

CS205

C / C++

Stéphane Faroult

ABOUT ME

Office: Room 914, Building A7, 9th floor
iPark, N° 1001 Xueyuan Road
if you want to meet me, send

me an email
Steffann *Farroo*
Stéphane Faroult

faroult@sustc.edu.cn

Class Material: <http://sakai.sustc.edu.cn/>

TEACHING ASSISTANT

Wang Wei (Vivian)



Contact info

- E-mail: vivian2017@aliyun.com
- Mobile: 13043458092
- Office: Room 913, Building A7, 9th floor
iPark, N° 1001 Xueyuan Road;

ABOUT THIS COURSE

	Monday	Tuesday	Wednesday	Thursday
8:00-9:50		LECTURE Room101, LiYuan Park 2		
10:10-12:00	LECTURE Room102, LiYuan Park 2		LECTURE Room102, LiYuan Park 2	LECTURE Room101, LiYuan Park 2
14:00-15:50		LAB Room205, Teaching Building 2		LAB Room205, Teaching Building 2
16:10-18:00		LAB Room205, Teaching Building 2		LAB Room205, Teaching Building 2

ABOUT THIS COURSE

C/C++ + Linux

<http://cygwin.org/>



The C programming language is historically closely associated with Unix systems (C was created for the development of the first Unix system, of which Linux is a descendent). This is why this course will also be an opportunity to take a look at Unix/Linux systems, and programming in these environments. You'll be able to code most of what is shown in this course on a Windows computer. However, if you only have a Windows machine, I recommend that you install Cygwin (free product) with the core development tools (gcc, make – they aren't installed by default, you must specifically select the packages). Cygwin runs on Windows but gives you a Linux "look and feel", and allows to code on a Windows machine using functions normally only available on Unix/Linux machines.

Cygwin has been installed on the lab computers.

ABOUT THIS COURSE

Expectations

Good understanding of C/C++
 Ability to write reasonably complex programs
 "Professional attitude"

I don't expect you to become C experts (it's said that you need 10,000 hours of practice to really master a topic – that's five years) but to be able to start confidently in a job requiring C programming.

ABOUT THIS COURSE

Exams test you on

They are tough on purpose.
 Not doing too well is normal ...

General knowledge about C/C++
 Ability to write pseudo-code for a moderately complex algorithm
 Being able to tell what a program does
 Finding errors in a program

LABS are **VERY** IMPORTANT

I believe in practice. If you manage to code lab assignments (even with a little help from me or teaching assistants), you'll progress a lot.

Mid Course Exam

Thursday, August 3rd

Final Exam

Thursday, August 17th

110 minutes, usual room, usual time

All exams are



Open book, open notes



Electronic devices forbidden

In the industry you aren't asked to know everything by heart. No electronic devices because I don't want any e-chat. Take notes of PDF documents, too big to print.

I'll provide an annotated version of lecture slides (PDF).

You can but don't need to take notes during lectures.

**Take notes from the course notes
AFTER lectures!**

This course will total about 1,400 slides. The course notes will total about 350 pages (4 slides per page). I advise you to create your own notes for the exam from the comprehensive course notes.

MidTerm Exam	20%
Final Exam	30%
Labs	40%
Quizzes	10%

The grade you will get for exams is a "raw" grade, where I consider that 40% is OK, 50% correct and 70% quite good. They will be adjusted (up) if need be for the final, official grades.

If you do well in labs, you'll pass

I consider that if you can code a correct program on your own, expectations are met. Exams are here to decide on whether people get a very good, good or passing grade.

DON'T FALL BEHIND WARN ME ASAP

The course keeps building on what has been seen. If at one point you lose foot, don't wait until the end of the course because it will be too late. If you don't get a lecture, try to ask somebody else to explain it to you, or come and see me **as soon as possible** (what "asap" means) so that I can show you how it works.

Collaboration policy

You are encouraged to discuss algorithms with others.

Code must be **personal**.

I know that some people like to work together, and that there is sometimes a fine line between group work and copy-and-paste. But even with the same algorithm, devised together, two people usually come out with obviously different programs. When I have identical programs, I divide the grades. You will always get a better grade submitting a bit late a personal work.

Honesty

Getting code from the internet for labs/ assignments is **perfectly OK**

IF YOU SAY IN A COMMENT WHERE IT COMES FROM

We all have a personal coding style, and it's incredibly easy to spot whether you wrote something or not. When you borrow, just say it. You don't need to reinvent the wheel.

