Deep C (and C++)

by Olve Maudal and Jon Jagger

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

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

October 2011

Suppose you are about to interview a candidate for a position as C programmer for various embedded platforms. As part of the interview you might want to check whether the candidate has a deep understanding of the programming language or not... here is a great code snippet to get the conversation started:

Suppose you are about to interview a candidate for a position as C programmer for various embedded platforms. As part of the interview you might want to check whether the candidate has a deep understanding of the programming language or not... here is a great code snippet to get the conversation started:

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

Suppose you are about to interview a candidate for a position as C programmer for various embedded platforms. As part of the interview you might want to check whether the candidate has a deep understanding of the programming language or not... here is a great code snippet to get the conversation started:

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

One candidate might say:

What will happen if you try to compile, link and run this program?

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

One candidate might say:

You must `#include <stdio.h>`, add a `return 0` and then it will compile and link. When executed it will print the value 42 on the screen.

What will happen if you try to compile, link and run this program?

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

One candidate might say:

You must `#include <stdio.h>`, add a `return 0` and then it will compile and link. When executed it will print the value 42 on the screen.

and there is nothing wrong with that answer...

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

But another candidate might use this as an opportunity to start demonstrating a deeper understanding. She might say things like:

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

But another candidate might use this as an opportunity to start demonstrating a deeper understanding. She might say things like:

You probably want to `#include <stdio.h>` which has an explicit declaration of `printf()`. The program will compile, link and run, and it will write the number `42` followed by a newline to the standard output stream.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and then she elaborates
a bit by saying:

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and then she elaborates a bit by saying:

A C++ compiler will refuse to compile this code as the language requires explicit declaration of all functions.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and then she elaborates a bit by saying:

A C++ compiler will refuse to compile this code as the language requires explicit declaration of all functions.

However a proper C compiler will create an implicit declaration for the function `printf()`, compile this code into an object file.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and then she elaborates a bit by saying:

A C++ compiler will refuse to compile this code as the language requires explicit declaration of all functions.

However a proper C compiler will create an implicit declaration for the function `printf()`, compile this code into an object file.

And when linked with a standard library, it will find a definition of `printf()` that accidentally will match the implicit declaration.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and then she elaborates a bit by saying:

A C++ compiler will refuse to compile this code as the language requires explicit declaration of all functions.

However a proper C compiler will create an implicit declaration for the function `printf()`, compile this code into an object file.

And when linked with a standard library, it will find a definition of `printf()` that accidentally will match the implicit declaration.

So the program above will actually compile, link and run.

What will happen if you try to compile, link and run this program?

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

and then she elaborates a bit by saying:

A C++ compiler will refuse to compile this code as the language requires explicit declaration of all functions.

However a proper C compiler will create an implicit declaration for the function `printf()`, compile this code into an object file.

And when linked with a standard library, it will find a definition of `printf()` that accidentally will match the implicit declaration.

So the program above will actually compile, link and run.

You might get a warning though.

What will happen if you try to compile, link and run this program?

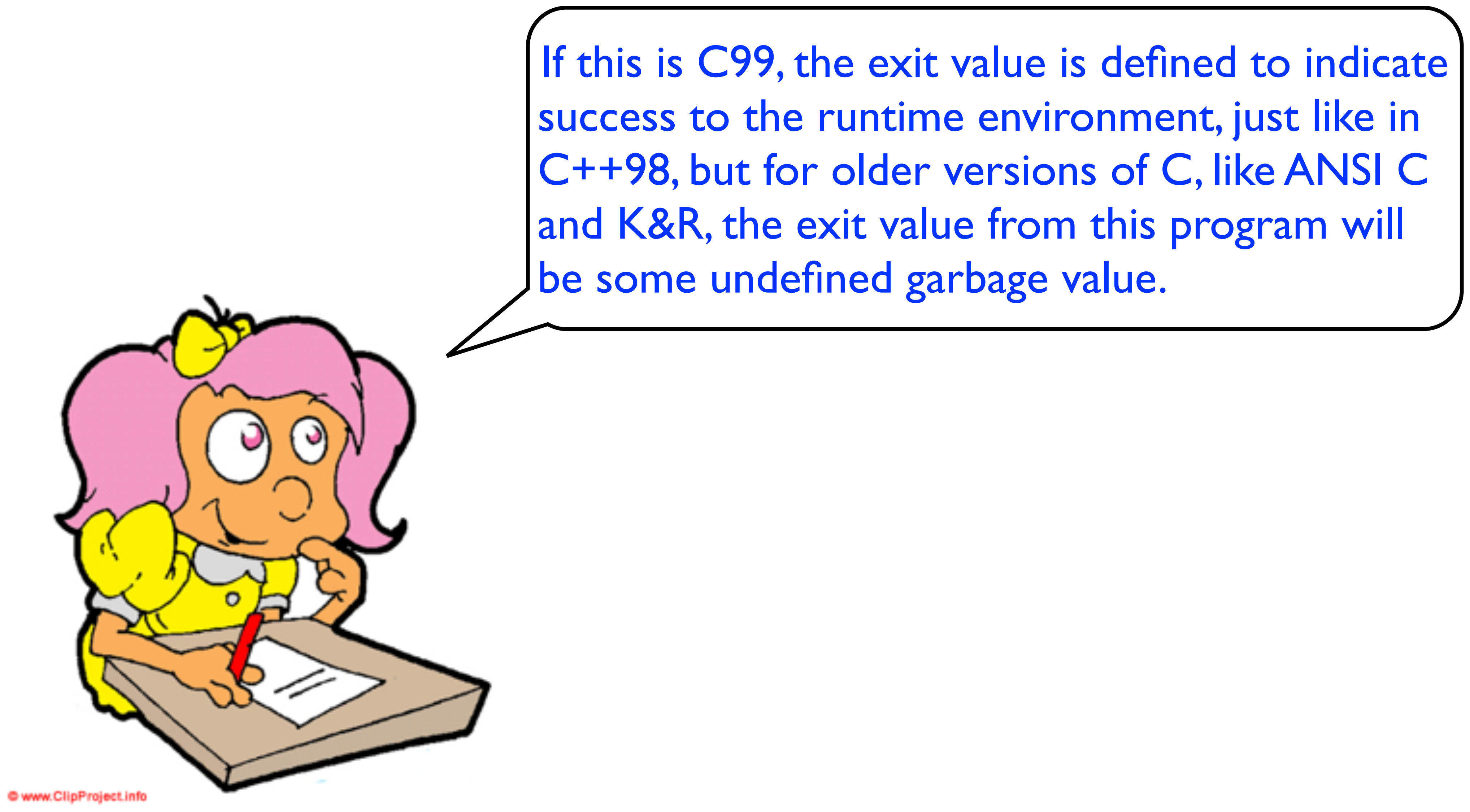
```
int main()
{
    int a = 42;
    printf("%d\n", a);
}
```

and while she is on the roll, she might continue with:

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and while she is on the roll, she might continue with:


If this is C99, the exit value is defined to indicate success to the runtime environment, just like in C++98, but for older versions of C, like ANSI C and K&R, the exit value from this program will be some undefined garbage value.
© www.ClipProject.info

What will happen if you try to compile, link and run this program?

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

and while she is on the roll, she might continue with:

If this is C99, the exit value is defined to indicate success to the runtime environment, just like in C++98, but for older versions of C, like ANSI C and K&R, the exit value from this program will be some undefined garbage value.

But since return values are often passed in a register I would not be surprised if the garbage value happens to be 3... since `printf()` will return 3, the number of characters written to standard out.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and while she is on the roll, she might continue with:

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and while she is on the roll, she might continue with:

And talking about C standards... if you want to show that you care about C programming, you should use `int main(void)` as your entry point - since the standard says so.

Using `void` to indicate no parameters is essential for declarations in C, eg a declaration ‘`int f();`’, says there is a function `f` that takes any number of arguments. While you probably meant to say ‘`int f(void);`’. Being explicit by using `void` also for function definitions does not hurt.

What will happen if you try to compile, link and run this program?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and to really show off...

Also, if you allow me to be a bit pedantic... the program is not really compliant, as the standard says that the source code must end with a newline.

What will happen if you try to compile, link and run this program?

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

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

ah, remember to include an explicit declaration of `printf()` as well

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

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

What will happen if you try to compile, link and run this program?

```c
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

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

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
```

What will happen if you try to compile, link and run this program?

```c
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

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

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

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

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
```

What will happen if you try to compile, link and run this program?

```c
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
$ ./a.out
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
$ ./a.out
42
```

What will happen if you try to compile, link and run this program?

```c
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
```

What will happen if you try to compile, link and run this program?

```
#include <stdio.h>

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

and here is what I get when compiling, linking and running the above program on my machine:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
0
```

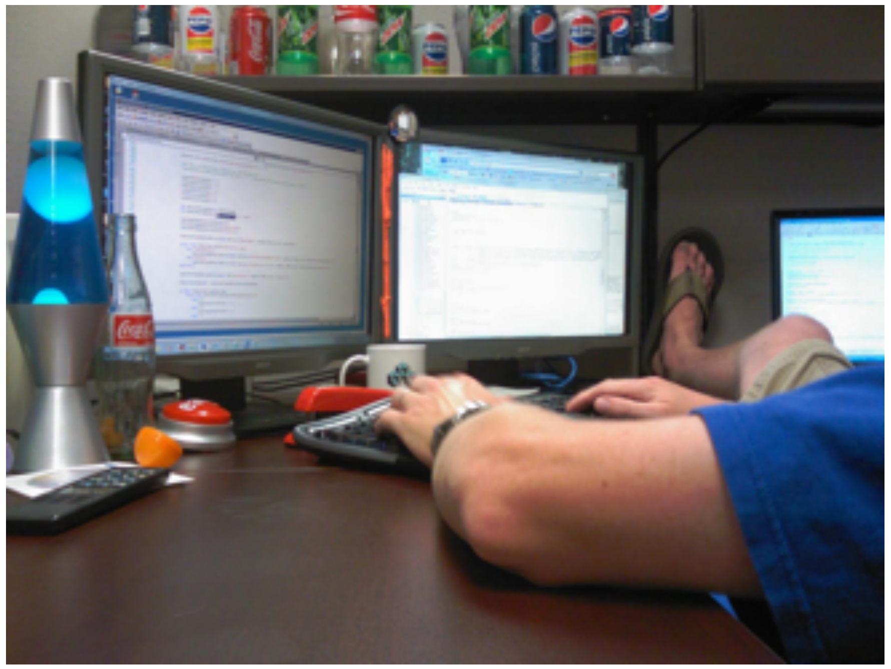
Is there any difference between these two candidates?

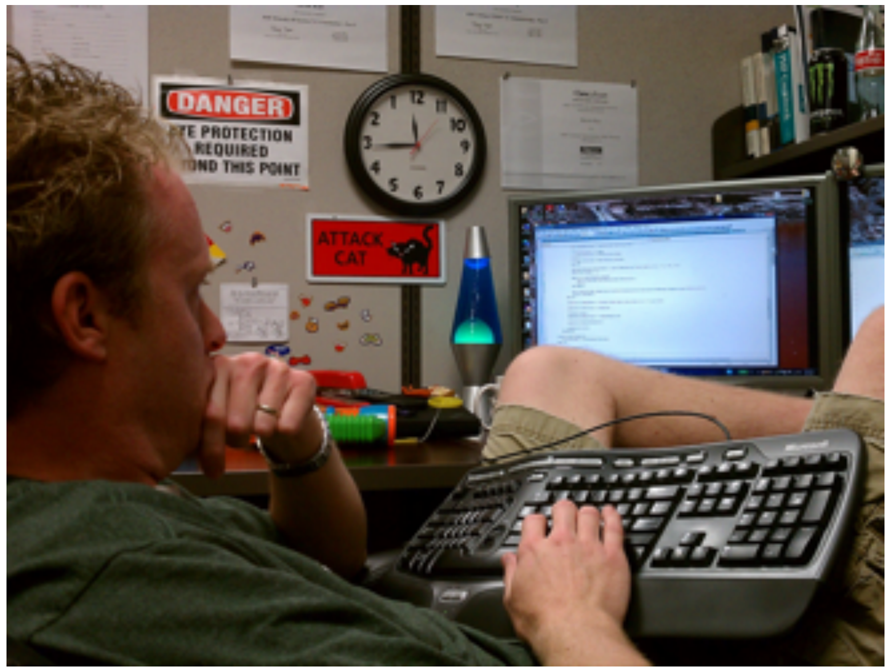
Is there any difference between these two candidates?

Not much, yet, but I really like her answers so far.

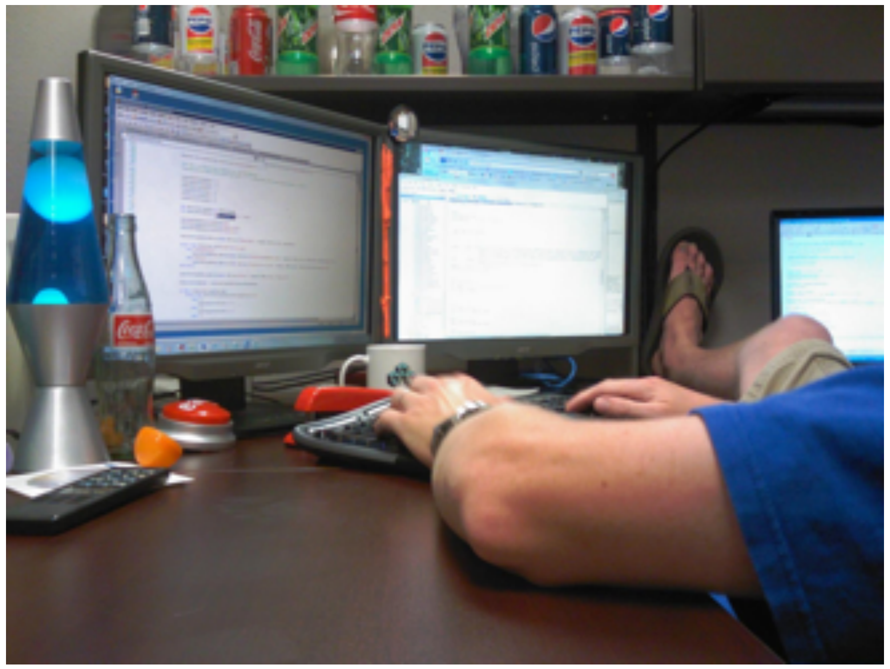
Now suppose they are not really candidates. Perhaps they are stereotypes for engineers working in your organization?

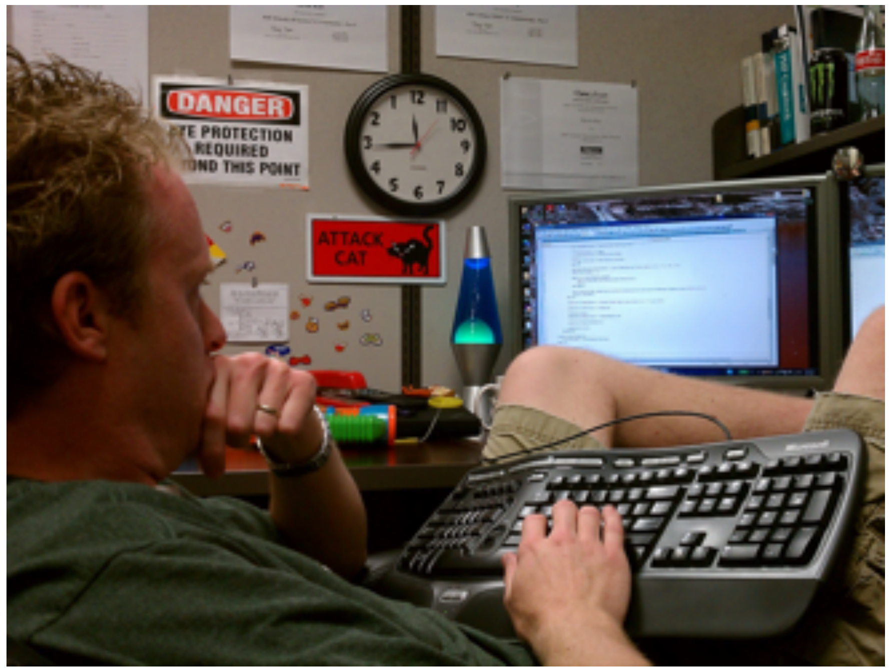
Now suppose they are not really candidates. Perhaps they are stereotypes for engineers working in your organization?



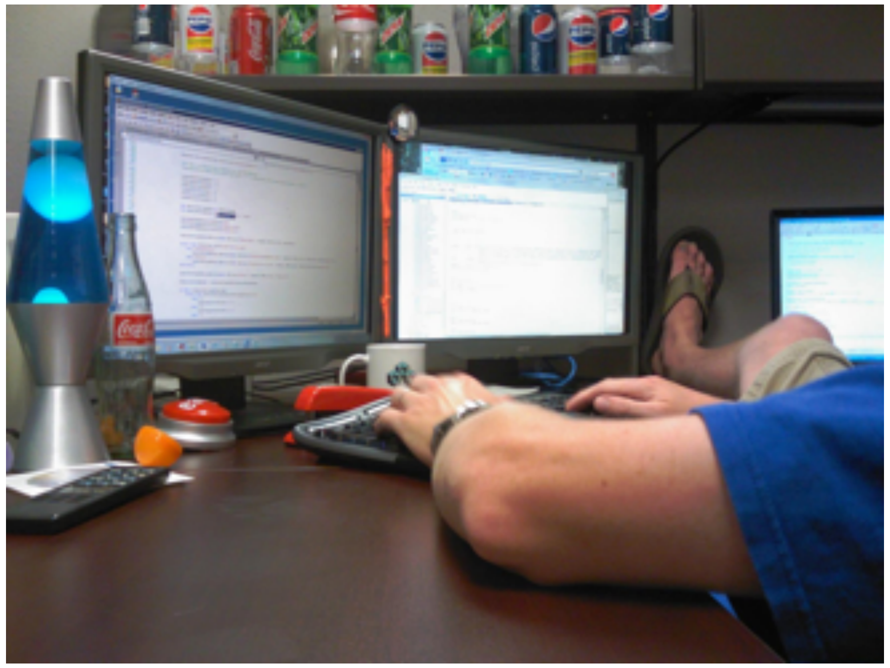


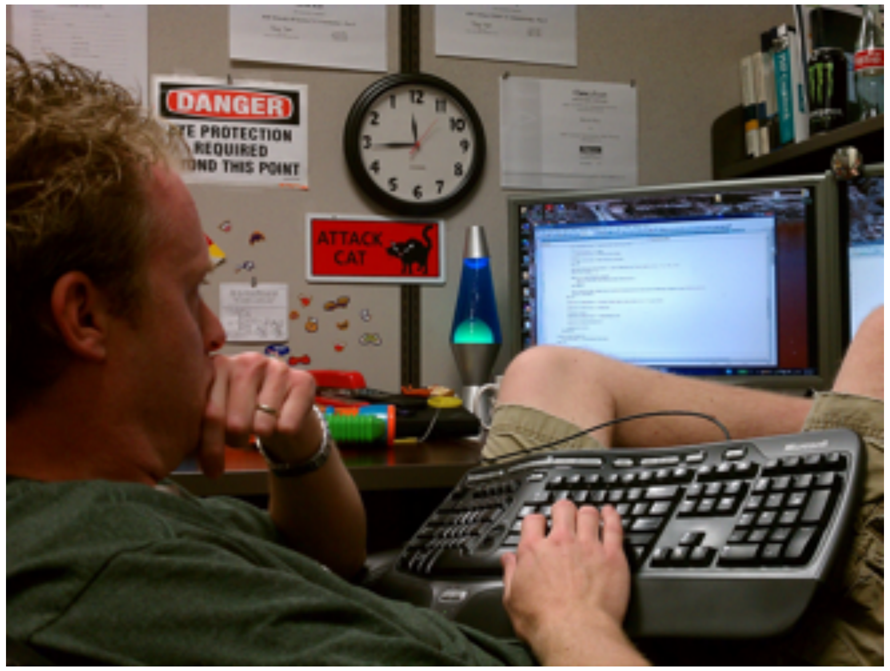
Now suppose they are not really candidates. Perhaps they are stereotypes for engineers working in your organization?





Would it be useful if most of your colleagues have a deep understanding of the programming language they are using?





Let’s find out how deep their knowledge of C and C++ is...

Let’s find out how deep their knowledge of C and C++ is...

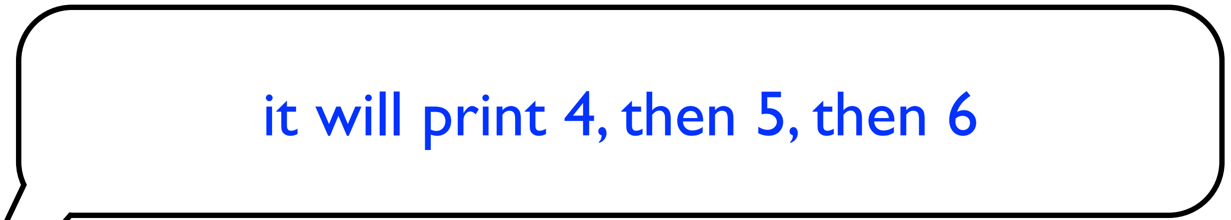
```c
#include <stdio.h>

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

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

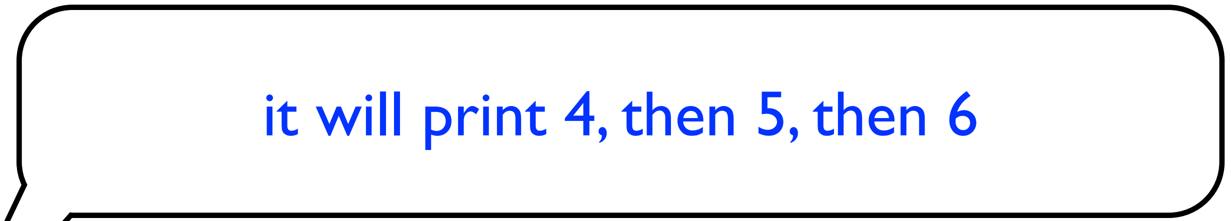
```
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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





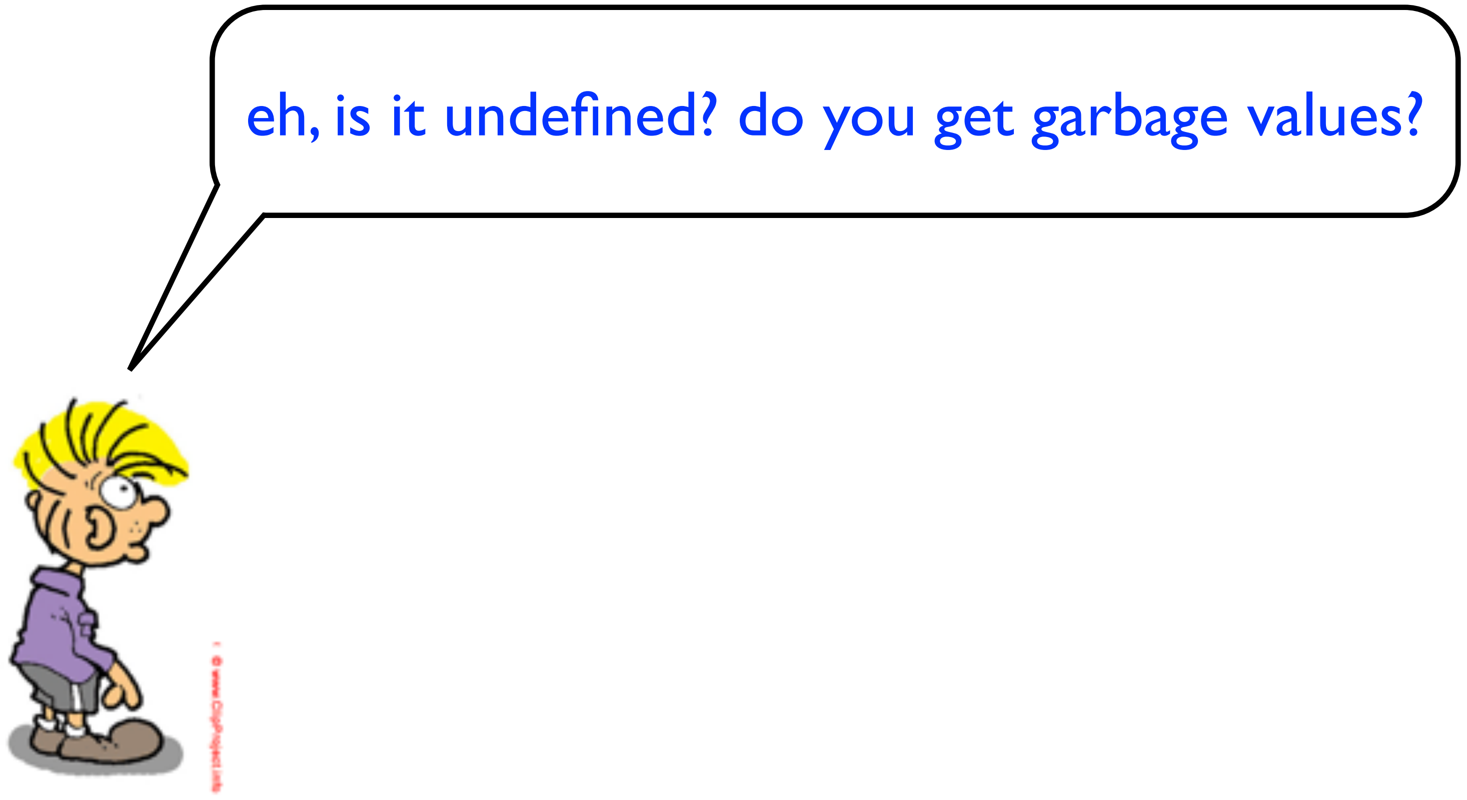
```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

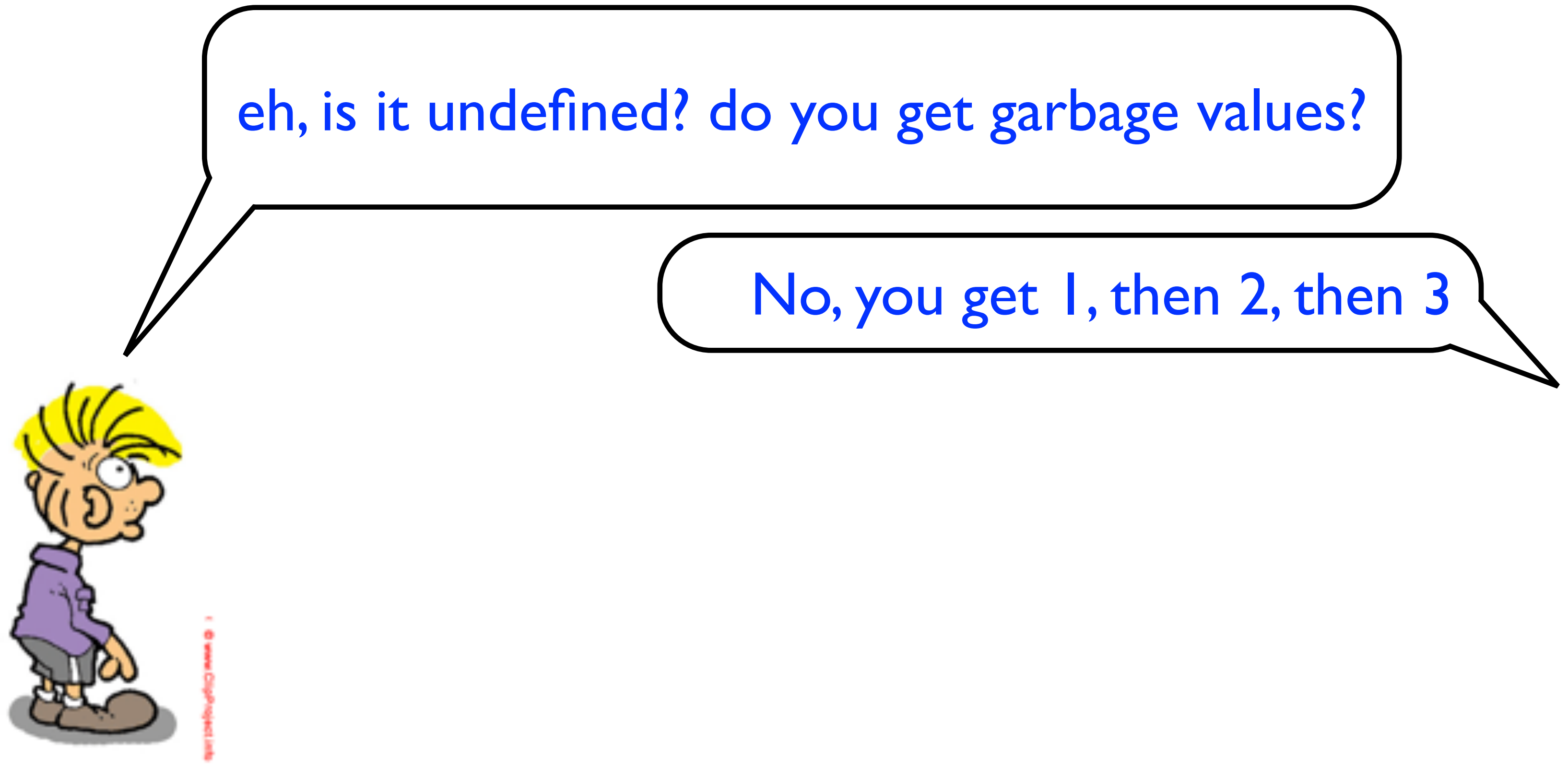
```
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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

```
#include <stdio.h>

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


eh, is it undefined? do you get garbage values?
No, you get 1, then 2, then 3

```c
#include <stdio.h>

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

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

eh, is it undefined? do you get garbage values?

