Deep C

by Olve Maudal

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

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

A 50 minute session at Scandinavian Developer Conference 2013
Tuesday, March 5, 2013


JAWS

Exercise

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

foo.c

```
#include <stdio.h>

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

Exercise

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

foo.c

```
#include <stdio.h>

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

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

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

foo.c

```c
#include <stdio.h>

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

Let’s add some flags for better diagnostics.

foo.c

```
#include <stdio.h>

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

Let’s add some flags for better diagnostics.

foo.c

```c
#include <stdio.h>

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

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

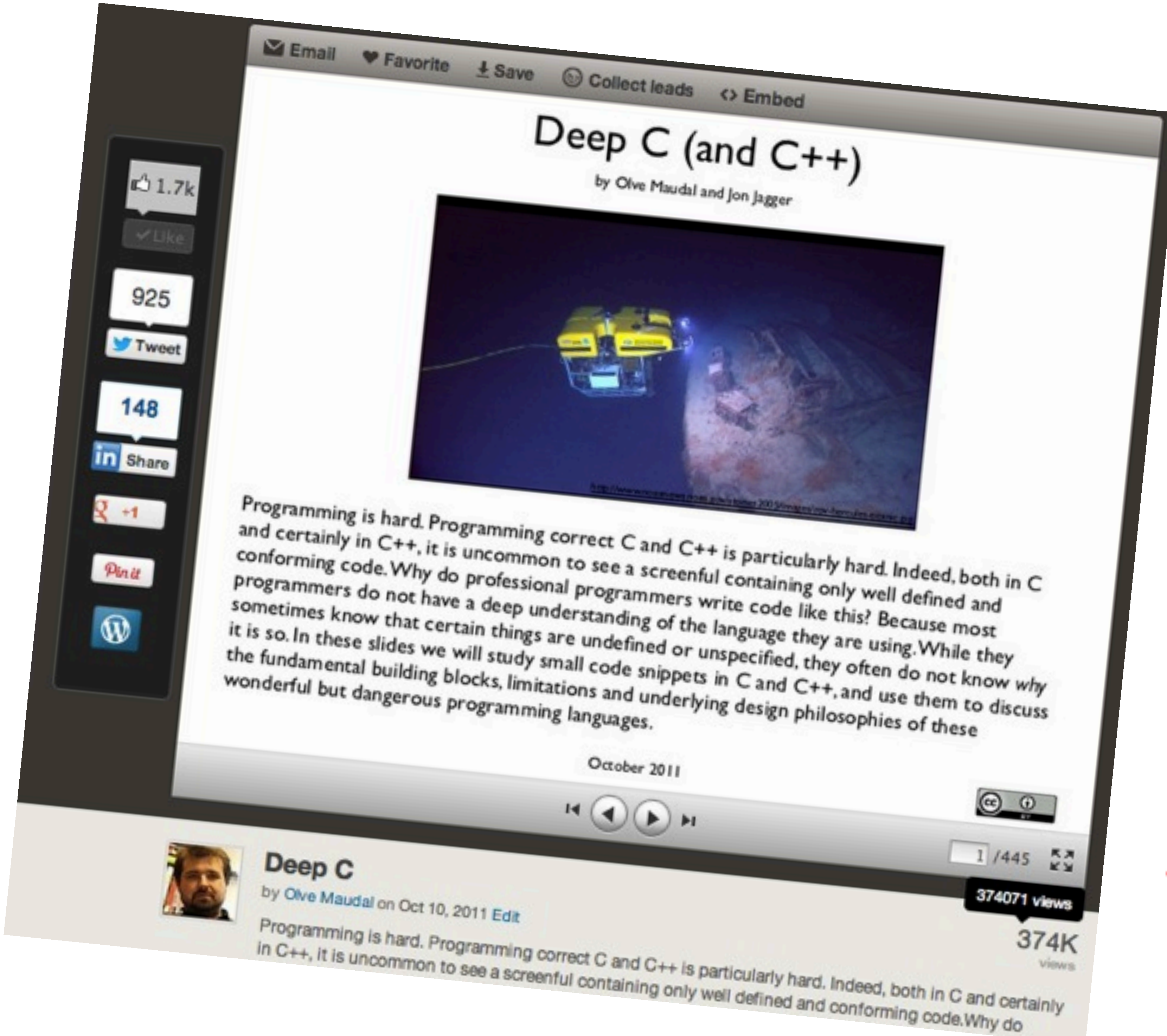
```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

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

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

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

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



```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```
4
```

```c
#include <stdio.h>

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

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

```
4
4
```

```c
#include <stdio.h>

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

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

```
4
4
4
```

```c
#include <stdio.h>

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

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

```
4
4
4
```

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```
4
```

```c
#include <stdio.h>

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

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

```
4
5
```

```c
#include <stdio.h>

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

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

```
4
5
6
```

```c
#include <stdio.h>

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

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

```
4
5
6
```

```c
#include <stdio.h>

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

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


```c
#include <stdio.h>

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

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

garbage, garbage, garbage?

```c
#include <stdio.h>

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

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

No. Variables with static storage duration are initialized to 0

garbage, garbage, garbage?

It is better to initialize explicitly.

```
#include <stdio.h>

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

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

No. Variables with static storage duration are initialized to 0

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

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

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

No. Variables with static storage duration are initialized to 0

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

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

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

```
1
```

No. Variables with static storage duration are initialized to 0

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

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

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

```
1
2
```

No. Variables with static storage duration are initialized to 0

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

garbage, garbage, garbage?

It is better to initialize explicitly.

```c
#include <stdio.h>

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

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

```
1
2
3
```

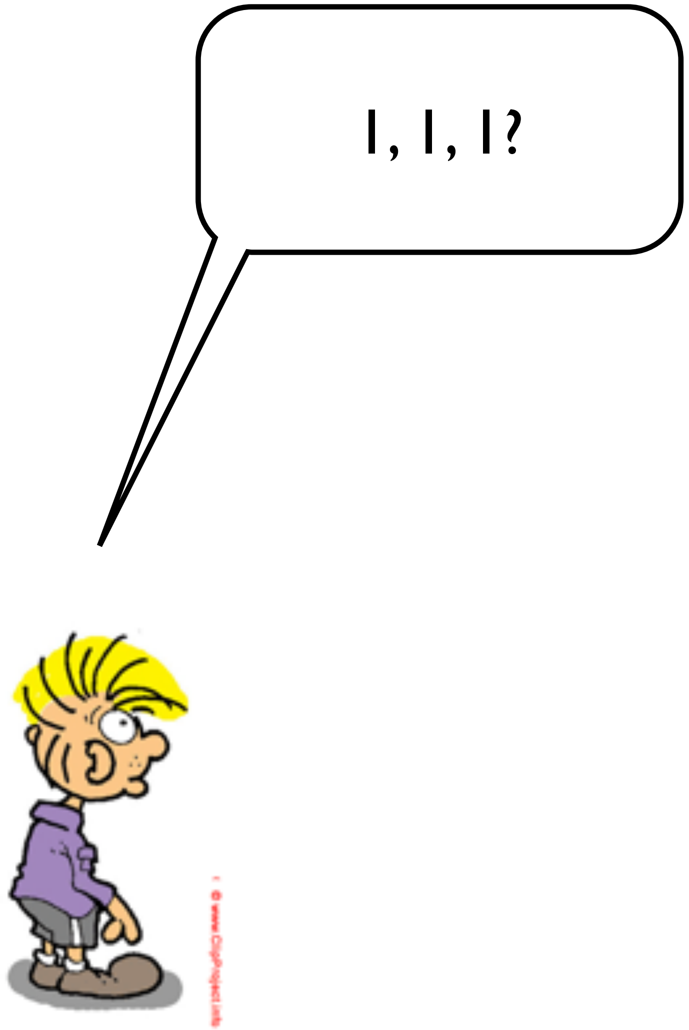
No. Variables with static storage duration are initialized to 0

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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




```
#include <stdio.h>

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

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

1, 1, 1?

```c
#include <stdio.h>

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

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

No, variables with automatic storage duration are not initialized implicitly

1, 1, 1?

Garbage, garbage, garbage?

```
#include <stdio.h>

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

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

No, variables with automatic storage duration are not initialized implicitly

I, I, I?

Garbage, garbage, garbage?

```c
#include <stdio.h>

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

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

No, variables with automatic storage duration are not initialized implicitly

Yes, in theory that is correct.

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

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

Because C is a braindead programming language?

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

Because C is a braindead programming language?

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

I, I, I?

Garbage, garbage, garbage?

```
#include <stdio.h>

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

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

No, variables with automatic storage duration are not initialized implicitly

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

I, I, I?

Garbage, garbage, garbage?

```c
#include <stdio.h>

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

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

```
1
```

No, variables with automatic storage duration are not initialized implicitly

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


```c
#include <stdio.h>

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

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

```
1
2
```

No, variables with automatic storage duration are not initialized implicitly

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

I, I, I?

Garbage, garbage, garbage?

```
#include <stdio.h>

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

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

```
1
2
3
```

No, variables with automatic storage duration are not initialized implicitly

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

I, I, I?

Garbage, garbage, garbage?

Ehh...

```
#include <stdio.h>

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

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

```
1
2
3
```

No, variables with automatic storage duration are not initialized implicitly

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

I, I, I?

Garbage, garbage, garbage?

Ehh...

