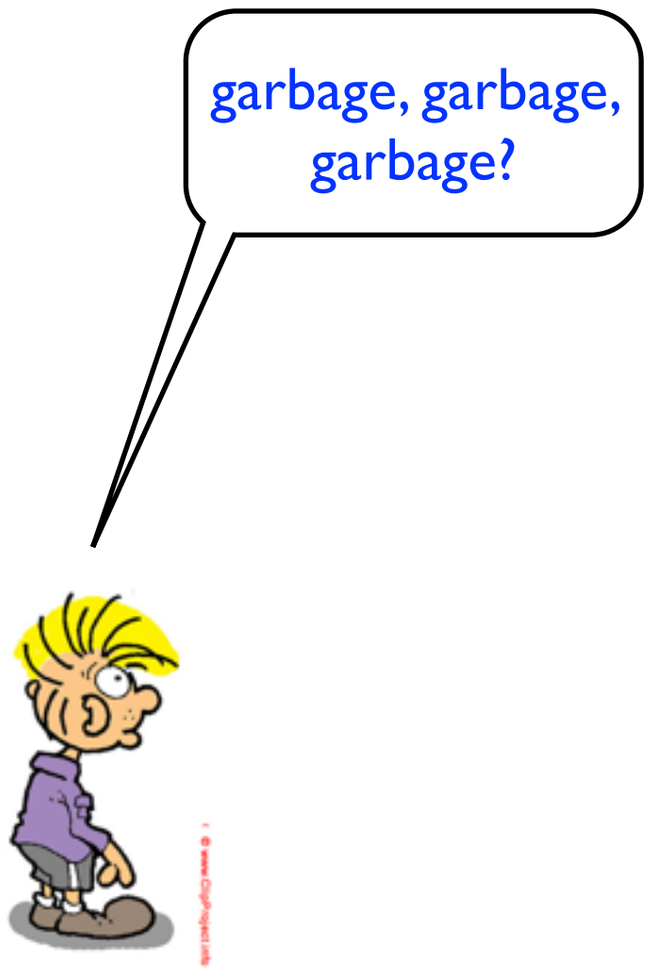
```cpp
#include <iostream>

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
```

```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();
}
```

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <iostream>

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

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

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

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

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

garbage, garbage, garbage?

It is better to initialize explicitly.

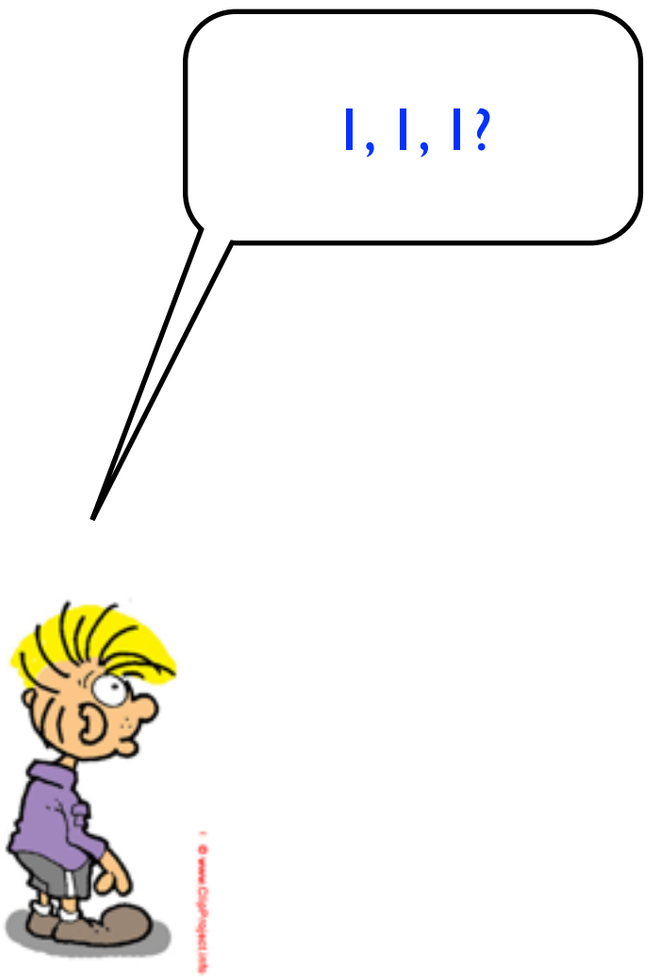
```
#include <iostream>

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

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

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

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

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

```
#include <iostream>

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

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

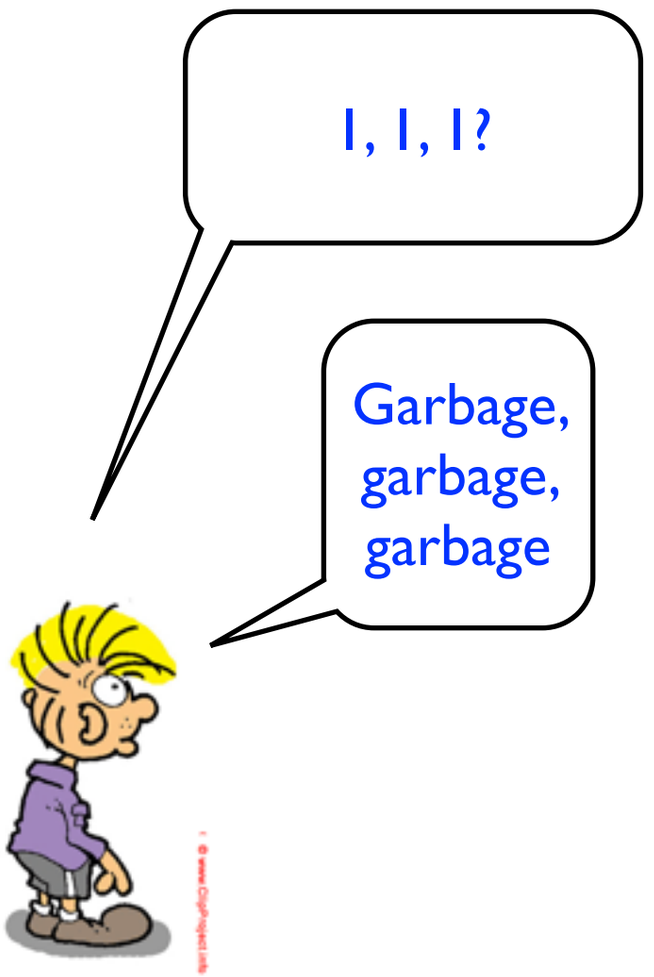



```
#include <iostream>

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

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

1, 1, 1?

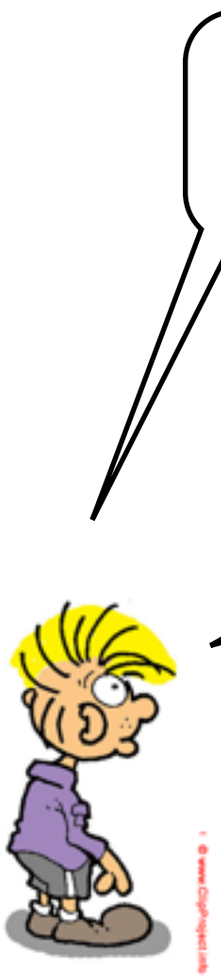
```
#include <iostream>

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




```
#include <iostream>

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

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

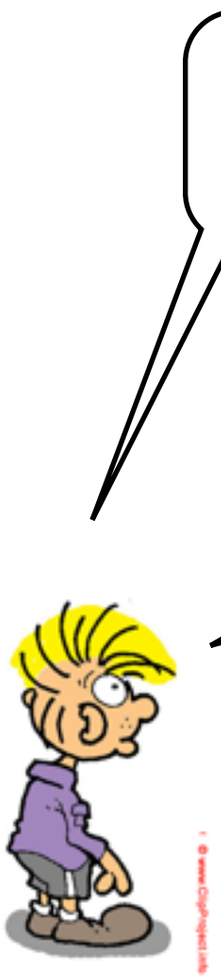
No, variables with automatic storage duration is not initialized implicitly




```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

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

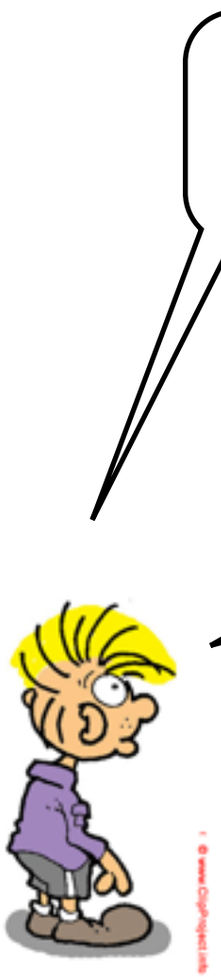



```
#include <iostream>

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

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

No, variables with automatic storage duration is not initialized implicitly

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

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

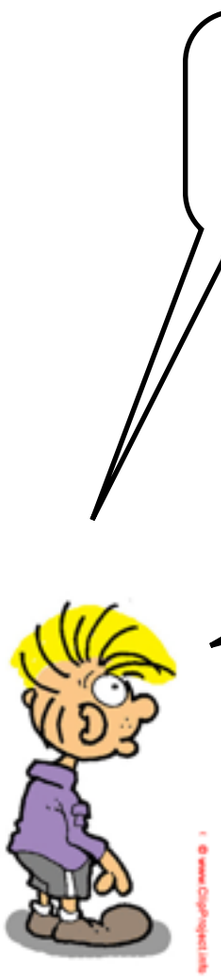



```
#include <iostream>

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
```

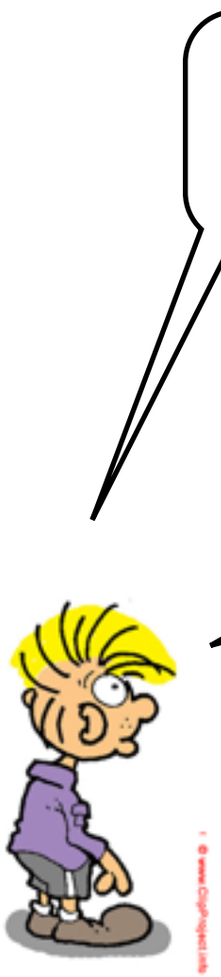



```
#include <iostream>

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
```

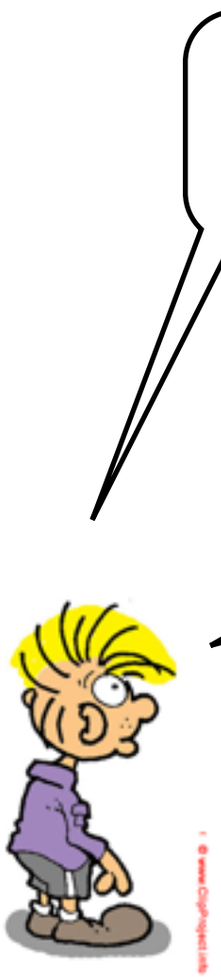



```
#include <iostream>

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
```

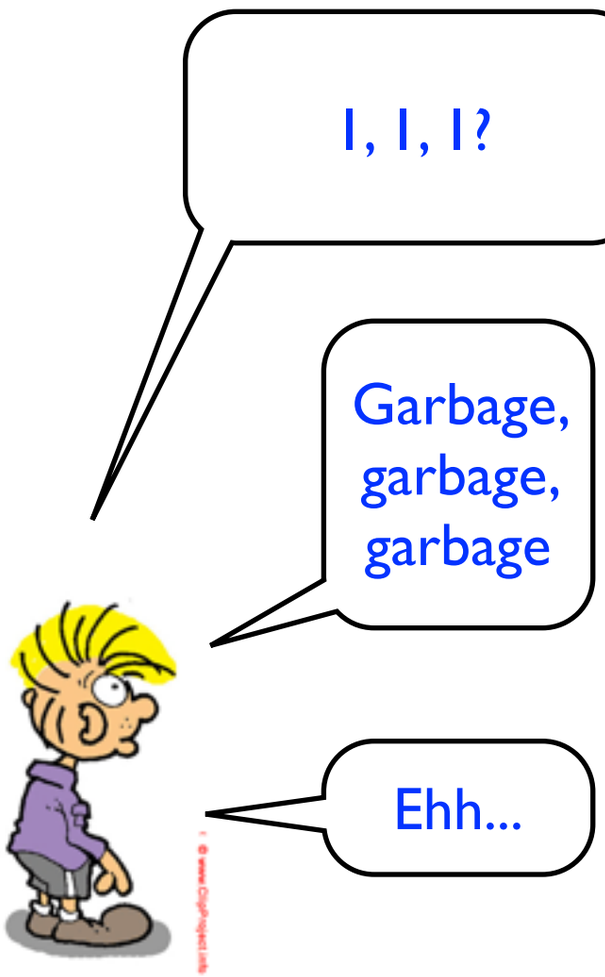



```
#include <iostream>

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

```
$ 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

any plausible explanation for this behaviour?

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

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();
}
```

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();
}
```

OK, let's add some flags

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

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();
}
```


```
#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
```

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!

```
#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?

eh?

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!

```
#include <iostream>

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

void bar()
{
    int a = 42;
}

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

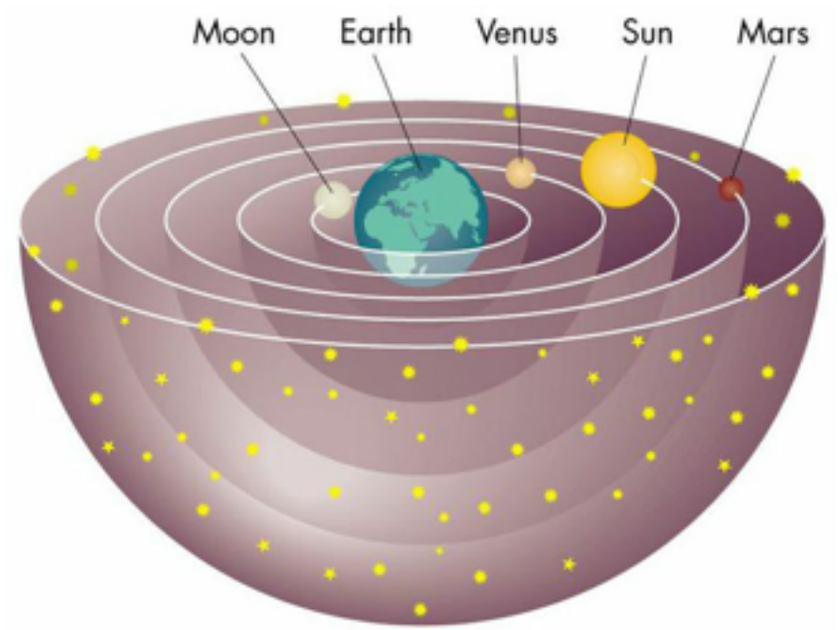
```
$ c++ foo.cpp && ./a.out
42
```

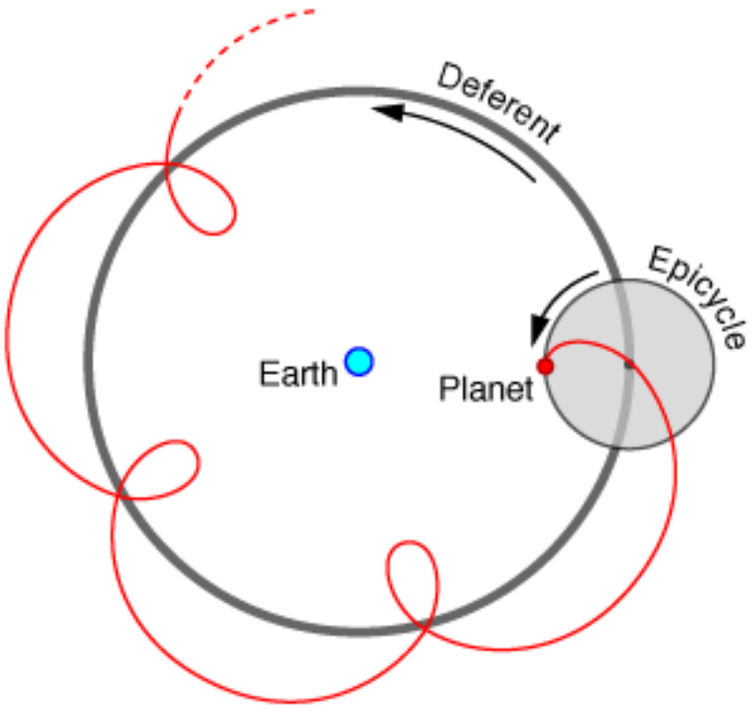
Can you explain this behaviour?

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.