No, you get 1, then 2, then 3

ok, I see... why?

```c
#include <stdio.h>

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

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

eh, is it undefined? do you get garbage values?

No, you get 1, then 2, then 3

ok, I see... why?

because static variables are set to 0

```
#include <stdio.h>

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

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

eh, is it undefined? do you get garbage values?

No, you get 1, then 2, then 3

ok, I see... why?

because static variables are set to 0

```c
#include <stdio.h>

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

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

eh, is it undefined? do you get garbage values?

No, you get 1, then 2, then 3

ok, I see... why?

because static variables are set to 0

the standard says that static variables are initialized to 0, so this should print 1, then 2, then 3

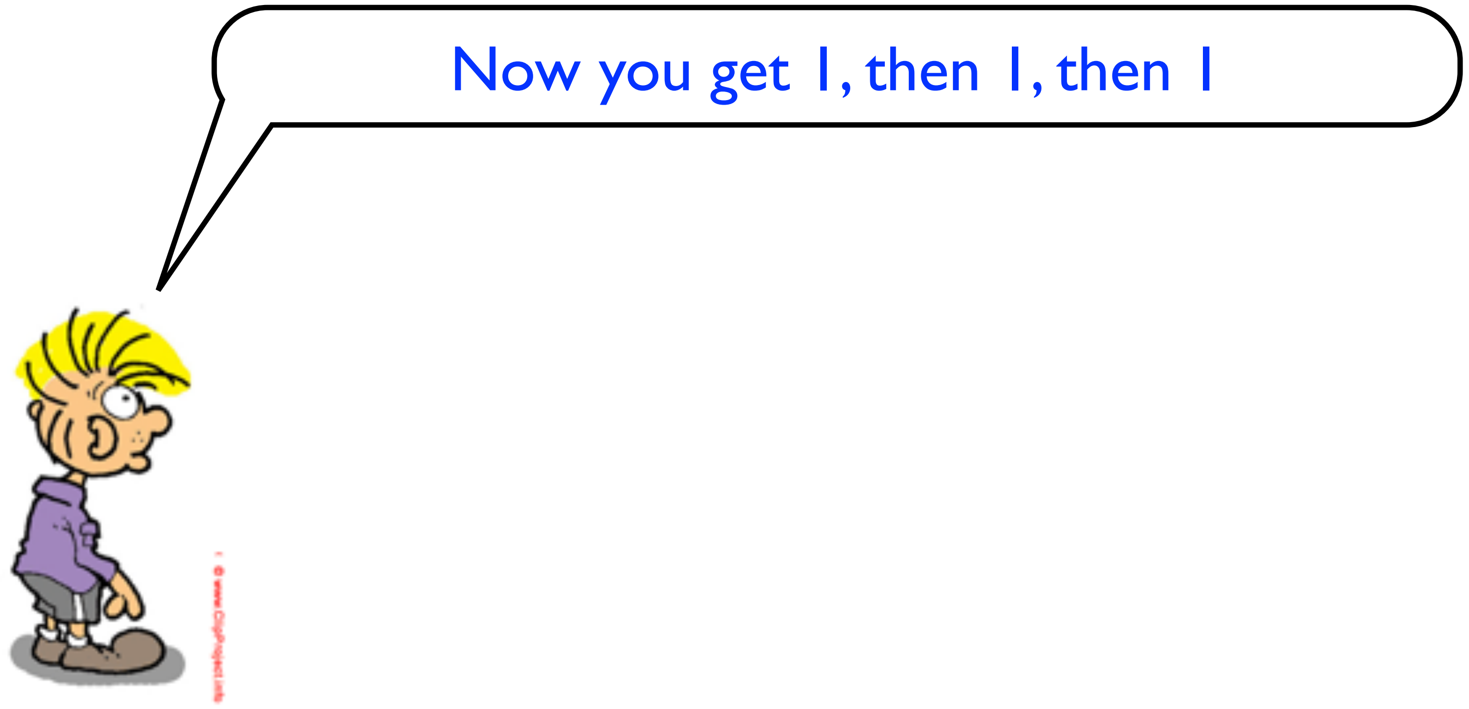
```c
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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

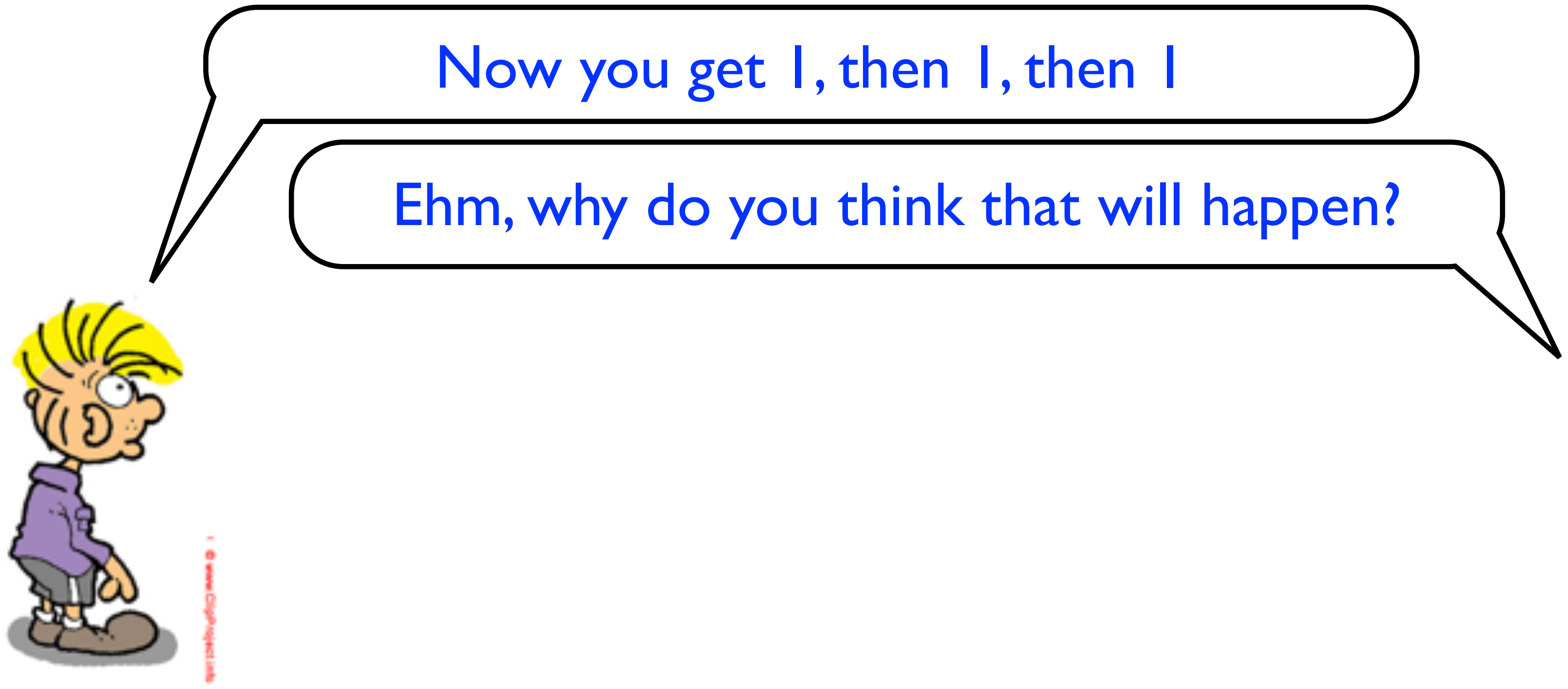
```c
#include <stdio.h>

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();
}
```

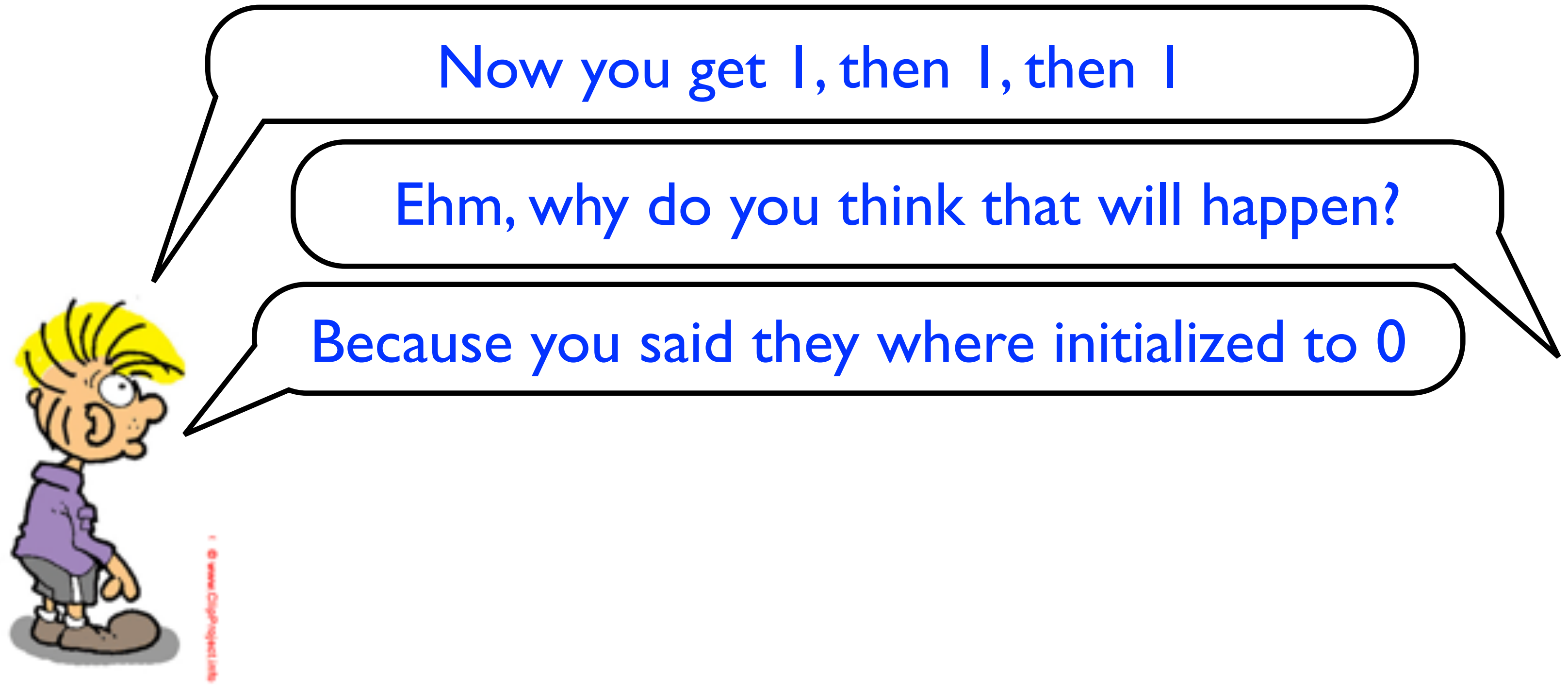
```
#include <stdio.h>

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

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



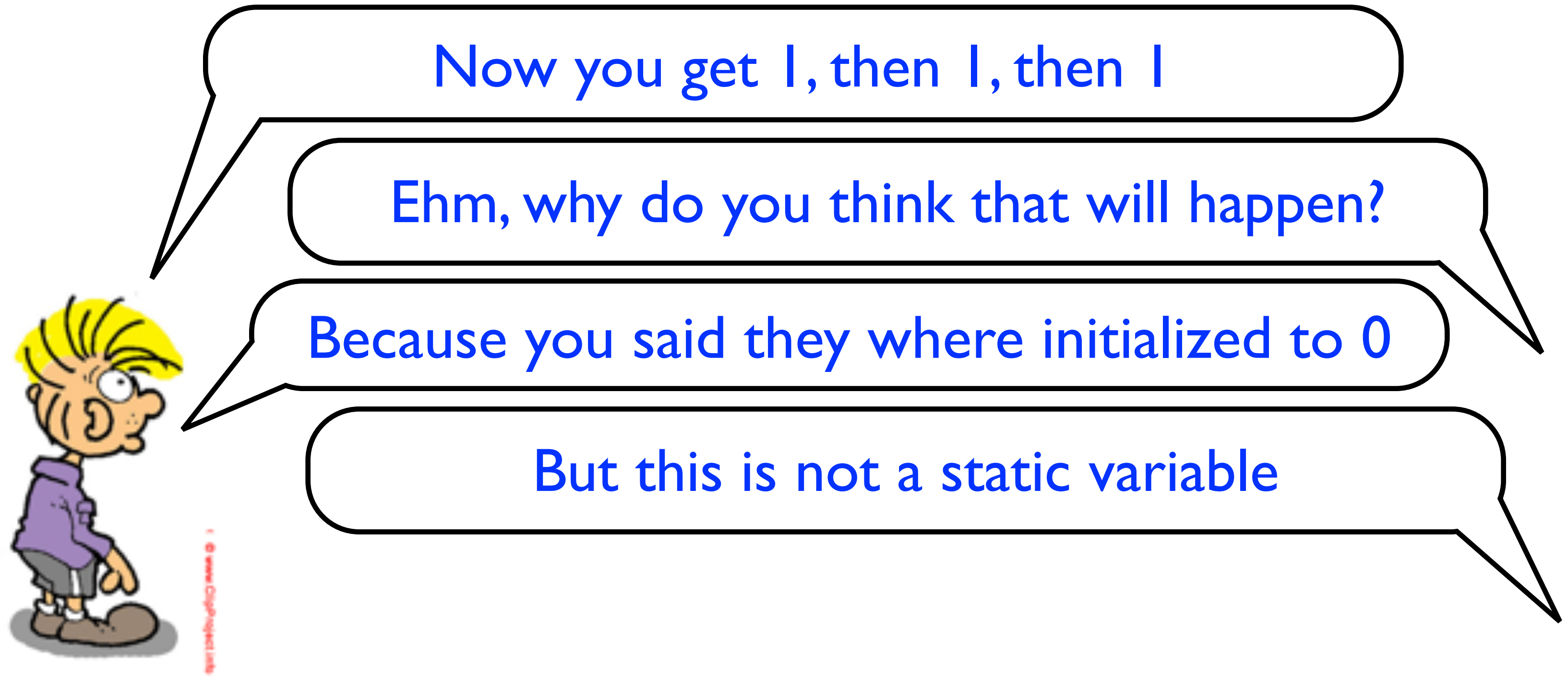
```
#include <stdio.h>

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

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


Now you get 1, then 1, then 1
Ehm, why do you think that will happen?

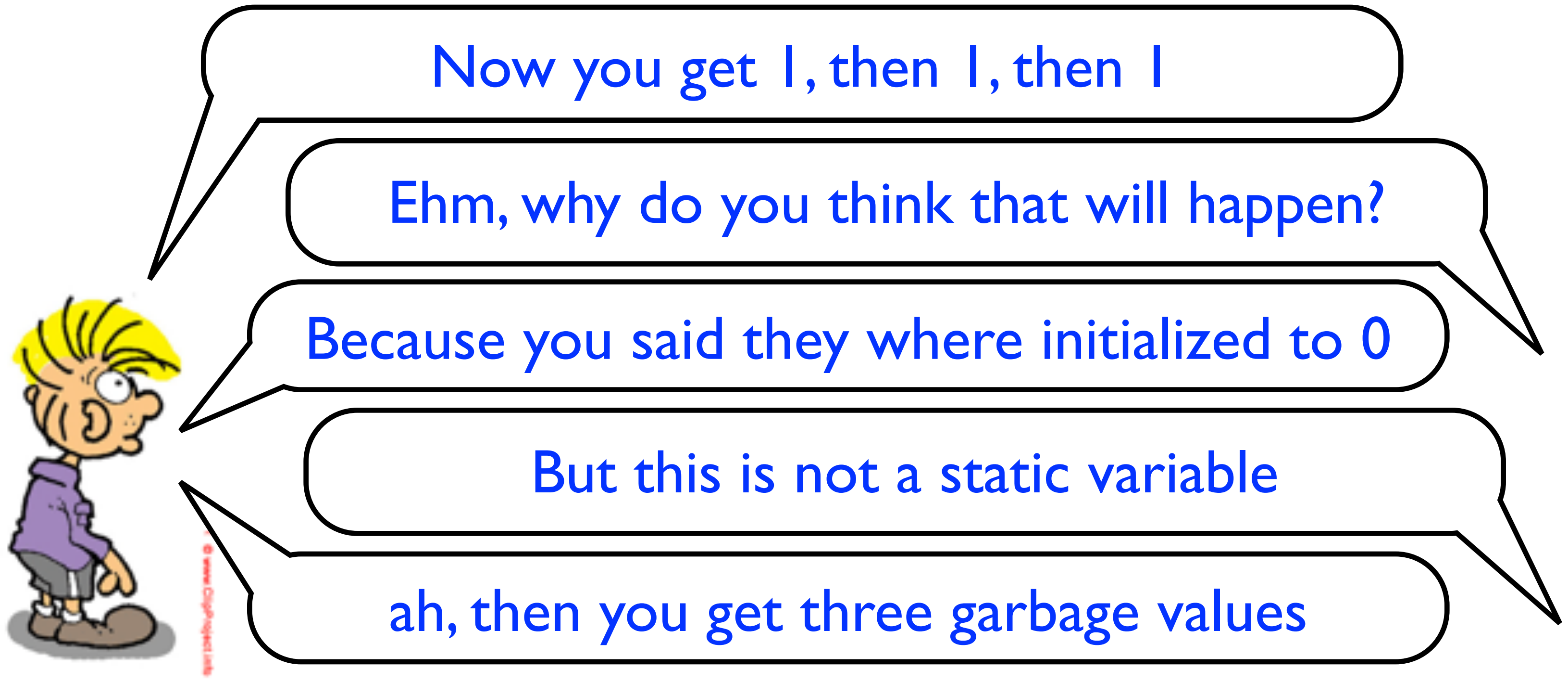
```
#include <stdio.h>

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

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


Now you get 1, then 1, then 1
Ehm, why do you think that will happen?
Because you said they where initialized to 0

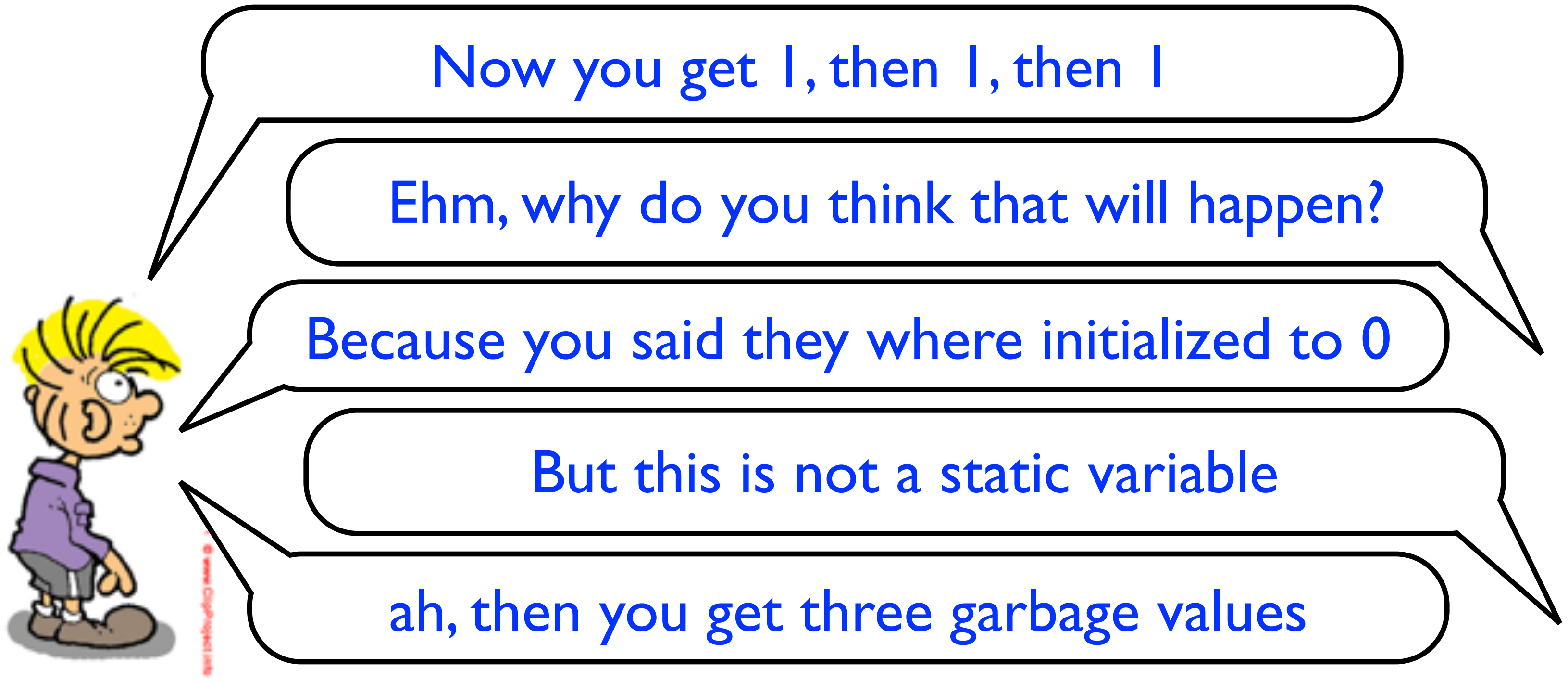
```c
#include <stdio.h>

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

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


Now you get 1, then 1, then 1
Ehm, why do you think that will happen?
Because you said they where initialized to 0
But this is not a static variable

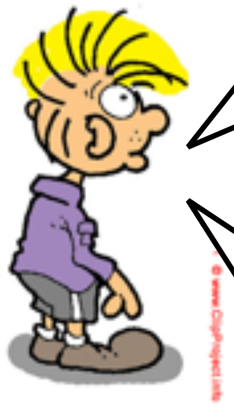
```
#include <stdio.h>

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

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


Now you get 1, then 1, then 1
Ehm, why do you think that will happen?
Because you said they where initialized to 0
But this is not a static variable
ah, then you get three garbage values

```c
#include <stdio.h>

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

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


Now you get 1, then 1, then 1
Ehm, why do you think that will happen?
Because you said they where initialized to 0
But this is not a static variable
ah, then you get three garbage values

```
#include <stdio.h>

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

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



Now you get 1, then 1, then 1

Ehm, why do you think that will happen?

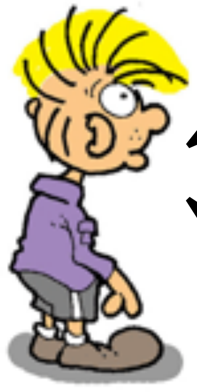
Because you said they where initialized to 0

But this is not a static variable

ah, then you get three garbage values

the value of `a` will be undefinded, so in theory you get three garbage values. In practice however, since auto variables are often allocated on an execution stack, `a` might get the same memory location each time and you might get three consecutive values... if you compile without optimization.