```c
#include <stdio.h>

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

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

```
1
2
3
```

No, variables with automatic storage duration are not initialized implicitly

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

any plausible explaination for this behaviour?

I, I, I?

Garbage, garbage, garbage?

Ehh...

```
#include <stdio.h>

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

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

```
1
2
3
```

No, variables with automatic storage duration are not initialized implicitly

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

any plausible explaination for this behaviour?

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

I, I, I?

Garbage, garbage, garbage?

Ehh...

```c
#include <stdio.h>

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

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

```
1
2
3
```

No, variables with automatic storage duration are not initialized implicitly

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

any plausible explaination for this behaviour?

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

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

```c
#include <stdio.h>

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

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

1, 1, 1?

Garbage, garbage, garbage?

No, variables with automatic storage duration are not initialized implicitly

Ehh...

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

```
1
2
3
```

any plausible explaination for this behaviour?

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

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

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

```c
#include <stdio.h>

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

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

```
1
2
3
```

1, 1, 1?

No, variables with automatic storage duration are not initialized implicitly

Garbage, garbage, garbage?

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

Ehh...

any plausible explaination for this behaviour?

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

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

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

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


```c
#include <stdio.h>

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

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

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

```
#include <stdio.h>

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

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

OK, let’s add some flags

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

```c
#include <stdio.h>

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

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

OK, let’s add some flags

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

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

```c
#include <stdio.h>

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

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

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

OK, let's add some flags

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

```
#include <stdio.h>

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

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

OK, let's add some flags

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

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

```c
#include <stdio.h>

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

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

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

OK, let's add some flags

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

```
#include <stdio.h>

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

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

OK, let's add some flags

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

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

Lousy compiler!

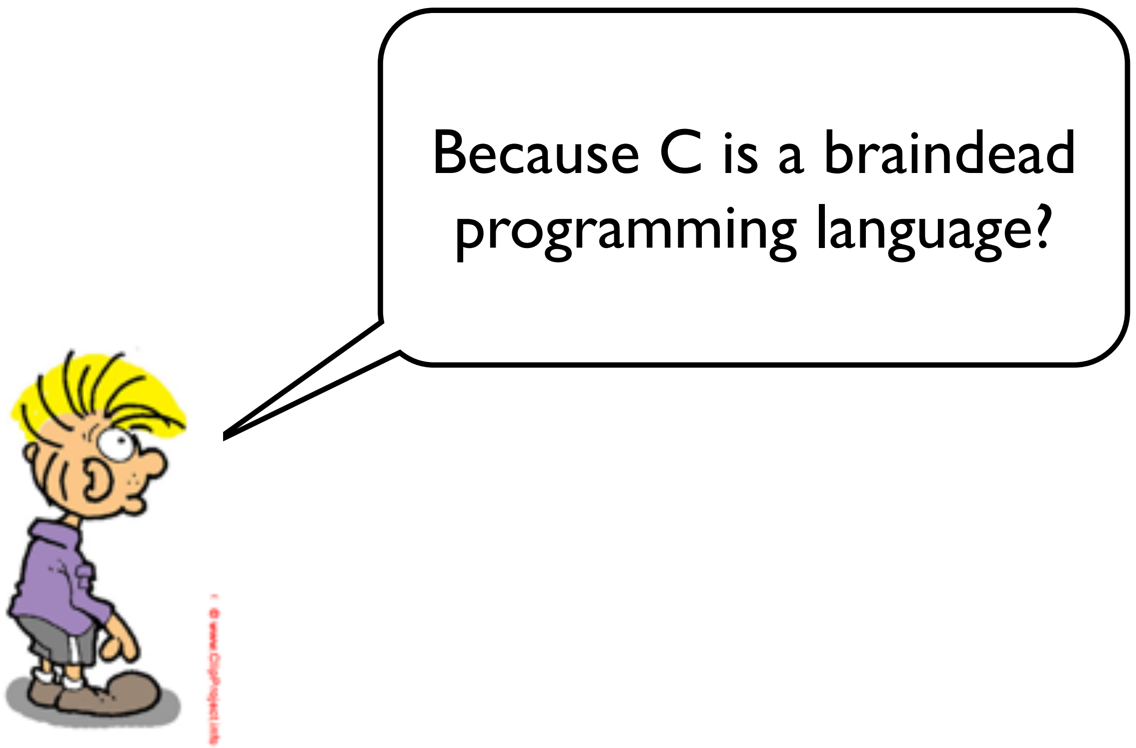
```c
#include <stdio.h>

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

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

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

OK, let's add some flags

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

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

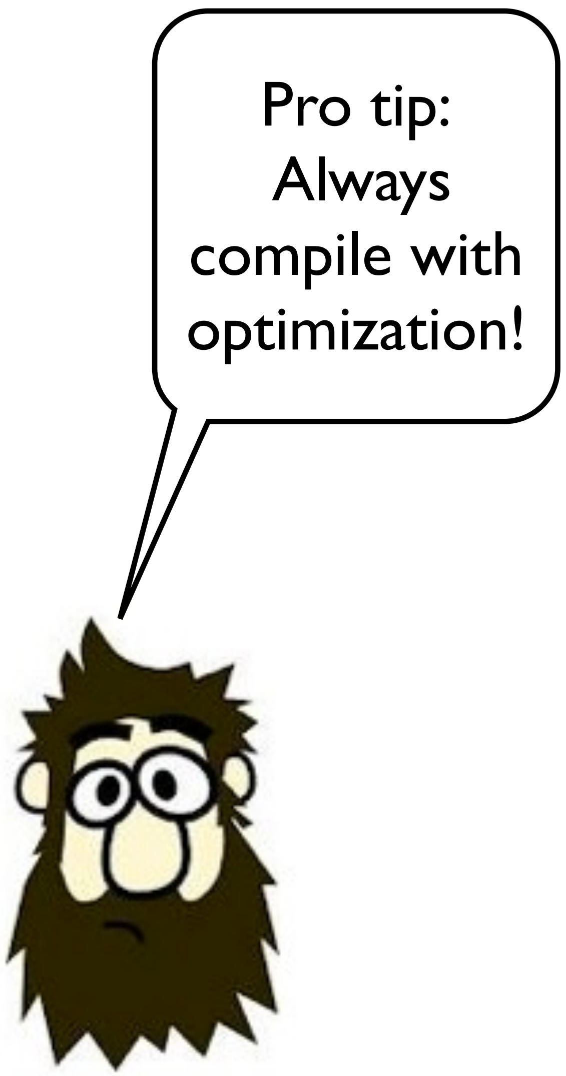


Why don’t the C standard require that you always get a warning or error on invalid code?

Because C is a braindead programming language?

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

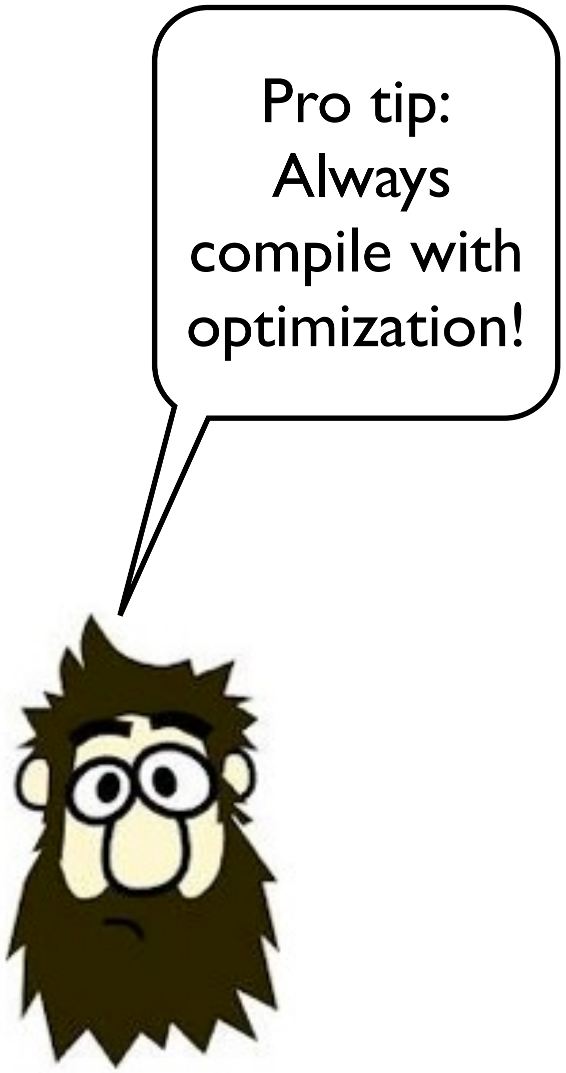
```
#include <stdio.h>

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

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