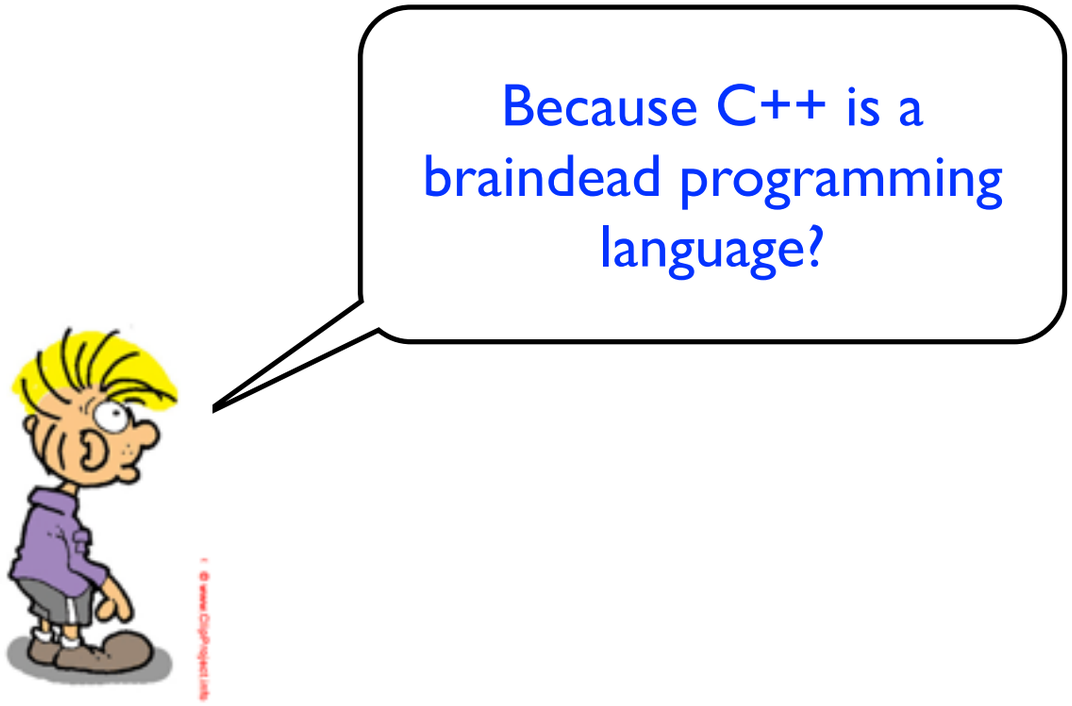
Yeah, sure...


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.

```
#include <iostream>

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

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

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

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

```cpp
#include <iostream>

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

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

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

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

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

```cpp
#include <iostream>

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

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

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

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

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

```cpp
#include <iostream>

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

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

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

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

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

```
#include <iostream>

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

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

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

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

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

but you might also get

```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
```

```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
```

```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

```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
```

```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
```

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;
}
```

```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
```

```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
```

```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;
}
```

What? Inconceivable!

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

```
#include <iostream>

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

What? Inconceivable!

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

This is a 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;
}
```

What? Inconceivable!

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

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

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

```
#include <iostream>

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

What? Inconceivable!

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

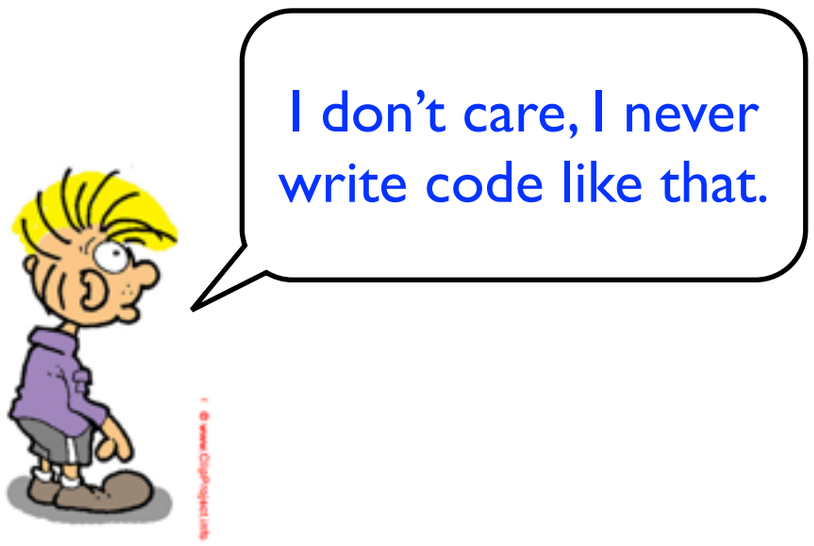
I agree this is crap code, but why is it wrong?

This is a 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**.

```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
```



```cpp
#include <iostream>

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

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

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

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...

```cpp
#include <iostream>

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

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

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

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

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...

```cpp
#include <iostream>

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

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

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

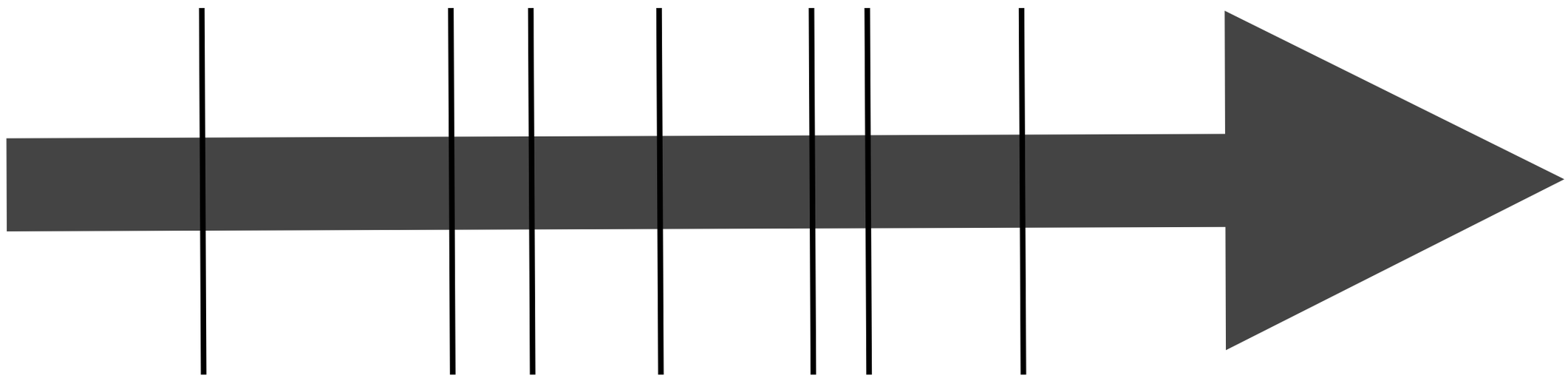
But why do we not get warning on this by default?

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...

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.

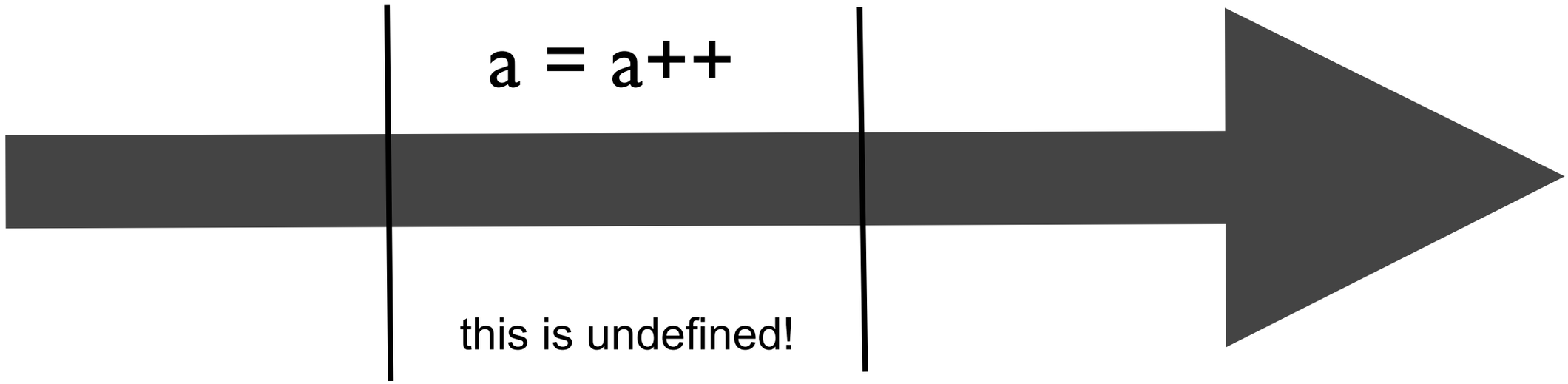
Sequence Points

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



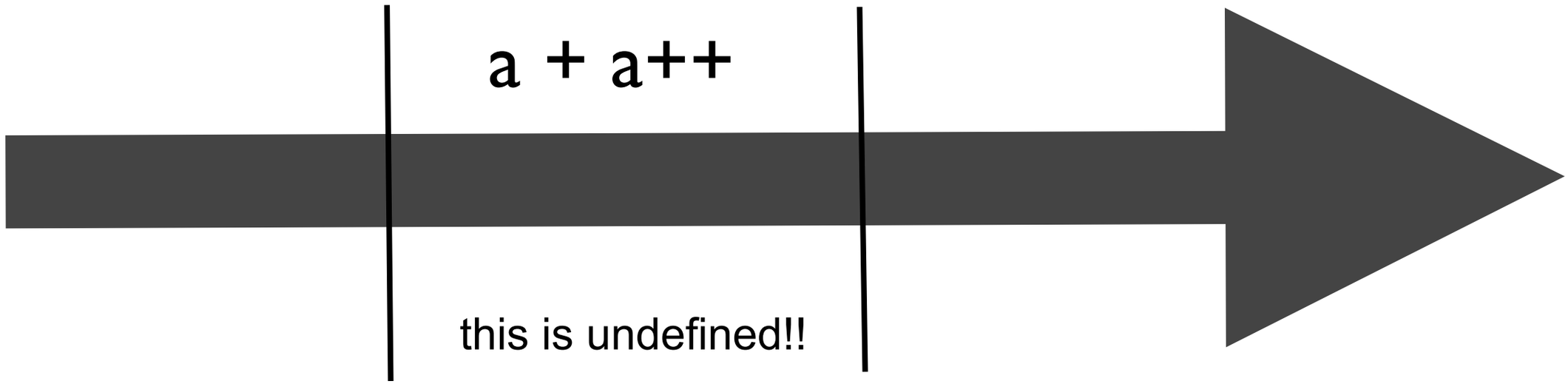
Sequence Points - Rule 1

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



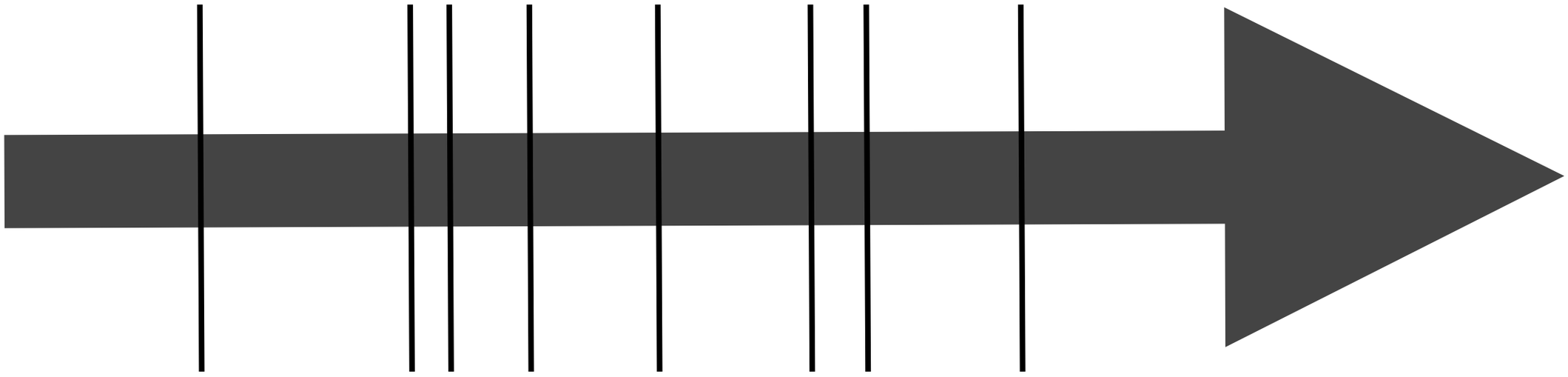
Sequence Points - Rule 2

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



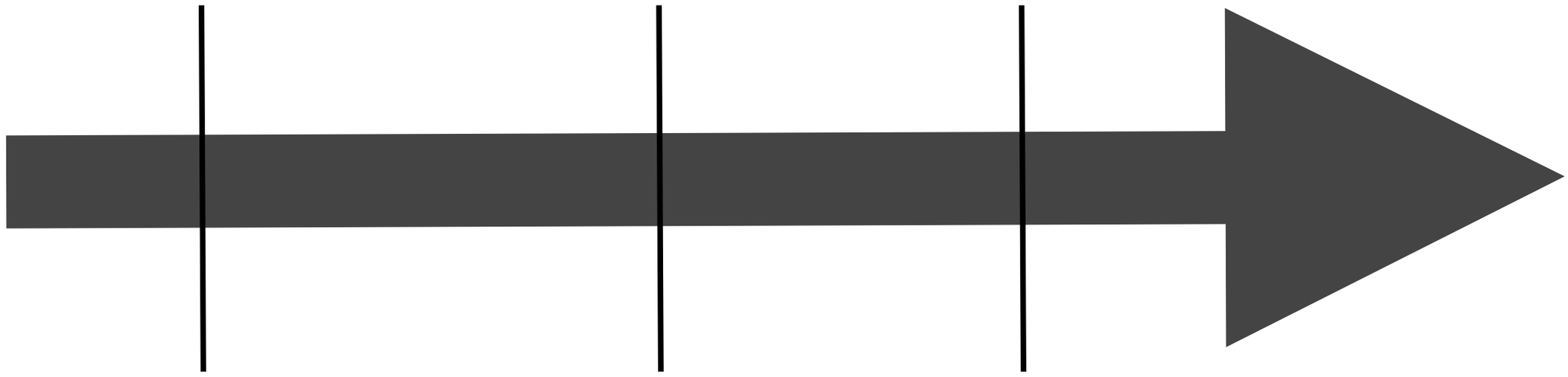
Sequence Points

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);
```

2

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

3

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

4

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

5

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

6

```
int a=41; a = foo(a++); printf("%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);
```

6

```
int a=41; a = foo(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);
```

6

```
int a=41; a = foo(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);
```

6

```
int a=41; a = foo(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);
```

6

```
int a=41; a = foo(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

6

```
int a=41; a = foo(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

6
```
int a=41; a = foo(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

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

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

```
#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();
}
```

```
#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();
}
```

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.

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

They are all morons!

```cpp
#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
```

They are all morons!

ehh...