```
#include <stdio.h>

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

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



Now you get 1, then 1, then 1

Ehm, why do you think that will happen?

Because you said they where initialized to 0

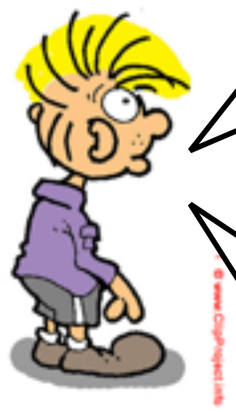
But this is not a static variable

ah, then you get three garbage values

the value of a will be undefinded, so in theory you get three garbage values. In practice however, since auto variables are often allocated on an execution stack, a might get the same memory location each time and you might get three consecutive values... if you compile without optimization.

on my machine I actually get, 1, then 2, then 3

```c
#include <stdio.h>

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

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



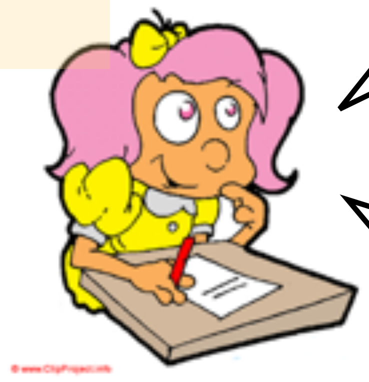
Now you get 1, then 1, then 1

Ehm, why do you think that will happen?

Because you said they where initialized to 0

But this is not a static variable

ah, then you get three garbage values



the value of `a` will be undefinded, so in theory you get three garbage values. In practice however, since auto variables are often allocated on an execution stack, `a` might get the same memory location each time and you might get three consecutive values... if you compile without optimization.

on my machine I actually get, 1, then 2, then 3

I am not surprised... if you compile in debug mode the runtime might try to be helpful and memset your stack memory to 0

```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();
}
```

Why do you think static variables are set to 0, while auto variables are not initialized?

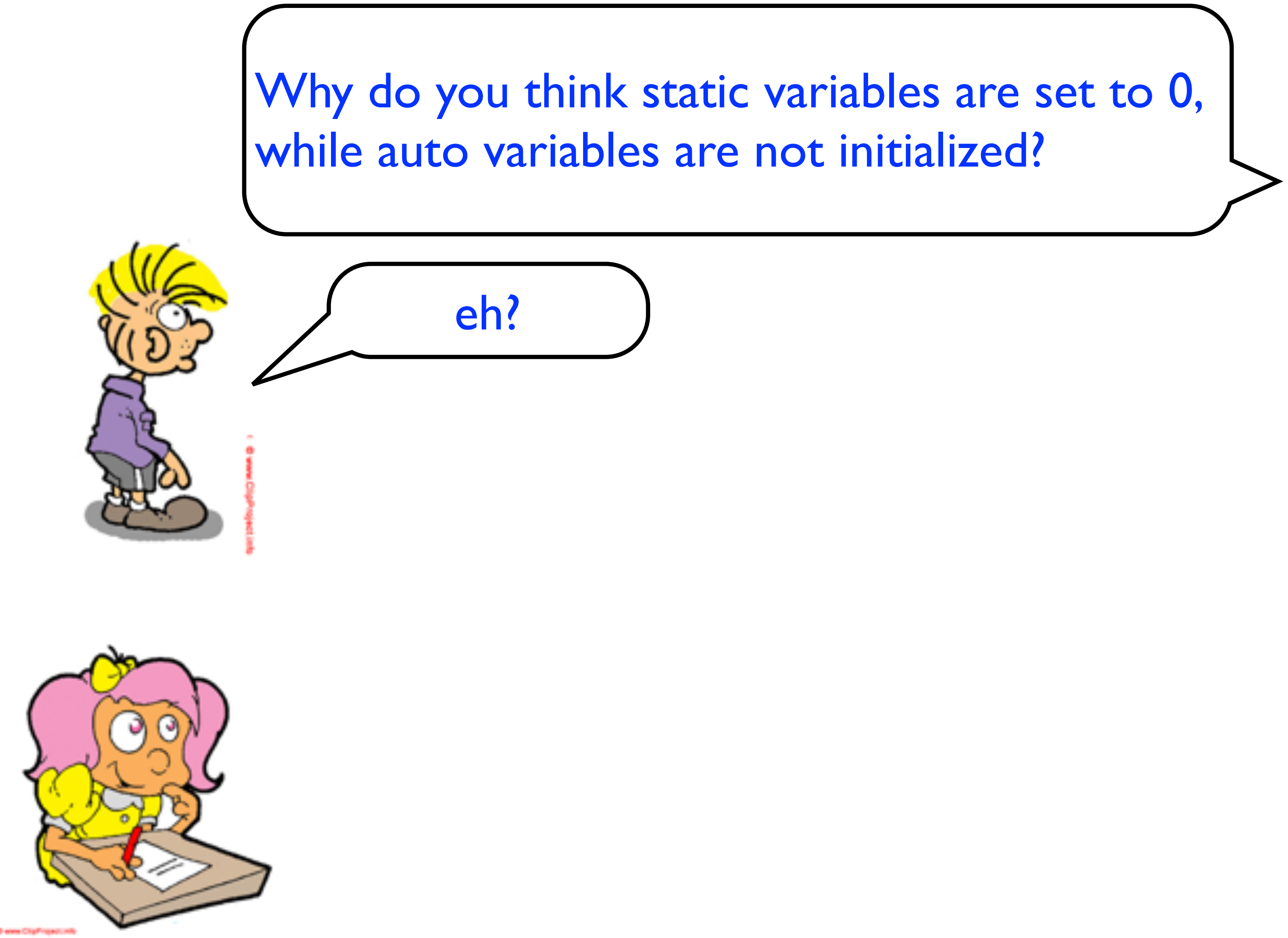
```
#include <stdio.h>

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

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

Why do you think static variables are set to 0, while auto variables are not initialized?

```
#include <stdio.h>

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

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

Why do you think static variables are set to 0, while auto variables are not initialized?

```
#include <stdio.h>

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

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


Why do you think static variables are set to 0, while auto variables are not initialized?
eh?

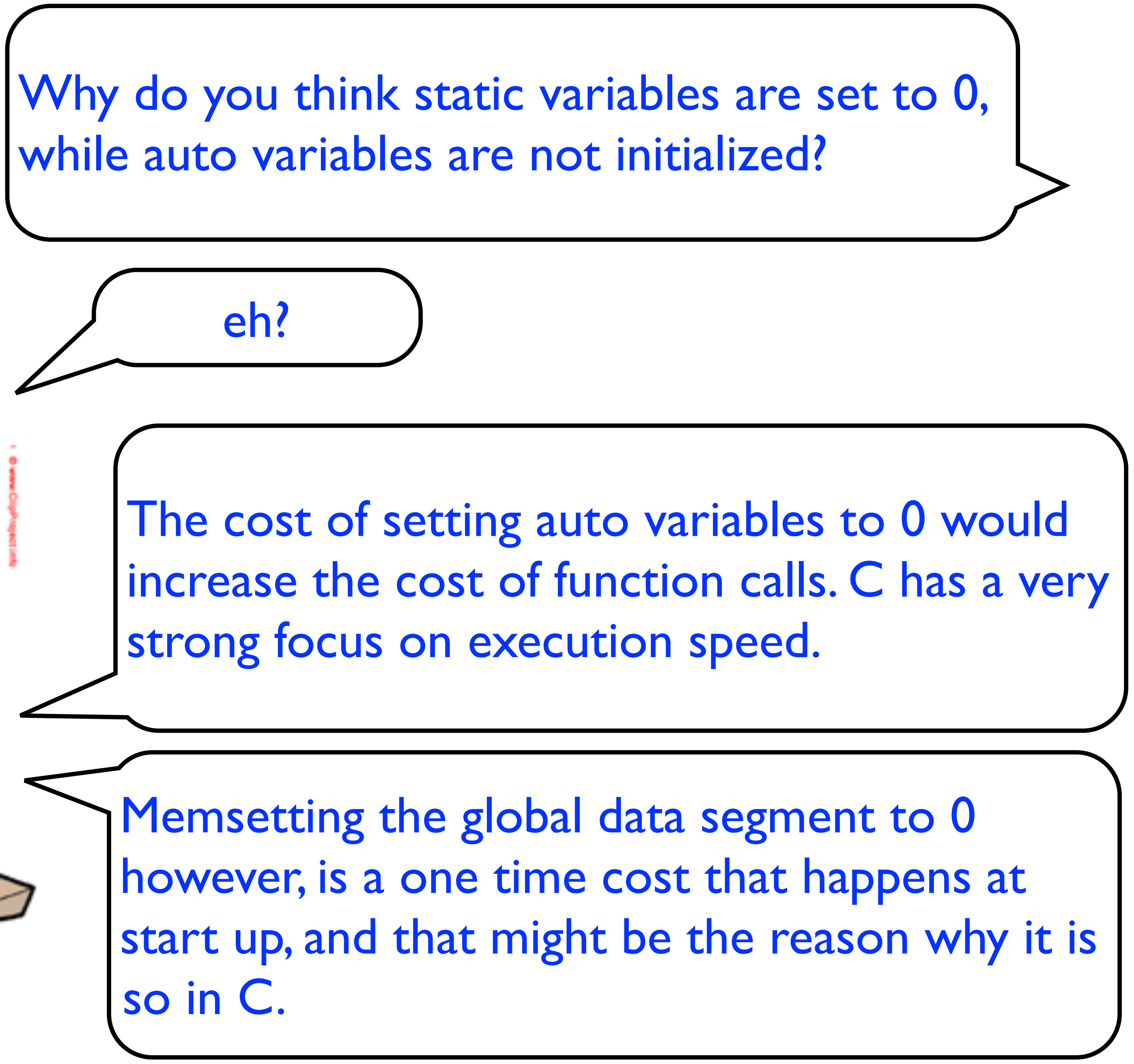
```
#include <stdio.h>

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

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

Why do you think static variables are set to 0, while auto variables are not initialized?

eh?

The cost of setting auto variables to 0 would increase the cost of function calls. C has a very strong focus on execution speed.

```
#include <stdio.h>

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

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


Why do you think static variables are set to 0, while auto variables are not initialized?
eh?
The cost of setting auto variables to 0 would increase the cost of function calls. C has a very strong focus on execution speed.
Memsetting the global data segment to 0 however, is a one time cost that happens at start up, and that might be the reason why it is so in C.

```
#include <stdio.h>

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

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

Why do you think static variables are set to 0, while auto variables are not initialized?

The cost of setting auto variables to 0 would increase the cost of function calls. C has a very strong focus on execution speed.

Memsetting the global data segment to 0 however, is a one time cost that happens at start up, and that might be the reason why it is so in C.

And to be precise, in C++ however, static variables are not set to 0, they are set to their default values... which for native types means 0.

```
#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();
}
```

```c
#include <stdio.h>



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

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

```
#include <stdio.h>

static int a;

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

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

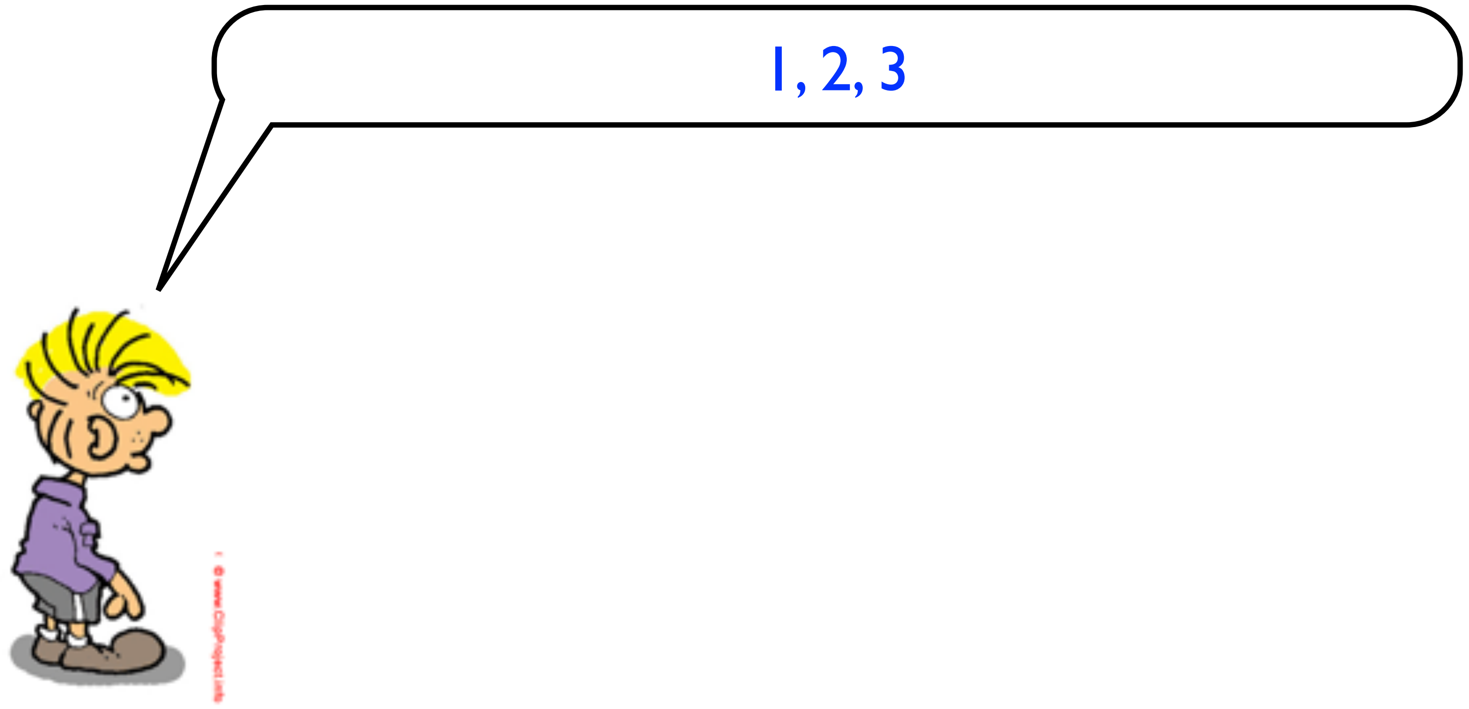
```
#include <stdio.h>

static int a;

void foo(void)
{

    ++a;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

static int a;

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

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

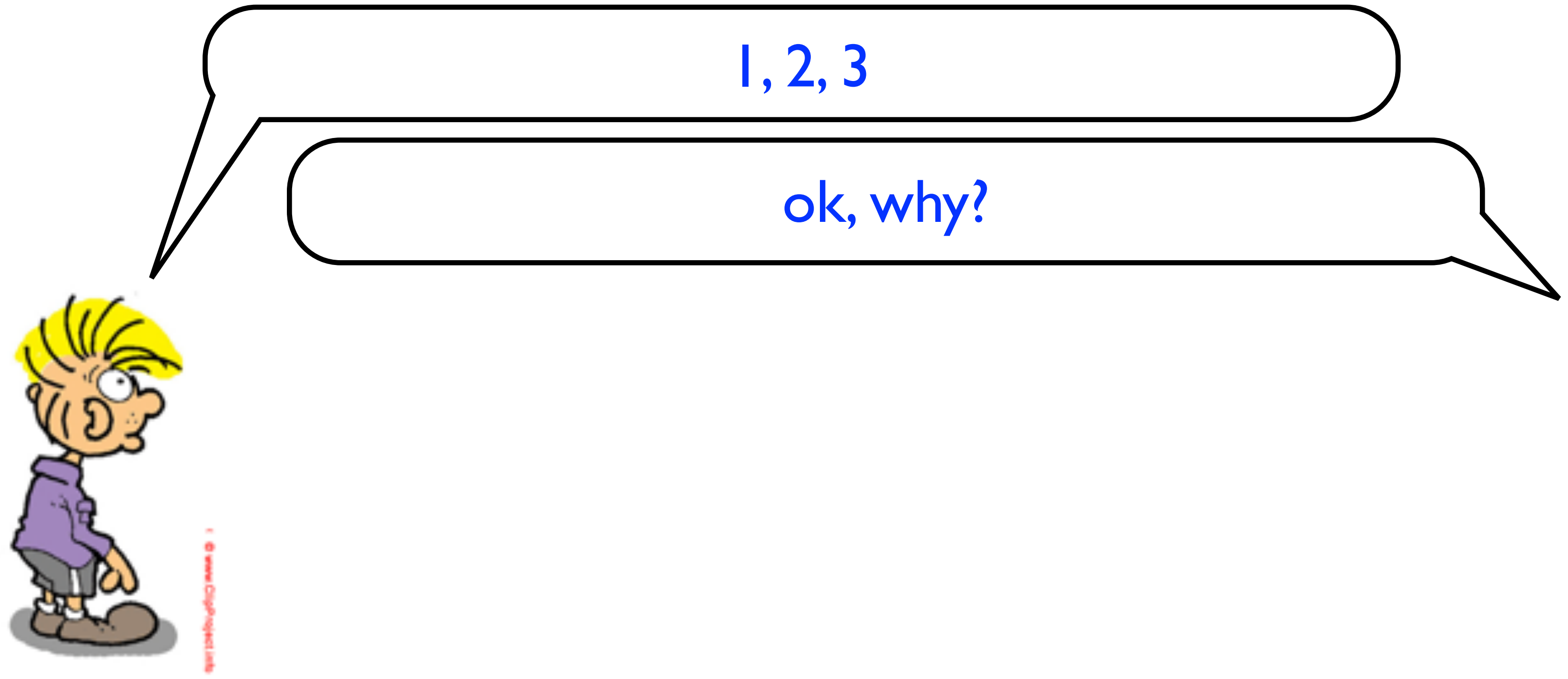
```
#include <stdio.h>

static int a;

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

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

```
#include <stdio.h>

static int a;

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

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

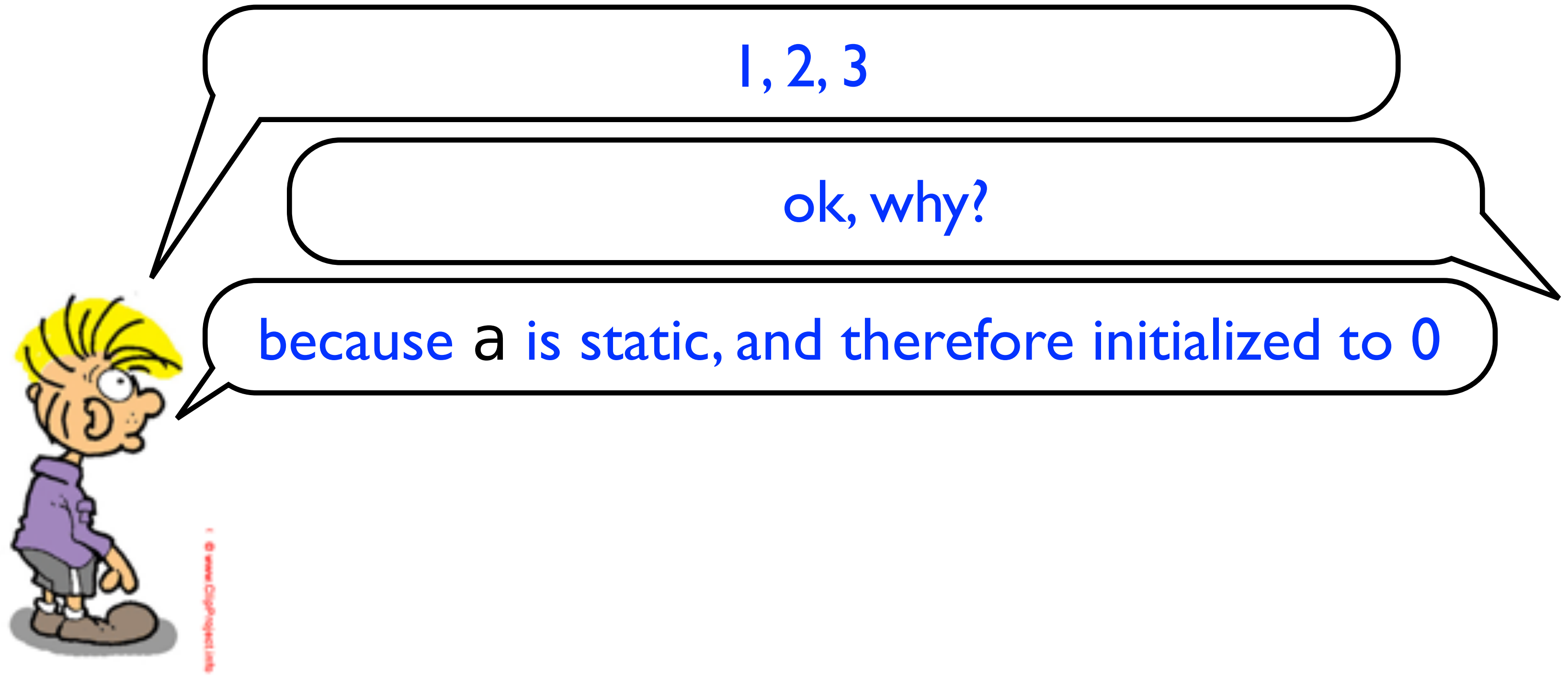
```
#include <stdio.h>

static int a;

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

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



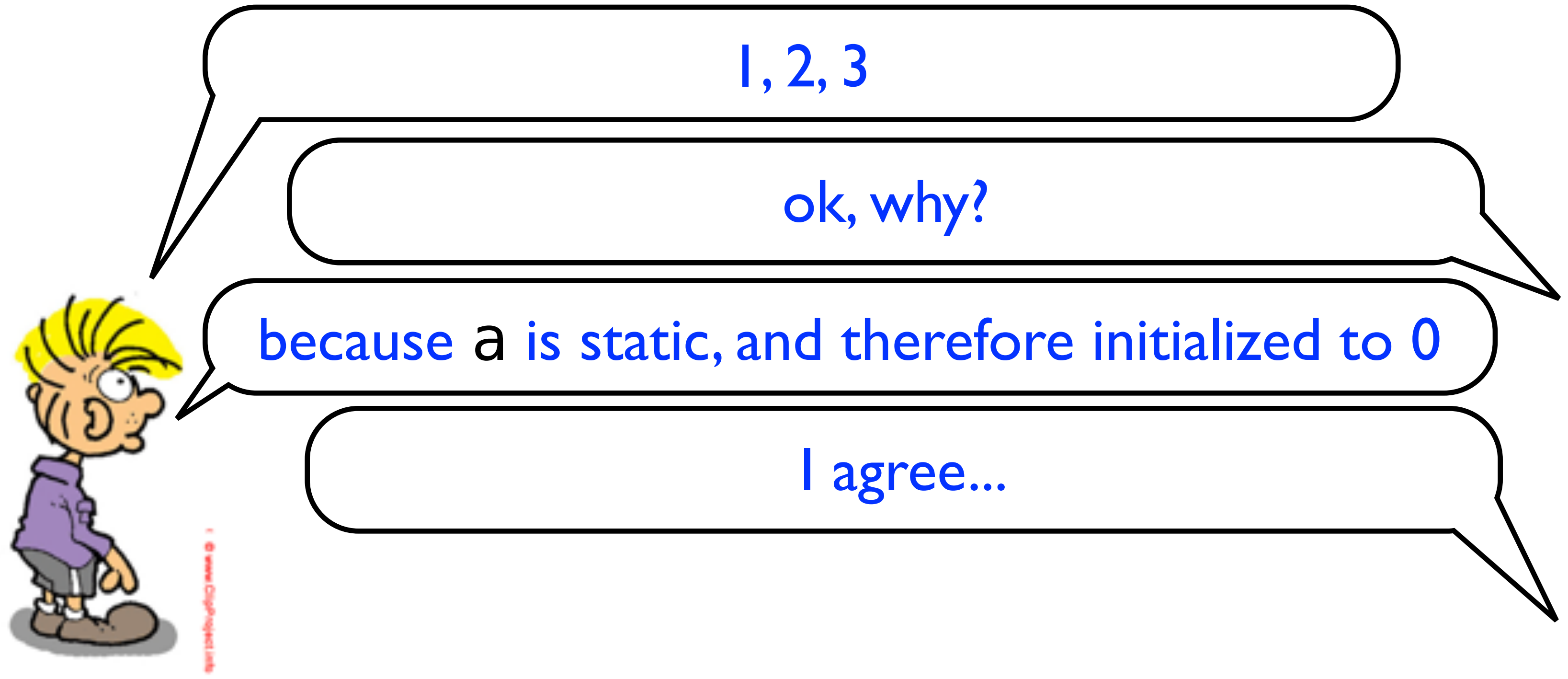
```c
#include <stdio.h>

static int a;