```c
#include <stdio.h>

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

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

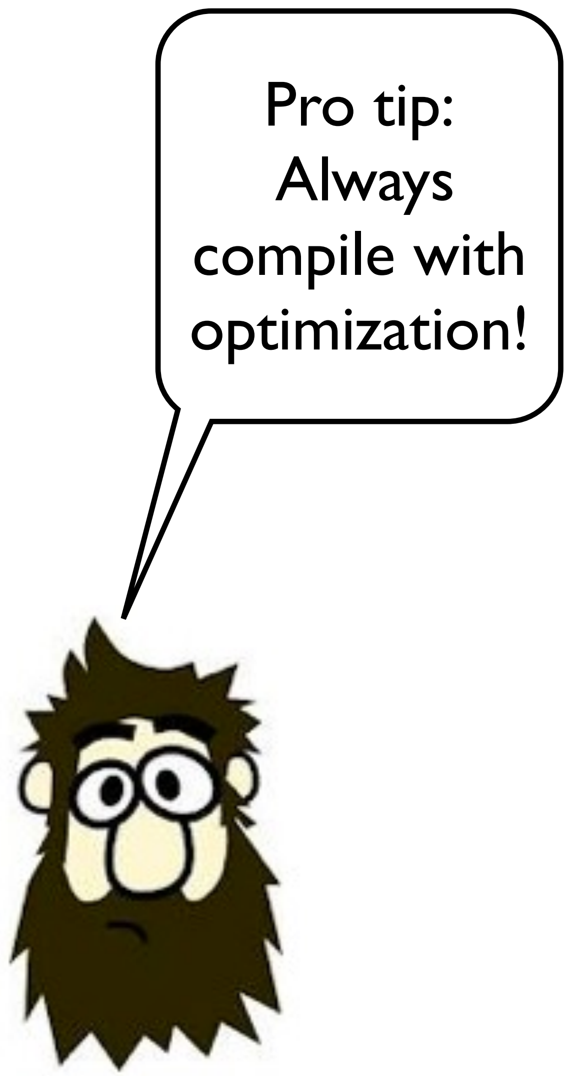

```c
#include <stdio.h>

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

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

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



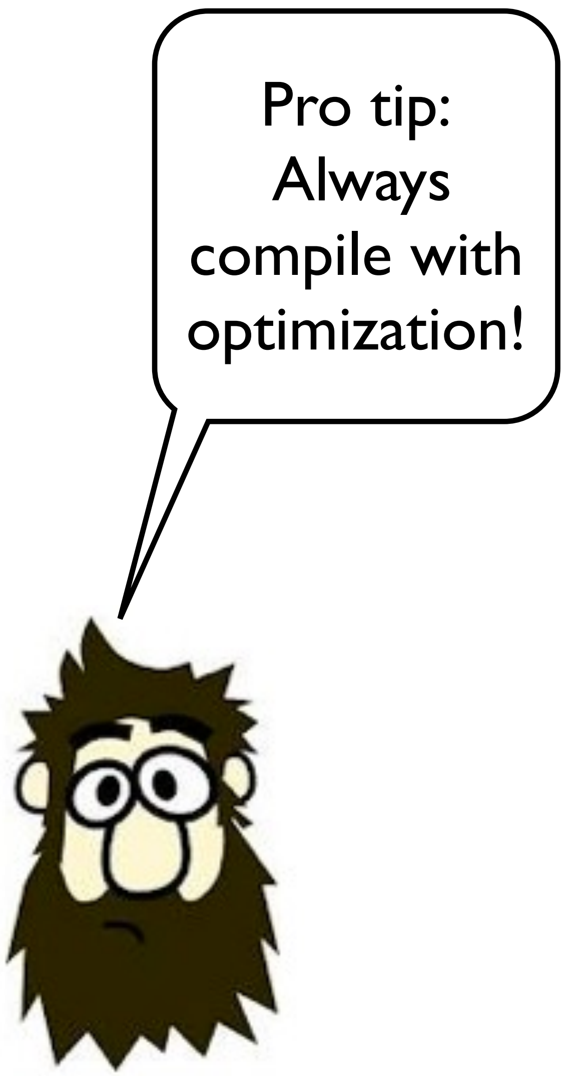

```c
#include <stdio.h>

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

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

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



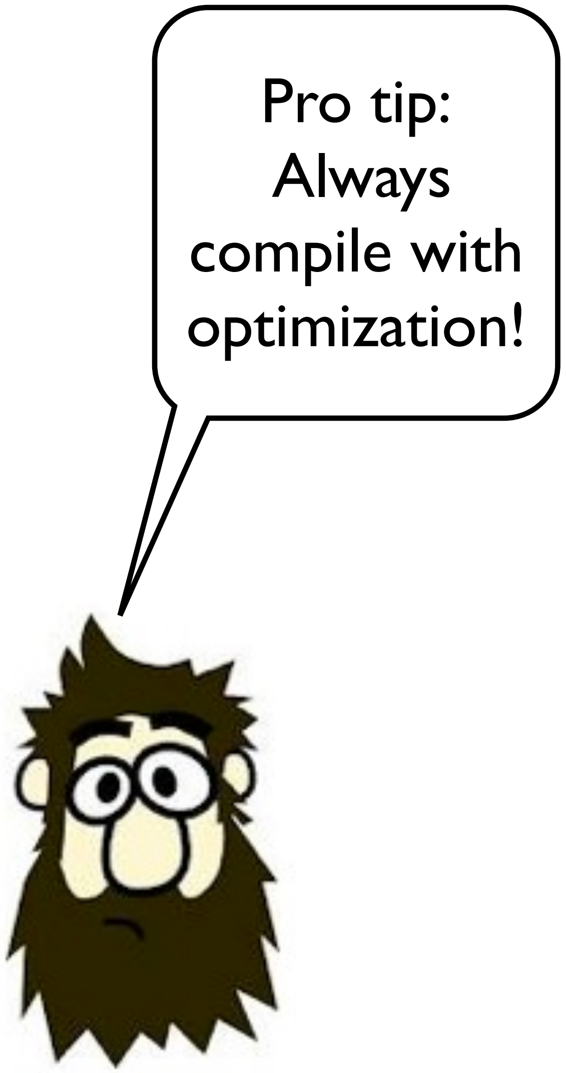

```c
#include <stdio.h>

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

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

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



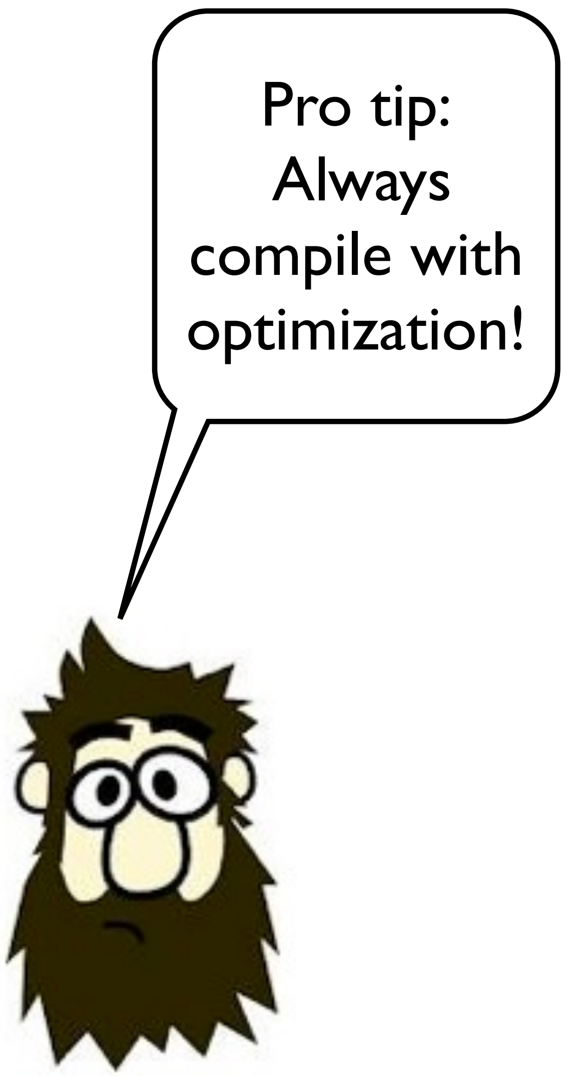

```c
#include <stdio.h>

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

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

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




```c
#include <stdio.h>

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

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

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

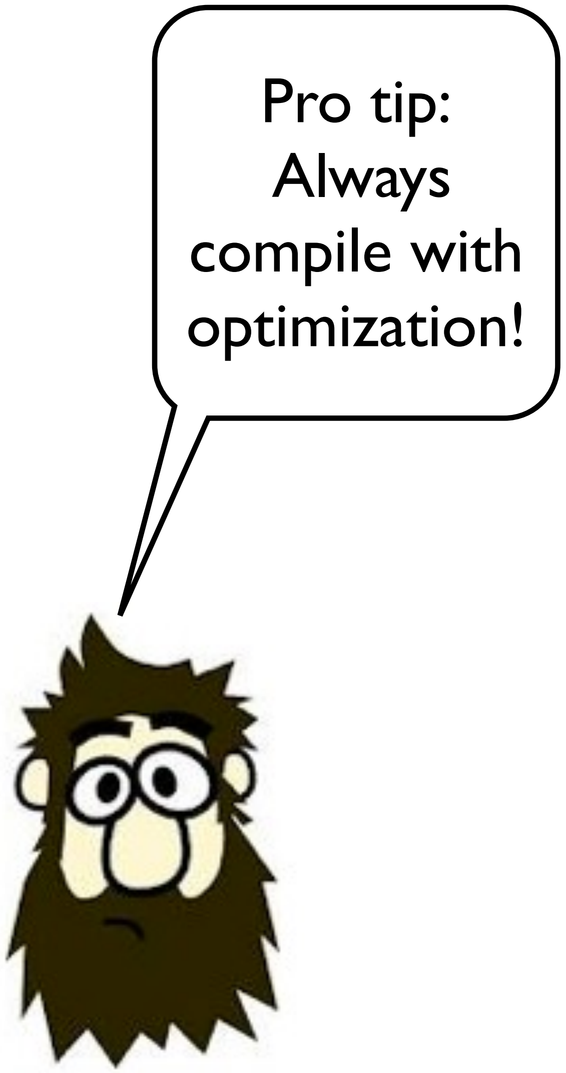