Honesty

Don't pretend or suggest that you are the author of something that you didn't write.

I'll be ferocious with people who cheat at exams.

Clear enough?

Quality Work

Robust Programs

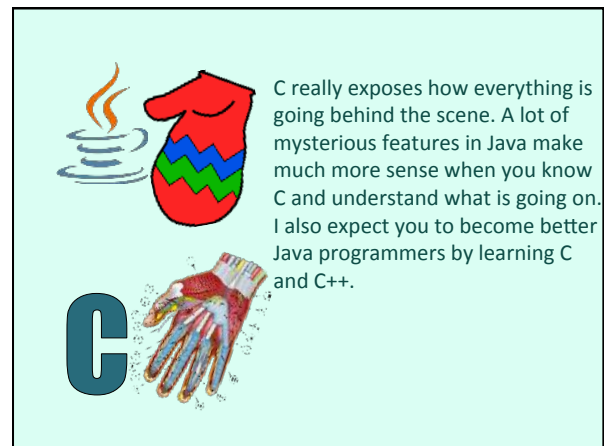
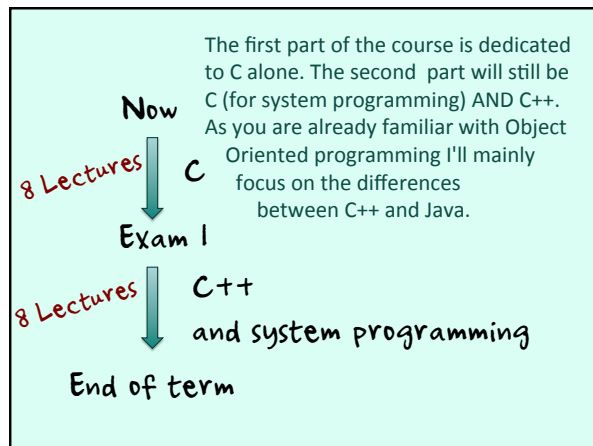
I insist on "robust" programs, which means programs that don't crash when something unexpected happens. It's easy to write a program that works well when everything is as expected. But if the user enters a filename, don't expect everything to work fine (there may be a typo in the name and you may not find it). Many users misread instructions or misunderstand them. Your program must behave correctly in all cases (even if it means exiting with a helpful message).

Quality Work

Robust Programs

Craftsmanship

Quality also means craftsmanship, and writing a program that is well designed and easy to use. There is a degree of subjectivity to it, but your program should feel "well finished" (no debugging messages in the final version!). Have it tested by a friend.



I won't be following it but this book (easily found on the web) has always been the reference. It's considered to be one of the best computer language books ever written , and one of the classics of computer literature.

Strongly recommended reading.



**Kernighan
& Ritchie**

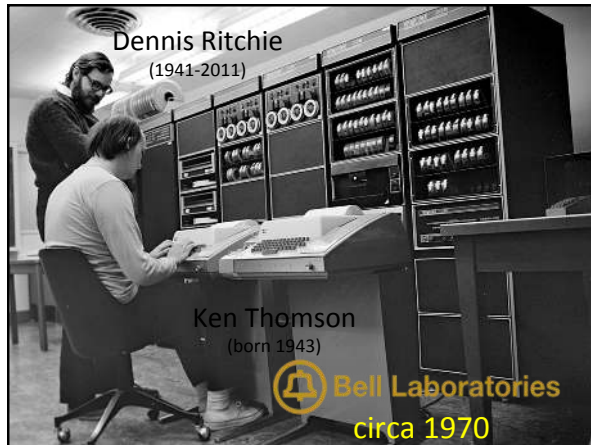


Bell Laboratories

circa 1970

The story of C starts in the Bell labs around 1970. At this time, Ken Thomson was developing a new operating system, UNIX (from which Linux and MacOSX derive, as well as other systems such as Solaris, HP-UX, AIX, ...) A high-level language to write some low-level operating system stuff was needed (more convenient than assembler), and Dennis Ritchie created C. Almost all big languages in use today (Java, C#, C++, Objective-C, PHP ...) derive from C, but C is still, more than 40 years later, one of the most actively used languages. If you are unconvinced, check this:

<http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>



Original "K&R" version

Various versions from
the late 1970s

There are several
versions of C, we'll use
the most common one,
known as "C89"

Standard defined by the American National
Standards Institute in the 1980s

C89

C++ created

Standard adopted by the International Standard
Organization in the late 1980s

Java created

C99 defined in 1999, C11 defined in 2011

Programming in C

Editor (vi, notepad)

Command line (gcc)

Build tools (make)

Or Integrated Development Environment

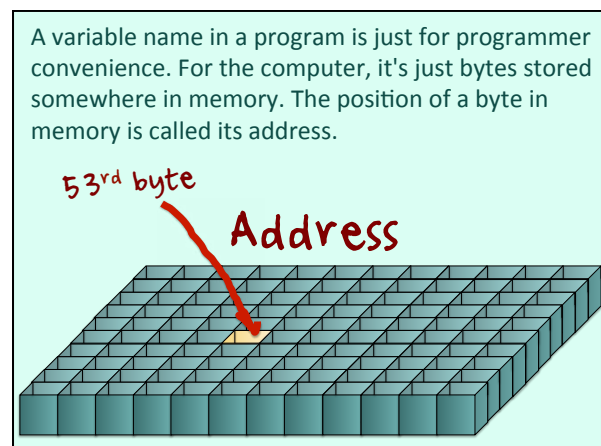
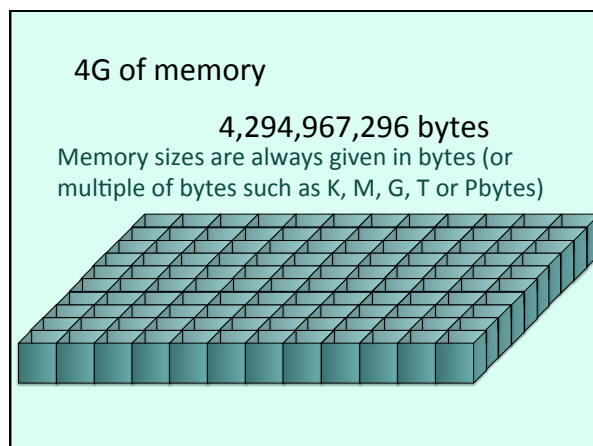
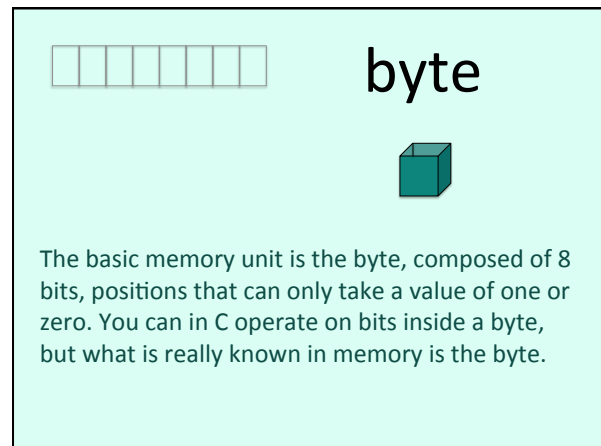
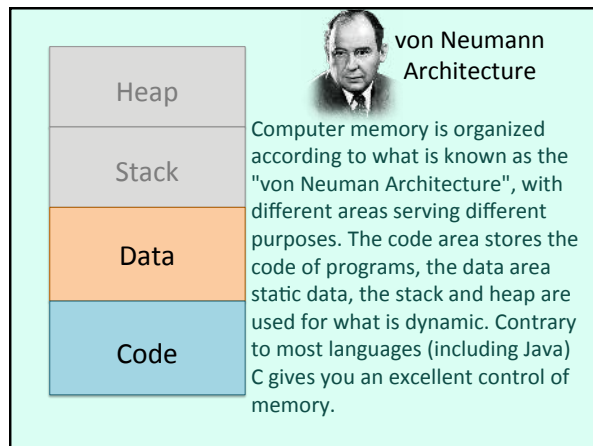
Visual Studio

XCode

You can program C
in different ways, in
a console or using an
IDE. Use what you
like best ...

Back to Basics

Because C is very close to the machine (it's the language
mostly used for writing operating systems and "low
layers", such as networking or databases) it's worth
talking a bit about how it works inside a computer and
concepts that are "abstracted" by some more recent or
more high-level languages.