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

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



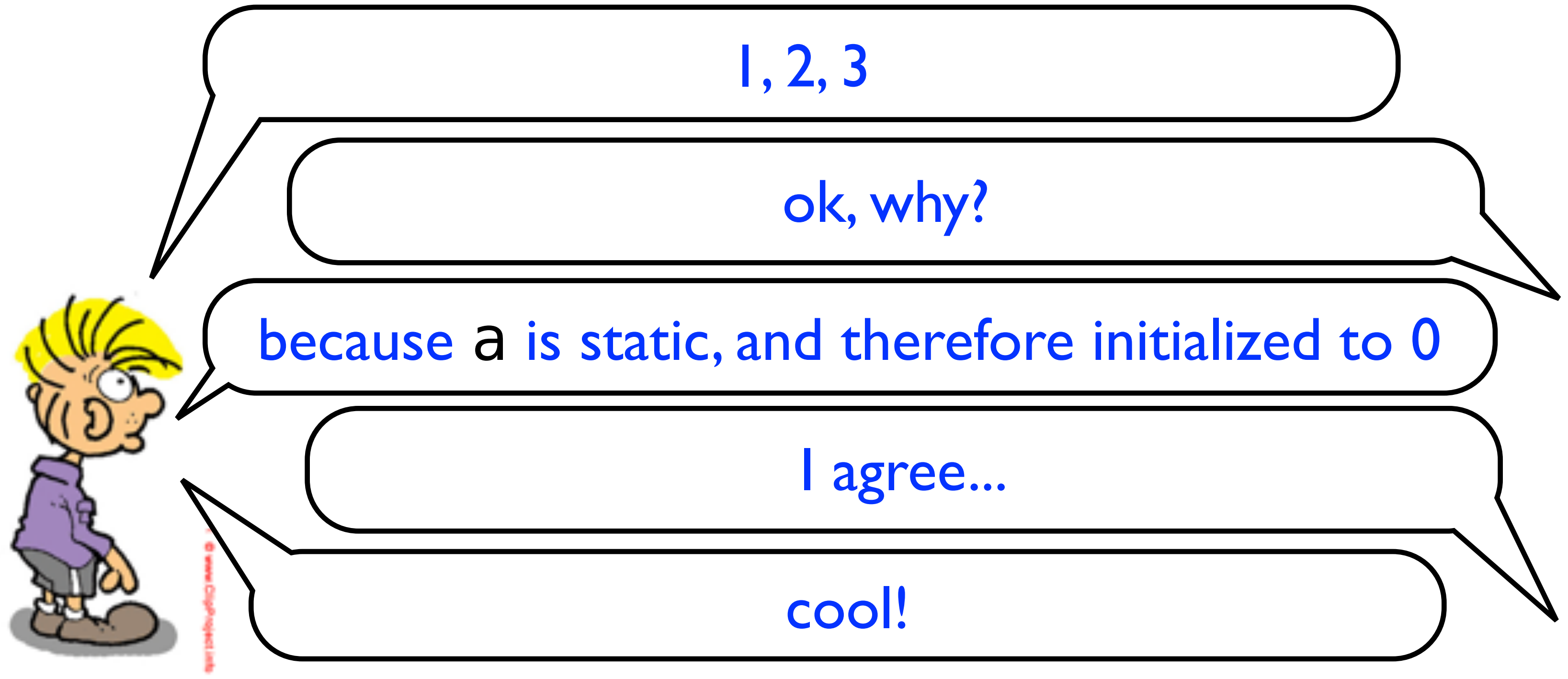
```
#include <stdio.h>

static int a;

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

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



```c
#include <stdio.h>

static int a;

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

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



```
#include <stdio.h>

static int a;

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

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



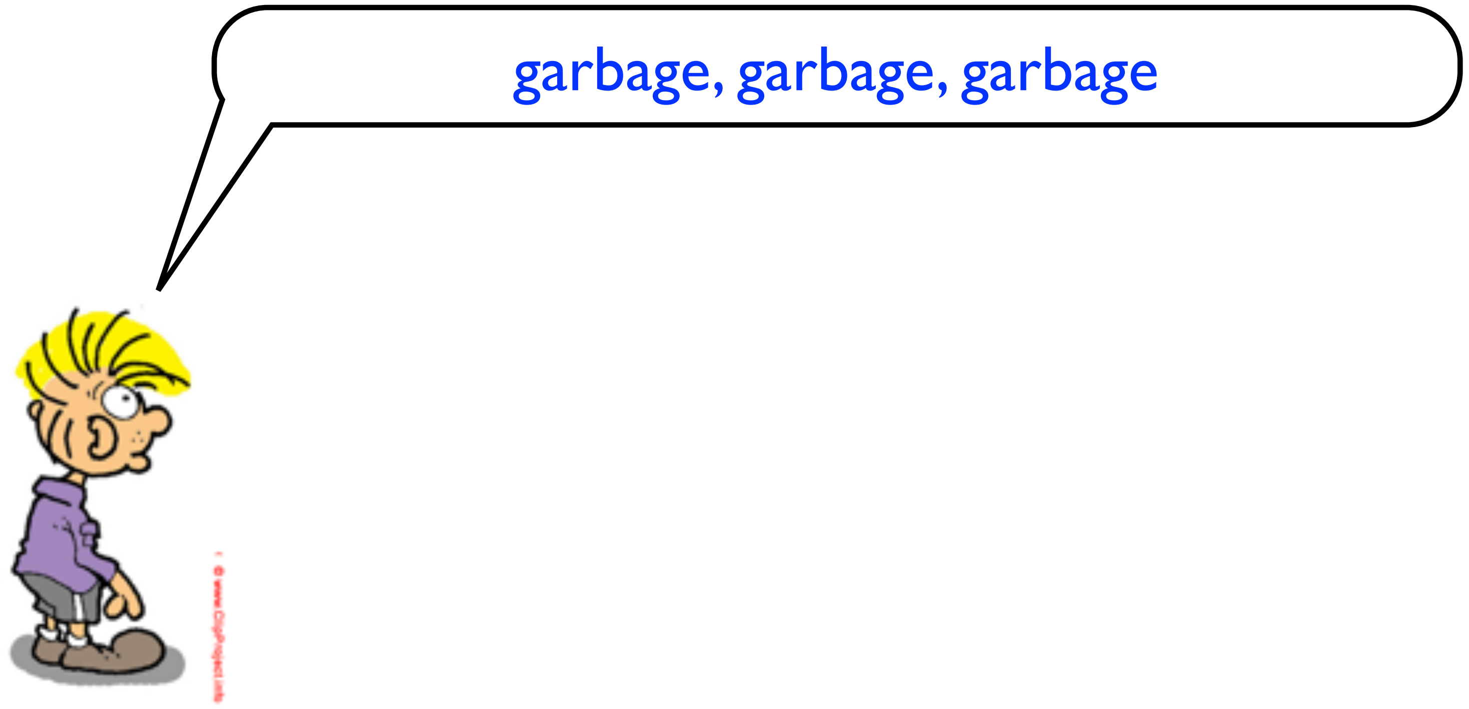
```c
#include <stdio.h>

static int a;

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

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

```
#include <stdio.h>

int a;

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

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

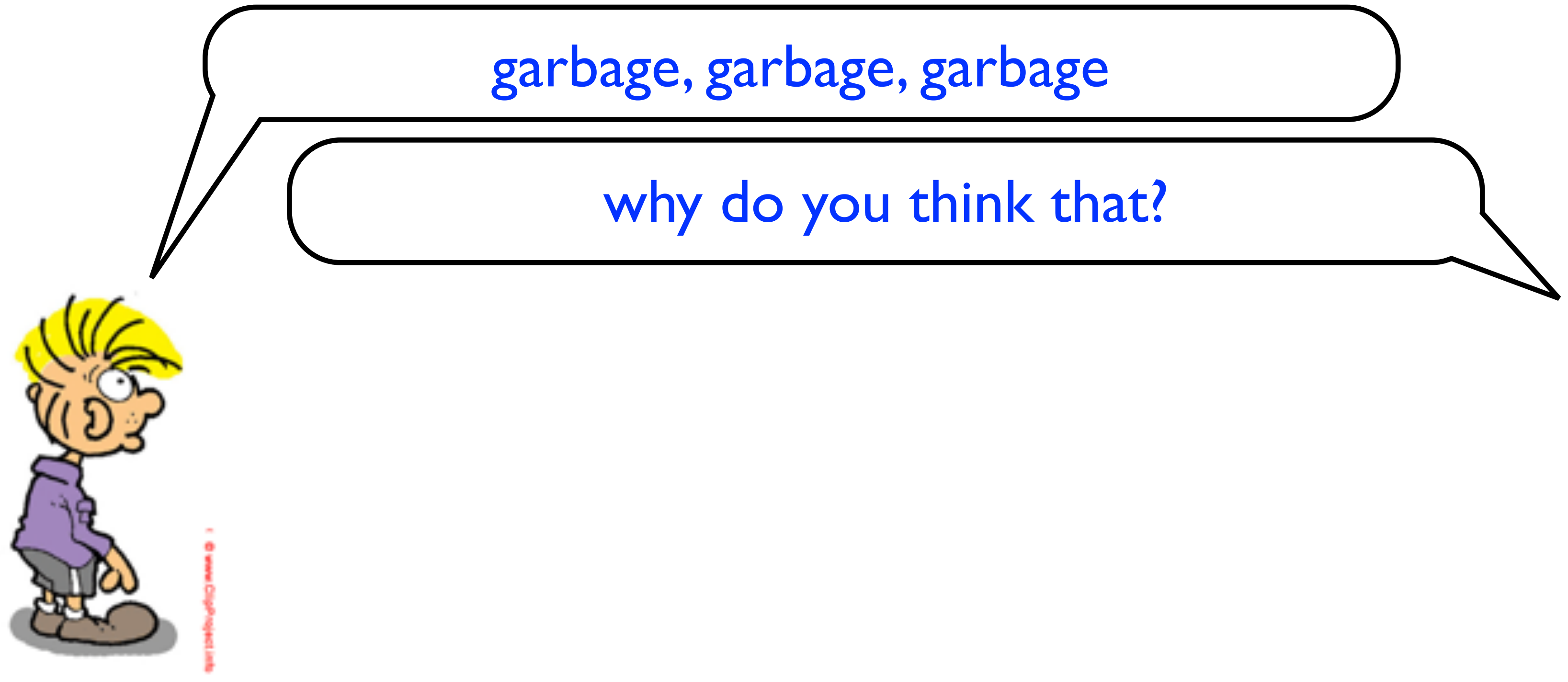
```c
#include <stdio.h>

int a;

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

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

```c
#include <stdio.h>

int a;

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

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

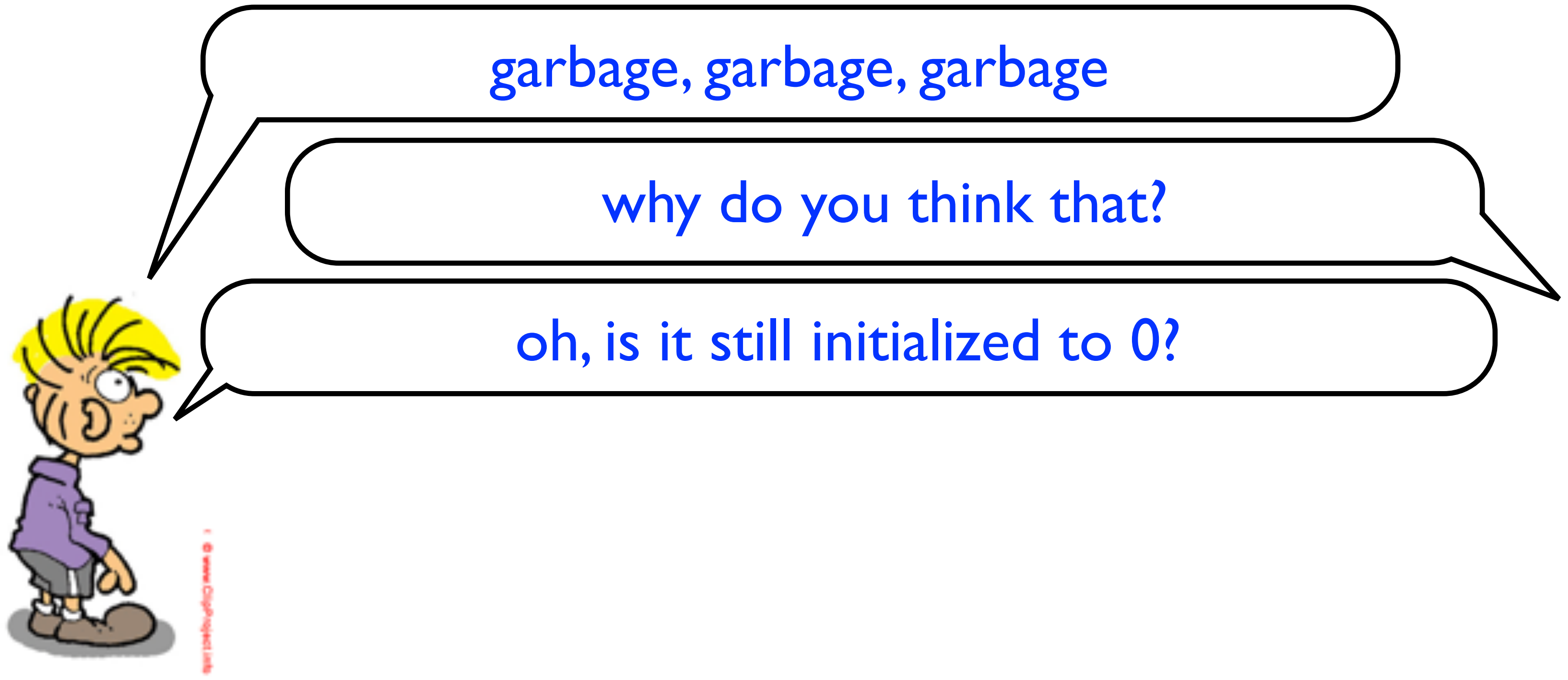
```
#include <stdio.h>

int a;

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

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



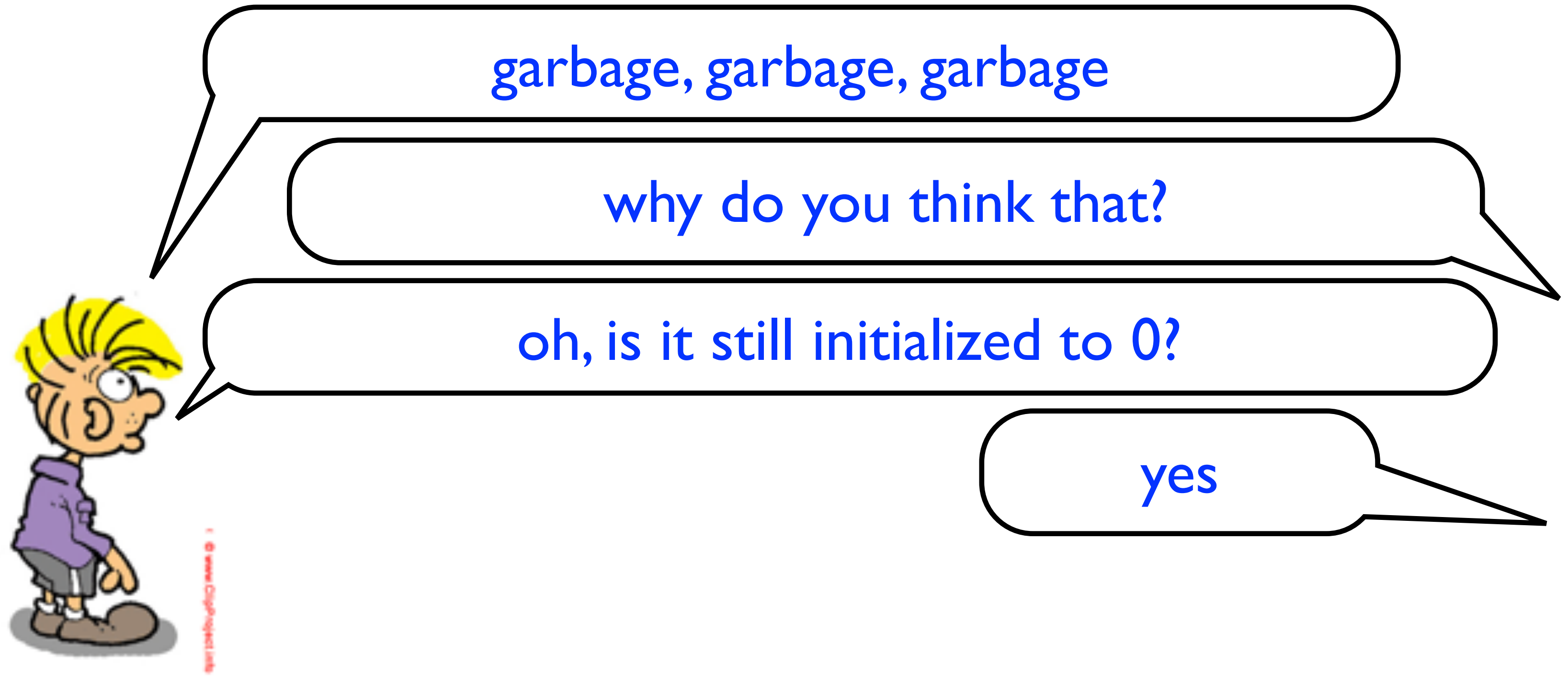
```
#include <stdio.h>

int a;

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

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



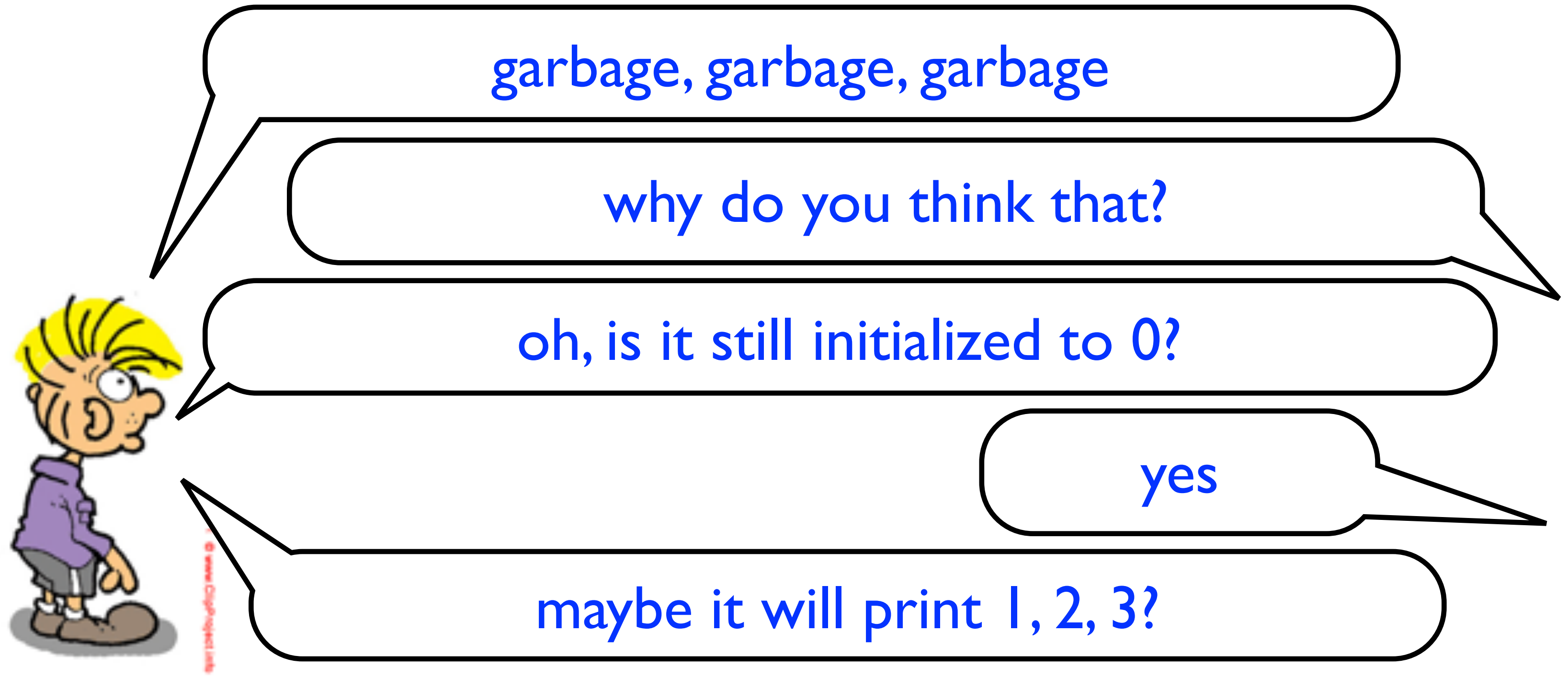
```
#include <stdio.h>

int a;

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

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



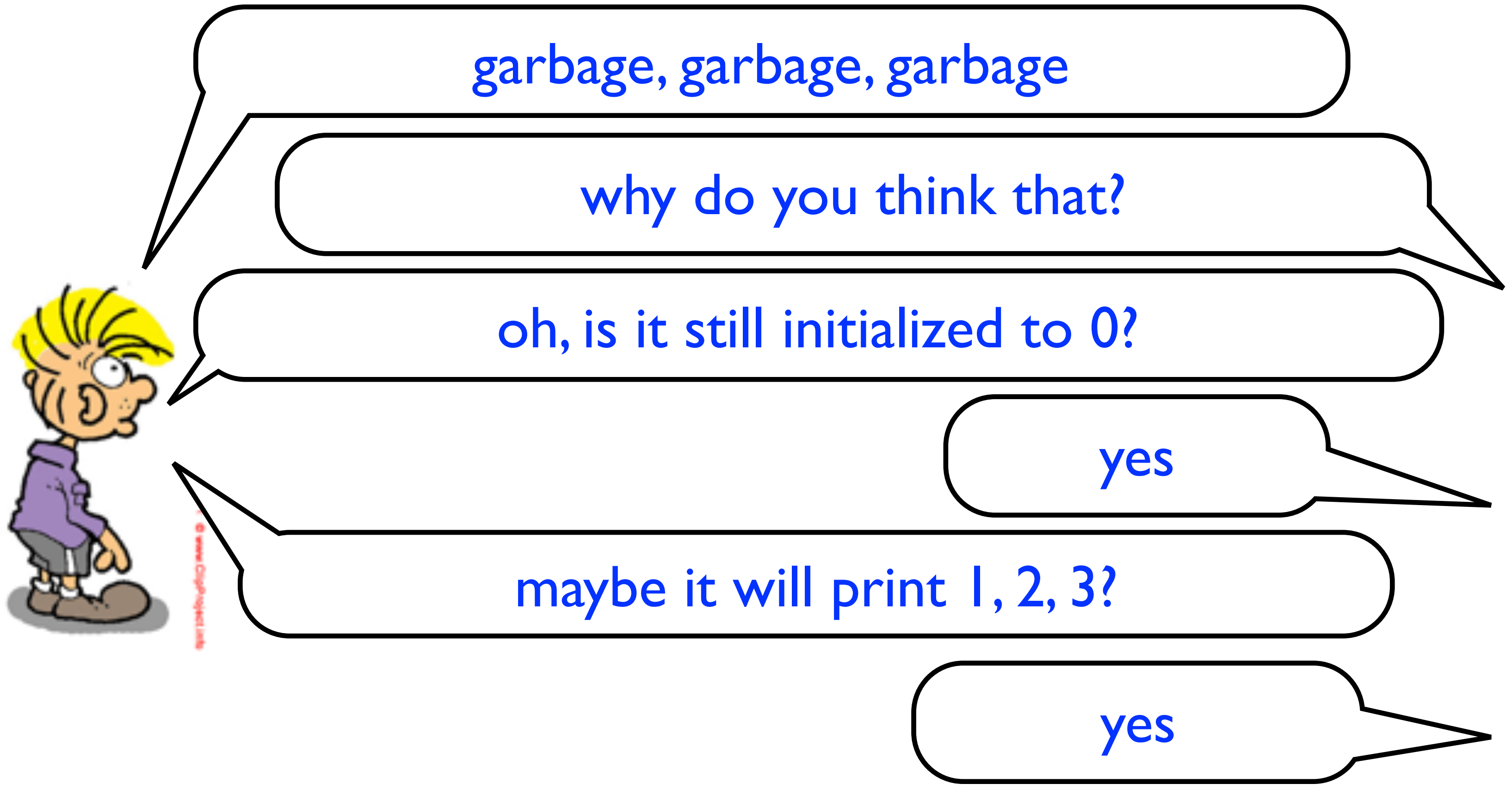
```
#include <stdio.h>

int a;

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

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



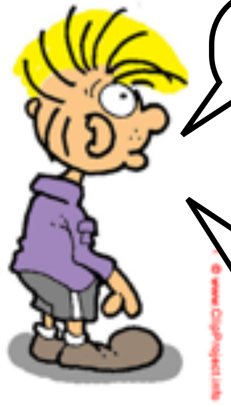
```
#include <stdio.h>

int a;

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

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


garbage, garbage, garbage
why do you think that?
oh, is it still initialized to 0?
yes
maybe it will print 1, 2, 3?

```
#include <stdio.h>

int a;

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

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


garbage, garbage, garbage
why do you think that?
oh, is it still initialized to 0?
yes
maybe it will print 1, 2, 3?
yes

```c
#include <stdio.h>

int a;

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

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



garbage, garbage, garbage

why do you think that?

oh, is it still initialized to 0?

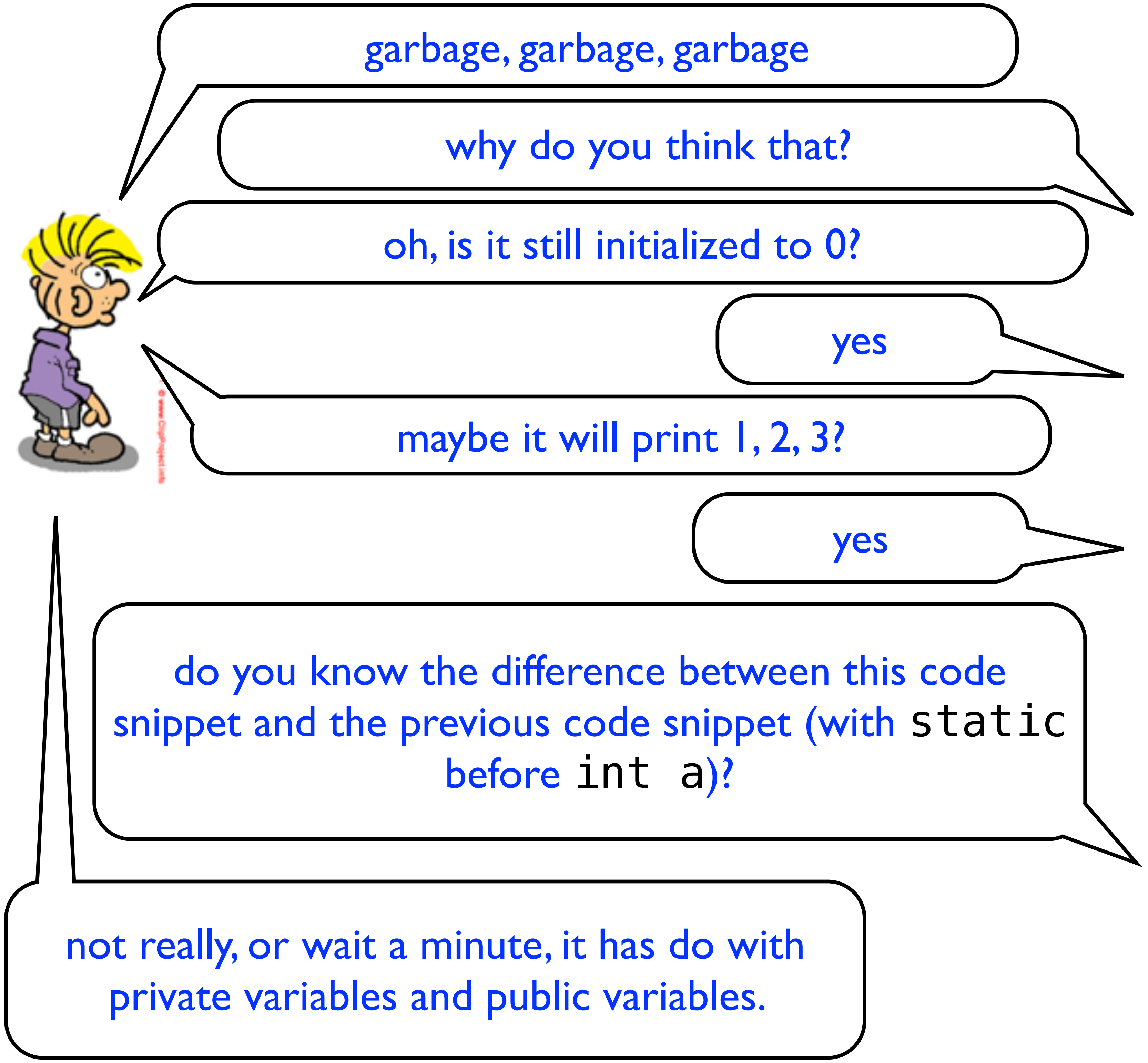
yes

maybe it will print 1, 2, 3?