```
#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
```

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 << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

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

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

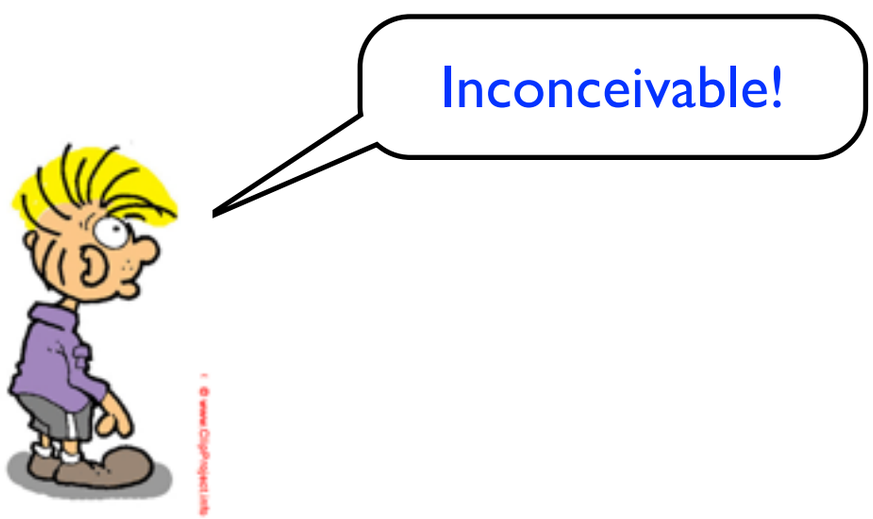
```
$ c++ foo.cpp && ./a.out
```

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

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
The answer is 44
```

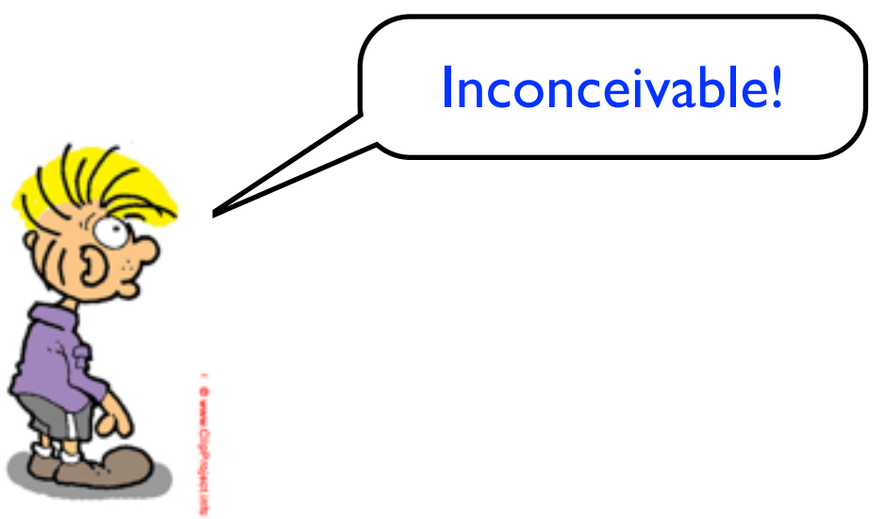
```
#include <iostream>
#include <climits>

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
The answer is 44
```



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

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
The answer is 44
```



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

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

int main() {
    int i = INT_MAX;
    int j = i + 1 - 1;
    if (j == INT_MAX)
      std::cout << "The answer is 42" << std::endl;
    else
      std::cout << "The answer is 43" << std::endl;
}
```

```
$ c++ foo.cpp && ./a.out
The answer is 44
```

Inconceivable!

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

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

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_a     $b          ; load value of b into register A
  compare_a  0           ; compare register A to 0
  jump_equal label1      ; skip next statement if A == 0
  call print_b_is_true   ; print "b is true"
label1:
  load_a     $b          ; load value of b into register A
  xor_a      1           ; xor register A with 1
  compare_a  0           ; compare register A to 0
  jump_equal label2      ; skip next statement if A == 0
  call print_b_is_false  ; 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_a     $b          ; load value of b into register A
  compare_a  0           ; compare register A to 0
  jump_equal label1      ; skip next statement if A == 0
  call print_b_is_true   ; print "b is true"
label1:
  load_a     $b          ; load value of b into register A
  xor_a      1           ; xor register A with 1
  compare_a  0           ; compare register A to 0
  jump_equal label2      ; skip next statement if A == 0
  call print_b_is_false  ; print "b is false"
label2:
```

this is approximately the code generated by gcc for an intel platform, try to imagine what will happen if b is 42

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_a     $b          ; load value of b into register A
  compare_a  0           ; compare register A to 0
  jump_equal label1      ; skip next statement if A == 0
  call print_b_is_true   ; print "b is true"
label1:
  load_a     $b          ; load value of b into register A
  xor_a      1           ; xor register A with 1
  compare_a  0           ; compare register A to 0
  jump_equal label2      ; skip next statement if A == 0
  call print_b_is_false  ; print "b is false"
label2:
```

this is approximately the code generated by gcc for an intel platform, try to imagine what will happen if b is 42

```
b is true
b is false
```

```
#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;
};

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

```cpp
#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
```

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

```cpp
#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

```cpp
#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

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

```cpp
#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

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?

```
#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.

```cpp
#include <iostream>

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

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

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

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

ok?

```
#include <iostream>

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

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

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

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

ok?

Lets add two more functions...

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

Then it will print 24. Two more pointers.

```
#include <iostream>

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

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

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

Then it will print 24. Two more pointers.

This is what I get on my machine

```cpp
#include <iostream>

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

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

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

Then it will print 24. Two more pointers.

This is what I get on my machine

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

```cpp
#include <iostream>

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

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

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

Then it will print 24. Two more pointers.

This is what I get on my machine

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

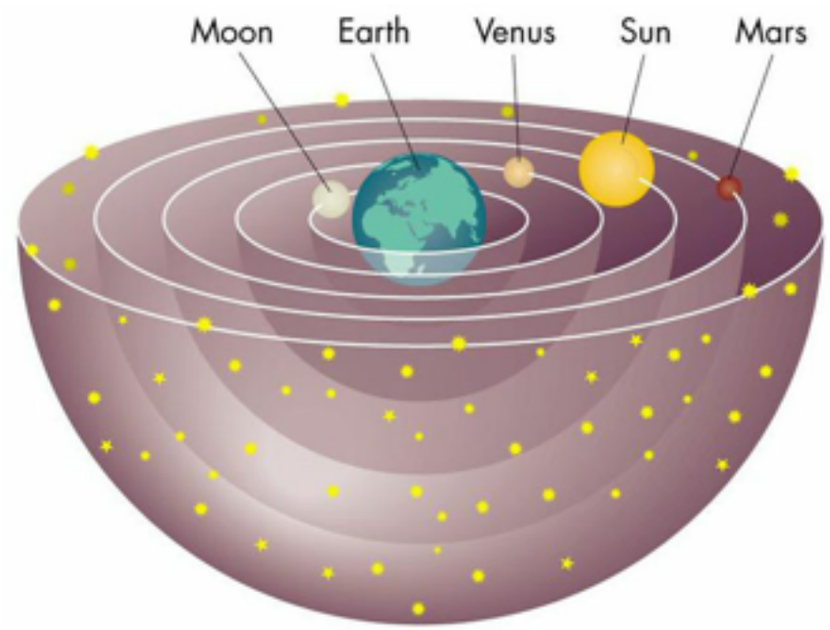
```cpp
#include <iostream>

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

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

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

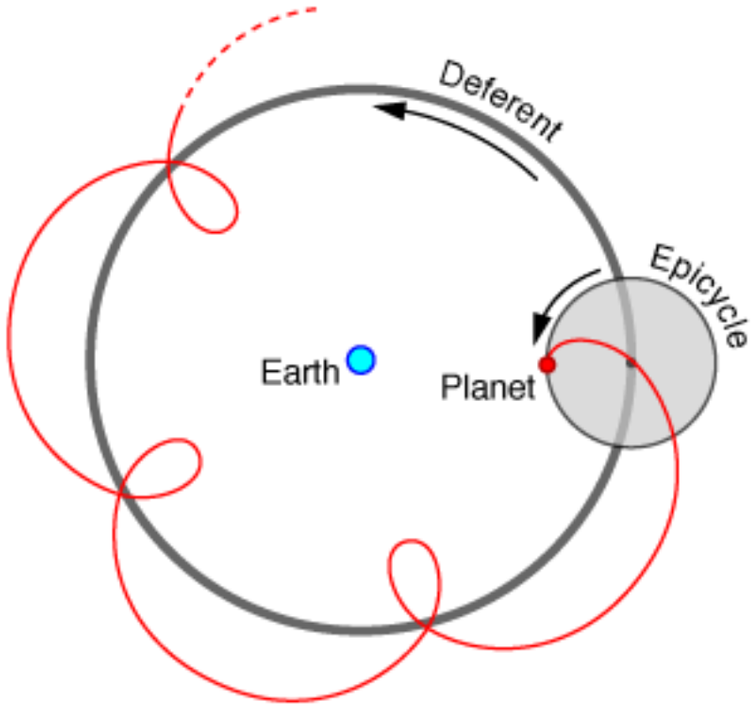
Then it will print 24. Two more pointers.

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
```

```cpp
#include <iostream>

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

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

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

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

```cpp
#include <iostream>

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

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

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

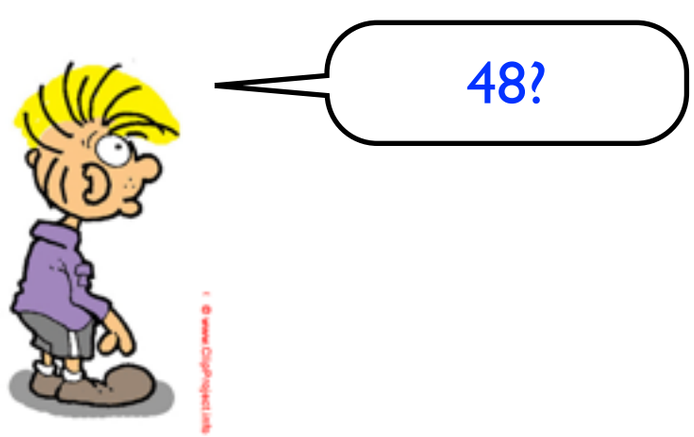

```
$ c++ foo.cpp && ./a.out
24
```

```cpp
#include <iostream>

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

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

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


```
$ 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;
}
```

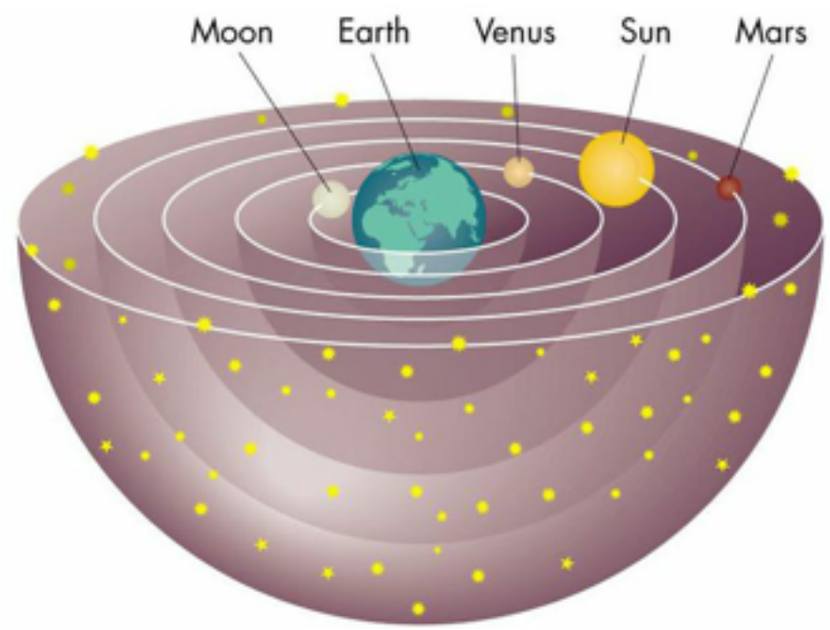


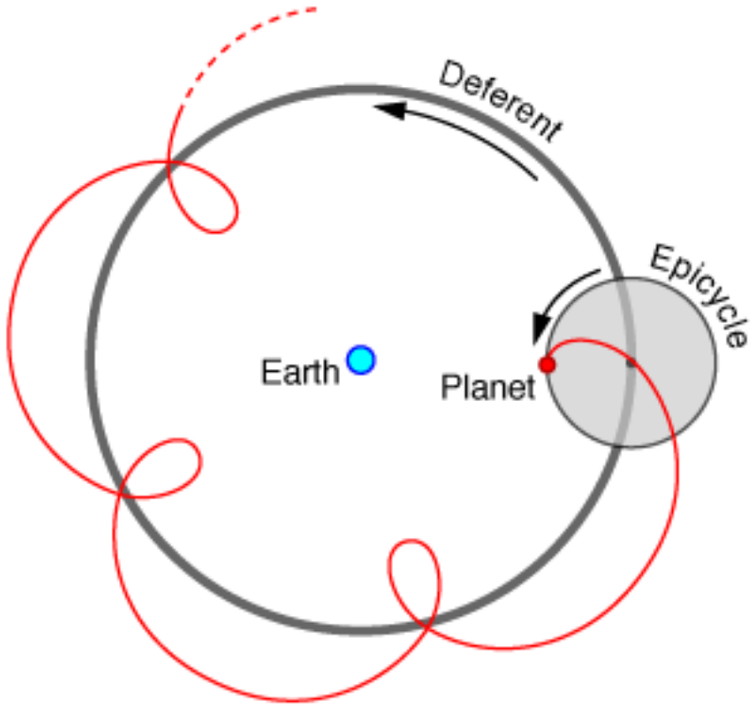
```cpp
#include <iostream>