01000000

However, bits are just codes (a combination of bits is supposed to have a specific meaning) and, depending on what you want to store, may be interpreted differently.

Thus, a byte containing 01000000 might be interpreted as an ASCII character code, in which case it corresponds to @, or as an integer value in base 2, $0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$, in which case it would mean 64. You cannot understand bits if you don't know what they are supposed to represent.

@ ?

64 ?

This is why you need to declare variables, which reserve room for storing data, and also tell about how to encode/decode (languages in which variables are not declared store everything as characters, and convert on the fly).

Declaration of **variables**

my_var

Data type

How to decode

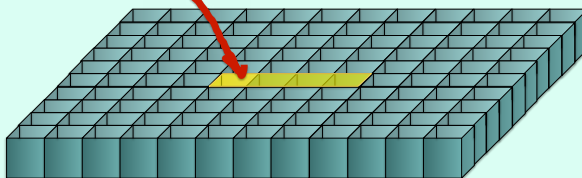
How many bytes

*Classic C naming style.
myVar also works ...*

Depending on what you want to store, you may need a different number of bytes.

If you declare an integer variable for instance, the compiler will reserve four bytes and will know that it's stored in base 2.

53rd byte my_var



IMPORTANT

2 ways to refer to data in memory

By variable name

By address

Although the computer always use addresses when it executes a program, in most languages you can refer to variables in the code only by using their name. In C, you can use either the variable name or its address. Of course, we are not interested by the actual address (which can change from execution to execution) but we can store them to special variables and do clever things, as you'll see later.

Variables that store addresses are called "pointers". They all have the same size, which depends on the computer architecture (32-bit: pointers are stored in 4 bytes, 64-bit: pointers are stored in 8 bytes)

Some variables contain addresses,

they are called "pointers"

my_var

ptr_my_var = &my_var

*ptr_my_var

Pointers have a type
(otherwise we wouldn't
know how to understand
what is at the address)

gives address

gives value at address

We'll see pointers
again (and again)
and in much more
detail ...

Data types

char c



Only ONE byte
Too small to store a
Chinese character!

No string type!

Text is stored in arrays of chars

char text[n]



Different
from Java

n

Although superficially most C data types look a lot like Java data types, there are many significant differences worth noting:

- A Java char is composed of two bytes. A C char is composed of a single byte. That means that you can store a Chinese character in a Java char, not in a C char.
- There is no boolean data type in C (more about this later)
- There is no String type in C. A C string of characters is a plain array of (single byte) chars.
- You may notice a small different of syntax with Java when you declare arrays. Square brackets (that enclose a size) follow the variable name and there is no "new".

Data types

char c



C isn't as strict with types as Java. A char is a smaller integer type in C. 'a' + 3 results in 'd', and any integer can be used to store a boolean result (false if it contains 0)

Can also be seen as a number (byte)

'a' + 3

Can also be seen as a boolean

0 false

<>0 true

Data types

char c



Depending on how big the integer value, you'll reserve a varying number of bytes. The numbers of bytes given here are pretty standard

-128 to +127

About -32000 to +32000

short n



but may be different

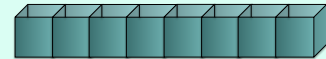
About -2 billion to +2 billion

int num



on some systems.

long bignum



Data types

Integer values (char included) support these operators:

Usual operators (+, -, *, /)

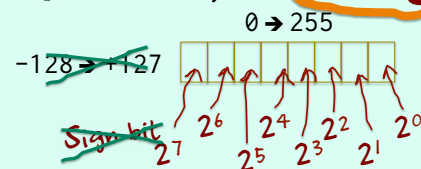
Modulo (%) remainder of integer division

Bit operations (&, |, ~)

Bit shift (>>, <<) Bit operations and shifts are mostly useful in Computer Engineering and very low-level operations.

There is something in C that doesn't exist in Java: integer values can be "unsigned", which means that the "bit sign" is considered like a regular bit.

Values can be **signed** (default) or **unsigned**



C was written at a time when memory was VERY expensive, and "unsigned" allowed to store bigger numbers in fewer bytes. Beware that some things that you would think are always positive sometimes are not: planes can fly below sea-level (negative altitude), and interest rates can be negative. You're better off working with signed values (char excepted).

unsigned

CAREFUL!