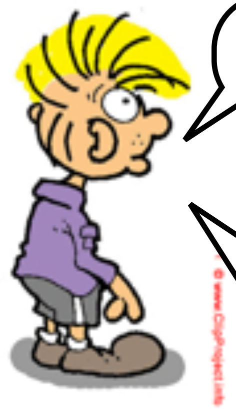
yes

do you know the difference between this code snippet and the previous code snippet (with `static` before `int a`)?

```
#include <stdio.h>

int a;

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

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


garbage, garbage, garbage
why do you think that?
oh, is it still initialized to 0?
yes
maybe it will print 1, 2, 3?
yes
do you know the difference between this code snippet and the previous code snippet (with static before int a)?
not really, or wait a minute, it has do with private variables and public variables.

```
#include <stdio.h>

int a;

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

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



garbage, garbage, garbage

why do you think that?

oh, is it still initialized to 0?

yes

maybe it will print 1, 2, 3?

yes

do you know the difference between this code snippet and the previous code snippet (with `static` before `int a`)?

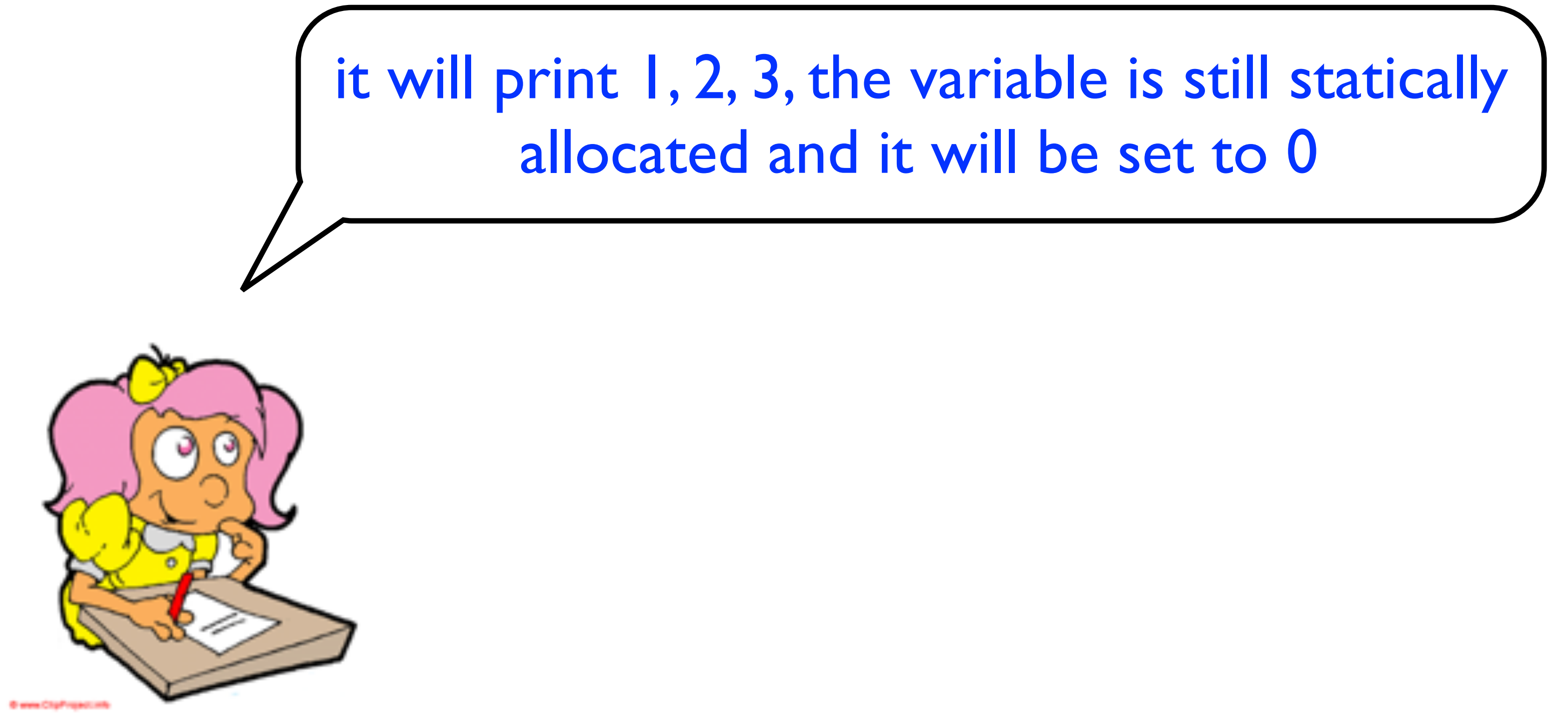
not really, or wait a minute, it has do with private variables and public variables.

yeah, something like that...

```c
#include <stdio.h>

int a;

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

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

```
#include <stdio.h>

int a;

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

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

```
#include <stdio.h>

int a;

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

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


it will print 1, 2, 3, the variable is still statically allocated and it will be set to 0

```
#include <stdio.h>

int a;

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

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

it will print 1, 2, 3, the variable is still statically allocated and it will be set to 0

do you know the difference between this code snippet and the previous code snippet (with `static` before `int a`)?

```
#include <stdio.h>

int a;

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

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

it will print 1, 2, 3, the variable is still statically allocated and it will be set to 0

do you know the difference between this code snippet and the previous code snippet (with `static` before `int a`)?

sure, it has to do with linker visibility. Here the variable is accessible from other compilation units, ie the linker can let another object file access this variable. If you add static in front, then the variable is local to this compilation unit and not visible through the linker.

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!

```
#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!

```
#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 behaviour?

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 behaviour?

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 behaviour?

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 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 <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 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...

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!

```
#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
```

Nice! I love it!

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
```

Nice! I love it!

You now want me to explain about execution stack or activation frames?

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
```

Nice! I love it!

You now want me to explain about execution stack or activation frames?

I guess you have already demonstrated that you understand it. But what do you think might happen if we optimize this code or use another compiler?

I am now going to show you something cool!

```
#include <stdio.h>

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

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

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

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

Nice! I love it!

You now want me to explain about execution stack or activation frames?

I guess you have already demonstrated that you understand it. But what do you think might happen if we optimize this code or use another compiler?

A lot of things might happen when the optimizer kicks in. In this case I would guess that the call to bar() can be skipped as it does not have any side effects. Also, I would not be surprised if the foo() is inlined in main(), ie no function call. (But since foo() has linker visibility the object code for the function must still be created just in case another object file wants to link with the function). Anyway, I suspect the value printed will be something else if you optimize the code.

```c
#include <stdio.h>

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

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

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

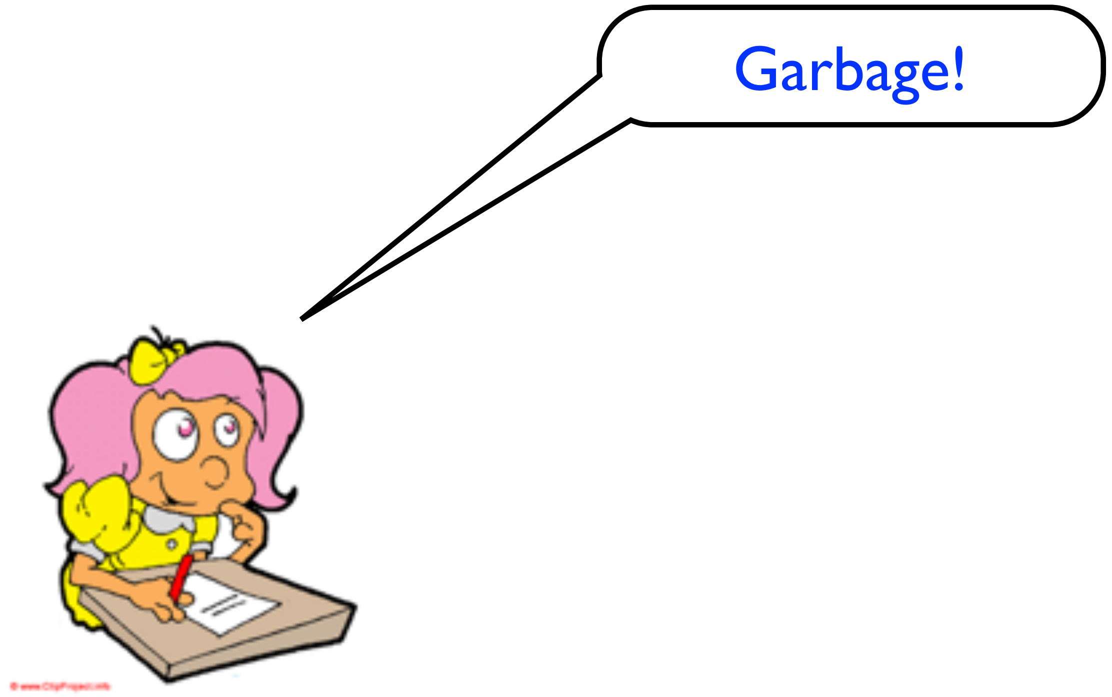
```
$ cc -O foo.c && ./a.out
1606415608
```

```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 -O foo.c && ./a.out
1606415608
```

```
#include <stdio.h>

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

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

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

```
$ cc -O foo.c && ./a.out
1606415608
```



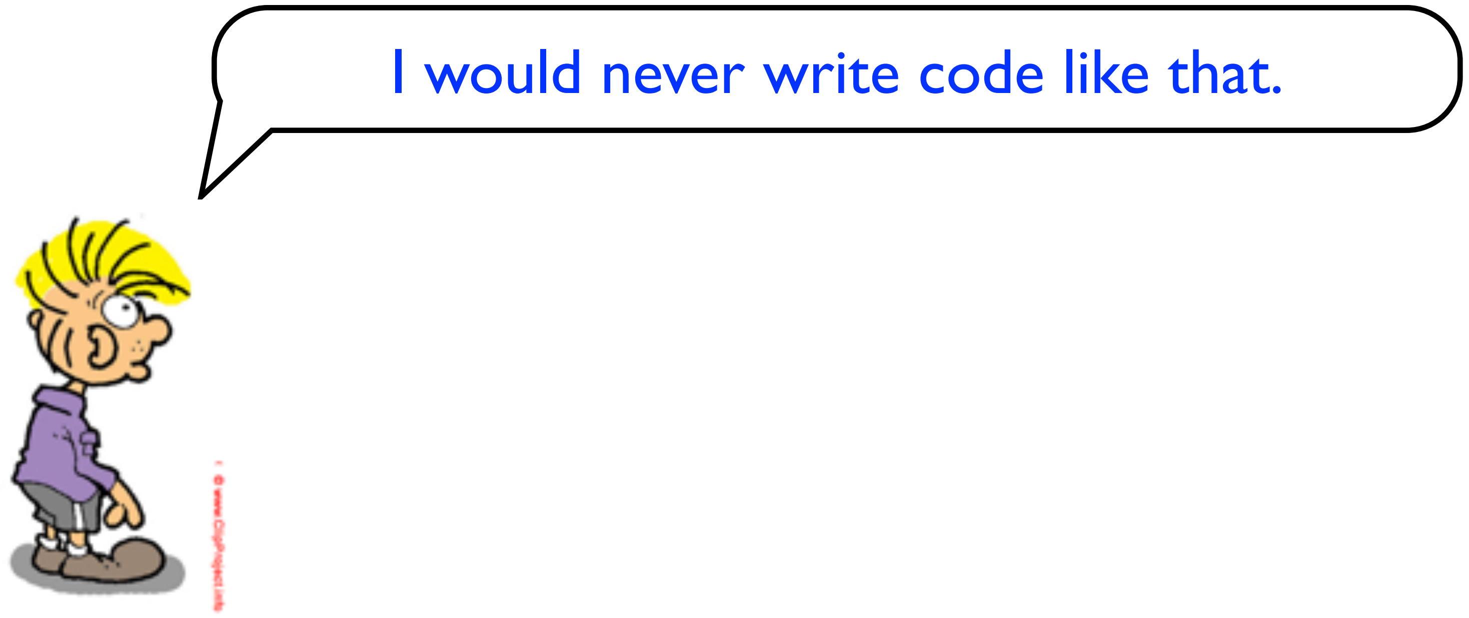
So what about this code snippet?

So what about this code snippet?

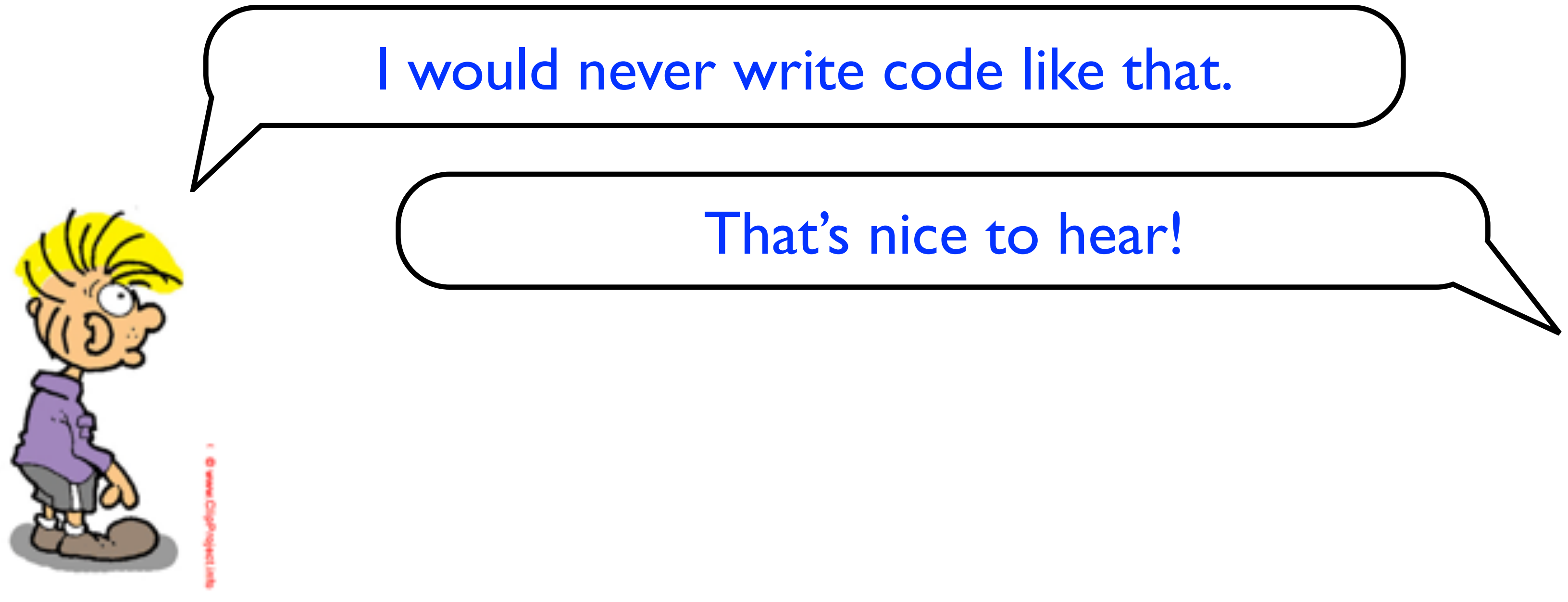
```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

```c
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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



```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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



```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

I would never write code like that.

That's nice to hear!

But I think the answer is 42

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

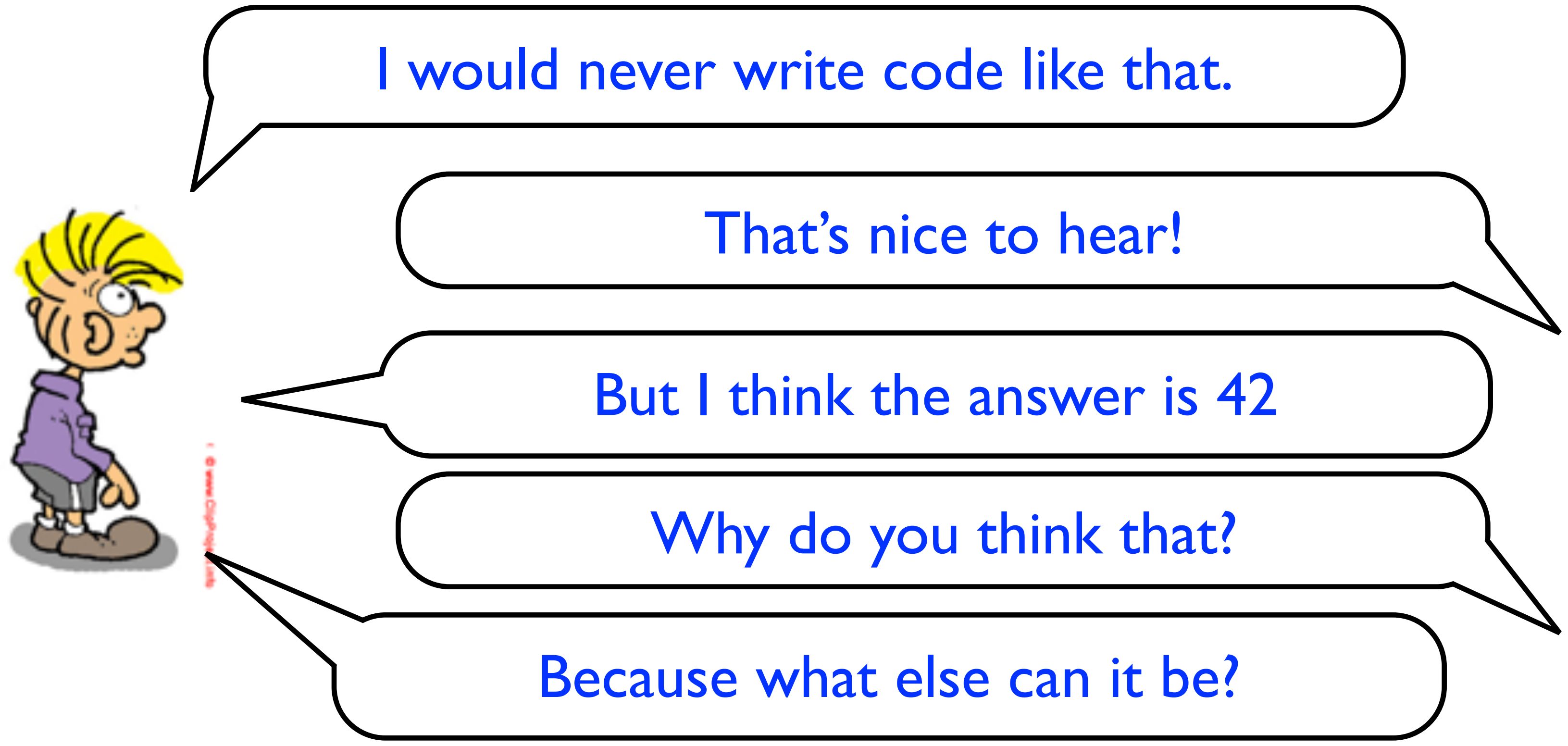
I would never write code like that.

That's nice to hear!

But I think the answer is 42

Why do you think that?

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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


I would never write code like that.
That's nice to hear!
But I think the answer is 42
Why do you think that?
Because what else can it be?

```c
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

I would never write code like that.

That's nice to hear!

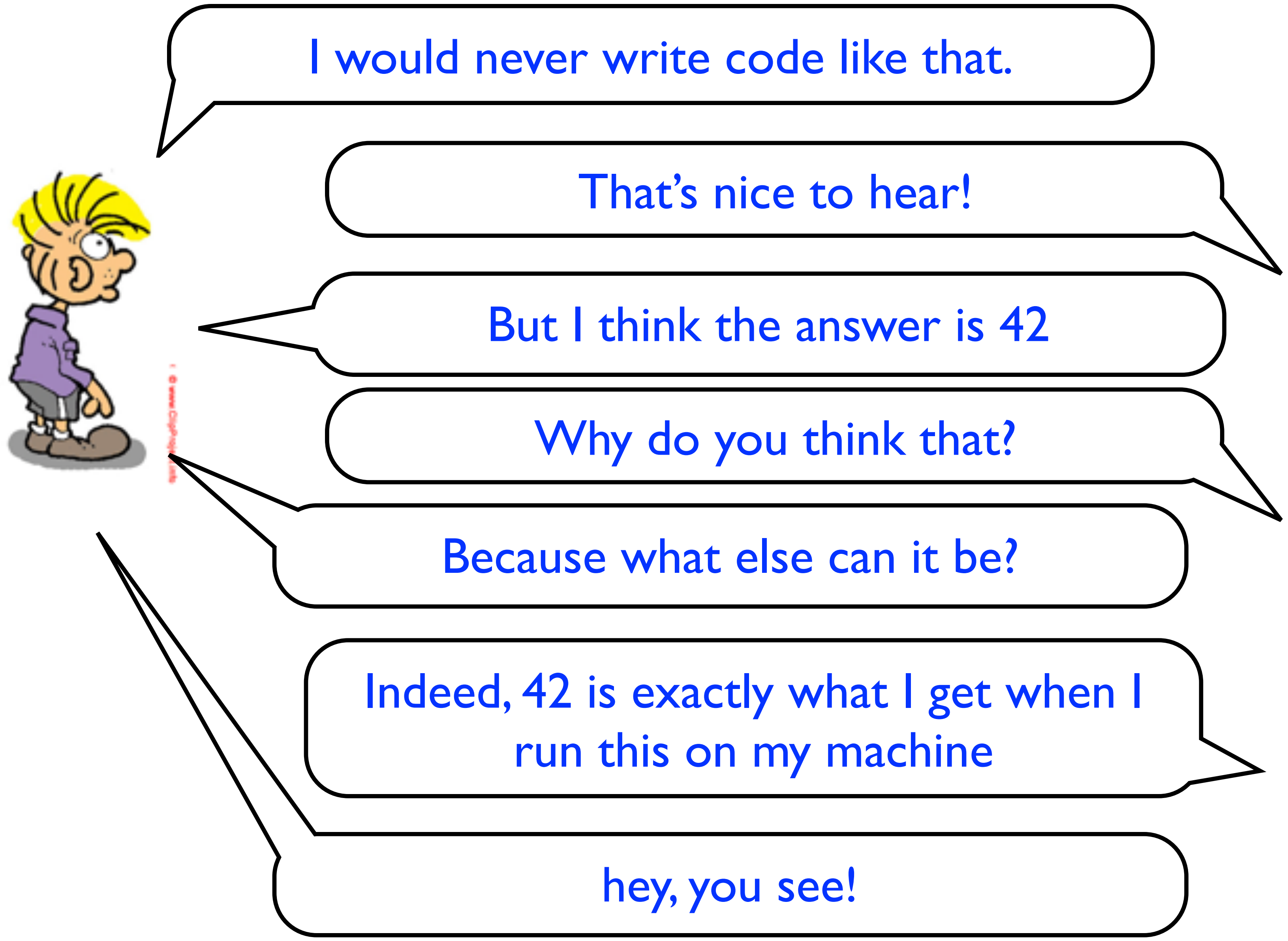
But I think the answer is 42

Why do you think that?

Because what else can it be?

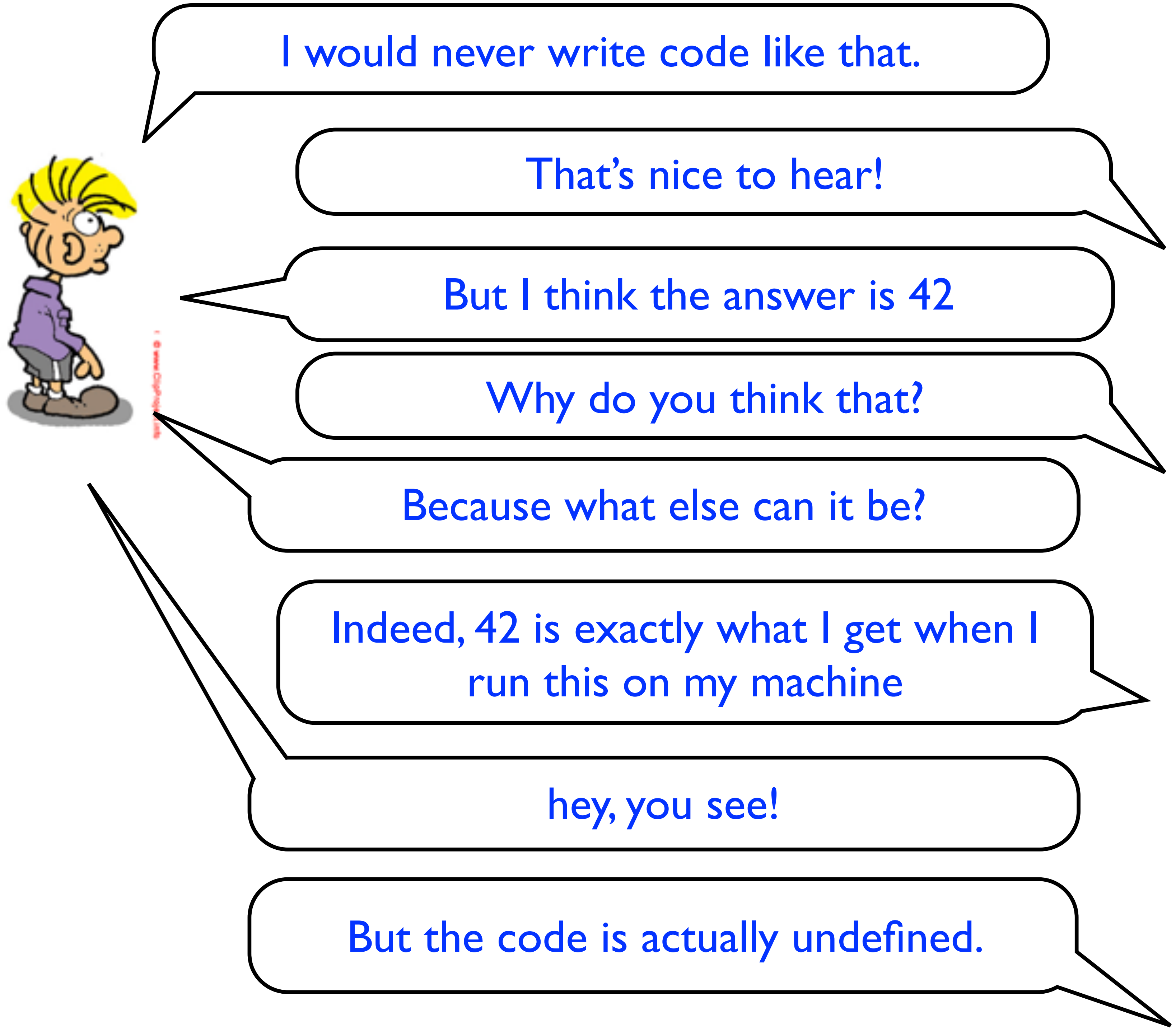
Indeed, 42 is exactly what I get when I run this on my machine