```c
#include <stdio.h>

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

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

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


```c
#include <stdio.h>

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

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

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

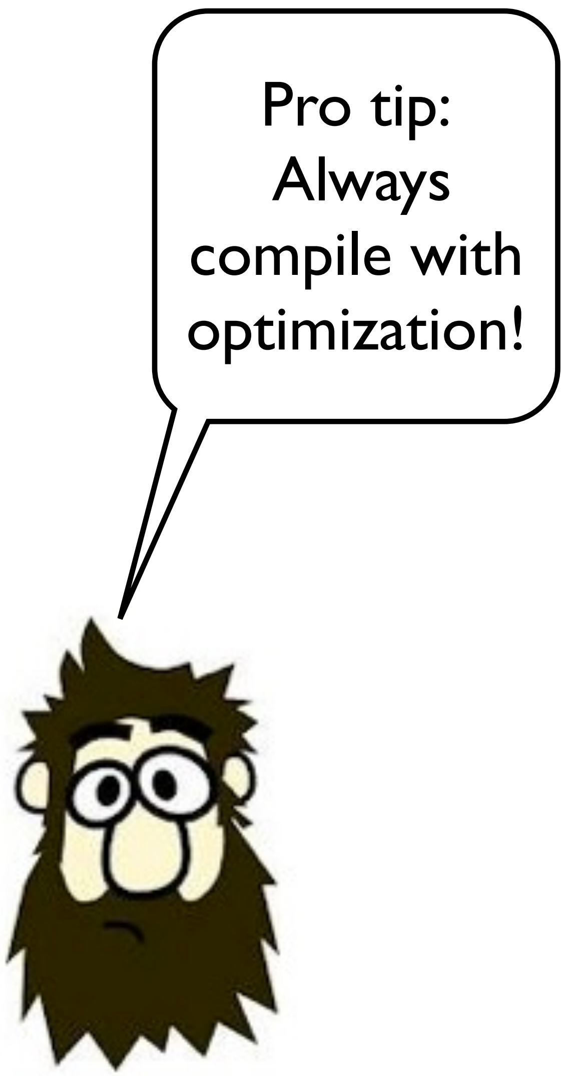



```c
#include <stdio.h>

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

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

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


```c
#include <stdio.h>

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

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

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




```c
#include <stdio.h>

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

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

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

I am now going to show you something cool!

I am now going to show you something cool!

```
#include <stdio.h>

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

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

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

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

Can you explain this behavior?

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

Can you explain this behavior?

If you can give a plausible explanation for this behavior, you should feel both good and bad. Bad because you obviously know something you are supposed to not know when programming in C. You make assumptions about the underlying implementation and architecture. Good because being able to understand such phenomenons are essential for troubleshooting C programs and for avoiding falling into all the traps laid out for you.

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

Can you explain this behavior?

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

Can you explain this behavior?

eh?

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

Can you explain this behavior?

eh?

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

I am now going to show you something cool!

```c
#include <stdio.h>

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

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

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

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

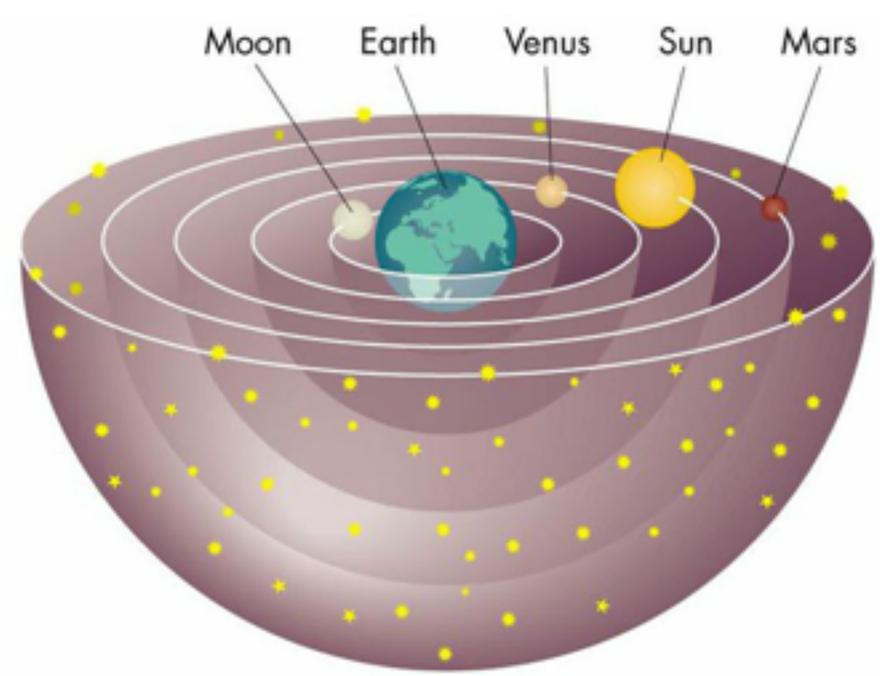
Can you explain this behavior?

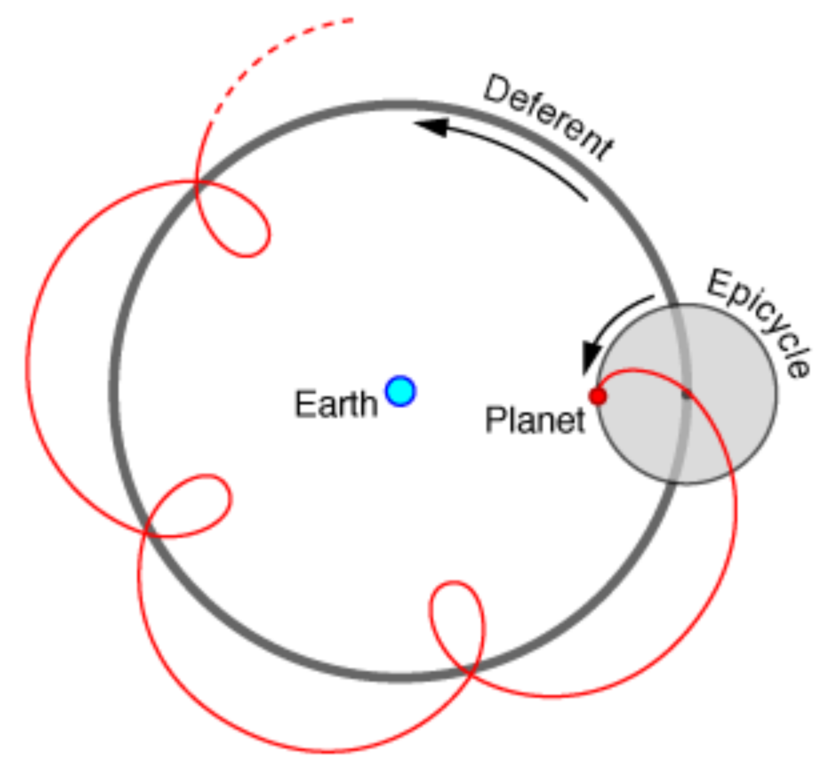
eh?

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

Yeah, sure...