Data types

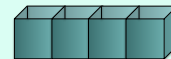
Sign (1 bit)

Exponent (8 bits)

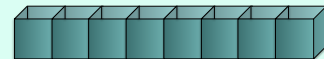
Radix (23 bits)

radix exponent
 0.314152×10^1
 ↑ Not stored in decimal but same idea

float **f**



double **d**



Exponent (11 bits)

Radix (52 bits)

In all languages, float values are stored in two parts, a radix (the first significant digits) and an exponent (a power, often of 64), with sometimes some tricks to make hardware operations more efficient.

You must remember that for many numbers (fractions, irrational numbers) there is always a rounding error. This is very important in engineering, because when you compute something, you want to avoid a method that does too many iterations and adds rounding errors at each step. Algorithms must be fast and precise enough.

One interesting feature of arrays in C is that they are closely related to pointers. When you declare an array, you reserve an area of consecutive memory the size of **Arrays** which is the size of one element times the number of elements.

my_var

Value

my_arr[n]

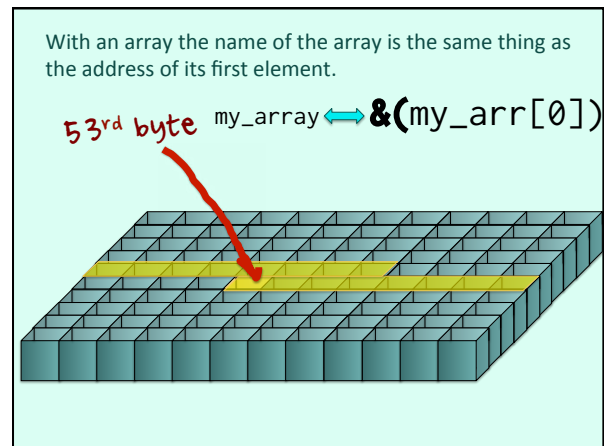
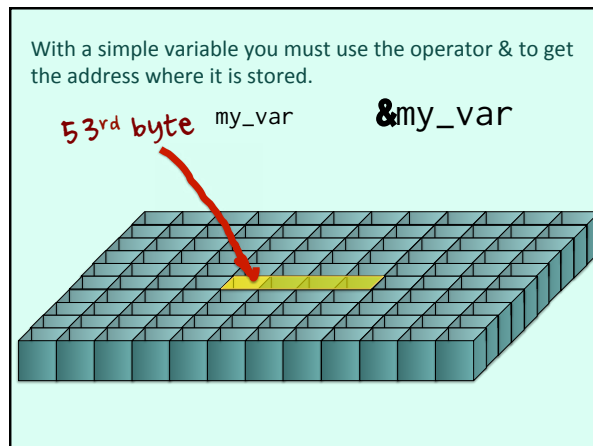
Value

In C, the name of the array is actually the address where this memory area starts (in Java too, but it's hidden)

my_arr

Address

In Java, the array declaration reserves no storage – "new" does it.



C arrays are much simpler animals than Java arrays, which are "objects". In C, arrays are just successive bytes in memory.

number of elements not fully "managed"

~~.length~~

There is no way in C to know where exactly an array ends, and no "length" property. A developer must manage arrays by oneself.

arrays.c

Arrays


```
#include <stdio.h>

int main() {
    int my_arr[4];

    my_arr[5] = 12;
    return;
}
```

A C compiler is able to notice when you are using a reference that is bigger than the size of the array.

Ouch!



Arrays

arrays.c


```
#include <stdio.h>

int main() {
    int my_arr[4];
    int n = 5;

    my_arr[n] = 12;
    return 0;
}
```

But if your index is a variable that can take any value, the compiler won't see anything and the program will compile.

Ouch again!



Arrays

```
$ gcc arrays.c -o arrays
$
```

When you run your program, you won't get in C any "Index out of bounds" exception. Your program will read bytes passed the end of your array, which may contain anything.

Using a piece of memory not reserved for the program ...

Anything can happen!

Your program will use garbage data; at that point everything becomes random. Your program may crash, or not.