```c
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

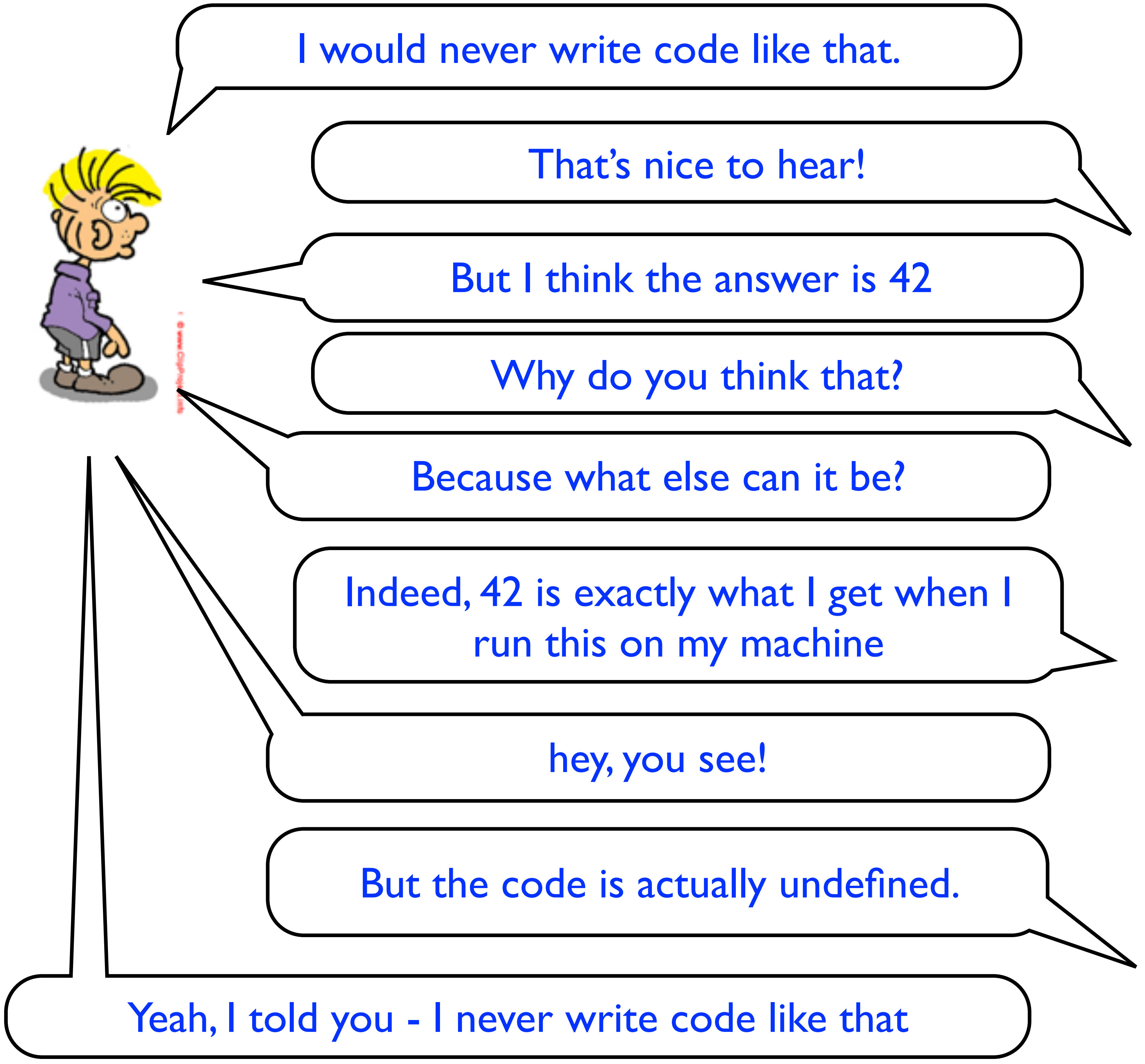


I would never write code like that.
That's nice to hear!
But I think the answer is 42
Why do you think that?
Because what else can it be?
Indeed, 42 is exactly what I get when I run this on my machine
hey, you see!

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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


I would never write code like that.
That's nice to hear!
But I think the answer is 42
Why do you think that?
Because what else can it be?
Indeed, 42 is exactly what I get when I run this on my machine
hey, you see!
But the code is actually undefined.

```c
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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


I would never write code like that.
That's nice to hear!
But I think the answer is 42
Why do you think that?
Because what else can it be?
Indeed, 42 is exactly what I get when I run this on my machine
hey, you see!
But the code is actually undefined.
Yeah, I told you - I never write code like that

```c
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

a gets an undefined value

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

a gets an undefined value

I don't get a warning when compiling it, and I do get 42

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

a gets an undefined value

I don't get a warning when compiling it, and I do get 42

Then you must increase the warning level, the value of a is certainly undefined after the assignment and increment because you violate one of the fundamental rules in C (and C++). The rules for sequencing says that you can only update a variable once between sequence points. Here you try to update it two times, and this causes a to become undefined.

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

a gets an undefined value

I don't get a warning when compiling it, and I do get 42

Then you must increase the warning level, the value of a is certainly undefined after the assignment and increment because you violate one of the fundamental rules in C (and C++). The rules for sequencing says that you can only update a variable once between sequence points. Here you try to update it two times, and this causes a to become undefined.

So you say a can be whatever? But I do get 42

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

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

a gets an undefined value

I don't get a warning when compiling it, and I do get 42

Then you must increase the warning level, the value of a is certainly undefined after the assignment and increment because you violate one of the fundamental rules in C (and C++). The rules for sequencing says that you can only update a variable once between sequence points. Here you try to update it two times, and this causes a to become undefined.

So you say a can be whatever? But I do get 42

Indeed! a can be 42, 41, 43, 0, 1099, or whatever... I am not surprised that your machine gives you 42... what else can it be here? Or perhaps the compiler choose 42 whenever a value is undefined ;-)

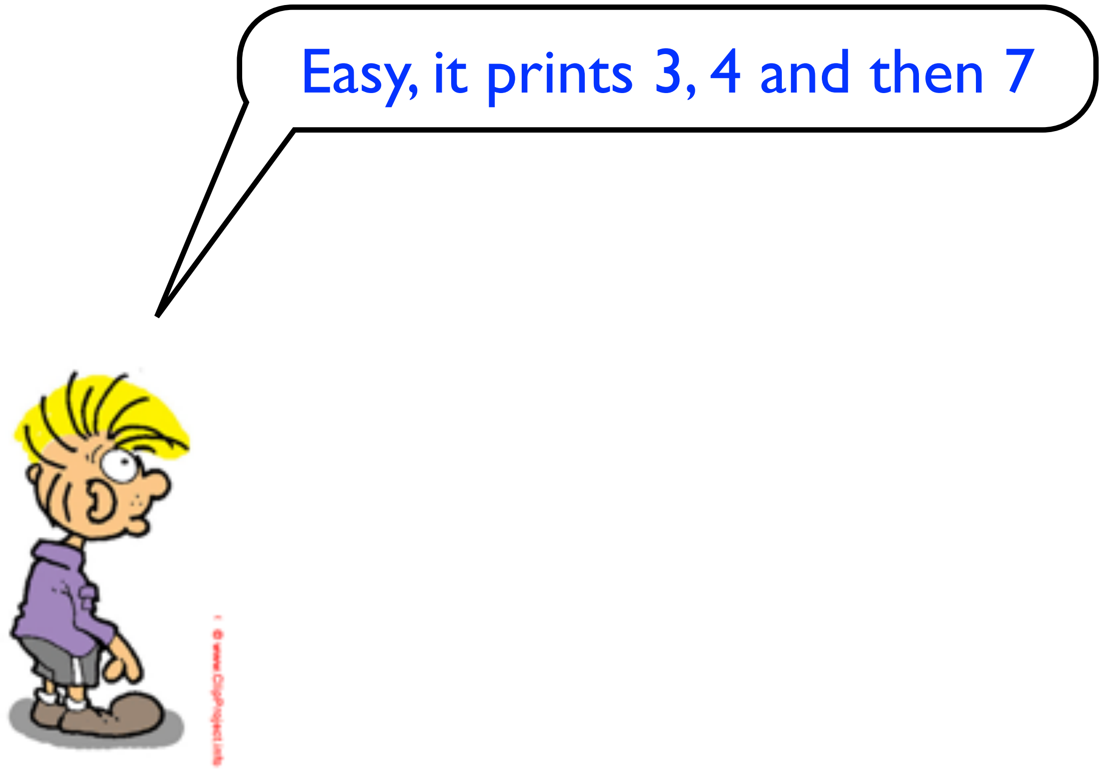
So what about this code snippet?

So what about this code snippet?

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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



```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

Easy, it prints 3, 4 and then 7

Actually, this could also be 4, 3 and then 7

```c
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

Easy, it prints 3, 4 and then 7

Actually, this could also be 4, 3 and then 7

Huh? Is the evaluation order undefined?

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

Easy, it prints 3, 4 and then 7

Actually, this could also be 4, 3 and then 7

Huh? Is the evaluation order undefined?

It is not really undefined, it is unspecified

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

Easy, it prints 3, 4 and then 7

Actually, this could also be 4, 3 and then 7

Huh? Is the evaluation order undefined?

It is not really undefined, it is unspecified

Well, whatever. Lousy compilers. I think it should give us a warning?

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

Easy, it prints 3, 4 and then 7

Actually, this could also be 4, 3 and then 7

Huh? Is the evaluation order undefined?

It is not really undefined, it is unspecified

Well, whatever. Lousy compilers. I think it should give us a warning?

A warning about what?

```c
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

The evaluation order of most expressions in C and C++ are unspecified, the compiler can choose to evaluate them in the order that is most optimal for the target platform. This has to do with sequencing again.

The code is conforming. This code will either print 3, 4, 7 or 4, 3, 7, depending on the compiler.

```
#include <stdio.h>

int b(void) { puts(“3”); return 3; }
int c(void) { puts(“4”); return 4; }

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

The evaluation order of most expressions in C and C++ are unspecified, the compiler can choose to evaluate them in the order that is most optimal for the target platform. This has to do with sequencing again.

The code is conforming. This code will either print 3, 4, 7 or 4, 3, 7, depending on the compiler.

Life would be so easy if more of my colleagues knew stuff like she does

© www.ClipProject.info

© www.ClipProject.info

At this point I think he has just revealed a shallow understanding of C programming, while she has excelled in her answers so far.

So what is it that she seems to understand better than most?

So what is it that she seems to understand better than most?

- Declaration and Definition

So what is it that she seems to understand better than most?

- Declaration and Definition
- Calling conventions and activation frames

So what is it that she seems to understand better than most?

- Declaration and Definition
- Calling conventions and activation frames
- Sequence points

So what is it that she seems to understand better than most?

- Declaration and Definition
- Calling conventions and activation frames
- Sequence points
- Memory model

So what is it that she seems to understand better than most?

- Declaration and Definition
- Calling conventions and activation frames
- Sequence points
- Memory model
- Optimization

So what is it that she seems to understand better than most?

- Declaration and Definition
- Calling conventions and activation frames
- Sequence points
- Memory model
- Optimization
- Knowledge of different C standards

We’d like to share some things about:

Sequence points
Different C standards

What do these code snippets print?

What do these code snippets print?

1

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

What do these code snippets print?

1

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

2

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

What do these code snippets print?

1

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

2

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

3

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

What do these code snippets print?

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

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

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

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

What do these code snippets print?

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

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

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

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

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

What do these code snippets print?

1

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

42

2

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

3

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

4

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

5

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

What do these code snippets print?

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

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

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

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

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

What do these code snippets print?

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

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

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

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

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

What do these code snippets print?

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

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

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

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

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

What do these code snippets print?

1

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

42

2

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

undefined

3

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

42

4

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

42

5

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

undefined

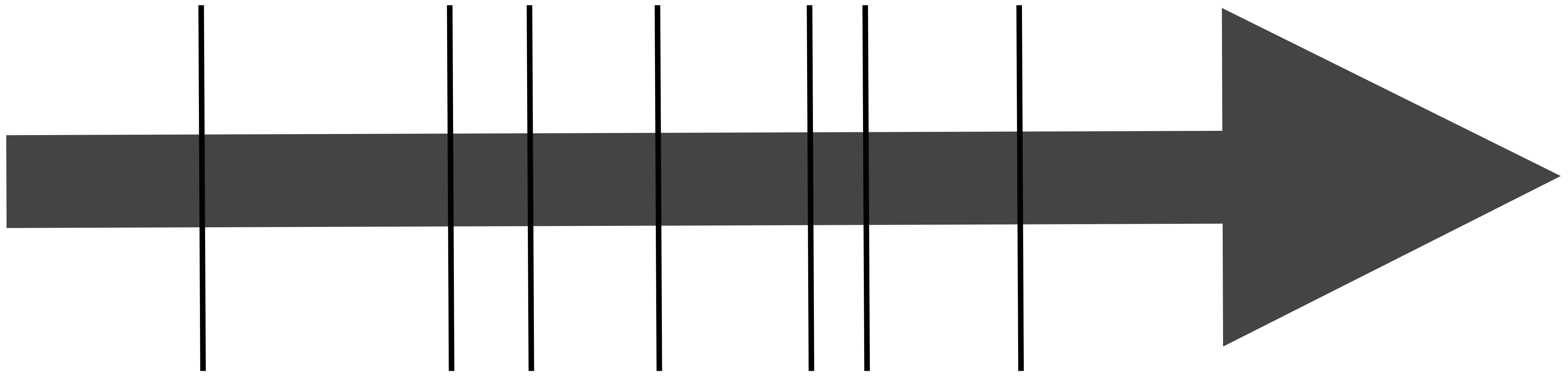
What do these code snippets print?

1	`int a=41; a++; printf("%d\n", a);`	42
2	`int a=41; a++ & printf("%d\n", a);`	undefined
3	`int a=41; a++ && printf("%d\n", a);`	42
4	`int a=41; if (a++ < 42) printf("%d\n", a);`	42
5	`int a=41; a = a++; printf("%d\n", a);`	undefined

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

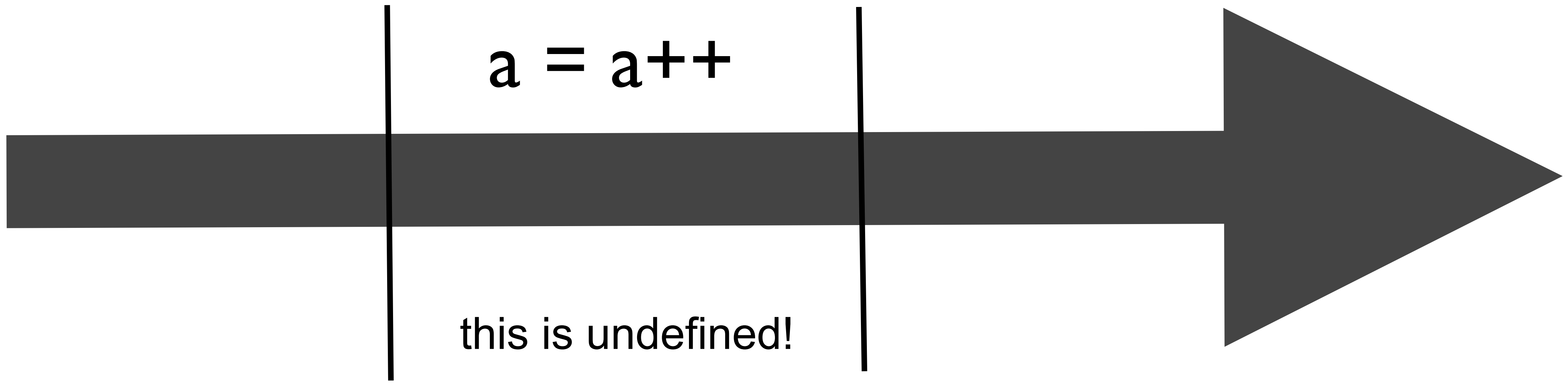
Sequence Points

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 (5.1.2.3)



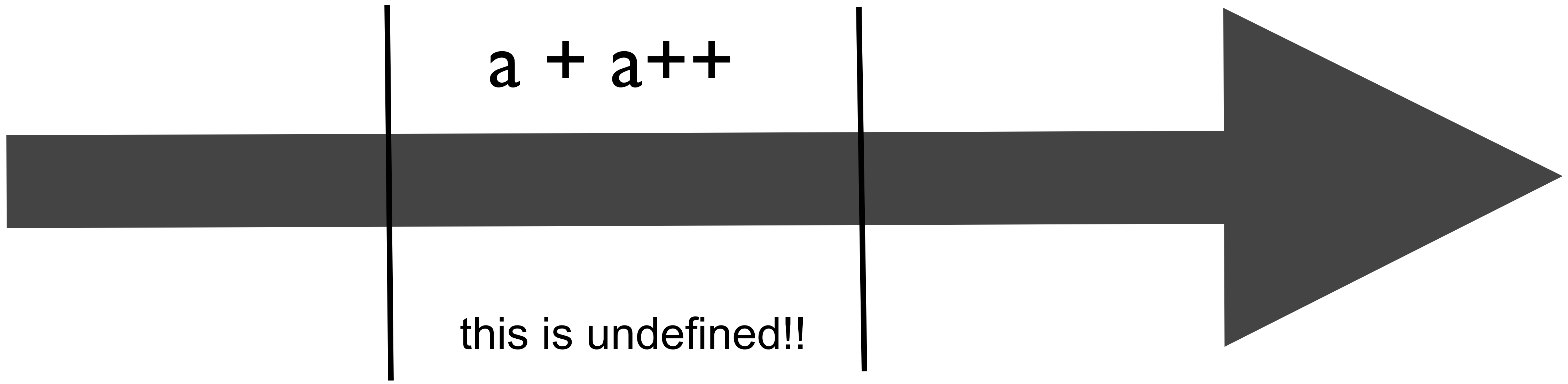
Sequence Points - Rule 1

Between the previous and next sequence point an object *shall* have its stored value modified at most once by the evaluation of an expression. (6.5)



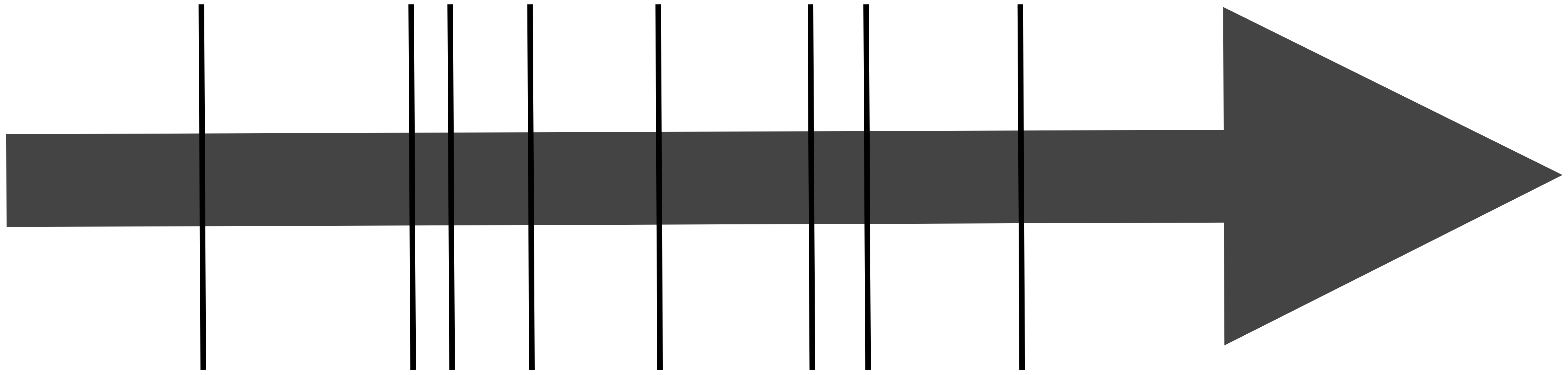
Sequence Points - Rule 2

Furthermore, the prior value *shall* be read only to determine the value to be stored. (6.5)



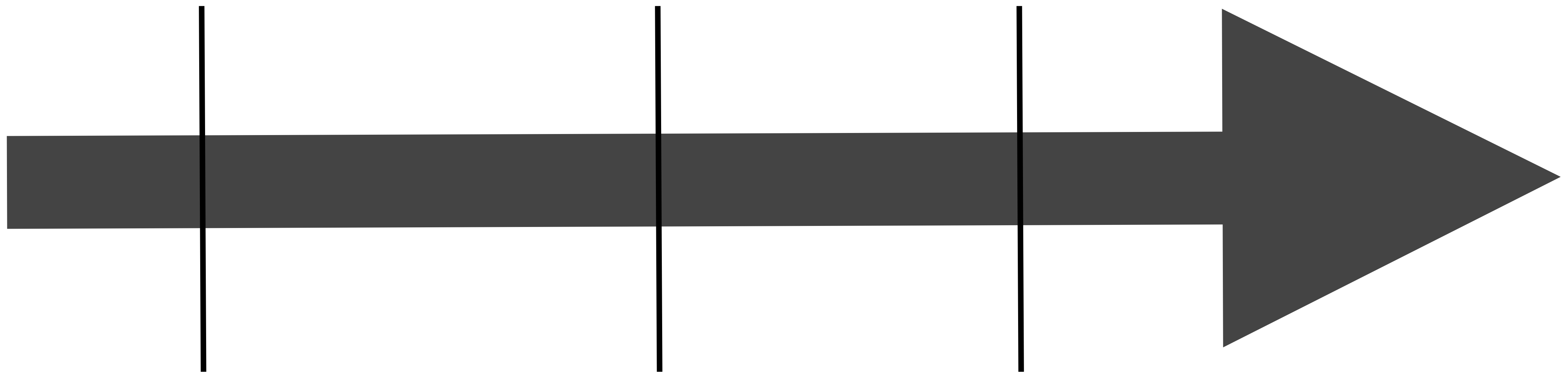
Sequence Points

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.

```
/* K&R C */

void say_it(a, s)
     int a;
     char s[];
{
     printf("%s %d\n", s, a);
}

main()
{
    int a = 42;
    puts("Welcome to classic C");
    say_it(a, "the answer is");
}
```

```
/* C89 */

void say_it(int a, char * s)
{
    printf("%s %d\n", s, a);
}

main()
{
    int a = 42;
    puts("Welcome to C89");
    say_it(a, "the answer is");
}
```

```
// C++ (C++98)

#include <cstdio>

struct X {
    int a;
    const char * s;
    explicit X(const char * s, int a = 42)
        : a(a), s(s) {}
    void say_it() const {
        std::printf("%s %d\n", s, a);
    }
};

int main()
{
    X("the answer is").say_it();
}
```

```
// C99

struct X
{
    int a;
    char * s;
};

int main(void)
{
    puts("Welcome to C99");
    struct X x = { .s = "the answer is", .a = 42 };
    printf("%s %d\n", x.s, x.a);
}
```

Let’s get back to our two developers...