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
```

```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

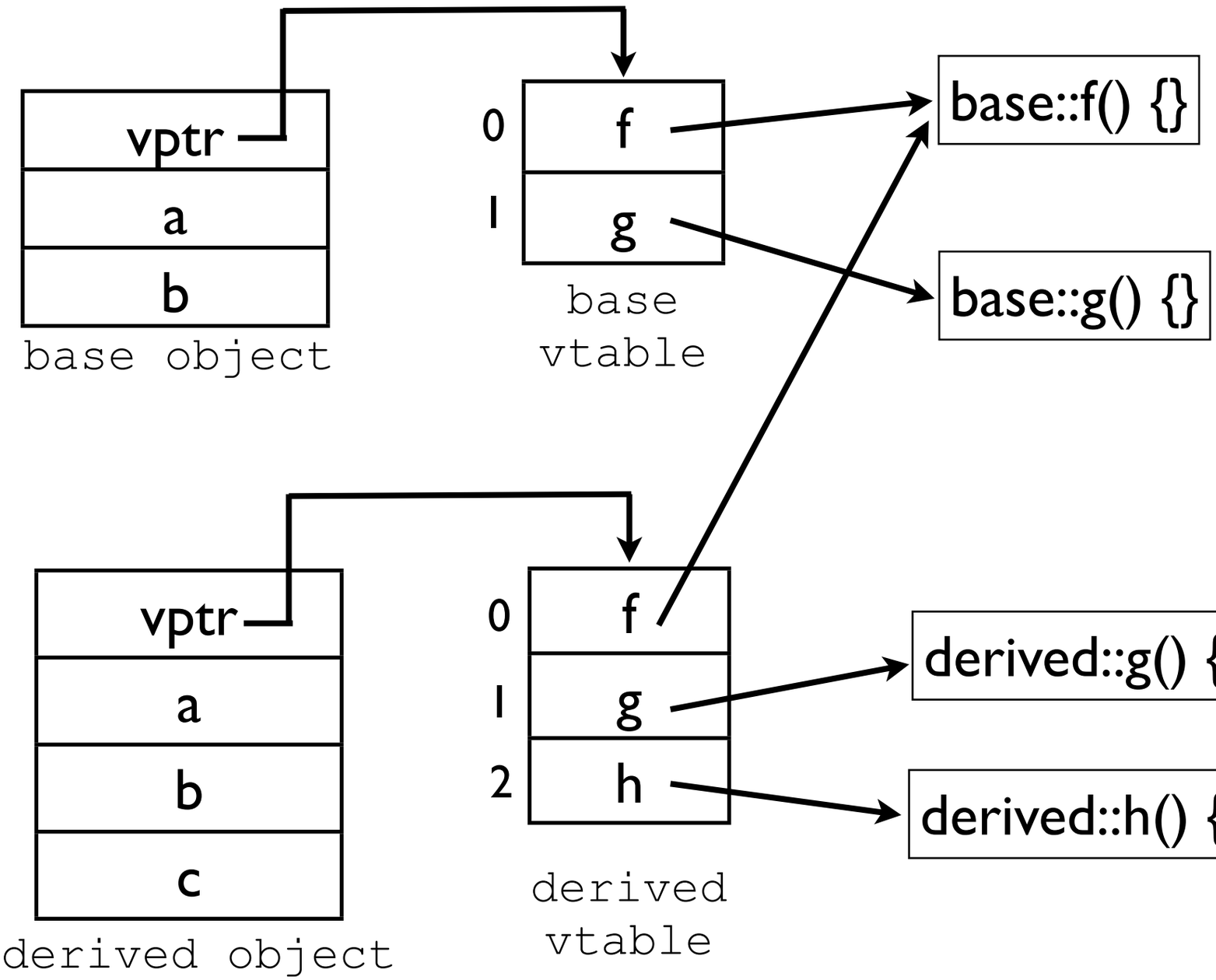

Deferent
Epicycle
Earth
Planet

```cpp
#include <iostream>

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

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

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

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

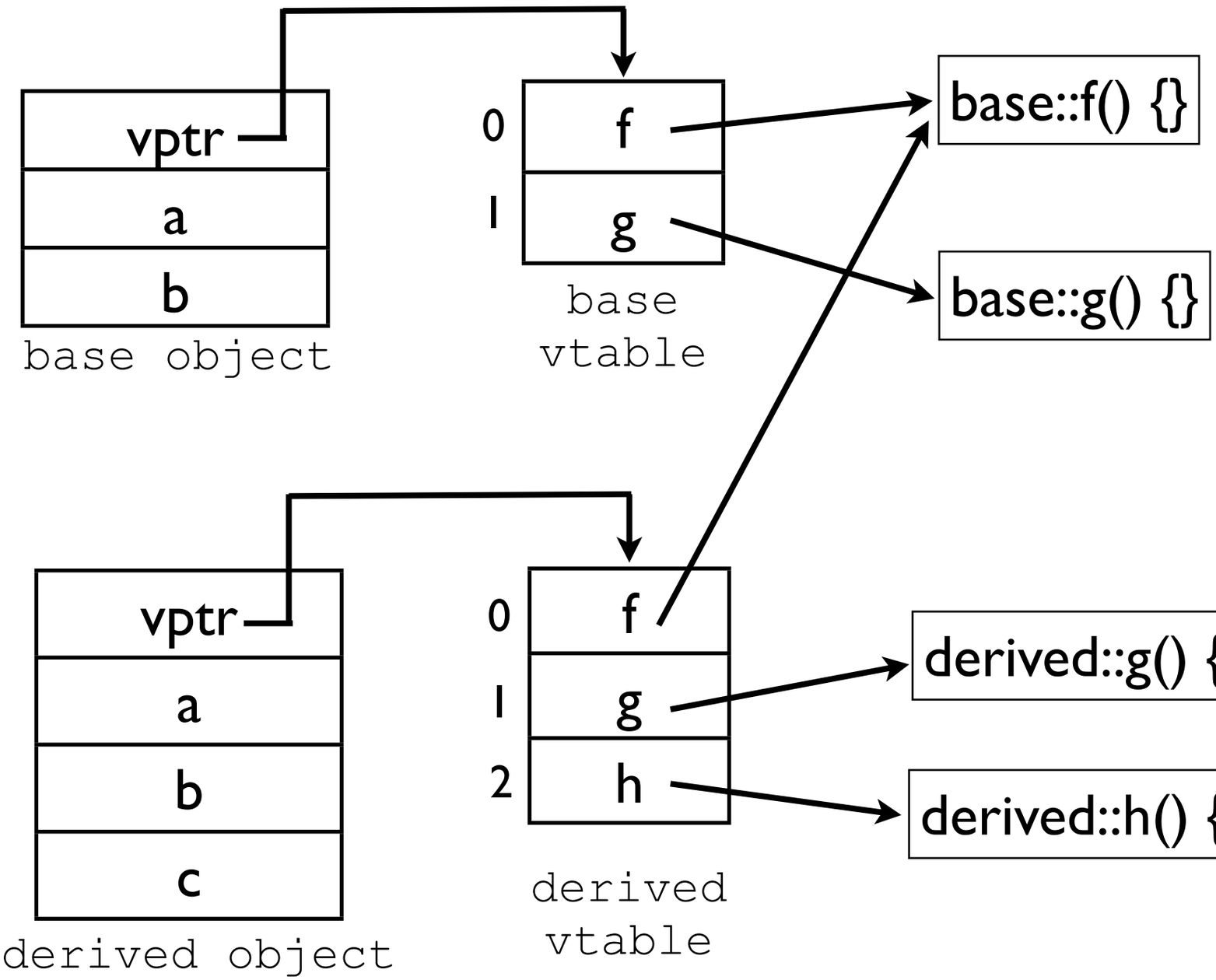
The vtable

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

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

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

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



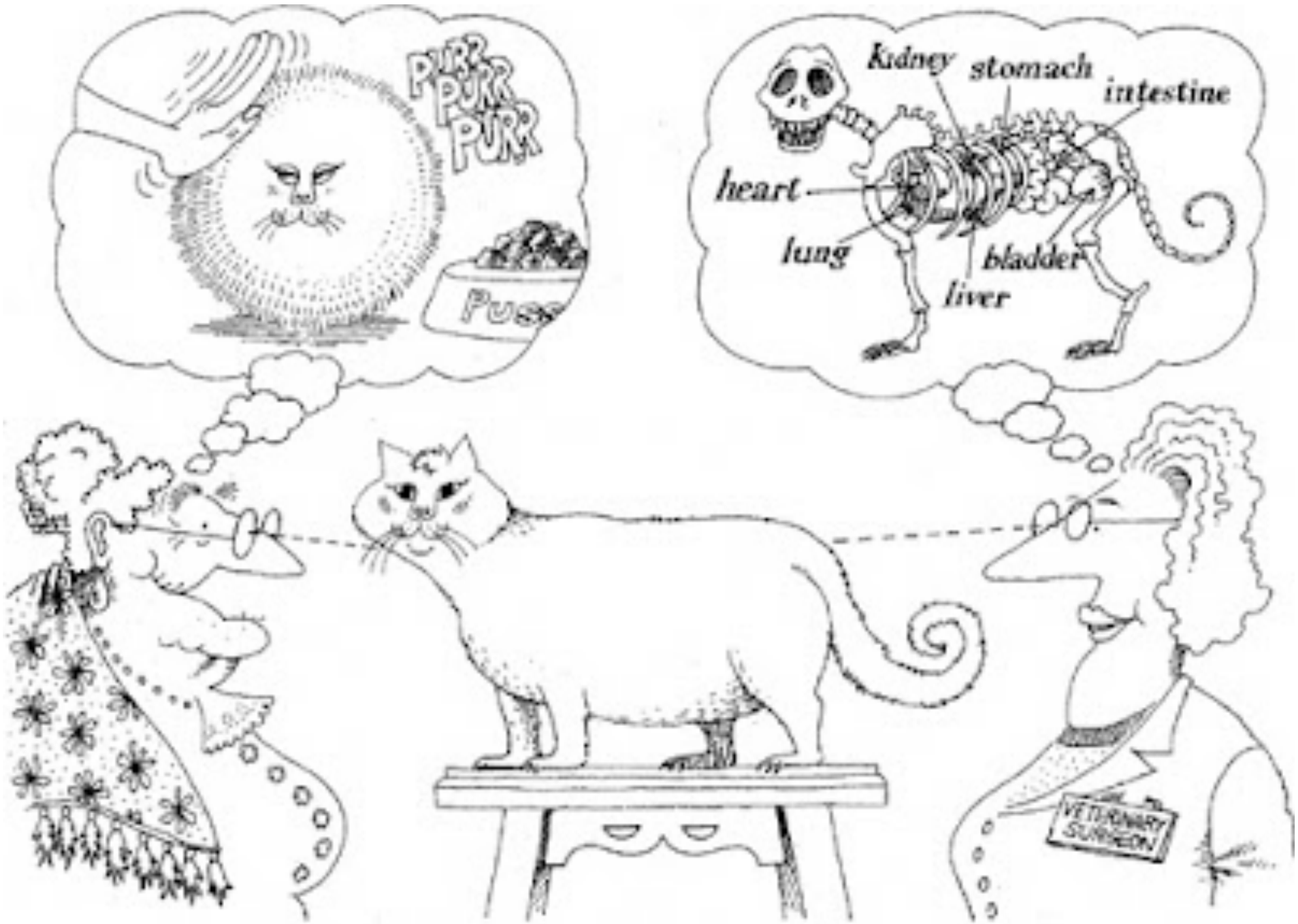
The vtable

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

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

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

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



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

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.

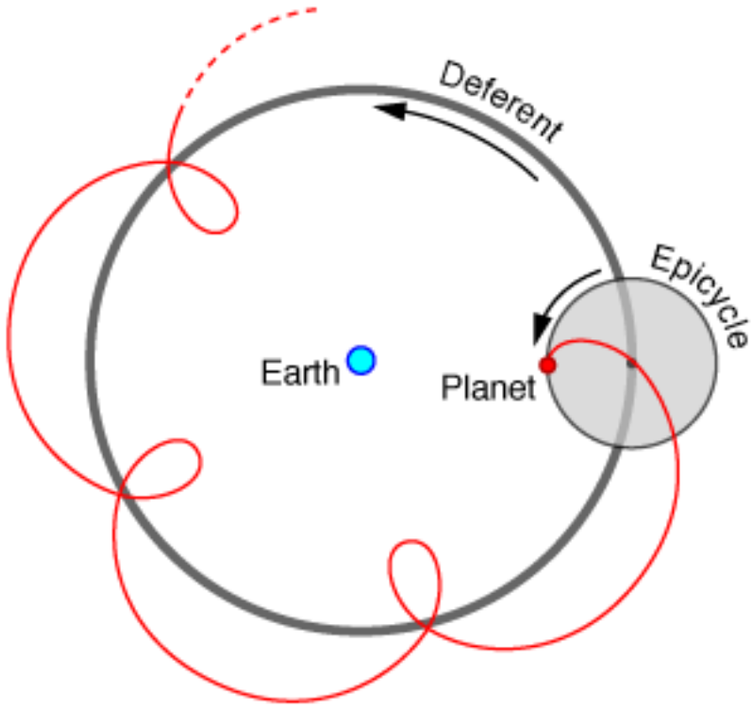
```
#include "B.hpp"

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

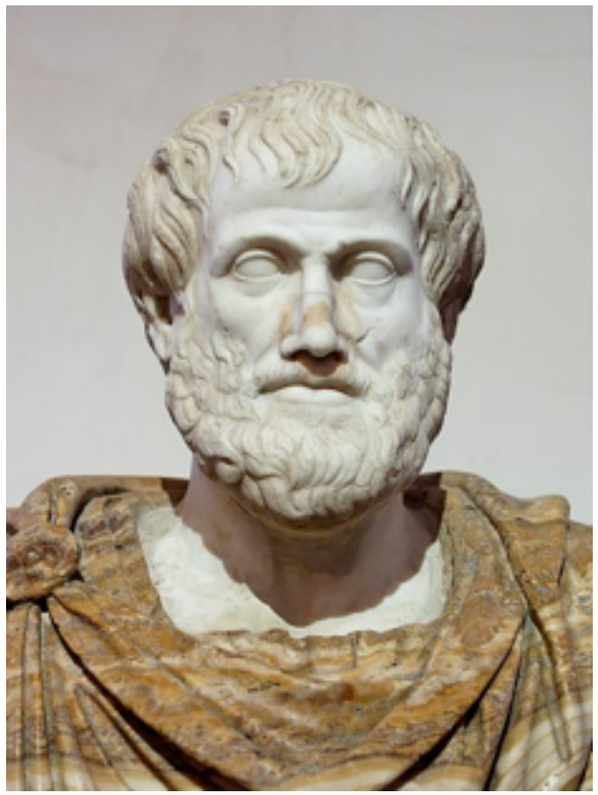
Summary

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

Summary

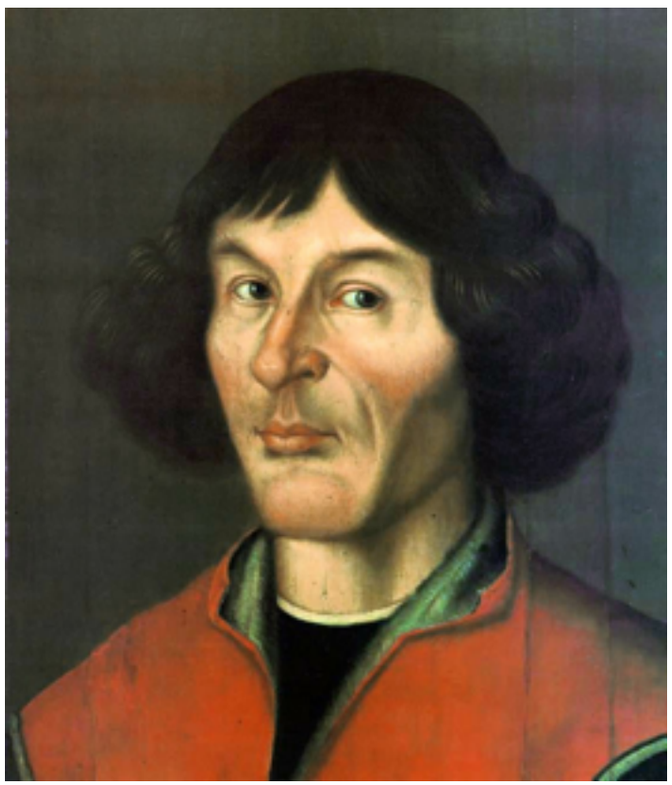
- memory model
- evaluation order
- sequence points
- undefined vs unspecified behavior
- optimization
- vtables
- object lifetimes
- rule of 3
- initialization of objects

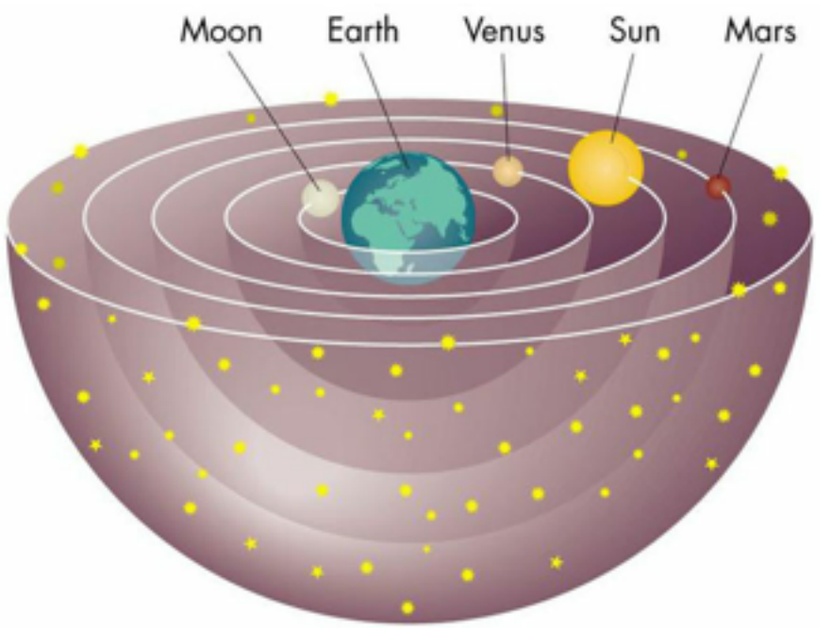
Aristotle (384 BC – 322 BC)



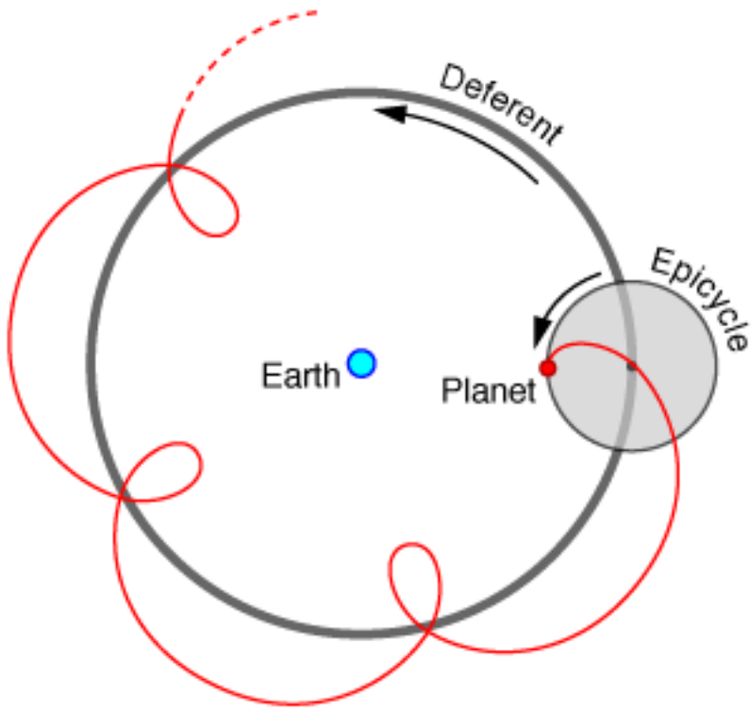
Ptolemy (90 AD – 168 AD)

Copernicus (1473 – 1543)

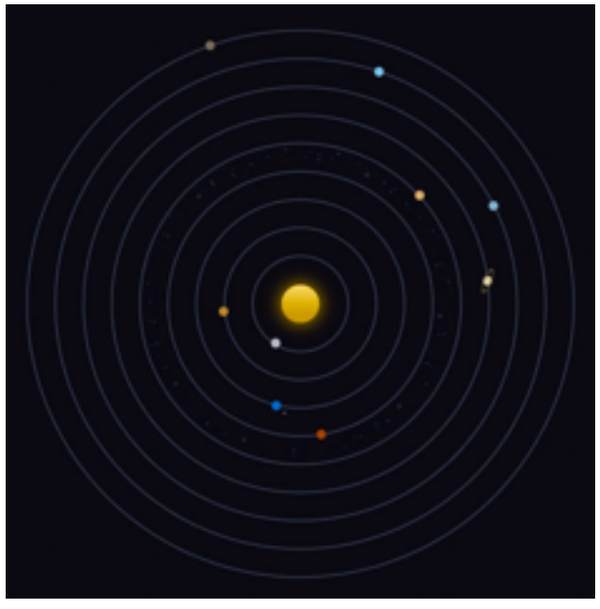




Aristotles Universe



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

Bonus material

A Brief Tour

tools and vocabulary

(a chapter from Deep C - a 3 day course by Jon Jagger & Olve Maudal)

not so

A Brief Tour

tools and vocabulary

(a chapter from Deep C - a 3 day course by Jon Jagger & Olve Maudal)

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
The answer is 42
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
The answer is 42
$ echo $?
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
The answer is 42
$ echo $?
0
```

Exercise: Hello World!

Type in this code. Compile and execute the program. What do you get?

hello.c

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
The answer is 42
$ echo $?
0
$
```

Was this the result you expected?