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

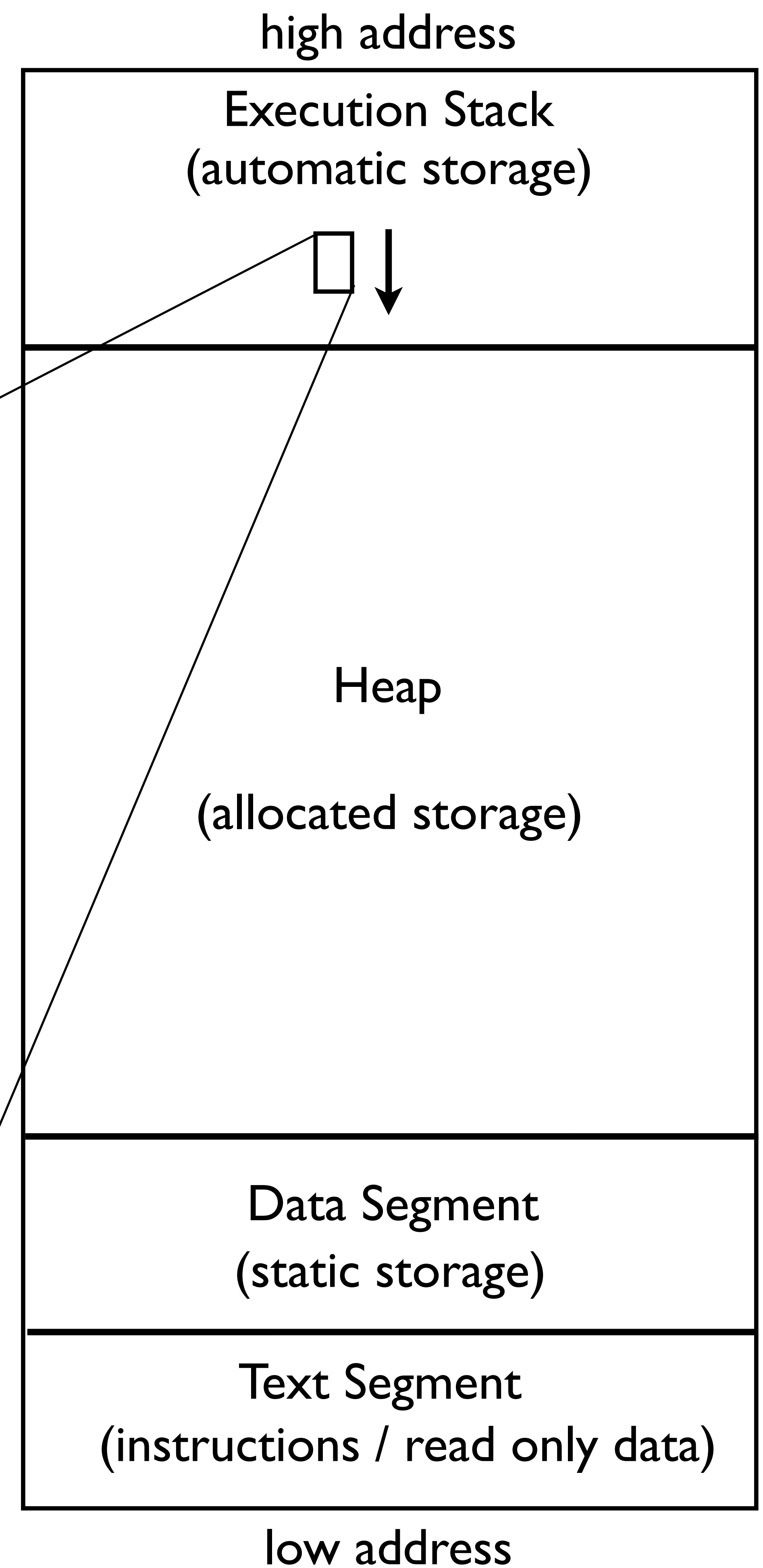




Memory Layout *

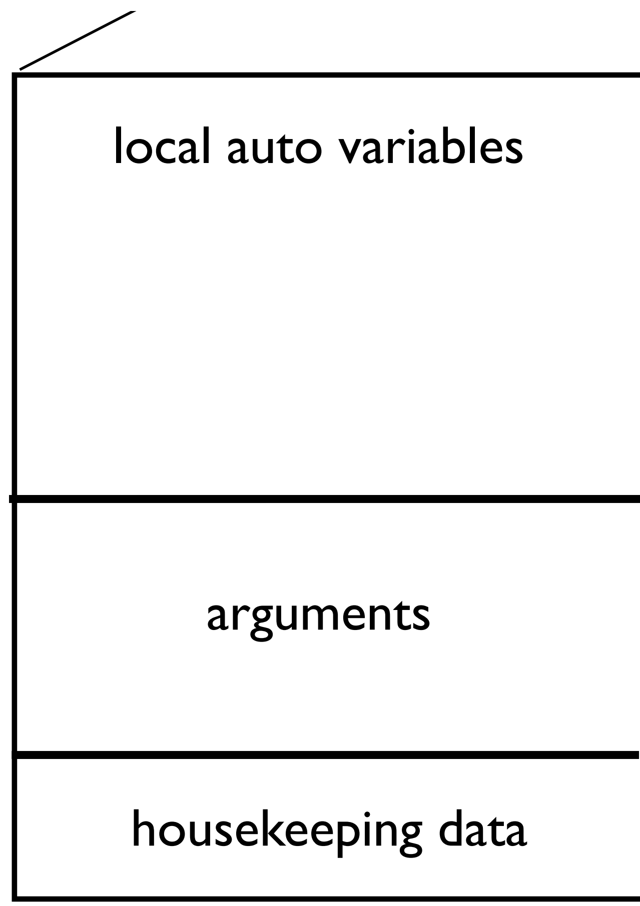
It is sometimes useful to assume that a C program uses a memory model where the instructions are stored in a **text segment**, and static variables are stored in a **data segment**. Automatic variables are allocated when needed together with housekeeping variables on an **execution stack** that is growing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

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



Activation Record

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



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

```c
#include <stdio.h>

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

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

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

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

```c
#include <stdio.h>

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

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

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

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

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

```c
#include <stdio.h>

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

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

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

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

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

```c
#include <stdio.h>

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

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

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

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

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

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

or

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

```
347
347
```

```c
#include <stdio.h>

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

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

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

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

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

but you might also get

```
437
347
```

or

```
437
437
```

or

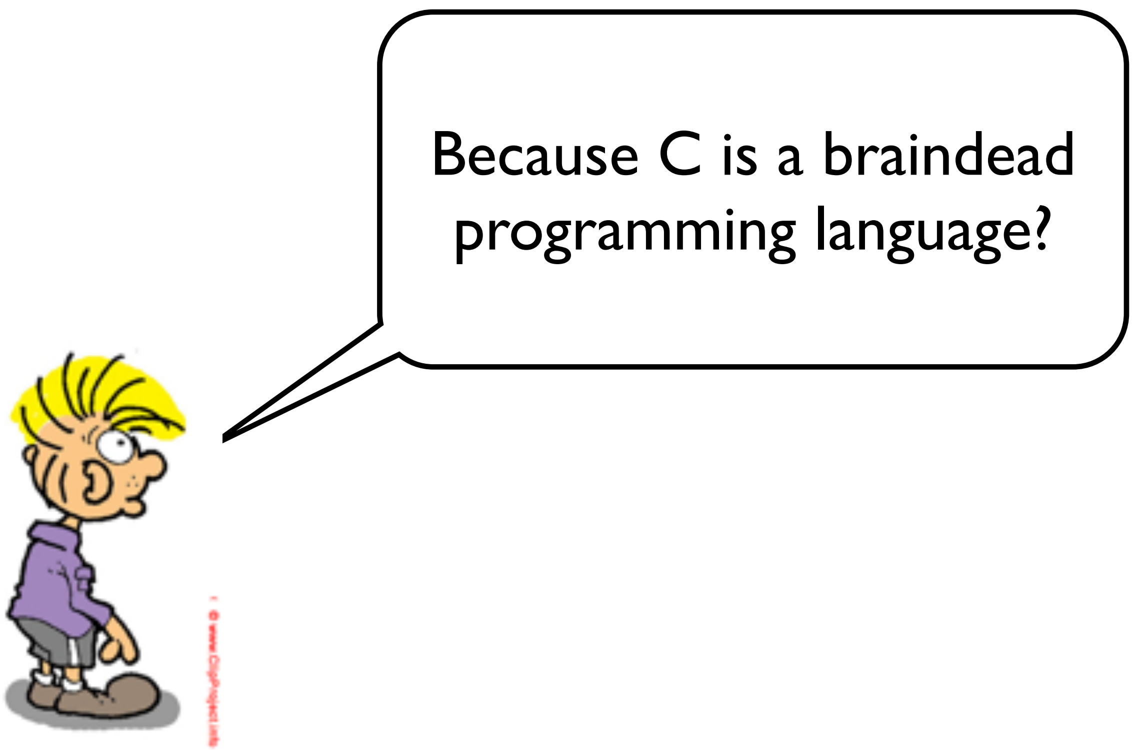
```
347
347
```

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

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf("%d\n", j);
}
```

```c
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

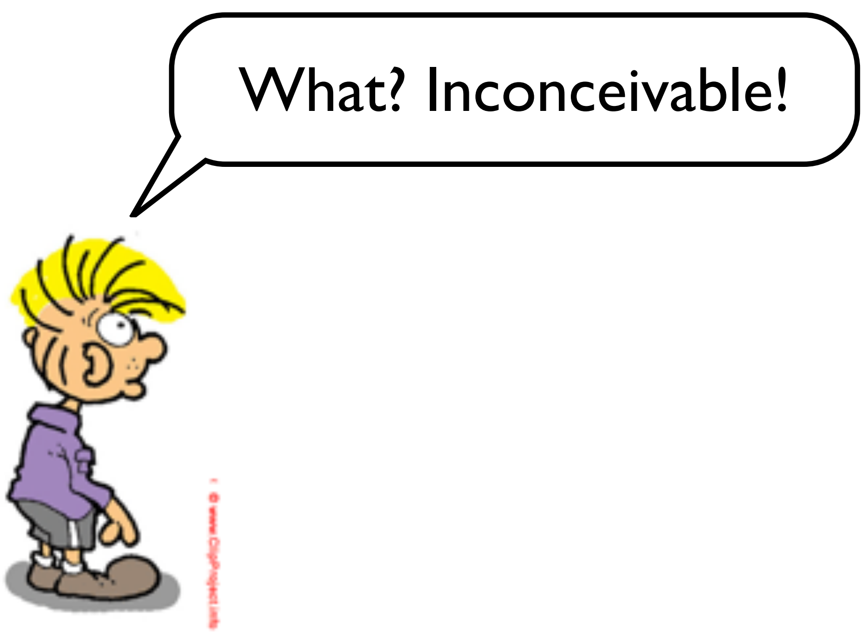
```
$ cc foo.cpp && ./a.out
```

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf("%d\n", j);
}
```

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

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

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



```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

What? Inconceivable!