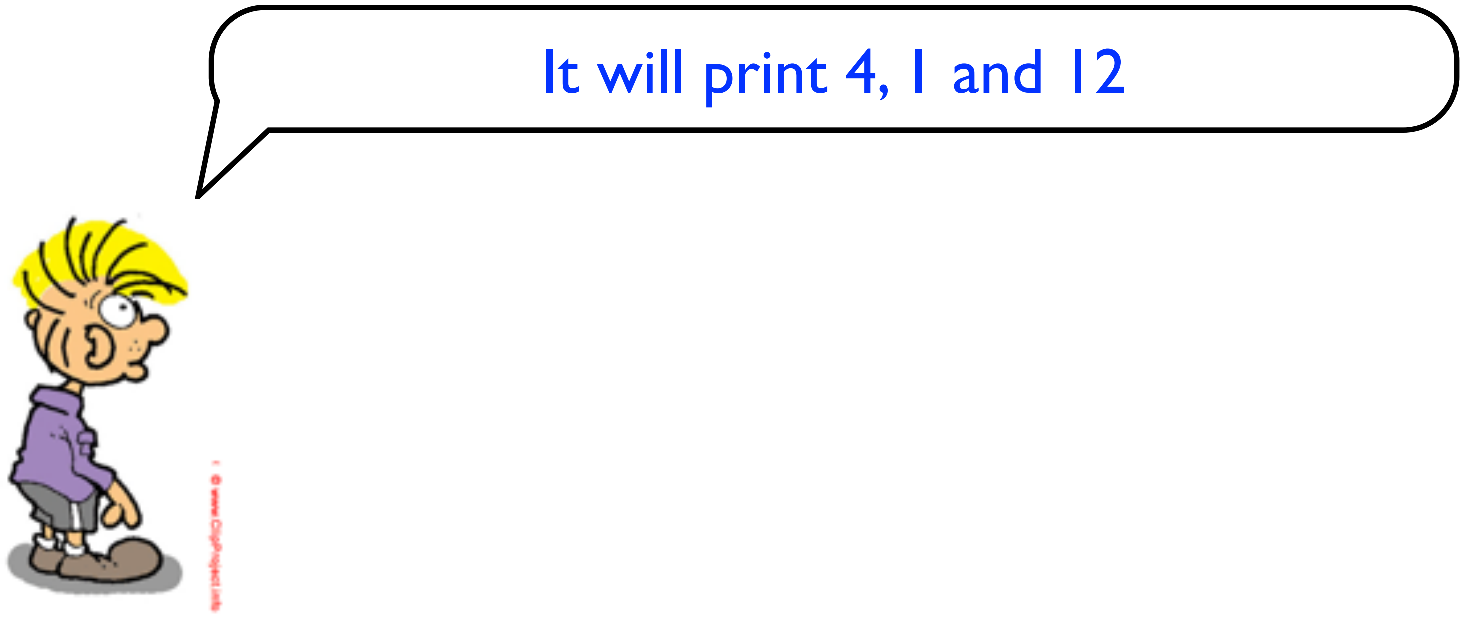
So what about this code snippet?

So what about this code snippet?

```
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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



```
#include <stdio.h>

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

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

It will print 4, 1 and 12

Indeed, it is exactly what I get on my machine

```
#include <stdio.h>

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

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

It will print 4, 1 and 12

Indeed, it is exactly what I get on my machine

Well of course, because `sizeof` returns the number of bytes. And in C int is 32 bits or 4 bytes, char is one byte and when the the size of structs are always rounded up to multiples of 4

```
#include <stdio.h>

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

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

It will print 4, 1 and 12

Indeed, it is exactly what I get on my machine

Well of course, because `sizeof` returns the number of bytes. And in C int is 32 bits or 4 bytes, char is one byte and when the the size of structs are always rounded up to multiples of 4

ok

```
#include <stdio.h>

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

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

It will print 4, 1 and 12

Indeed, it is exactly what I get on my machine

Well of course, because `sizeof` returns the number of bytes. And in C int is 32 bits or 4 bytes, char is one byte and when the the size of structs are always rounded up to multiples of 4

ok

do you want another ice cream?

```
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

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

```
#include <stdio.h>

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

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

Hmm... first of all, let's fix the code. The return type of `sizeof` is `size_t` which is not the same as int, so `%d` is a poor specifier to use in the format string for `printf` here

```
#include <stdio.h>

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

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

Hmm... first of all, let's fix the code. The return type of `sizeof` is `size_t` which is not the same as int, so %d is a poor specifier to use in the format string for `printf` here

ok, what should specifier should we use?

```
#include <stdio.h>

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

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

Hmm... first of all, let's fix the code. The return type of `sizeof` is `size_t` which is not the same as int, so %d is a poor specifier to use in the format string for `printf` here

ok, what should specifier should we use?

Thats a bit tricky. `size_t` is an unsigned integer type, but on say 32-bit machines it is usually an unsigned int and on 64-bit machines it is usually an unsigned long. In C99 however, they introduced a new specifier for printing size_t values, so `%zu` might be an option.

```
#include <stdio.h>

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

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

Hmm... first of all, let's fix the code. The return type of `sizeof` is `size_t` which is not the same as int, so %d is a poor specifier to use in the format string for `printf` here

ok, what should specifier should we use?

Thats a bit tricky. `size_t` is an unsigned integer type, but on say 32-bit machines it is usually an unsigned int and on 64-bit machines it is usually an unsigned long. In C99 however, they introduced a new specifier for printing size_t values, so `%zu` might be an option.

ok, let's fix the `printf` issue, and then you can try to answer the question

```
#include <stdio.h>

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

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

```
#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));
}
```

```
#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));
}
```

Now it all depends on the platform and the compile time options provided. The only thing we know for sure is that `sizeof` char is 1. Do you assume a 64-bit machine?

```
#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));
}
```

Now it all depends on the platform and the compile time options provided. The only thing we know for sure is that `sizeof` char is 1. Do you assume a 64-bit machine?

Yes, I have a 64-bit machine running in 32-bit compatibility mode.

```
#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));
}
```

Now it all depends on the platform and the compile time options provided. The only thing we know for sure is that `sizeof` char is 1. Do you assume a 64-bit machine?

Yes, I have a 64-bit machine running in 32-bit compatibility mode.

Then I would like to guess that this prints 4, 1, 12 due to word alignment

```
#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));
}
```

Now it all depends on the platform and the compile time options provided. The only thing we know for sure is that `sizeof` char is 1. Do you assume a 64-bit machine?

Yes, I have a 64-bit machine running in 32-bit compatibility mode.

Then I would like to guess that this prints 4, 1, 12 due to word alignment

But that of course also depends also on compilation flags. It could be 4, 1, 9 if you ask the compiler to pack the structs, eg `-fpack-struct` in gcc.

```
#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));
}
```

```
#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));
}
```

4, 1, 12 is indeed what I get on my machine. Why 12?

```
#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));
}
```

4, 1, 12 is indeed what I get on my machine. Why 12?

It is very expensive to work on subword data types, so the compiler will optimize the code by making sure that `c` is on a word boundary by adding some padding. Also elements in an array of `struct X` will now align on word-boundaries.

```
#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));
}
```

4, 1, 12 is indeed what I get on my machine. Why 12?

It is very expensive to work on subword data types, so the compiler will optimize the code by making sure that `c` is on a word boundary by adding some padding. Also elements in an array of `struct X` will now align on word-boundaries.

Why is it expensive to work on values that are not aligned?

```
#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));
}
```

4, 1, 12 is indeed what I get on my machine. Why 12?

It is very expensive to work on subword data types, so the compiler will optimize the code by making sure that `c` is on a word boundary by adding some padding. Also elements in an array of `struct X` will now align on word-boundaries.

Why is it expensive to work on values that are not aligned?

The instruction set of most processors are optimized for moving a word of data between memory and CPU. Suppose you want to change a value crossing a word boundary, you would need to read two words, mask out the value, change the value, mask and write back two words. Perhaps 10 times slower. Remember, C is focused on execution speed.

```
#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));
}
```

```
#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));
}
```

so what if I add a `char d` to the struct?

```
#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));
}
```

so what if I add a `char d` to the struct?

If you add it to the end of the struct, my guess is that the size of the struct becomes 16 on your machine. This is first of all because 13 would be a not so efficient size, what if you have an array of `struct X` objects? But if you add it just after `char b`, then 12 is a more plausible answer.

```
#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));
}
```

so what if I add a `char d` to the struct?

If you add it to the end of the struct, my guess is that the size of the struct becomes 16 on your machine. This is first of all because 13 would be a not so efficient size, what if you have an array of `struct X` objects? But if you add it just after `char b`, then 12 is a more plausible answer.

So why doesn't the compiler reorder the members in the structure to optimize memory usage, and execution speed?

```
#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));
}
```

so what if I add a `char d` to the struct?

If you add it to the end of the struct, my guess is that the size of the struct becomes 16 on your machine. This is first of all because 13 would be a not so efficient size, what if you have an array of `struct X` objects? But if you add it just after `char b`, then 12 is a more plausible answer.

So why doesn't the compiler reorder the members in the structure to optimize memory usage, and execution speed?

Some languages actually do that, but C and C++ don't.

```
#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));
}
```

```
#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));
}
```

so what if I add a `char * d` to the end of the struct?

```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));
}
```

so what if I add a `char * d` to the end of the struct?

You said your runtime was 64-bit, so a pointer is probably 8 bytes... Maybe the struct becomes 20? But perhaps the 64-bit pointer also needs alignment for efficiency? Maybe this code will print 4,1,24?

```
#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));
}
```

so what if I add a `char * d` to the end of the struct?

You said your runtime was 64-bit, so a pointer is probably 8 bytes... Maybe the struct becomes 20? But perhaps the 64-bit pointer also needs alignment for efficiency? Maybe this code will print 4,1,24?

Nice answer! It does not matter what I actually get on my machine. I like your argument and your insight.

So what is it that she seems to understand better than most?

So what is it that she seems to understand better than most?

- Some experience with 32-bit vs 64-bit issues

So what is it that she seems to understand better than most?

- Some experience with 32-bit vs 64-bit issues
- Memory alignment

So what is it that she seems to understand better than most?

- Some experience with 32-bit vs 64-bit issues
- Memory alignment
- CPU and memory optimization

So what is it that she seems to understand better than most?

- Some experience with 32-bit vs 64-bit issues
- Memory alignment
- CPU and memory optimization
- Spirit of C

We’d like to share some things about:

Memory model
Optimization
The spirit of C

Memory Model

static storage

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **static** has *static storage duration*. It’s lifetime is the entire execution of the program... (6.2.4)

```
int * immortal(void)
{
    static int storage = 42;
    return &storage;
}
```

Memory Model

automatic storage

An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*. ... It's lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (6.2.4)

```
int * zombie(void)
{
    auto int storage = 42;
    return &storage;
}
```

Memory Model

allocated storage

...storage allocated by calls to calloc, malloc, and realloc... The lifetime of an allocated object extends from the allocation to the dealloction. (7.20.3)

```
int * finite(void)
{
    int * ptr = malloc(sizeof *ptr);
    *ptr = 42;
    return ptr;
}
```

Optimization

By default you should compile with optimization ***on***. Forcing the compiler to work harder helps it find more potential problems.

opt.c

```
#include <stdio.h>

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

```
>cc -Wall opt.c
```

no warning!

opt.c

```
#include <stdio.h>

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

```
>cc -Wall -O opt.c
warning: 'a' is uninitialized
```

The Spirit of C

There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based

(C Rationale Introduction)

- Trust the programmer
- Keep the language small and simple
- Provide only one way to do an operation
- Make it fast, even if it is not guaranteed to be portable
- Maintain conceptual simplicity
- Don’t prevent the programmer from doing what needs to be done

Let’s ask our developers about C++

On a scale from 1 to 10, how do you rate your understanding of C++?

On a scale from 1 to 10, how do you rate your understanding of C++?

On a scale from 1 to 10, how do you rate your understanding of C++?

I rate myself as 8 or 9

On a scale from 1 to 10, how do you rate your understanding of C++?

I rate myself as 8 or 9

On a scale from 1 to 10, how do you rate your understanding of C++?

I rate myself as 8 or 9

4, maybe 5, I have just so much more to learn about C++

On a scale from 1 to 10, how do you rate your understanding of C++?

I rate myself as 8 or 9

4, maybe 5, I have just so much more to learn about C++

On a scale from 1 to 10, how do you rate your understanding of C++?

I rate myself as 8 or 9

4, maybe 5, I have just so much more to learn about C++

7

So what about this code snippet?

```
#include <iostream>

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

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

```
#include <iostream>

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

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

```
#include <iostream>

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

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

This struct is a POD (Plain Old Data) struct and it is guaranteed by the C++ standard to behave just like a struct in C.

```
#include <iostream>

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

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

This struct is a POD (Plain Old Data) struct and it is guaranteed by the C++ standard to behave just like a struct in C.

So on your machine? I guess this will still print 12.

```
#include <iostream>

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

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

This struct is a POD (Plain Old Data) struct and it is guaranteed by the C++ standard to behave just like a struct in C.

So on your machine? I guess this will still print 12.

And by the way, it looks weird to specify func(void) instead of func() as void is the default in C++. This is also true when defining the main function. Of course, no kittens are hurt by this, it just looks like the code is written by a die-hard C programmer struggling to learn C++

```
#include <iostream>

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

int main(void)
{
    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;
}
```

```
#include <iostream>

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

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

This program will print 12

```
#include <iostream>

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

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

This program will print 12

ok

```
#include <iostream>

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

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

This program will print 12

ok

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;


};

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;
}
```

```
#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;
}
```

Eh, can you do that in C++? I think you must use a class.

What is the difference between a class and a struct in C++?

```
#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;
}
```

Eh, can you do that in C++? I think you must use a class.

What is the difference between a class and a struct in C++?

Eh, in a class you can have member functions, while I don't think you can have member functions on structs. Or maybe you can? Is it the default visibility that is different?

```
#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;
}
```

Anyway, now this code will print 16. Because there will be a pointer to the function.

```
#include <iostream>

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

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

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

Anyway, 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;
}
```

Anyway, now this code will print 16. Because there will be a pointer to the function.

ok?

so what if I add two more functions?

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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


};

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#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;
}
```

I guess 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;
}
```

I guess it will print 24? Two more pointers?

On my machine it prints much less than 24

```
#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;
}
```

I guess it will print 24? Two more pointers?

On my machine it prints much less than 24

Ah, of course, it has a table of function pointers and only needs one pointer to the table! I do really have a deep understanding of this, I just forgot.

```
#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;
}
```

I guess it will print 24? Two more pointers?

On my machine it prints much less than 24

Ah, of course, it has a table of function pointers and only needs one pointer to the table! I do really have a deep understanding of this, I just forgot.

Actually, on my machine this code still prints 12.

```
#include <iostream>

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

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

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

I guess it will print 24? Two more pointers?

On my machine it prints much less than 24

Ah, of course, it has a table of function pointers and only needs one pointer to the table! I do really have a deep understanding of this, I just forgot.

Actually, on my machine this code still prints 12.

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

So what do you think this code prints?

```
#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;
}
```

So what do you think this code prints?

On you machine? I guess 12 again?

```
#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;
}
```

So what do you think this code prints?

On you machine? I guess 12 again?

Ok, why?

```
#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;
}
```

So what do you think this code prints?

On you machine? I guess 12 again?

Ok, why?

Because adding member functions like this does not change the size of the struct. 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;
}
```

So what do you think this code prints?

On you machine? I guess 12 again?

Ok, why?

Because adding member functions like this does not change the size of the struct. 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(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++; }
```

Like this?

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++; }
```

Like this?

Yeah, just like that, and now it is obvious that functions like this do not change the size of the type and 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;
}
```

```
#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;
}
```

So what happens now?

```
#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;
}
```

So what happens now?

The size of the type will probably grow. The C++ standard does not say much about how virtual classes and overriding should be implemented, but a common approach is to create a virtual table and then you need a pointer to it. So in this case add 8 bytes? Does it print 20?

```
#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;
}
```

So what happens now?

The size of the type will probably grow. The C++ standard does not say much about how virtual classes and overriding should be implemented, but a common approach is to create a virtual table and then you need a pointer to it. So in this case add 8 bytes? Does it print 20?

I get 24 when I run this code snippet

```
#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;
}
```

So what happens now?

The size of the type will probably grow. The C++ standard does not say much about how virtual classes and overriding should be implemented, but a common approach is to create a virtual table and then you need a pointer to it. So in this case add 8 bytes? Does it print 20?

I get 24 when I run this code snippet

Ah, don't worry. It is probably just some extra padding to align the pointer

```
#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;
}
```

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

```
#include <iostream>

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

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

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

So what happens now?

```
#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;
}
```

So what happens now?

My guess is that it still prints 24, as you only need one vtable per class.

```
#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;
}
```

So what happens now?

My guess is that it still prints 24, as you only need one vtable per class.

ok, what is a vtable?

```
#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;
}
```

So what happens now?

My guess is that it still prints 24, as you only need one vtable per class.

ok, what is a vtable?

It is a common implementation technique to support one type of polymorphism in C++. It is basically a jump table for function calls, and with it you can override functions when doing class inheritance let’s consider another code snippet...

```
#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.

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?

Are you thinking about using ‘delete[]’ instead of ‘delete’ when deleting an array of objects? Well, I am experienced enough to know that it is not really needed, modern compilers will handle that.

```
#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?

Are you thinking about using ‘delete[]’ instead of ‘delete’ when deleting an array of objects? Well, I am experienced enough to know that it is not really needed, modern compilers will handle that.

Ok? What about the “rule of three”? Do you need to support or disable copying of this object?

```
#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?

Are you thinking about using 'delete[]' instead of 'delete' when deleting an array of objects? Well, I am experienced enough to know that it is not really needed, modern compilers will handle that.

Ok? What about the "rule of three"? Do you need to support or disable copying of this object?

Yeah, whatever... never heard of the tree-rule but of course if people copy this object they might get problems. But I guess that is the spirit of C++... give programmers a really hard time.

```
#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_;
};
```

And by the way, 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_;
};
```

And by the way, 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...

```
#include "B.hpp"

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

And by the way, 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...

another ice cream perhaps?

```
#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.

Oh, where should I start... let's focus on the most important stuff first

```
#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.

Oh, where should I start... let's focus on the most important stuff first

In the destructor. If you use operator new[] you should destroy with operator delete[]. With operator delete[] the allocated memory will be deallocated after the destructor for **every** object in the array will be called. Eg, as it stands now, the B constructor will be called sz times, but the B destructor will only be called once. In this case, bad things will happen if B allocates resources that need to be released in its destructor.

```
#include "B.hpp"

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

And the next thing is the often referred to as the "rule of three". If you need a destructor, you probably also need to either implement or disable the copy constructor and the assignment operator, the default ones created by the compiler are probably not correct.

```
#include "B.hpp"

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

And the next thing is the often referred to as the "rule of three". If you need a destructor, you probably also need to either implement or disable the copy constructor and the assignment operator, the default ones created by the compiler are probably not correct.

A perhaps smaller issue, but also important, is to use the member initializer list to initialize an object. In the example above it does not really matter much, but when member objects are more complex it makes sense to explicitly initialize the members (using the initalizer list), rather than letting the object implicitly initialize all its member objects to default values, and **then** assign them some particular value.

```
#include "B.hpp"

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

And the next thing is the often referred to as the "rule of three". If you need a destructor, you probably also need to either implement or disable the copy constructor and the assignment operator, the default ones created by the compiler are probably not correct.

A perhaps smaller issue, but also important, is to use the member initializer list to initialize an object. In the example above it does not really matter much, but when member objects are more complex it makes sense to explicitly initialize the members (using the initalizer list), rather than letting the object implicitly initialize all its member objects to default values, and **then** assign them some particular value.

Please fix the code and I will tell you more...

```cpp
#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_;
};
```

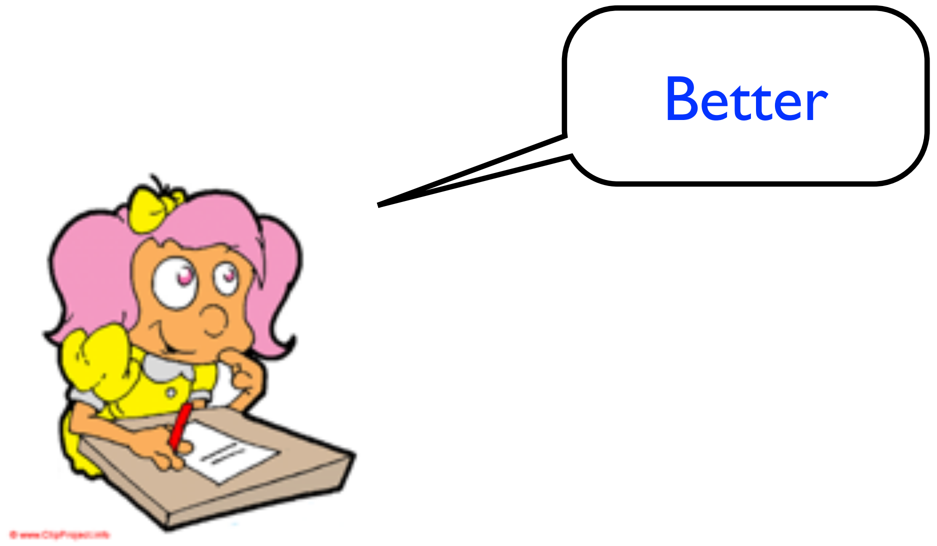
```cpp
#include "B.hpp"

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_;
};
```

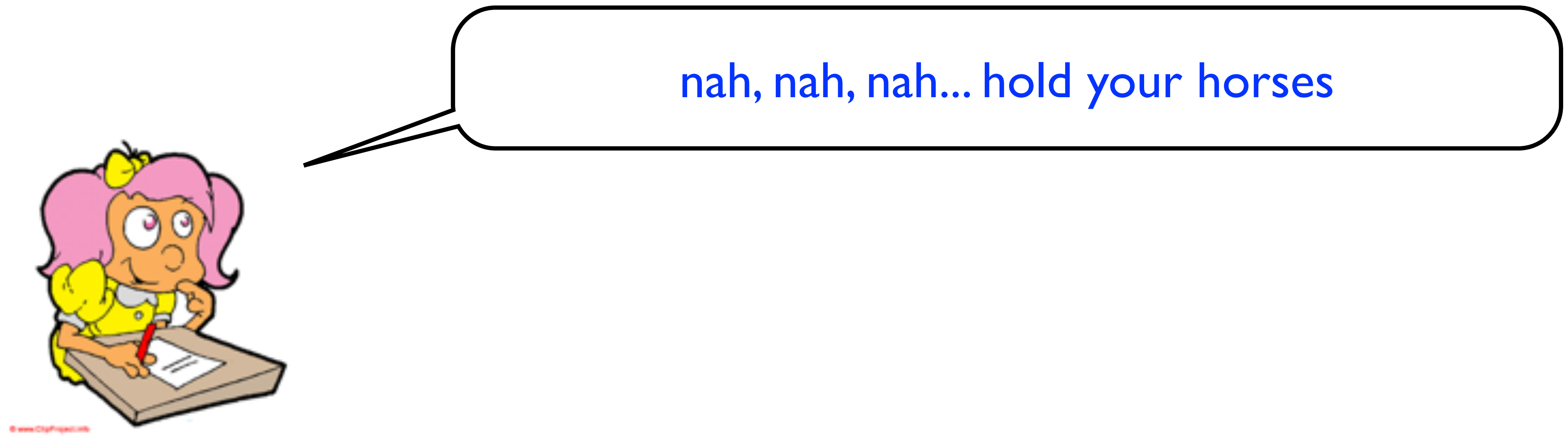
```
#include "B.hpp"

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_;
};
```

```
#include "B.hpp"

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

nah, nah, nah... hold your horses

What is the point of having a virtual destructor on a class like this? There are no virtual functions so it does not make sense to inherit from it. I know that there are programmers who do inherit from non-virtual classes, but I suspect they have misunderstood a key concept of object orientation. I suggest you remove the virtual specifier from the destructor, it indicates that the class is designed to be used as a base class - while it obviously is not.

```
#include "B.hpp"

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

nah, nah, nah... hold your horses

What is the point of having a virtual destructor on a class like this? There are no virtual functions so it does not make sense to inherit from it. I know that there are programmers who do inherit from non-virtual classes, but I suspect they have misunderstood a key concept of object orientation. I suggest you remove the virtual specifier from the destructor, it indicates that the class is designed to be used as a base class - while it obviously is not.

why don't you fix the initializer list issue instead

```
#include "B.hpp"

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

```cpp
#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_;
};
```

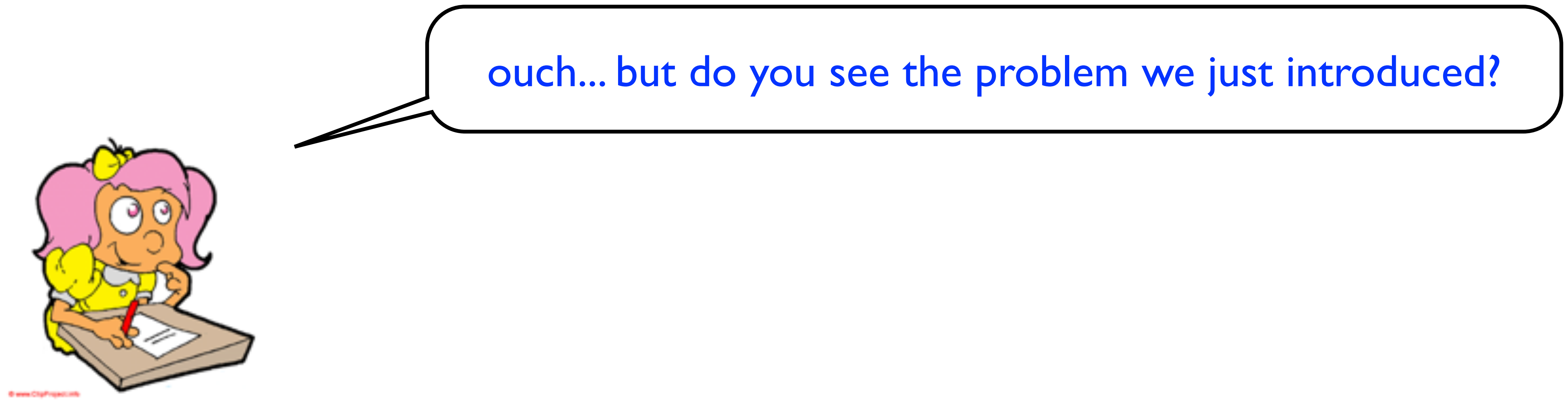
```
#include "B.hpp"

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

```cpp
#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_;
};
```

now we have an initializer list...

```
#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_;
};
```

ouch... but do you see the problem we just introduced?

Are you compiling with -Wall? You should consider -Wextra -pedantic and -Weffc++ as well

```
#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_;
};
```

ouch... but do you see the problem we just introduced?

Are you compiling with -Wall? You should consider -Wextra -pedantic and -Weffc++ as well

Without warning flags you might not notice the mistake here. But if you increase the warning levelsl it will scream the problem in your face...

```
#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_;
};
```

ouch... but do you see the problem we just introduced?

Are you compiling with -Wall? You should consider -Wextra -pedantic and -Weffc++ as well

Without warning flags you might not notice the mistake here. But if you increase the warning levelsl it will scream the problem in your face...

A nice rule of thumb is to always write the member initializers in the order they are defined. In this case, when v(new B[sz_]) is evaluated sz_ is undefined, and **then** sz_ is initialized with sz. Actually, these things are just too common in C++ code.

```
#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_;
};
```

```cpp
#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_;
};
```

Now this looks better! Is there anything else that needs to be improved? Perhaps some small stuff that I would like to mention...

```
#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_;
};
```

Now this looks better! Is there anything else that needs to be improved? Perhaps some small stuff that I would like to mention...