Was this the result you expected?

Of course it was!

Was this the result you expected?

Of course it was!

But let's do a dry run and step through the code

Was this the result you expected?

Of course it was!

But let's do a dry run and step through the code

(we will look at the compilation step later.)

Was this the result you expected?

Of course it was!

But let's do a dry run and step through the code

(we will look at the compilation step later.)

The following is an example of what ***might*** happen when executing this code:

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

After some basic initialization the run-time environment will call the main function.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

then preparation for the call to `calc()` starts by evaluating all the arguments to be passed to the function.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{                          ?      ?            ?
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

guess which argument is evaluated first?

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

unlike other popular programming laugages, the order of evaluation is mostly unspecified in C. In this case the compiler or runtime environment might choose to evaluate `life()` first

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe,   6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe,   6   , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe,   6   , everything());
    printf("The answer is %d\n", a);
}
```

`universe` is replaced with 7.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

now we are ready to call the `calc()` function. This can be done by pushing arguments on an execution stack, reserve space for the return value and perhaps some housekeeping values.

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

? ? ?

now we are ready to call the `calc()` function. This can be done by pushing arguments on an execution stack, reserve space for the return value and perhaps some housekeeping values.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

? ? ?

guess which argument is pushed first?

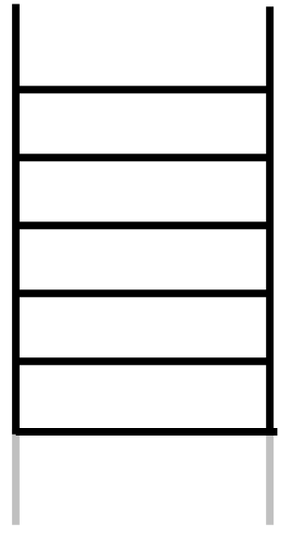
now we are ready to call the `calc()` function. This can be done by pushing arguments on an execution stack, reserve space for the return value and perhaps some housekeeping values.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...

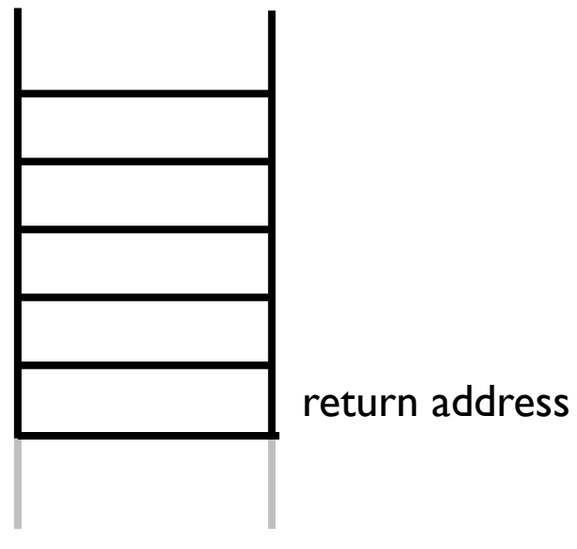
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...



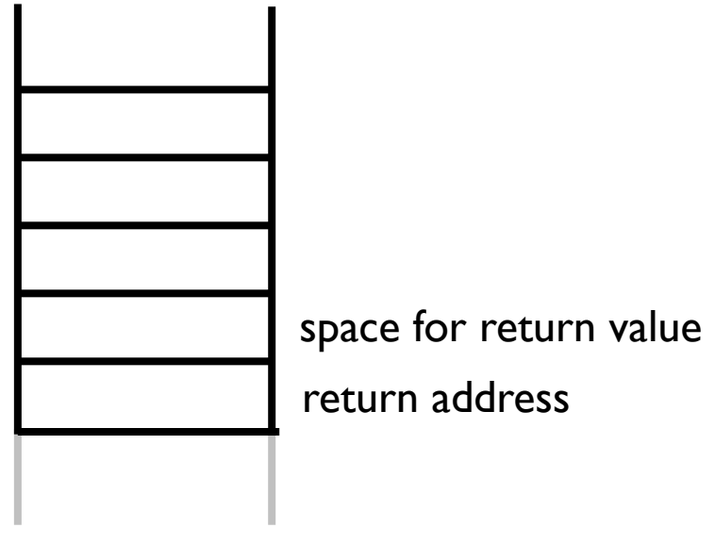
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...



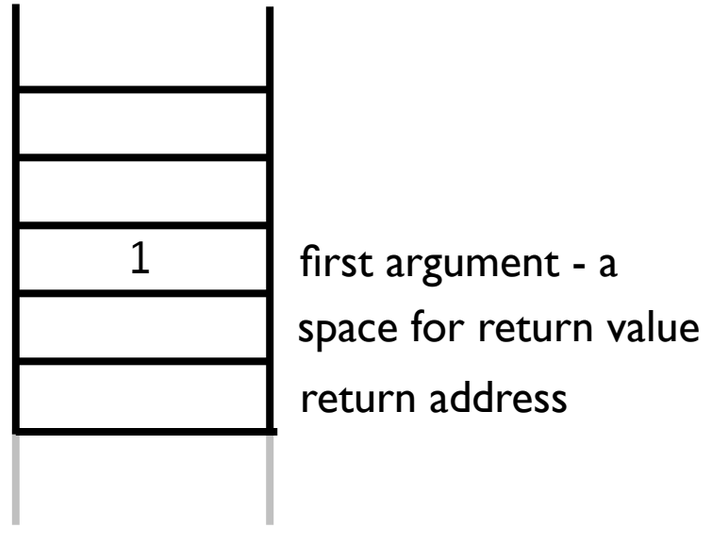
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...



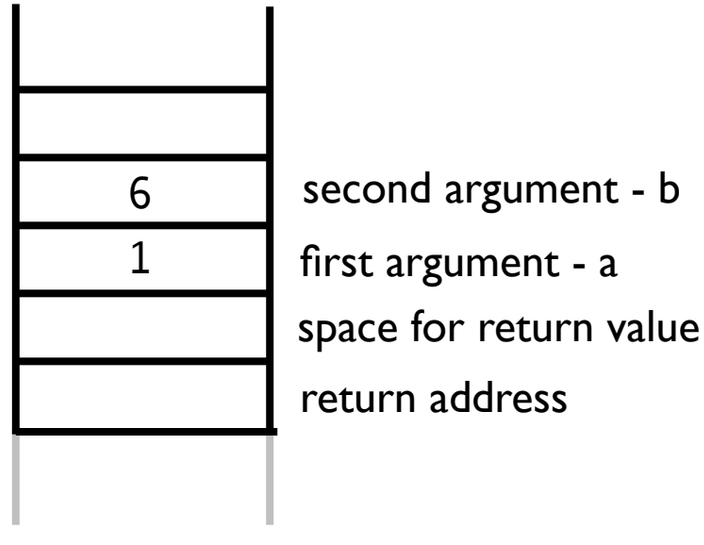
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...



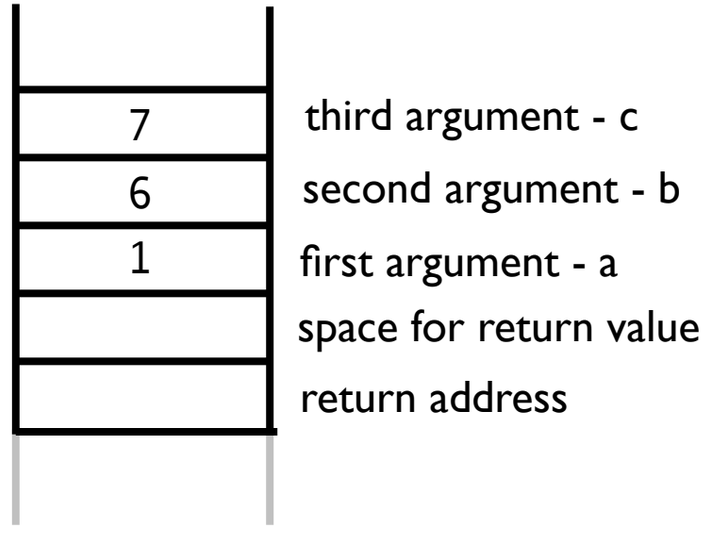
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...

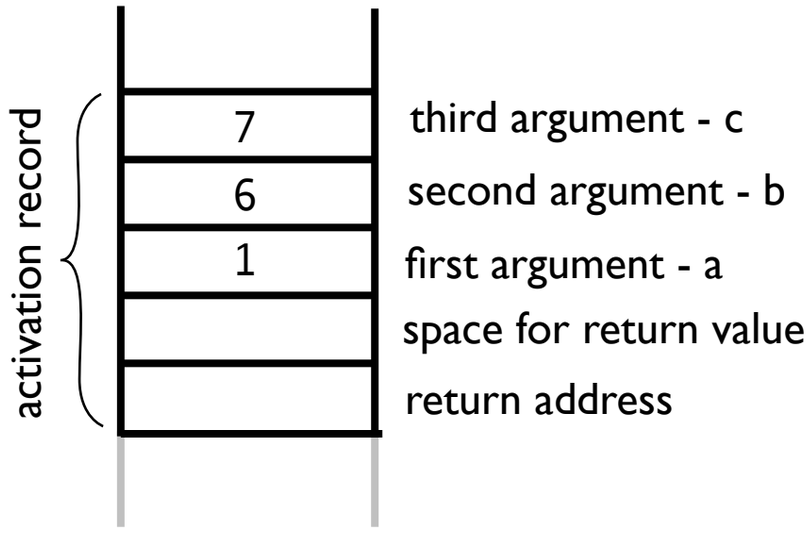


```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...

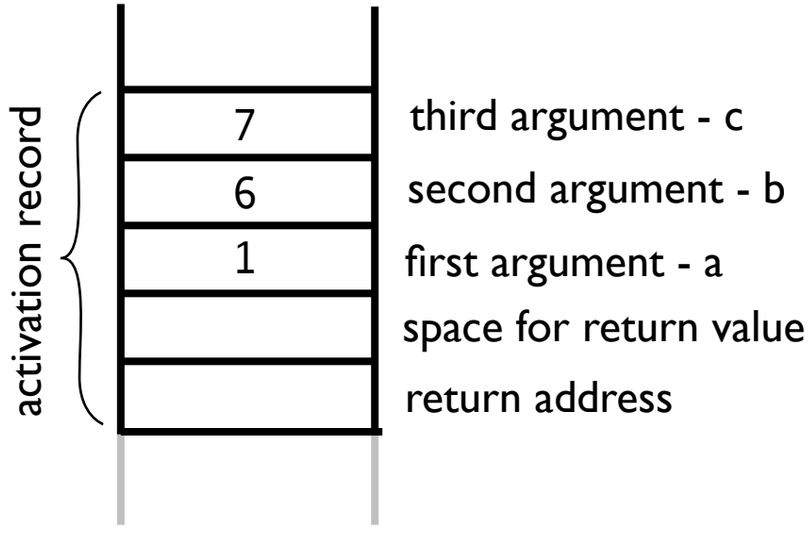


```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```



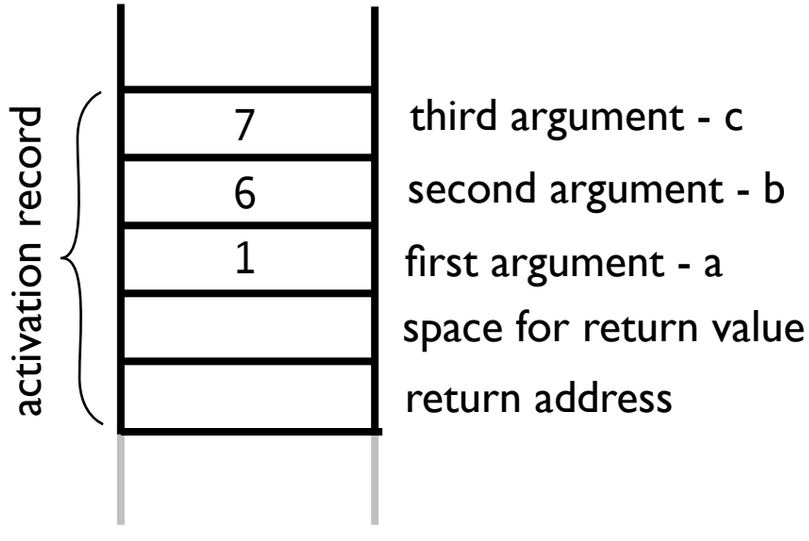
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

and when the "activation record" is populated, the program can jump into the function.



```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

and when the "activation record" is populated, the program can jump into the function.



```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

and then evaluate the expression.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return 7 * 6 / 1;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

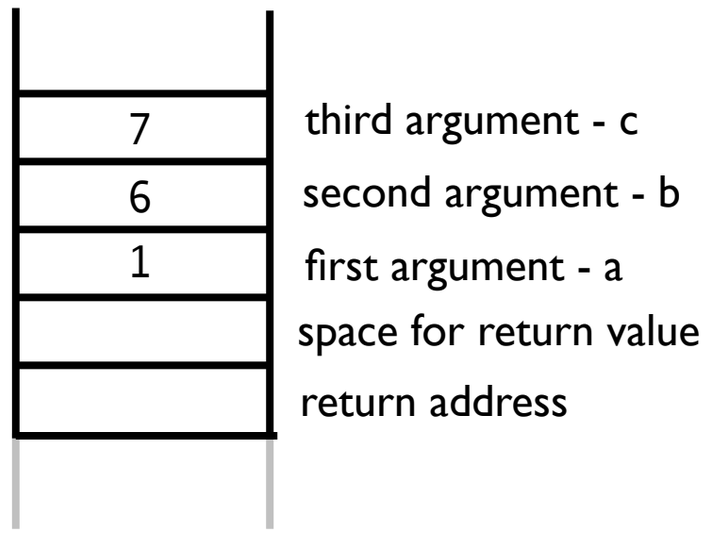
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return 7 * 6 / 1;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return   42  / 1;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

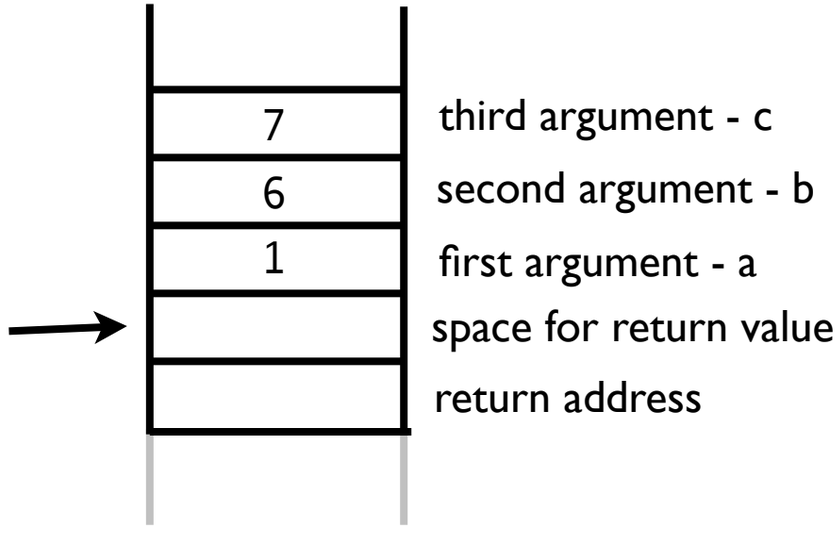
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```

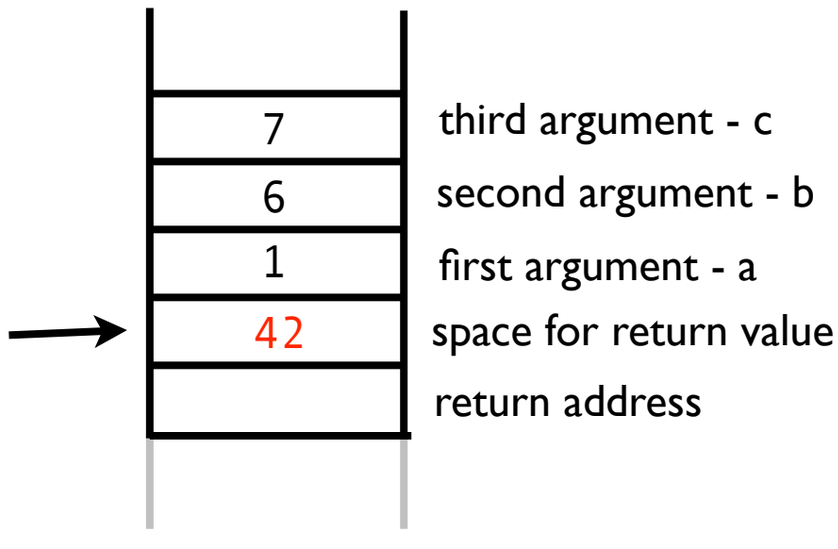
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1       );
    printf("The answer is %d\n", a);
}
```