```
$ cc 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 <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

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

What? Inconceivable!

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!

```c
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf("%d\n", j);
}
```

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

What? Inconceivable!

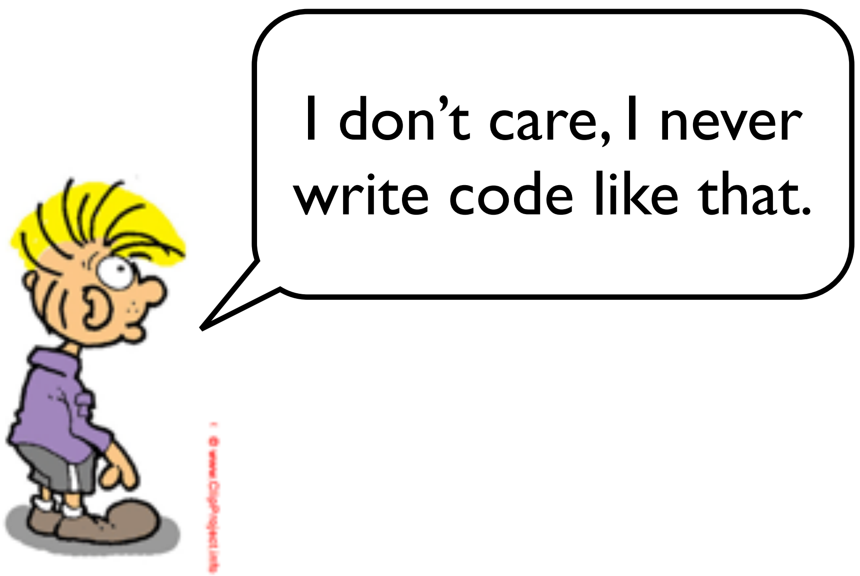
I agree this is crap code, but why is it wrong?

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

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

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

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



```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf("%d\n", j);
}
```

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

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

```
$ cc 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...

```
#include <stdio.h>

int main(void) {
    int v[6] = {4,6,2,9};
    int i = 2;
    int j = i * 3 + v[i++];
    printf(“%d\n”, j);
}
```

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

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

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

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

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

What do these code snippets print?

What do these code snippets print?

1

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

What do these code snippets print?

1

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

2

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

What do these code snippets print?

1

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

2

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

3

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

What do these code snippets print?

1

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

2

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

3

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

4

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

What do these code snippets print?

1

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

2

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

3

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

4

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

5

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

What do these code snippets print?

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

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

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

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

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

6
```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

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

42

2

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

3

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

4

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

5

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

6

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

What do these code snippets print?

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

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

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

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

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

6
```
int a=41; a = foo(a++); printf("42\n");
```

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

5

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

6

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

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

42

5

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

6

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

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

42

5

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

undefined

6

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

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

42

5

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

undefined

6

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

?

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

42

5

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

undefined

6

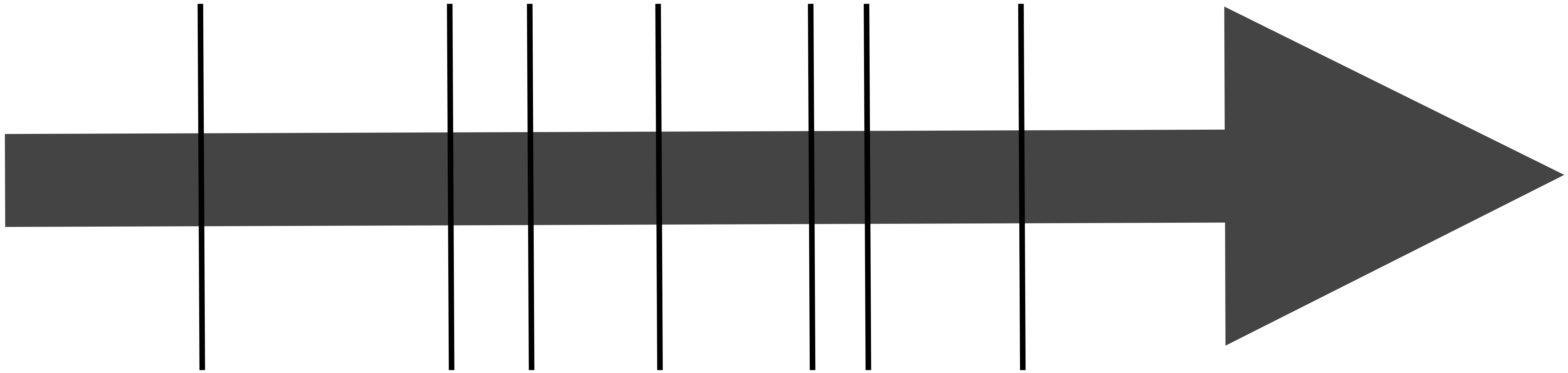
```
int a=41; a = foo(a++); printf("42\n");
```

?