When I see bald pointers in C++ it is usually a bad sign. A lot of good C++ programmers tend to avoid using them like this. In this case of course, it looks like v is a candidate for being an STL vector or something like that.

```
#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_;
};
```

Now this looks better! Is there anything else that needs to be improved? Perhaps some small stuff that I would like to mention...

When I see bald pointers in C++ it is usually a bad sign. A lot of good C++ programmers tend to avoid using them like this. In this case of course, it looks like v is a candidate for being an STL vector or something like that.

You seem to use different naming conventions for private member variables, but as long as it is private stuff I think you can do whatever you want. But I guess either postfixing all member variables with _ is fine, so is prefixing with m_, but you should never just prefix with _ because you might stumble into reserved naming conventions for C, Posix and/or compilers.

So what is it that she seems to understand better than most?

So what is it that she seems to understand better than most?

- the connection between C and C++

So what is it that she seems to understand better than most?

- the connection between C and C++
- some techniques for polymorphism

So what is it that she seems to understand better than most?

- the connection between C and C++
- some techniques for polymorphism
- how to initialize objects properly

So what is it that she seems to understand better than most?

- the connection between C and C++
- some techniques for polymorphism
- how to initialize objects properly
- rule of three

So what is it that she seems to understand better than most?

- the connection between C and C++
- some techniques for polymorphism
- how to initialize objects properly
- rule of three
- operator new[] and delete[]

So what is it that she seems to understand better than most?

- the connection between C and C++
- some techniques for polymorphism
- how to initialize objects properly
- rule of three
- operator new[] and delete[]
- common naming conventions

We'd like to share some things about:

Object lifetime
The rule of three
The vtable

proper object initialization

assignment is not the same as initialization

```
struct A
{
    A() { puts("A()"); }
    A(int v) { puts("A(int)"); }
   ~A() { puts("~A()"); }
};

struct X
{
    X(int  v) { a=v; }
    X(long v) : a(v) { }
    A a;
};

int main()
{
    puts("bad style");
    { X slow(int(2)); }
    puts("good style");
    { X fast(long(2)); }
}
```

```
bad style
A()
A(int)
~A()
~A()
good style
A(int)
~A()
```

object lifetime

A basic principle of C++ is that the operations performed when an object's life ends are the exact reverse of the operations performed when the object's life starts.

```
struct A
{
    A() { puts("A()"); }
   ~A() { puts("~A()"); }
};

struct B
{
    B() { puts("B()"); }
   ~B() { puts("~B()"); }
};

struct C
{
    A a;
    B b;
};
```

```
int main()
{
    C obj;
}
```

```
A()
B()
~B()
~A()
```

object lifetime

A basic principle of C++ is that the operations performed when an object's life ends are the exact reverse of the operations performed when the object's life starts.

```
struct A
{
    A() : id(count++)
    {
       printf("A(%d)\n", id);
    }
    ~A()
    {
        printf("~A(%d)\n", id);
    }
    int id;
    static int count;
};
```

```
int main()
{
    A array[4];
}
```

```
A(0)
A(1)
A(2)
A(3)
~A(3)
~A(2)
~A(1)
~A(0)
```

object lifetime

A basic principle of C++ is that the operations performed when an object's life ends are the exact reverse of the operations performed when the object's life starts.

```
struct A
{
    A() : id(count++)
    {
       printf("A(%d)\n", id);
    }
    ~A()
    {
        printf("~A(%d)\n", id);
    }
    int id;
    static int count;
};
```

```
int main()
{
    A * array = new A[4];
    delete[] array;
}
```

```
A(0)
A(1)
A(2)
A(3)
~A(3)
~A(2)
~A(1)
~A(0)
```

```
int main()
{
    A * array = new A[4];
    delete array;
}
```

```
A(0)
A(1)
A(2)
A(3)
~A(0)
```

The Rule of Three

If a class defines a copy constructor

```
class wibble_ptr {
public:
    wibble_ptr()
        : ptr(new wibble), count(new int(1)) {
    }
    wibble_ptr(const wibble_ptr & other)
        : ptr(other.ptr), count(other.count) {
        (*count)++;
    }
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
private:
    wibble * ptr;
    int * count;
};
```

The Rule of Three

If a class defines a copy constructor, a copy assignment operator

```
class wibble_ptr {
public:
    wibble_ptr()
        : ptr(new wibble), count(new int(1)) {
    }
    .
    .
    .
    .
    wibble_ptr & operator=(const wibble_ptr & rhs) {
        wibble_ptr copy(rhs);
        swap(copy);
        return *this;
    }
    .
    .
    .
    .
    .
private:
    wibble * ptr;
    int * count;
};
```

The Rule of Three

If a class defines a copy constructor, a copy assignment operator, or a destructor

```
class wibble_ptr {
public:
    wibble_ptr()
        : ptr(new wibble), count(new int(1)) {
    }
    .
    .
    .
    .
    .
    .
    .
    .
    .
    ~wibble_ptr() {
        if (--(*count) == 0)
            delete ptr;
    }
    .
private:
    wibble * ptr;
    int * count;
};
```

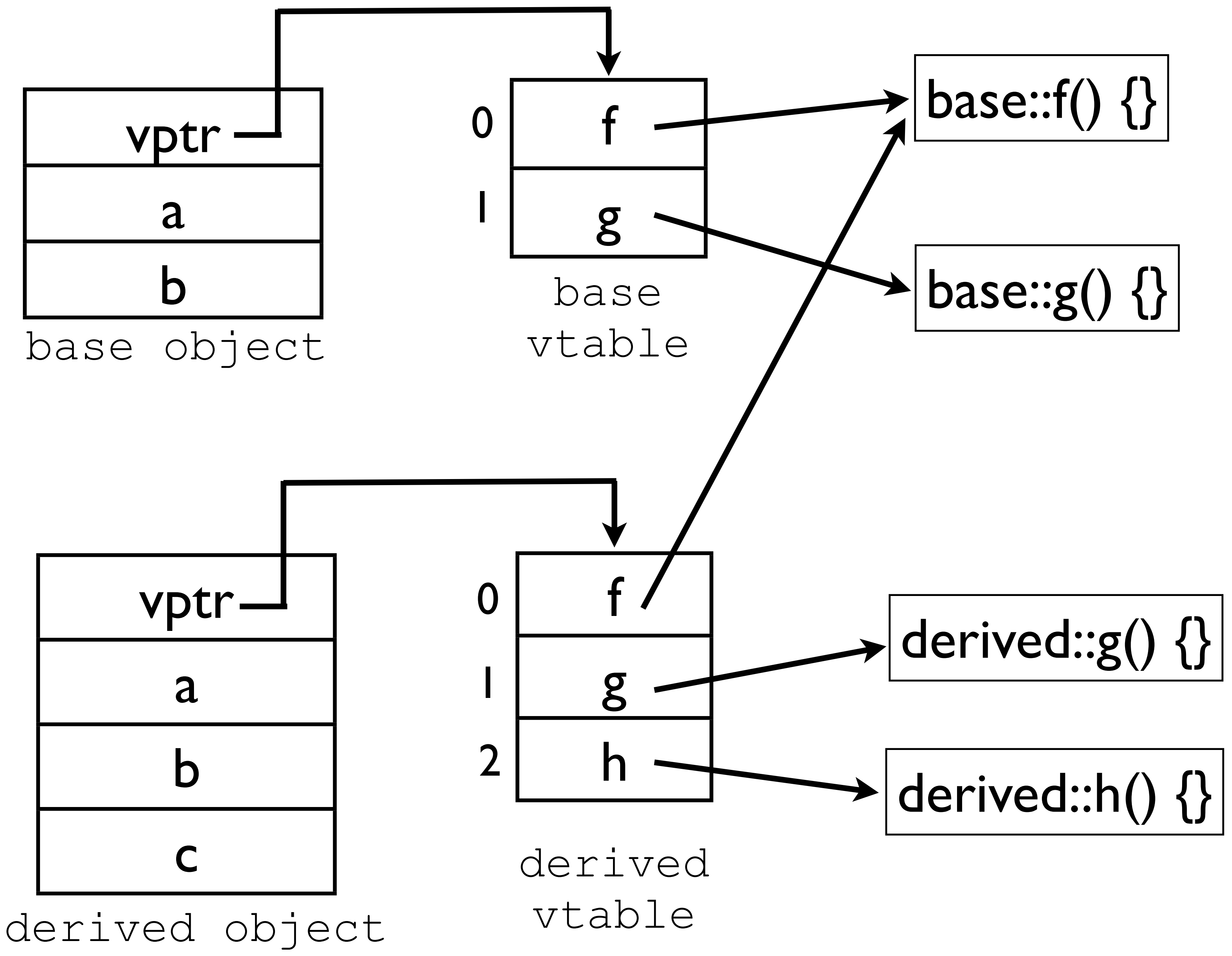
The Rule of Three

If a class defines a copy constructor, a copy assignment operator, or a destructor, then it should define all three.

```
class wibble_ptr {
public:
    wibble_ptr()
        : ptr(new wibble), count(new int(1)) {
    }
    wibble_ptr(const wibble_ptr & other)
        : ptr(other.ptr), count(other.count) {
        (*count)++;
    }
    wibble_ptr & operator=(const wibble_ptr & rhs) {
        wibble_ptr copy(rhs);
        swap(copy);
        return *this;
    }
    ~wibble_ptr() {
        if (--(*count) == 0)
            delete ptr;
    }
    ...
private:
    wibble * ptr;
    int * count;
};
```

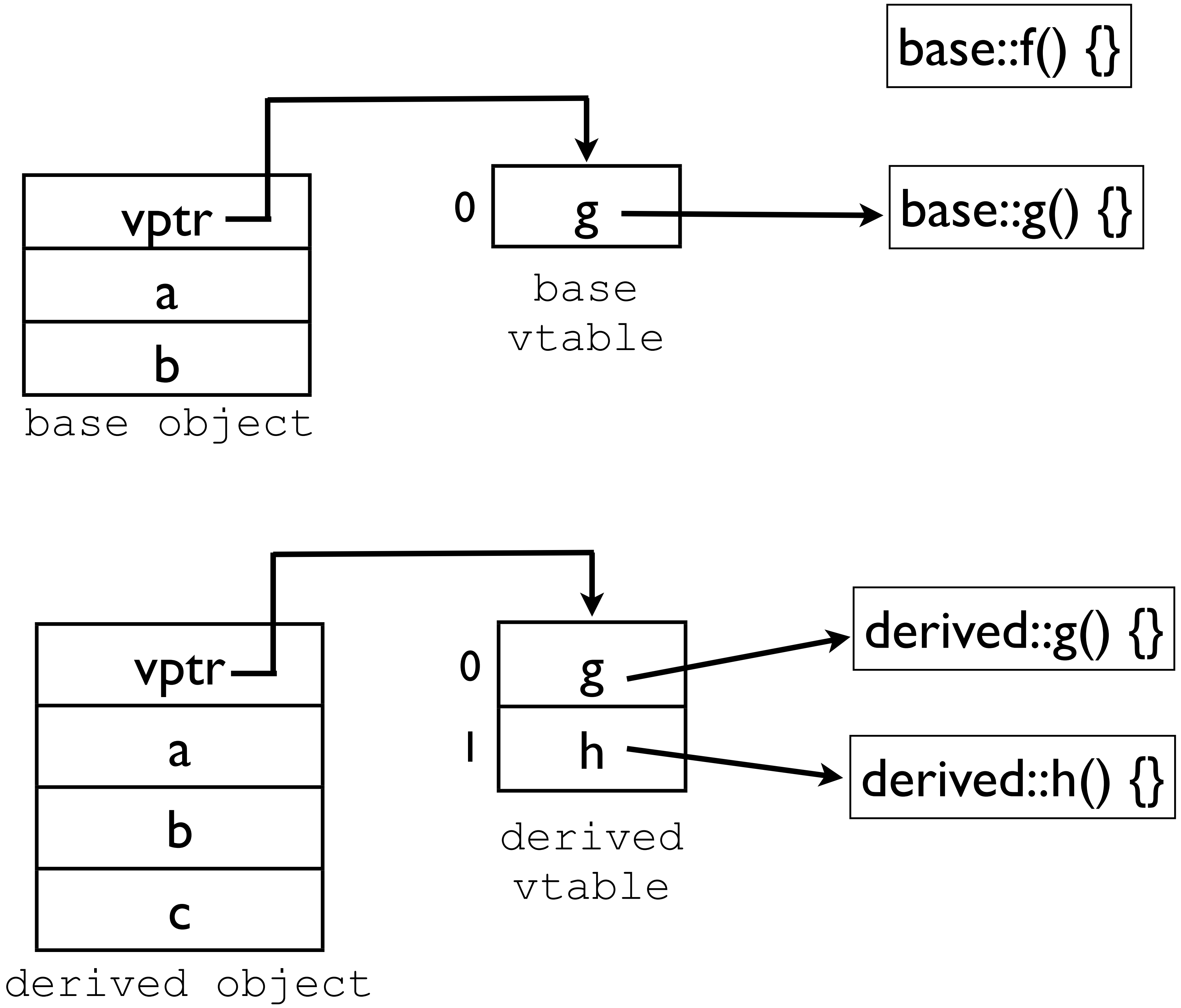
The vtable

```
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());
}
```

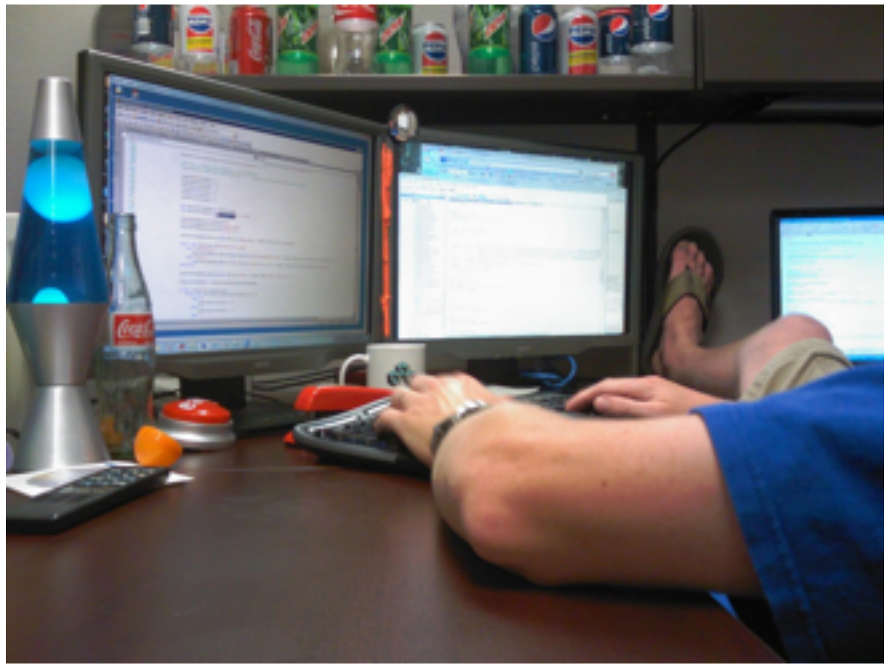


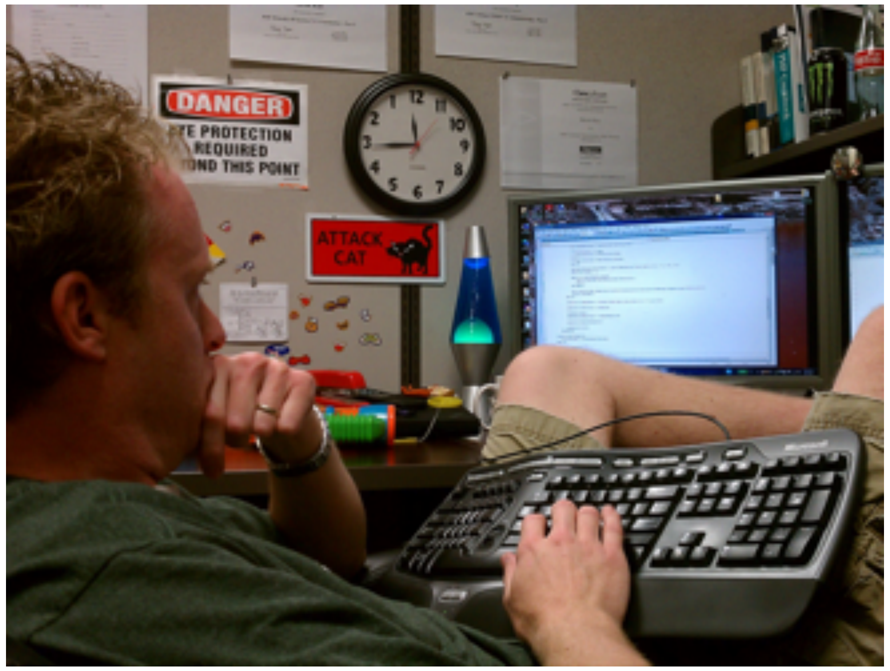
The vtable

```
struct base
{
              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());
}
```



Would it be useful if more of your colleagues have a deep understanding of the programming language they are using?





We are not suggesting that **all** your C and C++ programmers in your organization need a deep understanding of the language. But you certainly need a critical mass of programmers that care about their profession and constantly keep updating themselves and always strive for a better understanding of their programming language.

Let’s get back to our two developers...

So what is the biggest difference between these two developers?

So what is the biggest difference between these two developers?

Current knowledge of the language?

So what is the biggest difference between these two developers?

Current knowledge of the language?   No!

So what is the biggest difference between these two developers?

Current knowledge of the language? No!

It is their attitude to learning!

When was the last time you did a course about programming?

When was the last time you did a course about programming?

What do you mean? I learned programming at university and now I am learning by doing. What more do you need?

When was the last time you did a course about programming?

What do you mean? I learned programming at university and now I am learning by doing. What more do you need?

So what kinds of books are you reading?

When was the last time you did a course about programming?

What do you mean? I learned programming at university and now I am learning by doing. What more do you need?

So what kinds of books are you reading?

Books? I don't need books. I look up stuff on internet when I need it.

When was the last time you did a course about programming?

What do you mean? I learned programming at university and now I am learning by doing. What more do you need?

So what kinds of books are you reading?

Books? I don't need books. I look up stuff on internet when I need it.

Do you discuss programming with your colleagues?

When was the last time you did a course about programming?

What do you mean? I learned programming at university and now I am learning by doing. What more do you need?

So what kinds of books are you reading?

Books? I don't need books. I look up stuff on internet when I need it.

Do you discuss programming with your colleagues?

They are all stupid, I have nothing to learn from them...

You seem to know a lot about C and C++? How come?

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

I read books. Lots of books. Did you know that James Grenning just came out with a great book about Test-Driven Development in C?

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

I read books. Lots of books. Did you know that James Grenning just came out with a great book about Test-Driven Development in C?

I have to admit that I visit WG14 and WG21 once in a while

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

I read books. Lots of books. Did you know that James Grenning just came out with a great book about Test-Driven Development in C?

I have to admit that I visit WG14 and WG21 once in a while

I am a member of ACCU, for those who care about professionalism in programming, I read Overload, C Vu and discussions on accu-general

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

I read books. Lots of books. Did you know that James Grenning just came out with a great book about Test-Driven Development in C?

I have to admit that I visit WG14 and WG21 once in a while

I am a member of ACCU, for those who care about professionalism in programming, I read Overload, C Vu and discussions on accu-general

And whenever I get a chance I attend classes teaching C and C++. It not always because I learn so much from the slides and the teacher, it is often through discussions with other learners that I expand my knowledge.

You seem to know a lot about C and C++? How come?

I am learning new things every day, I really enjoy it.

I occasionally follow C and C++ discussions on stack overflow, comp.lang.c and comp.lang.c++

I am a member of a local C and C++ Users Group, we have meetings once in a while

I read books. Lots of books. Did you know that James Grenning just came out with a great book about Test-Driven Development in C?

I have to admit that I visit WG14 and WG21 once in a while

I am a member of ACCU, for those who care about professionalism in programming, I read Overload, C Vu and discussions on accu-general

And whenever I get a chance I attend classes teaching C and C++. It not always because I learn so much from the slides and the teacher, it is often through discussions with other learners that I expand my knowledge.

But perhaps the best source of knowledge is working closely with my colleagues and try to learn from them while contributing with my knowledge.

Summary

Summary

- compiler and linker

Summary

- compiler and linker
- declaration vs definition

Summary

- compiler and linker
- declaration vs definition
- activation frame

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++
- proper initialization of objects

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++
- proper initialization of objects
- object lifetimes

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++
- proper initialization of objects
- object lifetimes
- vtables

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++
- proper initialization of objects
- object lifetimes
- vtables
- rule of 3

Summary

- compiler and linker
- declaration vs definition
- activation frame
- memory segments
- memory alignment
- sequence points
- evaluation order
- undefined vs unspecified
- optimization
- something about C++
- proper initialization of objects
- object lifetimes
- vtables
- rule of 3
- ... and something about attitude and professionalism

!

Eh?

Eh?

Yes?

Eh?

Yes?

I really love programming, but I realize now that perhaps I am not behaving as a true professional. Any advice on how to get started to get a deep knowledge of C and C++?

Eh?

Yes?

I really love programming, but I realize now that perhaps I am not behaving as a true professional. Any advice on how to get started to get a deep knowledge of C and C++?

First of all you must realize that programming is a continuous learning process, it does not matter how much you know, there is always much more to learn. The next thing to realize is that professional programming is first of all a team activity, you must work and develop together with your colleagues. Think about programming as a team sport, where nobody can win a whole match alone.

Eh?

Yes?

I really love programming, but I realize now that perhaps I am not behaving as a true professional. Any advice on how to get started to get a deep knowledge of C and C++?

First of all you must realize that programming is a continuous learning process, it does not matter how much you know, there is always much more to learn. The next thing to realize is that professional programming is first of all a team activity, you must work and develop together with your colleagues. Think about programming as a team sport, where nobody can win a whole match alone.

Ok, I need to think about that...

Eh?

Yes?

I really love programming, but I realize now that perhaps I am not behaving as a true professional. Any advice on how to get started to get a deep knowledge of C and C++?

First of all you must realize that programming is a continuous learning process, it does not matter how much you know, there is always much more to learn. The next thing to realize is that professional programming is first of all a team activity, you must work and develop together with your colleagues. Think about programming as a team sport, where nobody can win a whole match alone.

Ok, I need to think about that...

Having said that. Make it a habit to once in a while take a look at the assembly output actually produced by snippets of C and C++. There are a lot of surprising things to discover. Use a debugger, step through code, study how memory is used and look at the instructions actually executed by the processor.

© www.ClipProject.info

Any books, sites, courses and conferences about C and C++ you would like to recommend?

Any books, sites, courses and conferences about C and C++ you would like to recommend?

To learn modern ways of developing software, I recommend “Test-Driven Development for Embedded C” by James Grenning. For deep C knowledge, Peter van der Linden wrote a book called “Expert C programming” two decades ago, but the content is still quite relevant. For C ++ I recommend you start with “Effective C++” by Scott Meyers and “C++ coding standards” by Herb Sutter and Andrei Alexandrescu.

Any books, sites, courses and conferences about C and C++ you would like to recommend?

To learn modern ways of developing software, I recommend “Test-Driven Development for Embedded C” by James Grenning. For deep C knowledge, Peter van der Linden wrote a book called “Expert C programming” two decades ago, but the content is still quite relevant. For C++ I recommend you start with “Effective C++” by Scott Meyers and “C++ coding standards” by Herb Sutter and Andrei Alexandrescu.

Also, whenever you get a chance to go to a course about C and C++, do so. If your attitude is right, there is just so much to learn from both the instructor and the other students at the course.

Any books, sites, courses and conferences about C and C++ you would like to recommend?

To learn modern ways of developing software, I recommend "Test-Driven Development for Embedded C" by James Grenning. For deep C knowledge, Peter van der Linden wrote a book called "Expert C programming" two decades ago, but the content is still quite relevant. For C++ I recommend you start with "Effective C++" by Scott Meyers and "C++ coding standards" by Herb Sutter and Andrei Alexandrescu.

Also, whenever you get a chance to go to a course about C and C++, do so. If your attitude is right, there is just so much to learn from both the instructor and the other students at the course.

And finally, I would recommend that you get yourself involved in user groups and communities for programmers. In particular I would recommend the ACCU (accu.org), they are very much focused on C and C++ programming. Did you know they host a conference in Oxford every spring where professional programmers from all around the world meet to discuss programming for a week? Perhaps I see you there in April next year?

Any books, sites, courses and conferences about C and C++ you would like to recommend?

To learn modern ways of developing software, I recommend “Test-Driven Development for Embedded C” by James Grenning. For deep C knowledge, Peter van der Linden wrote a book called “Expert C programming” two decades ago, but the content is still quite relevant. For C ++ I recommend you start with “Effective C++” by Scott Meyers and “C++ coding standards” by Herb Sutter and Andrei Alexandrescu.

Also, whenever you get a chance to go to a course about C and C++, do so. If your attitude is right, there is just so much to learn from both the instructor and the other students at the course.

And finally, I would recommend that you get yourself involved in user groups and communities for programmers. In particular I would recommend the ACCU (accu.org), they are very much focused on C and C++ programming. Did you know they host a conference in Oxford every spring where professional programmers from all around the world meet to discuss programming for a week? Perhaps I see you there in April next year?

Thanks!

Any books, sites, courses and conferences about C and C++ you would like to recommend?

To learn modern ways of developing software, I recommend "Test-Driven Development for Embedded C" by James Grenning. For deep C knowledge, Peter van der Linden wrote a book called "Expert C programming" two decades ago, but the content is still quite relevant. For C++ I recommend you start with "Effective C++" by Scott Meyers and "C++ coding standards" by Herb Sutter and Andrei Alexandrescu.

Also, whenever you get a chance to go to a course about C and C++, do so. If your attitude is right, there is just so much to learn from both the instructor and the other students at the course.

And finally, I would recommend that you get yourself involved in user groups and communities for programmers. In particular I would recommend the ACCU (accu.org), they are very much focused on C and C++ programming. Did you know they host a conference in Oxford every spring where professional programmers from all around the world meet to discuss programming for a week? Perhaps I see you there in April next year?

Thanks!

Good luck!

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