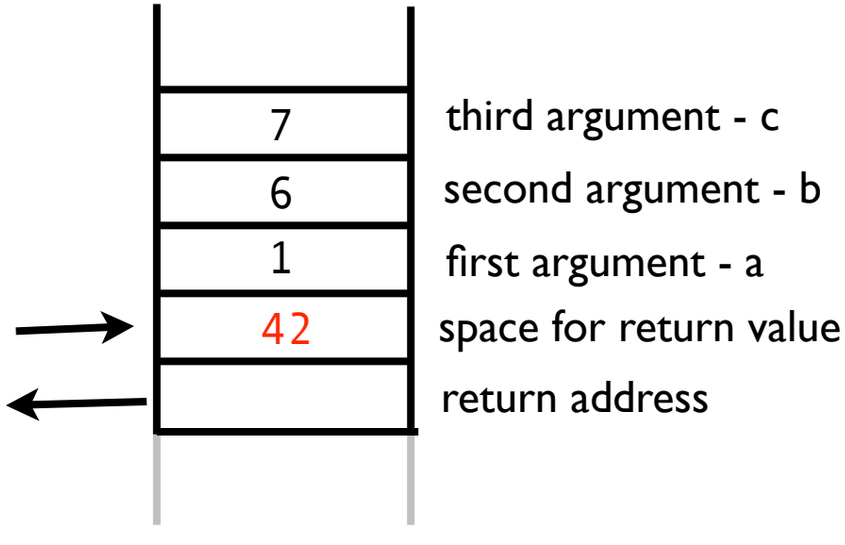
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1        );
    printf("The answer is %d\n", a);
}
```



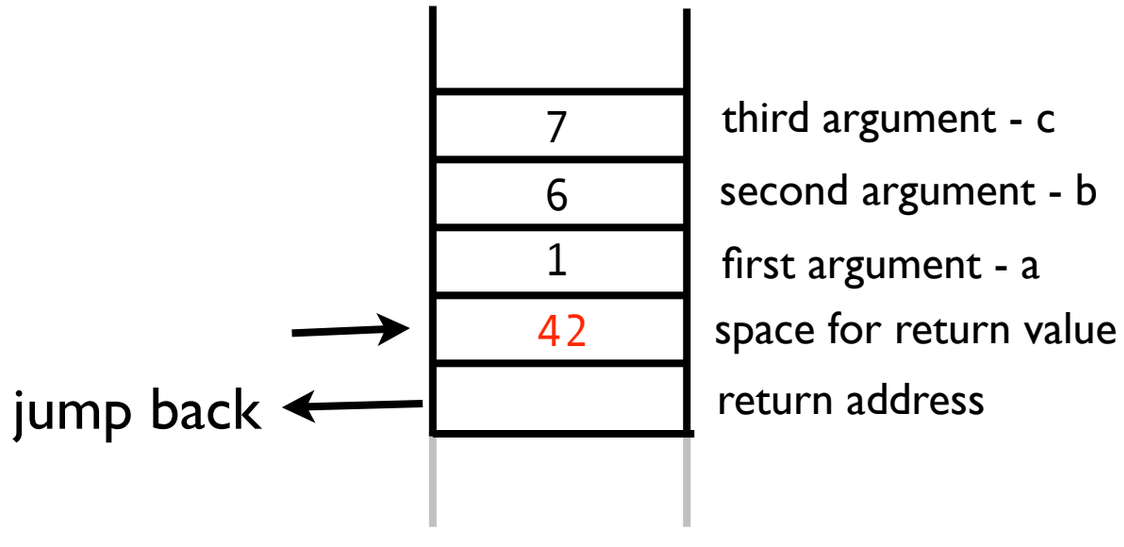
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1        );
    printf("The answer is %d\n", a);
}
```



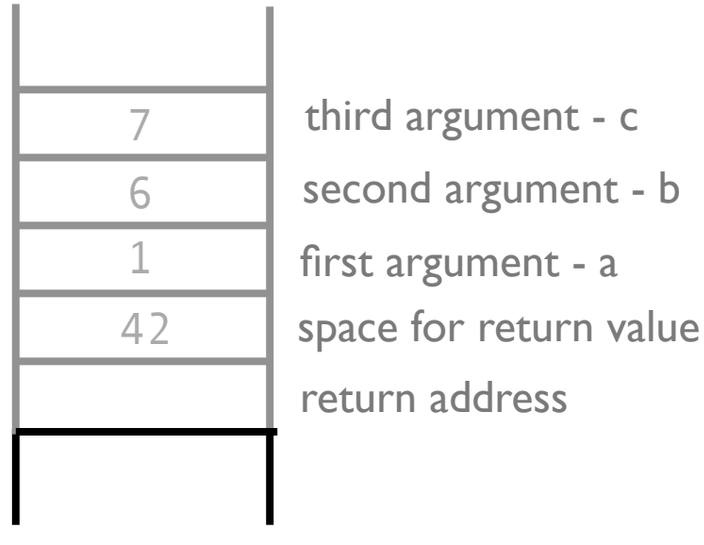
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1        );
    printf("The answer is %d\n", a);
}
```



```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1        );
    printf("The answer is %d\n", a);
}
```



```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,       1        );
    printf("The answer is %d\n", a);
}
```



```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

and then 42 and the pointer to the character string is pushed on the execution stack before the library function `printf()` is called

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

The `printf()` function writes out to the standard output stream

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

```
The answer is 42
```

The `printf()` function writes out to the standard output stream

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

```
The answer is 42
```

The `printf()` function writes out to the standard output stream

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

```
The answer is 42
$ echo $?
0
```

and a default value to indicate success, in this case 0, is returned back to the run-time environment

Was this exactly what you expected?

Was this exactly what you expected?

Good!

Was this exactly what you expected?

Good!

This was just an example of what ***might*** happen.

Was this exactly what you expected?

Good!

This was just an example of what ***might*** happen.

Because...

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

maybe the compiler is clever and see that `life()` and `everything()` always returns 6 and 1

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

maybe the compiler is clever and see that `life()` and `everything()` always returns 6 and 1

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

maybe the compiler is clever and see that `life()` and `everything()` always returns 6 and 1

and then by inlining `calc()` perhaps the compiler choose to optimize the code into...

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", a);
}
```

and since nobody else uses the functions perhaps the compiler decides to not create code for `life()` and `calc()`

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", a);
}
```

and since nobody else uses the functions perhaps the compiler decides to not create code for `life()` and `calc()`

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", a);
}
```

and since nobody else uses the functions perhaps the compiler decides to not create code for `life()` and `calc()`

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", a);
}
```

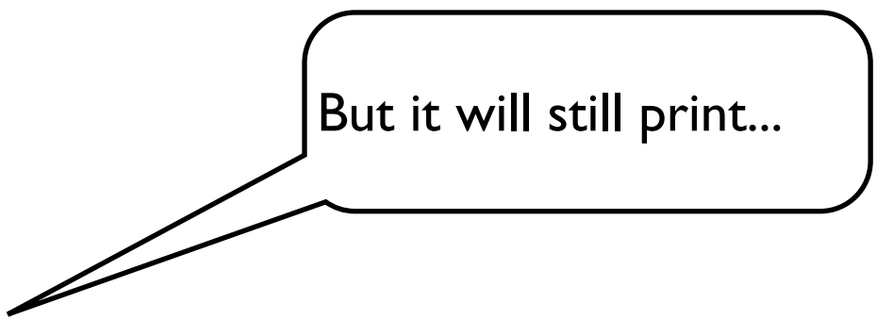
and since nobody else uses the functions perhaps the compiler decides to not create code for `life()` and `calc()`

and since variable `a` is used only here, then it might skip creating object a and just evaluate the expression as part of the `printf()` expression.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", universe * 6 / 1);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", universe * 6 / 1);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", universe * 6 / 1);
}
```



```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", universe * 6 / 1);
}
```

But it will still print...

```
The answer is 42
$ echo $?
0
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = universe * 6 / 1;
    printf("The answer is %d\n", universe * 6 / 1);
}
```

But it will still print...

```
The answer is 42
$ echo $?
0
```

The key take away from this session is that things like execution stack and calling conventions are not dictated by the standard. The evaluation order of expressions and arguments is mostly unspecified. And the optimizer might rearrange the execution of the code significantly. In C, nearly everything can happen, and will happen, internally as long as the external behavior is satisfied.

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 such a wide range of hardware!

Behavior

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

int main(void)
{
    // 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.

just to illustrate the point. What do you think will happen when we run this program?

```
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

just to illustrate the point. What do you think will happen when we run this program?

```
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

According to the C standard, this program will print:

just to illustrate the point. What do you think will happen when we run this program?

```
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

According to the C standard, this program will print:

```
ba7
```

just to illustrate the point. What do you think will happen when we run this program?

```
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

According to the C standard, this program will print:

```
ba7
```

or

```
ab7
```

just to illustrate the point. What do you think will happen when we run this program?

```c
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

According to the C standard, this program will print:

```
ba7
```

or

```
ab7
```

Unlike most modern programming languages, the evaluation order of most expressions are ***not*** specified in C

And while we are on a roll... What do you think might happen when we run this program?

And while we are on a roll... What do you think might happen when we run this program?

```
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

And while we are on a roll... What do you think might happen when we run this program?

```
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

you might get:

And while we are on a roll... What do you think might happen when we run this program?

```
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

you might get:

```
42
0
```

And while we are on a roll... What do you think might happen when we run this program?

```
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

you might get:

```
42
0
```

If you break the rules of the language, the behaviour of the whole program is undefined. Anything can happen! And the compiler will often not be able to give you a warning.

And while we are on a roll... What do you think might happen when we run this program?

```
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

you might get:

```
42
0
```

Try it!

If you break the rules of the language, the behaviour of the whole program is undefined. Anything can happen! And the compiler will often not be able to give you a warning.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

The violation here is that we are violating the rules of sequencing, which, among other things says that a variable can not be updated twice between two sequence points.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

Here is what I get on my machine:

```
$ gcc -O -Wall -Wextra -pedantic foo.c
$ ./a.out
0
4
$
```

The violation here is that we are violating the rules of sequencing, which, among other things says that a variable can not be updated twice between two sequence points.

We just had a glimpse of what ***might*** happen when code is executed.

Scary stuff? Not really, but with only a shallow understanding of the language it is easy to make big mistakes.

The goal of this course is to give you a deep understanding of C, by starting from scratch an relearn the language in a systematic way.

First, lets start to establish a vocabulary. To do that, we need an even more contrived example program...

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
$ cc foo.c
$ ./a.out
Hello everyone
$ echo $?
0
```

Why do we need this?

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
$ cc foo.c
$ ./a.out
Hello everyone
$ echo $?
0
```

Why do we need this?

and what does this mean?

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
$ cc foo.c
$ ./a.out
Hello everyone
$ echo $?
0
```

Why do we need this?

and what does this mean?

how is a for loop *really* working?

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
$ cc foo.c
$ ./a.out
Hello everyone
$ echo $?
0
```

Why do we need this?

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

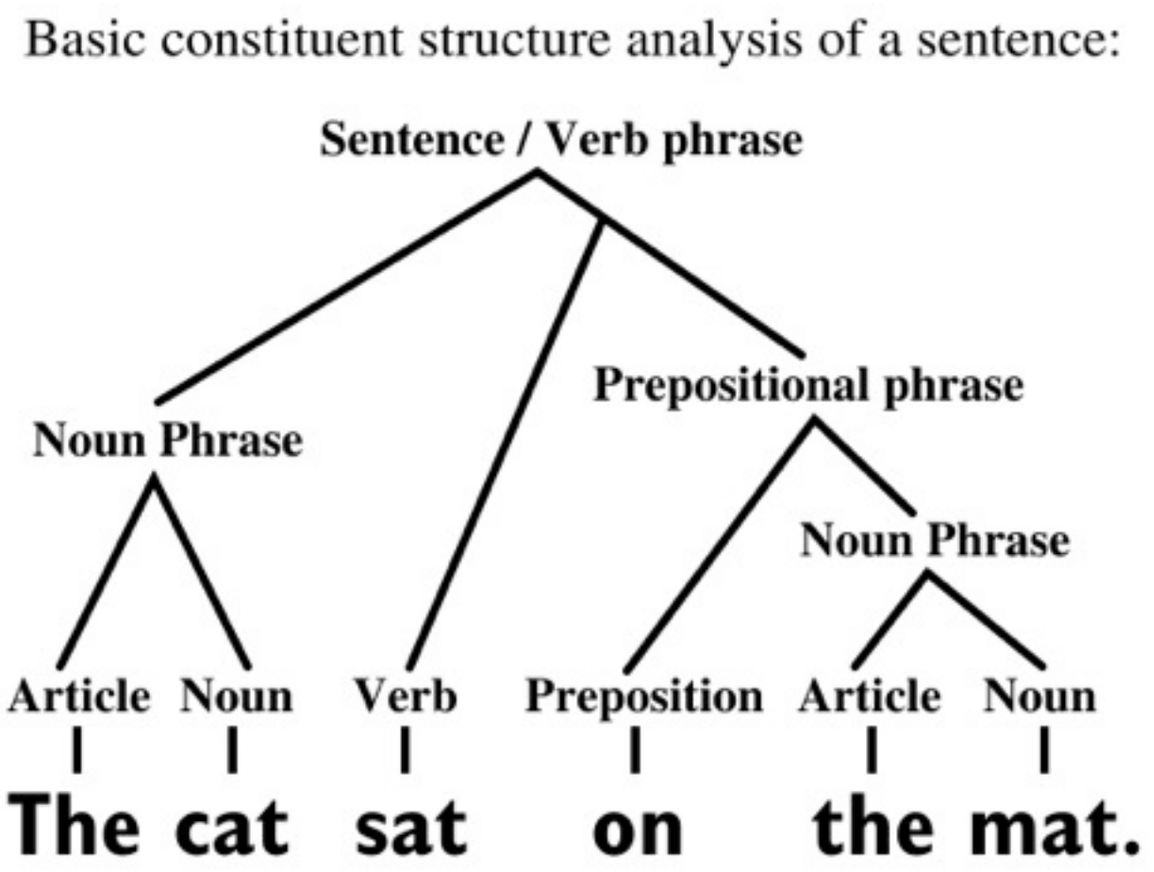
and what does this mean?

how many arguments does this function accept?

how is a for loop *really* working?