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

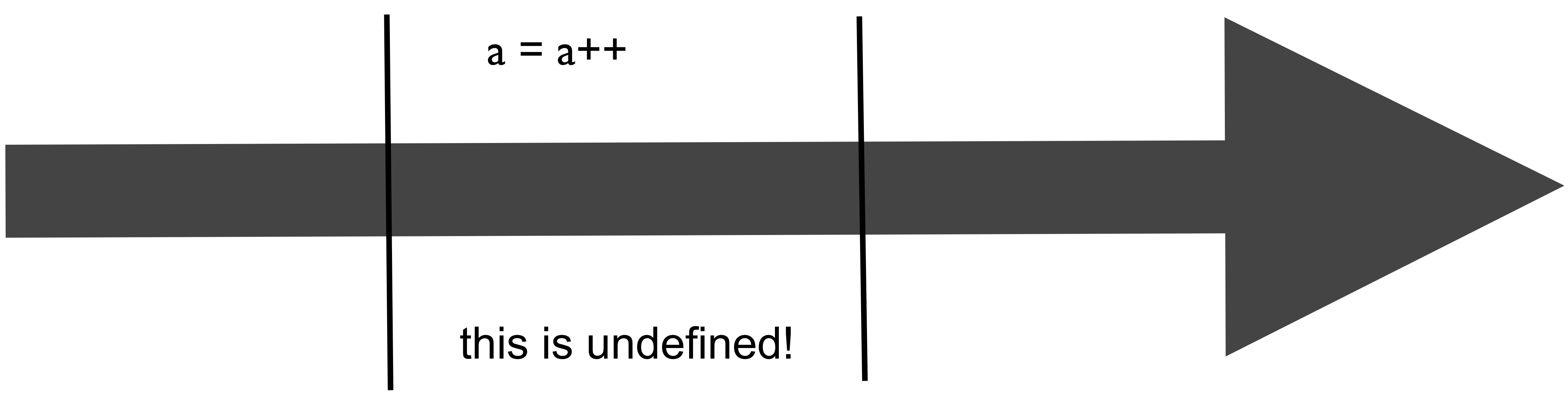
Sequence Points

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



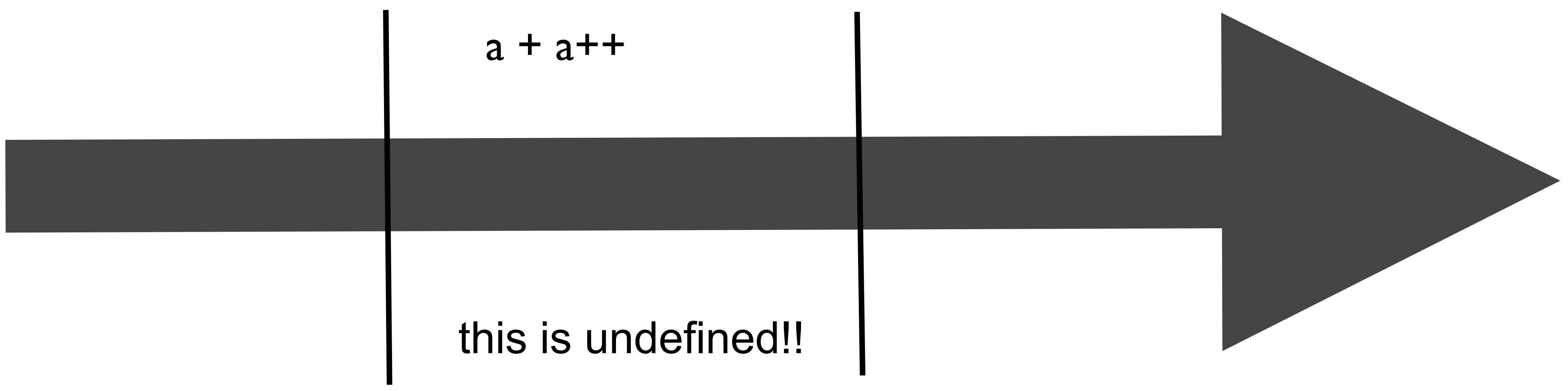
Sequence Points - Rule 1

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



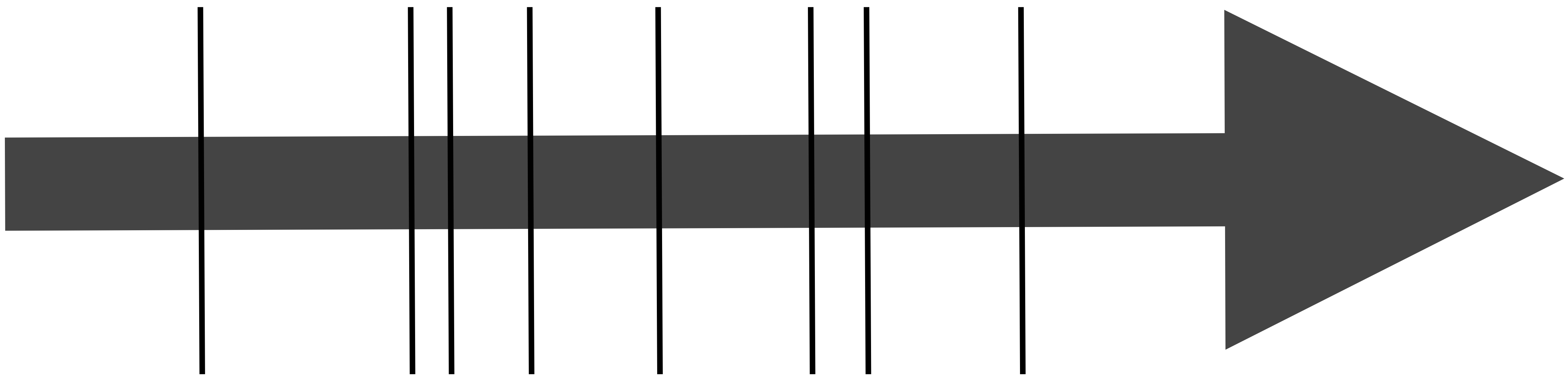
Sequence Points - Rule 2

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



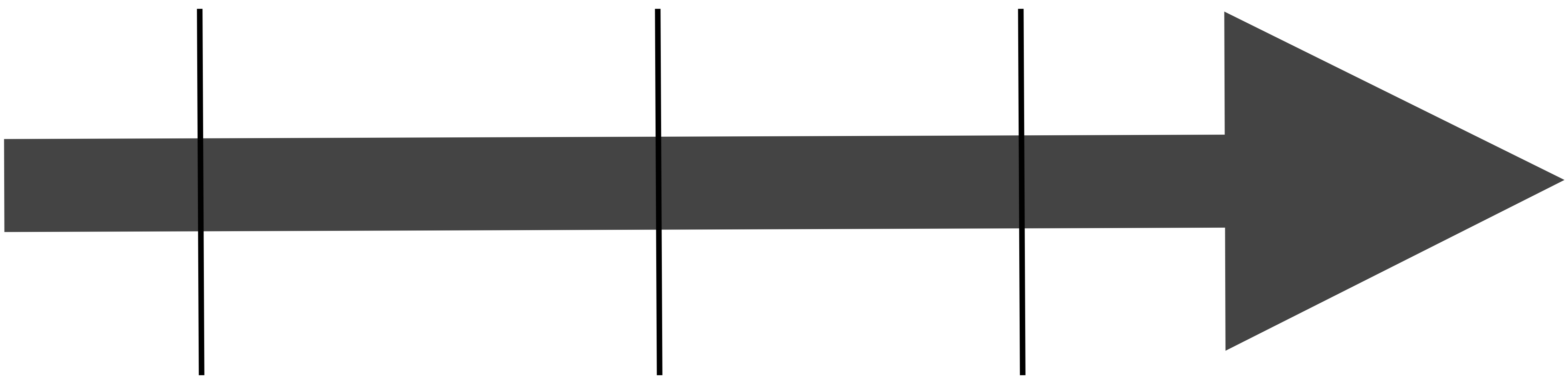
Sequence Points

A lot of developers think C has _many_ sequence points



Sequence Points

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



This helps to maximize optimization opportunities for the compiler.

Sequence points in C

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

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

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

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

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

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

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

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

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

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```
$ cc foo.c
```

```c
#include <stdio.h>

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

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

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

```c
#include <stdio.h>

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

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

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

```c
#include <stdio.h>

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

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

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

```c
#include <stdio.h>

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

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

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

```c
#include <stdio.h>

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

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

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

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

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

```
#include <stdio.h>

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

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

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

They are all morons!

They are all morons!

```c
#include <stdio.h>

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

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

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

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

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

They are all morons!

ehh...

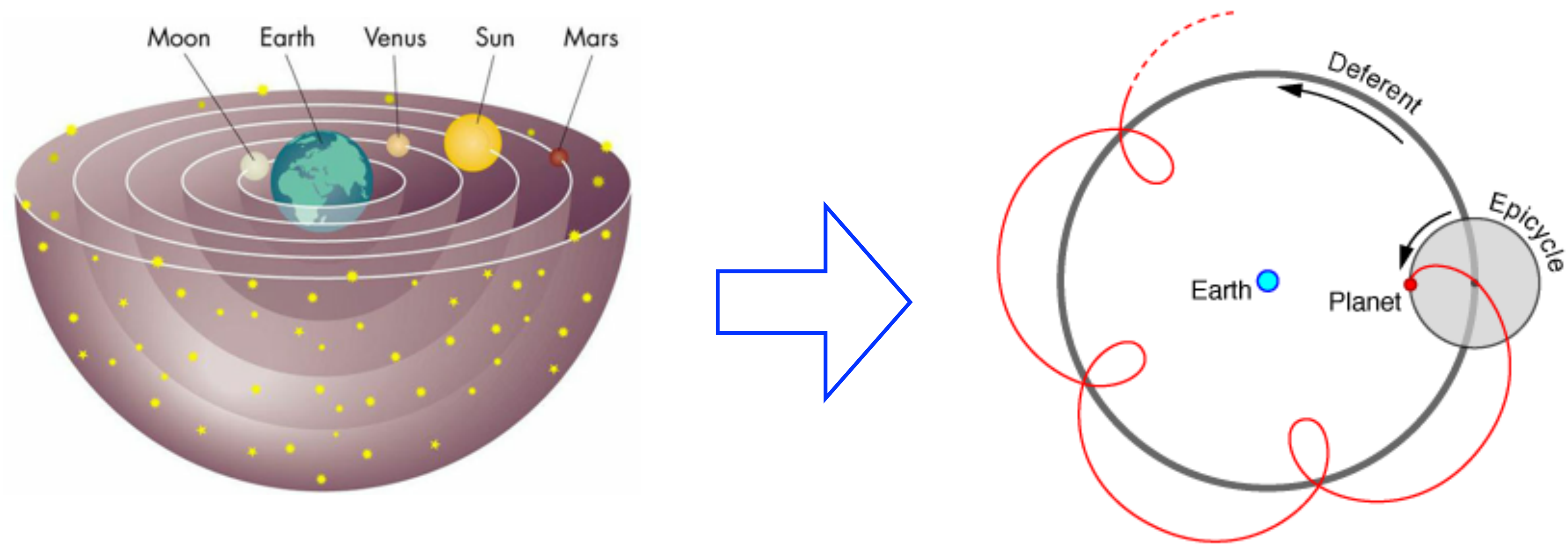
```c
#include <stdio.h>

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

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

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

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

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

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



Behavior

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

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

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

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

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

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

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

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

Behavior

... and, locale-specific behavior

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

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

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

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

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

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

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

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

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

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

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

deep_thought.c

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

deep_thought.c

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

main.c

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

int the_answer(int);

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

deep_thought.c

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

main.c

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

int the_answer(int);

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

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

deep_thought.c

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

main.c

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

int the_answer(int);

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

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

deep_thought.c

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

main.c

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

int the_answer(int);

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

```
$ cc main.c deep_thought.c && ./a.out
The anwser is:
3.1415926535897932384626433832795028841971
```

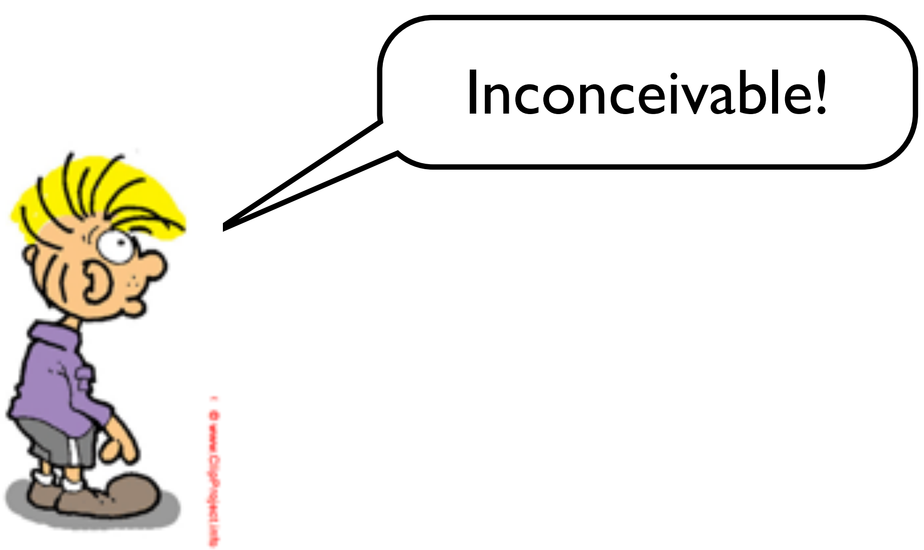
deep_thought.c

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

main.c

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

int the_answer(int);

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