```
$ cc foo.c
$ ./a.out
Hello everyone
$ echo $?
0
```

To get a deep understanding of any language you need to be able to ‘break’ it down and analyse it, at least you need to recognize the words used when experts are discussing among themselves



http://en.wikipedia.org/wiki/File:Basic_constituent_structure_analysis_English_sentence.svg

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

keyword

keyword

keyword

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

keyword

keyword

keyword

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

function prototype

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

function prototype

function definition

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

function prototype (declaration)

function definition

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

identifier with internal linkage

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

linkage specification

identifier with internal linkage

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

linkage specification

identifier with internal linkage

identifier with external linkage

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration with initialization

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration with initialization

declaration without initialization

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration with initialization

declaration without initialization

expression

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration without initialization

declaration with initialization

assignment expression

expression

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration without initialization

declaration with initialization

assignment expression

expression

expression statement

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

declaration without initialization

declaration with initialization

expression

assignment expression

expression statement

statement

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

type specifier

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

type qualifier

type specifier

storage class specifier

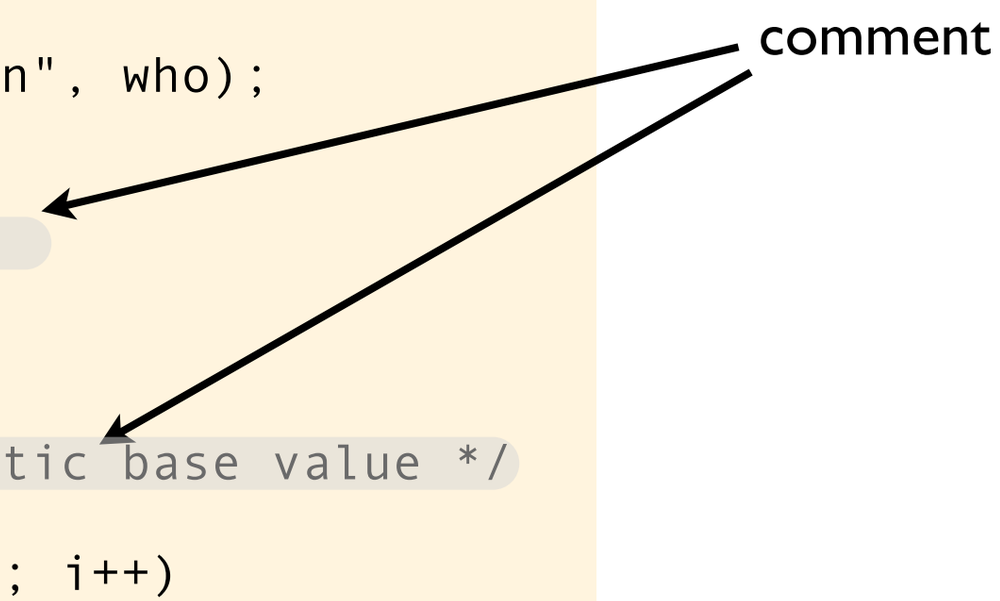
type qualifier

type specifier

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```



pre-processor directive

comment

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

pre-processor directive

comment

macro name

function prototype

pre-processor directive

linkage specification

declaration without initialization

function definition

comment

internal linkage

external linkage

expression

storage class specifier

declaration with initialization

expression statement

assignment expression

keyword

type qualifier

statement

type specifier

macro name

```c
int printf(const char * restrict, ...);
void exit(int);
#define EXIT_FAILURE 1

static void say_hello(const char * who)
{
    printf("Hello %s\n", who);
}

// compute the answer
int the_answer()
{
    register int a;
    int b = 6; /* mystic base value */
    a = 0;
    for (int i=0; i<7; i++)
        a += b;
    return a;
}

const int life_universe_everything = 42;

int main(void)
{
    say_hello("everyone");
    int a = the_answer();
    if (a != life_universe_everything)
        exit(EXIT_FAILURE);
}
```

A glimpse into tools often used when developing C

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
           answer);
}
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -c dt.c
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -c dt.c
$ cc -c theanswer.c
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
            answer);
}
```

```
$ cc -c dt.c
$ cc -c theanswer.c
$ cc -o theanswer theanswer.o dt.o
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
            answer);
}
```

```
$ cc -c dt.c
$ cc -c theanswer.c
$ cc -o theanswer theanswer.o dt.o
$ ./theanswer
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
            answer);
}
```

```
$ cc -c dt.c
$ cc -c theanswer.c
$ cc -o theanswer theanswer.o dt.o
$ ./theanswer
The answer is 42
```

Exercise: Deep thought, Part 1

dt.c

```
#include "dt.h"

int dt_base_value;
#define MULTIPLIER 7
static int dt_answer;

static void run_computer(void)
{
    dt_answer = dt_base_value * MULTIPLIER;
}

int dt_get_answer(void)
{
    run_computer();
    return dt_answer;
}
```

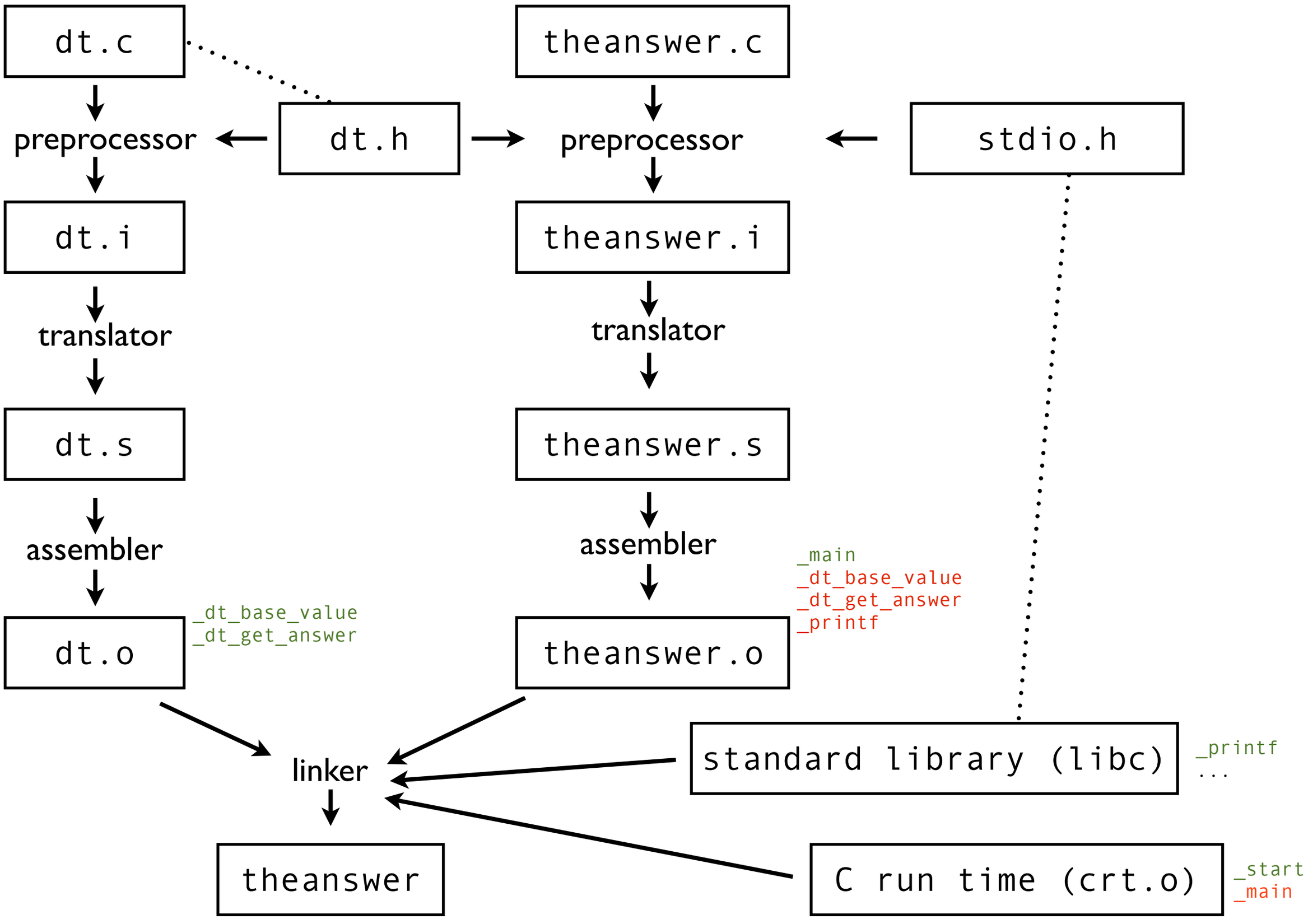
dt.h

```
extern int dt_base_value;
int dt_get_answer(void);
```

theanswer.c

```
#include "dt.h"
#include <stdio.h>

int main(void)
{
    dt_base_value = 6;
    int answer = dt_get_answer();
    printf("The answer is %d\n",
            answer);
}
```

```
$ cc -c dt.c
$ cc -c theanswer.c
$ cc -o theanswer theanswer.o dt.o
$ ./theanswer
The answer is 42
$
```


dt.c
dt.h
preprocessor
dt.i
translator
dt.s
assembler
dt.o
_dt_base_value
_dt_get_answer
theanswer.c
preprocessor
theanswer.i
translator
theanswer.s
assembler
theanswer.o
_main
_dt_base_value
_dt_get_answer
_printf
stdio.h
standard library (libc)
_printf
...
linker
theanswer
C run time (crt.o)
_start
_main

Exercise: Deep thought, Part 2

dt.c

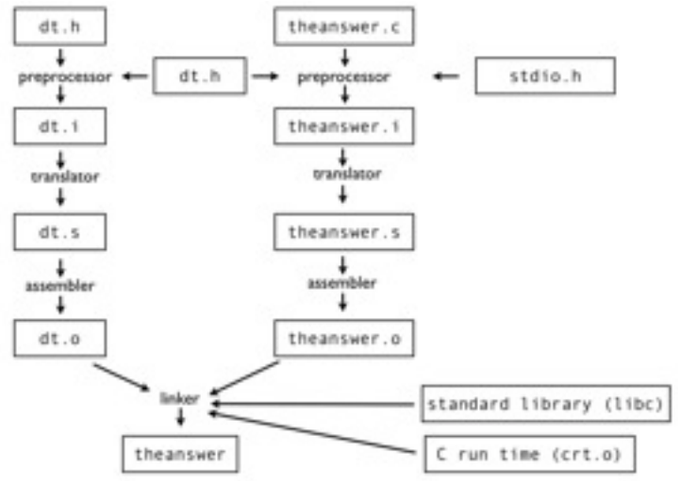
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```



Exercise: Deep thought, Part 2

dt.c

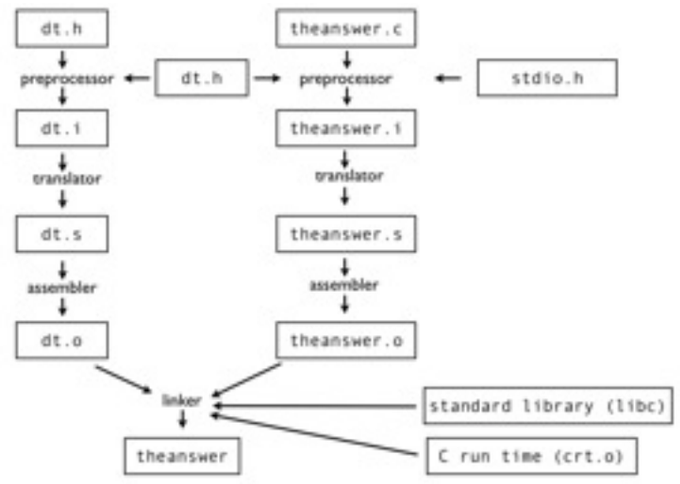
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
```

Exercise: Deep thought, Part 2

dt.c

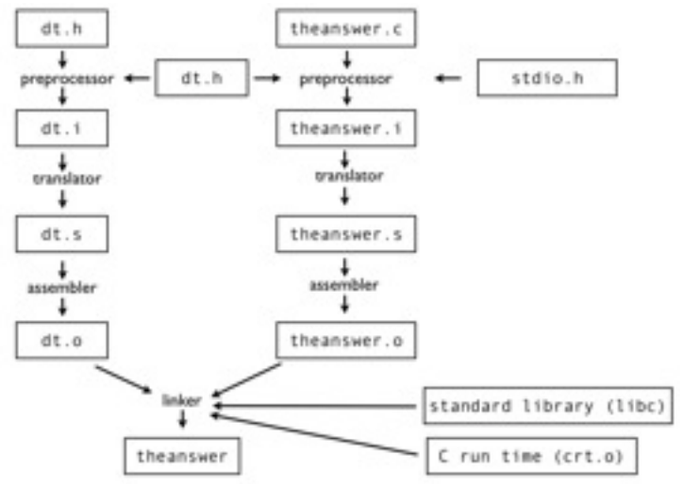
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
```

Exercise: Deep thought, Part 2

dt.c

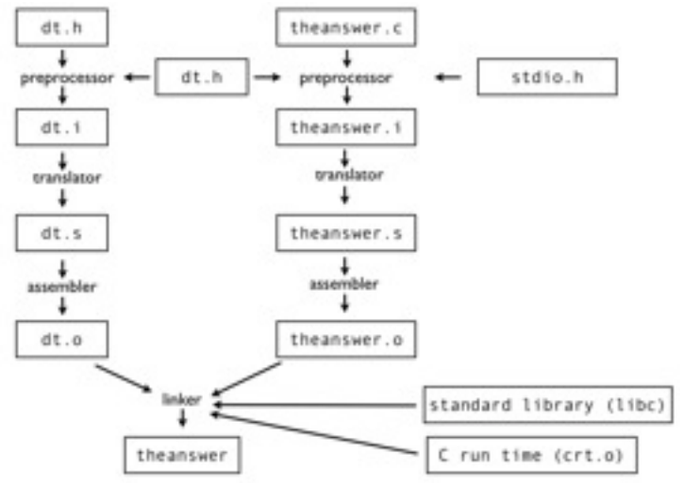
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
```

Exercise: Deep thought, Part 2

dt.c

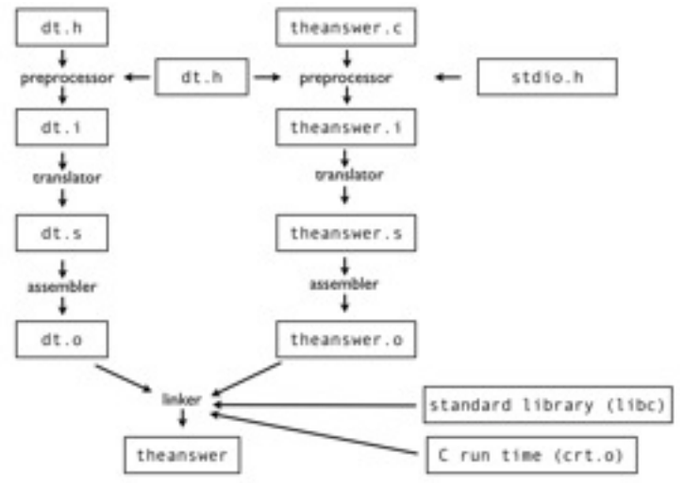
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
```

Exercise: Deep thought, Part 2

dt.c

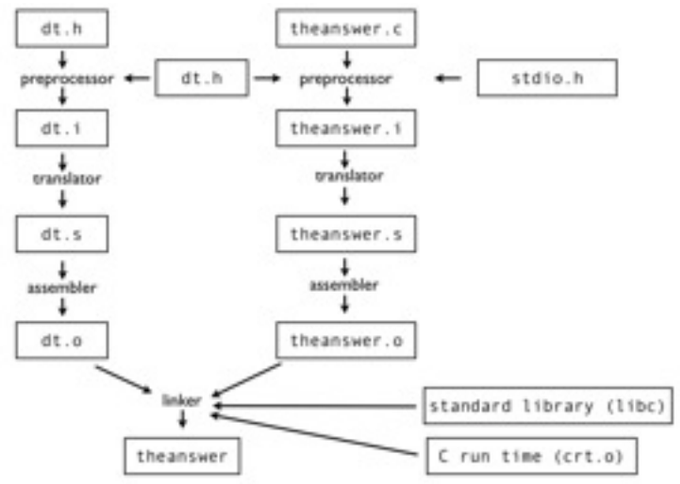
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
```

Exercise: Deep thought, Part 2

dt.c

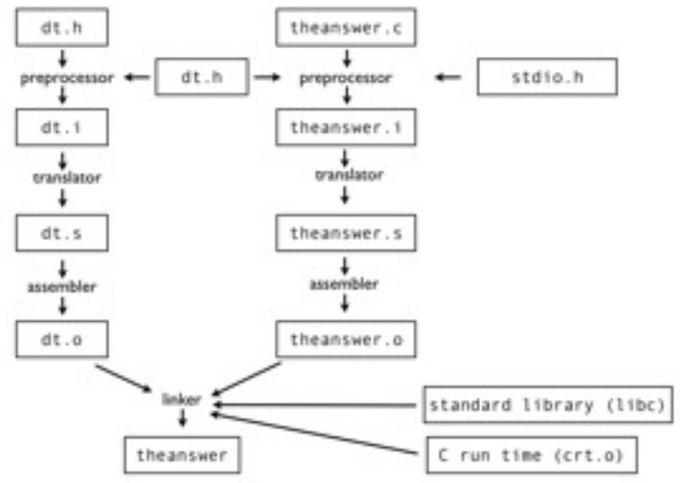
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o
```