```
$ cc main.c deep_thought.c && ./a.out
The anwser is:
3.1415926535897932384626433832795028841971
```



deep_thought.c

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

main.c

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

int the_answer(int);

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

```
$ cc main.c deep_thought.c && ./a.out
The anwser is:
3.1415926535897932384626433832795028841971
```

deep_thought.c

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

main.c

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

int the_answer(int);

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

```
$ cc main.c deep_thought.c && ./a.out
The anwser is:
3.1415926535897932384626433832795028841971
```

Inconceivable!

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

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

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

Exercise

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

foo.c

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

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

Exercise

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

foo.c

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

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

main.c

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

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

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
```

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
```

Exercise

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

foo.c

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

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

main.c

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

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

bar.c

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
$ gcc foo.c bar.c main.c
```

Exercise

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

foo.c

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

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

main.c

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

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

bar.c

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
$ gcc foo.c bar.c main.c
$ ./a.out
```

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
$ gcc foo.c bar.c main.c
$ ./a.out
true
```

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
$ gcc foo.c bar.c main.c
$ ./a.out
true
false
```

Exercise

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

bar.c

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

foo.c

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

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

main.c

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

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

This is what I get on my computer (Mac OS 10.8.2, gcc 4.7.2)

```
$ gcc -v
gcc version 4.7.2 (GCC)
$ gcc foo.c bar.c main.c
$ ./a.out
true
false
$
```

A real story of “anything can happen”

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

A real story of "anything can happen"

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

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

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

A real story of "anything can happen"

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

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

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

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

A real story of "anything can happen"

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

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

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

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

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

a few words about memory

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

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

int main(void)
{
    struct X a = {42,'a',1337};
    struct X b = {42,'a',1337};

    if (memcmp(&a, &b, sizeof a) == 0)
        printf("equal\n");
    else
        printf("not equal\n");
}
```

a few words about memory

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

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

int main(void)
{
    struct X a = {42,'a',1337};
    struct X b = {42,'a',1337};

    if (memcmp(&a, &b, sizeof a) == 0)
        printf("equal\n");
    else
        printf("not equal\n");
}
```

This might happen:

```
$ cc -O2 foo.c && ./a.out
equal
$ cc -O3 foo.c && ./a.out
not equal
$
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
```

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

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

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

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

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

```
$ cc -fpack-struct foo.c
```

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

```
$ cc -fpack-struct foo.c
$ ./a.out
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
```

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

On my machine (Mac OS 10.8.2 x86_64):

```
$ cc foo.c
$ ./a.out
4
1
12
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
9
```

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

packed struct

a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned

a	a	a	a
b	.	.	.
c	c	c	c

`sizeof(struct X) == 12`

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

packed struct

a	a	a	a
b	c	c	c
c			

`sizeof(struct X) == 9`

memory aligned

a	a	a	a
b	.	.	.
c	c	c	c

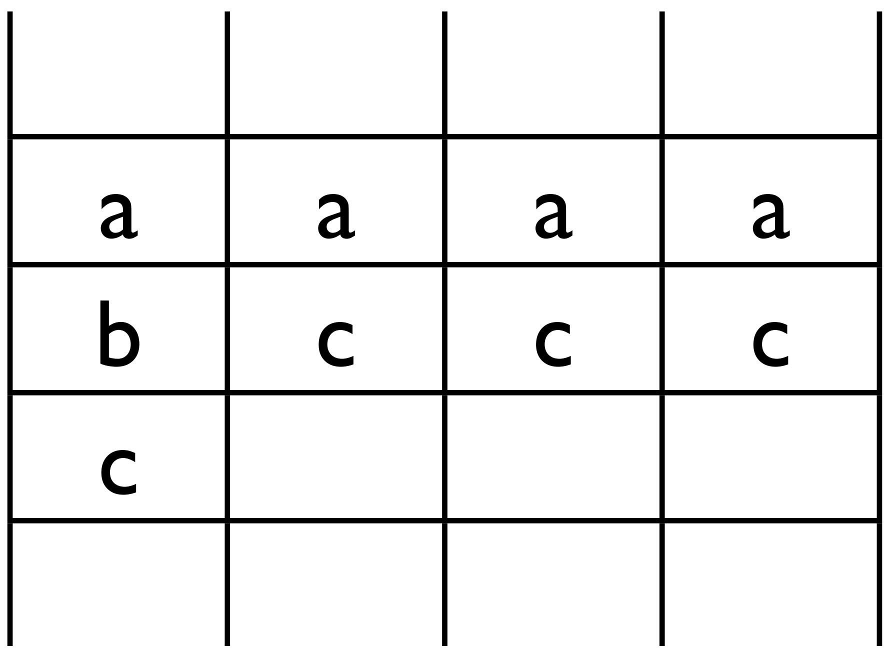
`sizeof(struct X) == 12`

Imagine how the assembly code for this snippet would look like:

```c
void foo(struct X * x) {
  x->c += 42;
}
```

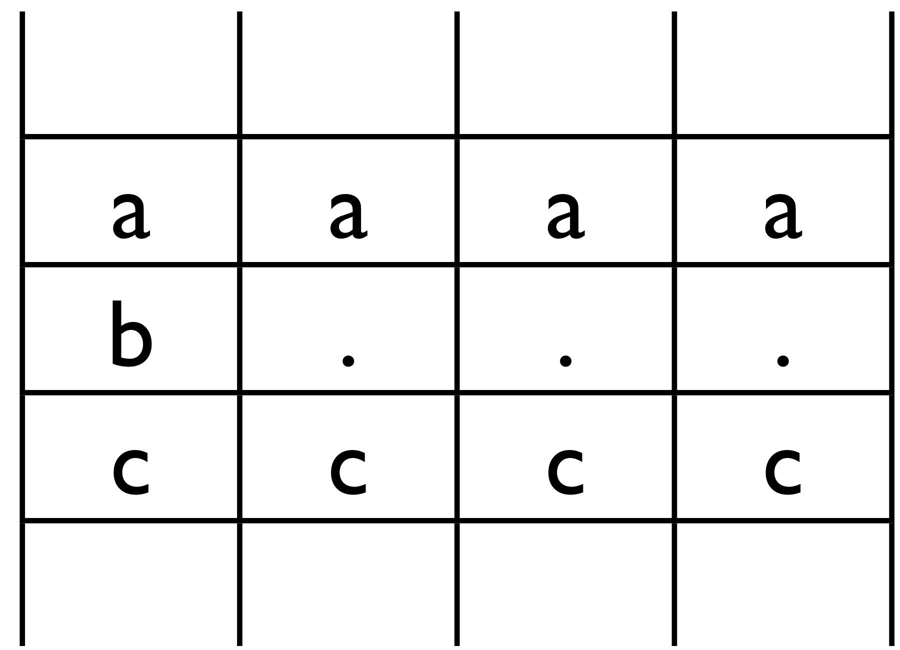
```
struct X {
    int a;
    char b;
    int c;
};
```

packed struct ?




```
sizeof(struct X) == 9
```

memory aligned




```
sizeof(struct X) == 12
```

Imagine how the assembly code for this snippet would look like:

```
void foo(struct X * x) {
  x->c += 42;
}
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
```

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

```
$ cc foo.c
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

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

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

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

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

```
$ cc foo.c
$ ./a.out
4
1
```

a few words about memory

```
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

```
$ cc foo.c
$ ./a.out
4
1
8
```

a few words about memory

```c
#include <stdio.h>

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

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(char *));
    printf("%zu\n", sizeof(struct X));
}
```

```
$ cc -fpack-struct foo.c
$ ./a.out
4
1
8
17
```

```
$ cc foo.c
$ ./a.out
4
1
8
24
```

So what’s wrong with this code?

foo.c

```
#include <stdio.h>

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

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

So what’s wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code ?

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

So what's wrong with this code?

foo.c

```
#include <stdio.h>

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

It is crap code

The standard says that this is invalid code

Update a variable multiple times between two semicolons

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

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

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

does not make sense.

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

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

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

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



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




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






cité
de
l'espace
esa
cnes


© AFP

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

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

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

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

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

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

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

snippet from pftn.c in pcc 1.0.0.RELEASE 20110221

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

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

C and C++ are not really high level languages, they are more like portable assemblers. When programming in C and C++ you *must* have a understanding of what happens under the hood! And if you don't have a decent understanding of it, then you are doomed to create lots of bugs...

C and C++ are not really high level languages, they are more like portable assemblers. When programming in C and C++ you *must* have a understanding of what happens under the hood! And if you don't have a decent understanding of it, then you are doomed to create lots of bugs...

But if you *do* have a useful mental model of what happens under the hood, then...

http://www.sharpshirter.com/assets/images/sharkpunchashgrey1.jpg

!

The spirit of C

trust the programmer
- let them do what needs to be done
- the programmer is in charge not the compiler

keep the language small and simple
- small amount of code → small amount of assembler
- provide only one way to do an operation
- new inventions are not entertained

make it fast, even if its not portable
- target efficient code generation
- int preference, int promotion rules
- sequence points, maximum leeway to compiler

rich expression support
- lots of operators
- expressions combine into larger expressions