Exercise: Deep thought, Part 2

dt.c

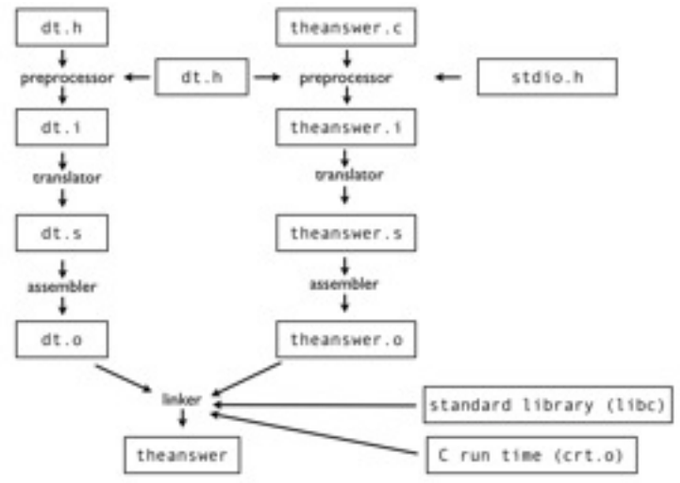
```c
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```c
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```c
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
```

Exercise: Deep thought, Part 2

dt.c

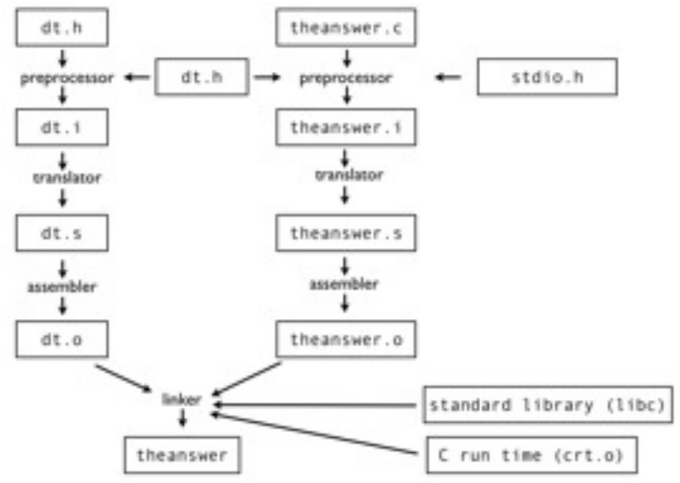
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
$ ls theanswer.*
```

Exercise: Deep thought, Part 2

dt.c

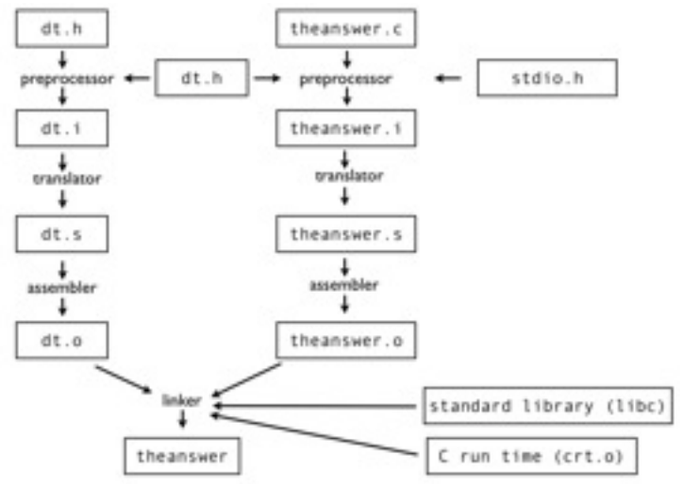
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
$ ls theanswer.*
$ nm theanswer.o
```

Exercise: Deep thought, Part 2

dt.c

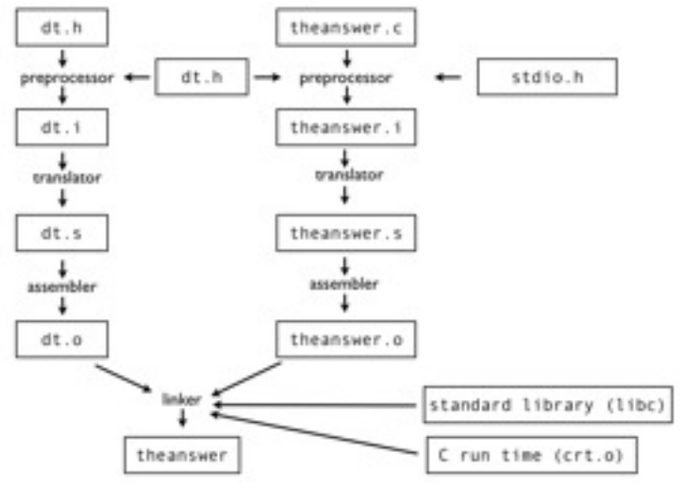
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
$ ls theanswer.*
$ nm theanswer.o

$ ld -lc -o theanswer dt.o theanswer.o /usr/lib/crt1.o
```

Exercise: Deep thought, Part 2

dt.c

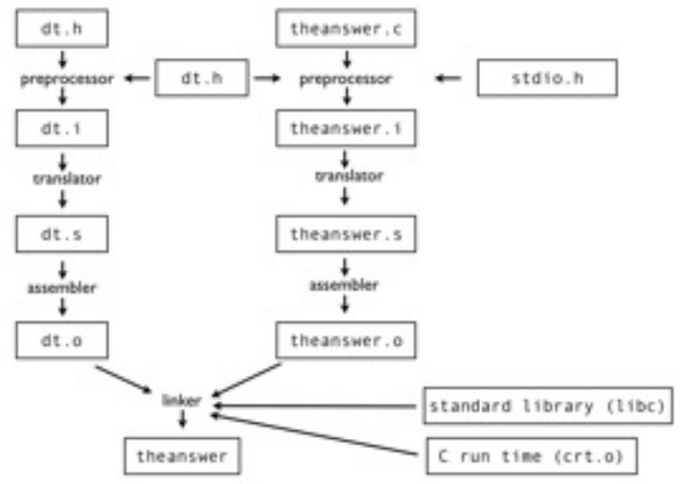
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
$ ls theanswer.*
$ nm theanswer.o

$ ld -lc -o theanswer dt.o theanswer.o /usr/lib/crt1.o
$ ./theanswer
```

Exercise: Deep thought, Part 2

dt.c

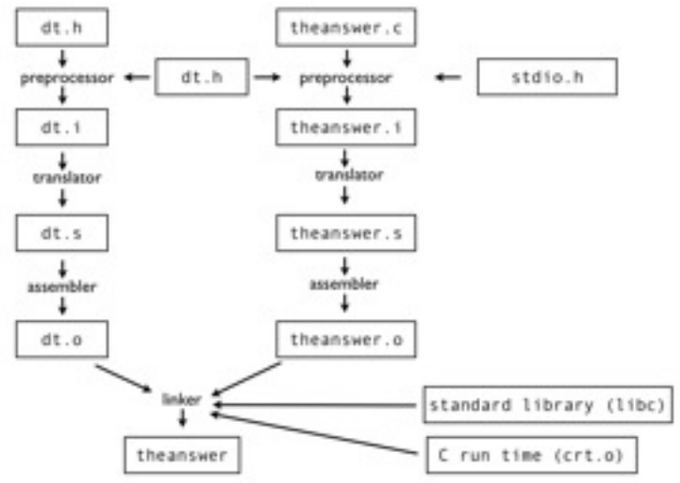
```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```




```
$ cc -E dt.c >dt.i
$ cat dt.i
$ cc -S dt.i
$ cat dt.s
$ cc -c dt.s
$ nm dt.o

$ cc -c -save-temps theanswer.c
$ ls theanswer.*
$ nm theanswer.o

$ ld -lc -o theanswer dt.o theanswer.o /usr/lib/crt1.o
$ ./theanswer
The answer is 42
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
(gdb) disassemble run_computer
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
(gdb) disassemble run_computer
(gdb) set disassembly-flavor intel
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
(gdb) disassemble run_computer
(gdb) set disassembly-flavor intel
(gdb) disassemble run_computer
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
(gdb) disassemble run_computer
(gdb) set disassembly-flavor intel
(gdb) disassemble run_computer
(gdb) help
```

Exercise: Deep thought, Part 3

dt.c

```
#include "dt.h"

int dt_base_value;
static int dt_answer;

static void run_computer(int multiplier)
{
    dt_answer = dt_base_value * multiplier;
}

void dt_init(void)
{
    dt_base_value = 6;
}

int dt_compute_answer(void)
{
    run_computer(7);
    return dt_answer;
}
```

dt.h

```
void dt_init(void);
int dt_compute_answer(void);
```

theanswer.c

```
#include <stdio.h>
#include "dt.h"

int main(void)
{
    dt_init();
    int answer = dt_compute_answer();
    printf("The answer is %d\n",
           answer);
}
```

```
$ cc -g -o theanswer dt.c theanswer.c
$ gdb theanswer
(gdb) run
(gdb) break run_computer
(gdb) set dt_base_value = 8
(gdb) cont
(gdb) disassemble run_computer
(gdb) set disassembly-flavor intel
(gdb) disassemble run_computer
(gdb) help
(gdb) quit
```

A few words about memory, activation frames and storage durations

Memory Layout and Activation Record

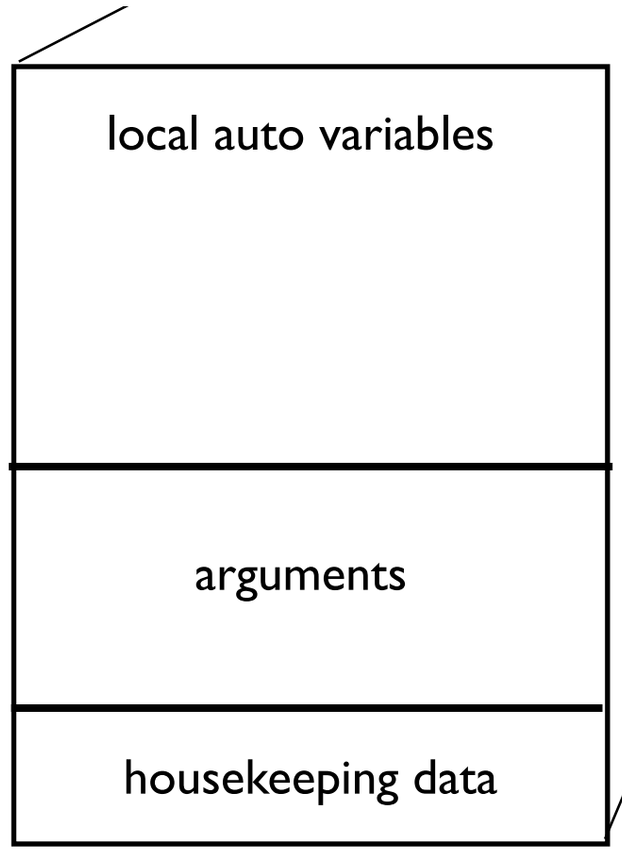
(This is a simplified view of how a possible memory layout might look like. Some architectures and run-time environment uses a very different layout and the C standard does not dictate how it should look like. However, without actually knowing the memory layout on your target machine, the description here is good enough as a conceptual model)

Under the hood

Memory Layout *

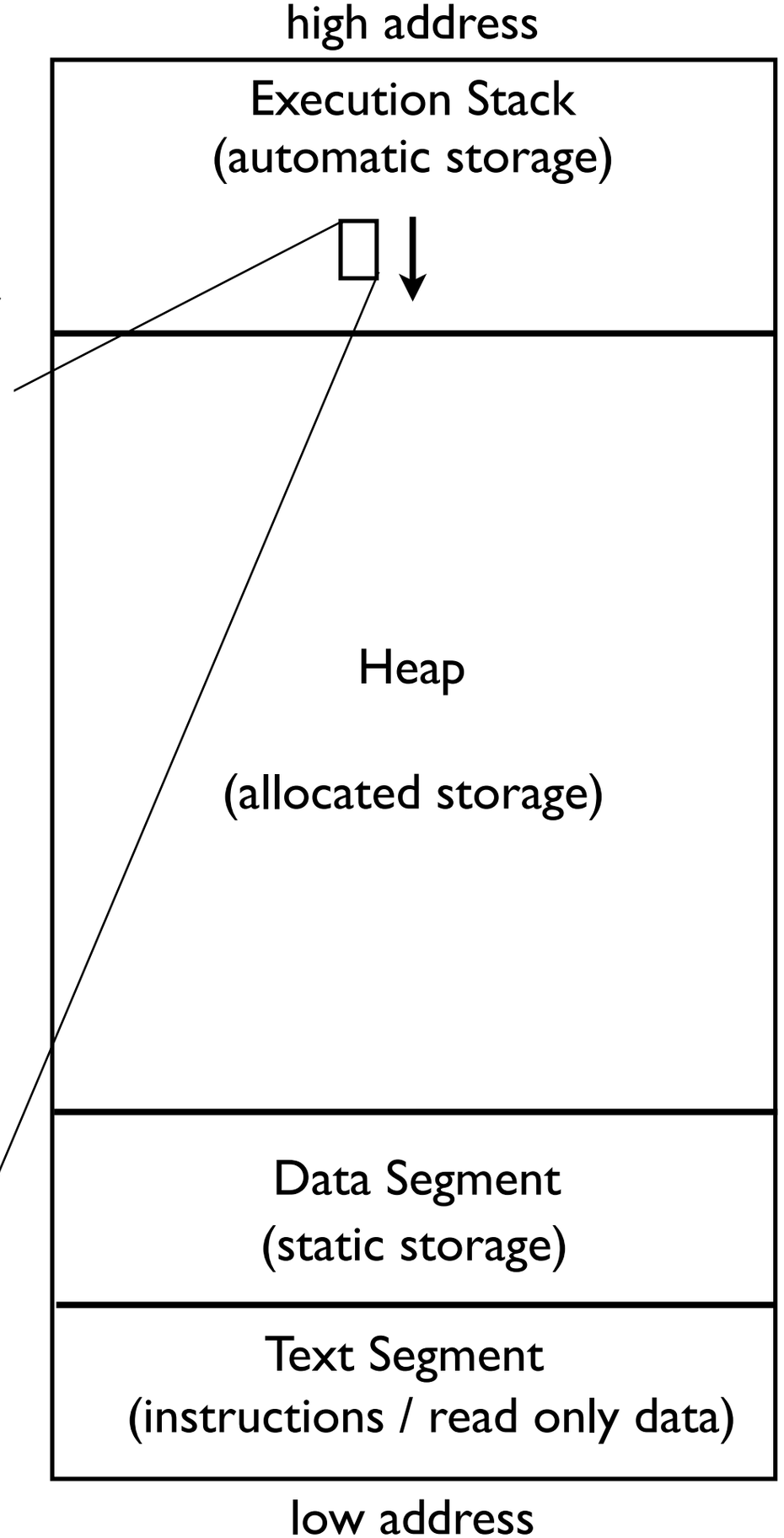
It is sometimes useful to assume that a C program uses a memory model where the instructions are stored in a **text segment**, and static variables are stored in a **data segment**. Automatic variables are allocated when needed together with housekeeping variables on an **execution stack** that is growing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

Automatic storage duration

Variables declared inside a “block”, eg the function body, come into existence when entering the block and disappears when exiting the block.* These objects have automatic storage class and they get an indeterminate initial value. Reading an indeterminate value causes undefined behavior. Referring to an object outside it’s lifetime also causes undefined behavior.

lifetime of object a, c and value

lifetime of object b

```
void function(int value)
{
    int a;
    // ...
    {
        // ...
        int b;
        // ...
    }
    // ...
    int c;
    // ...
}
```

(*) unless declared with the `static` keyword, see other slide

Static storage duration

A variable declared outside a function scope, or with the `static` keyword inside a function, comes into existence and is initialized at program startup and its lifetime does not end until the program exits.

initialized to 0 at program startup

```
int global_var;

int next_value(void)
{
    static int local_var = 39;
    local_var += 1;
    return local_var;
}

int main(void)
{
    next_value();
    next_value();
    printf("The answer is %d\n",
           next_value());
    exit(0);
}
```

initialized to 39 at program startup

increase the value of local_var every time this line expression is evaluated.

lifetime of static variables, from program startup to program exit

```
The answer is 42
```

Allocated storage duration

lifetime of the object that b is pointing to

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

static int * b;

int main(void)
{
    printf("The answer is");
    b = malloc(sizeof b);
    *b = 42;
    printf(" %d\n", *b);
    free(b);
    b = 0;
}
```

```
The answer is 42
```

Summary

- hello world!
- behaviour
- vocabulary of the language
- compiler, translator, assembler, linker
- standard library and C run-time
- memory layout and execution stack