Strings

The fact that the length isn't a built-in property of an array leads to a very special handling of strings. Strings are arrays, but something is required to tell where they end.

```
char my_char = 'H';
char my_text[5] = {'H', 'e', 'l', 'l', 'o'};
```

This is WRONG

End-of-string marker

'\0' = byte with all bits set to zero

You will declare an array that will be big enough to accommodate the biggest string you want to store (for instance, say 20 characters for a western surname). Most surnames, though, will be shorter. As Chinese characters are stored on three bytes in UTF-8 (the most common encoding on the web), you need 9 chars to store a 3-character Chinese surname. A special character (and it's ONE character even if it's represented by two) is used to indicate the end of the actual string stored. **IT NEEDS TO BE STORED TOO.** Everything past it in the array will be ignored.

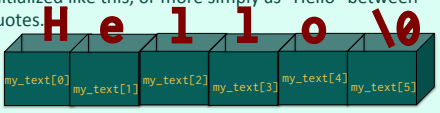
Strings

Always one more than needed
IMPORTANT

```
char my_char = 'H';
```

```
char my_text[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

It can be initialized like this, or more simply as "Hello" between DOUBLE quotes.



A C string must always be one byte bigger than what you want to store, to accommodate the end-marker. Without the end-marker, programs crash.

'A' One character

"A" Two characters

{ 'A', '\0' }

There is also in Java this difference between single quotes enclosing a char, and double quotes enclosing a string. In C, single quote means single character (as \0 is a single character, you will refer to it as '\0'). Double quotes mean string, in other words mean that there is implicitly a \0 after the visible letters. You can process strings either with special functions (we'll see them later) or as plain arrays.

We'll see how to handle asian (and other) languages later



I have hinted that Chinese characters require several bytes of storage. Depending on the encoding, it can be two (frequently used in China) or three (the Web). Many Western characters (such as é, ø, ü) are coded on more than one byte on the web as well, like Russian.

As languages that you know derive from C, syntax will look familiar (but C was the first to introduce this)

In a C program:

Instructions terminated by **;**

Block of instructions **{ }**

However, you cannot in C define a function inside a function. All functions are at the same level.

Functions are NOT nested

As a syntax example, let's write a very simple program that inputs a number of kilometers and converts it to miles.

When you compile a C program, there are in fact three phases:

- First a preprocessor runs over your code. The preprocessor simply replaces text. It looks for lines that start with a #, which are preprocessor instructions.
- `#include` says to fetch a header file and insert it into the code. `#define` gives a name to a constant value. Whenever the name is found in the code, it will be replaced by the value.
- Then the compiler turns the code into an "object file"
- then the linker adds the code of standard C functions and turns the program into an "executable" program.

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



to_miles.c

`#include` is very different from `import`. Header files are text files that just define things.

Well-known location (eg /usr/include)

Defines constants – eg

```
#define SEEK_SET 0
```

Defines "grammatical rules" – eg

```
extern int printf(const char *__restrict __format, ...);
```

These function prototypes allow the compiler to check that you have datatypes and number of parameters right.

No executable instructions

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

to_miles.c

```
int main() {
```

```
    char input[80];
```

```
    float km;
```

```
    printf("Please enter a number of kilometers: ");
```

```
    fgets(input, 80, stdin);
```

```
    sscanf(input, "%f", &km);
```

↑ Address

I am reading here a line using a function called `fgets` (also used for reading from files), then scanning the string read for a float value. `sscanf` reads from a string, `scanf` (single s) directly from the keyboard.

A C program must ALWAYS have a function called `main()`. Guess where Java took the idea? NOT static (the word also exists in C but has a different meaning from Java).

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

to_miles.c

```
int main() {
```

```
    char input[80];
```

```
    float km;
```

```
    printf("Please enter a number of kilometers: ");
```

```
    fgets(input, 80, stdin);
```

```
    sscanf(input, "%f", &km);
```

```
    printf("%f kilometers = %f miles\n", km, km * 0.62);
```

```
    return 0;
```

Beware: on Windows it closes windows before you can read ...

Returning 0 is a convention that means "OK". You can, like in Java, have a void main function, but returning a int value is recommended.


```
$ ./to_miles
Please enter a number of kilometers: 40000
40000.000000 kilometers = 24800.000000 miles
$
```

On Linux (or Mac) or with Cygwin I build my program in a console using:

```
gcc -o to_miles to_miles.c
```

-o is followed by the name I give to the executable program. If I omit it the program will be called a.out (a.exe with Cygwin).

Note that ./ is here to specify that the program is located in the current directory. A Linux system looks for commands into various directories, but the current one isn't necessarily in the list.

```
$ gcc to_miles.c -o to_miles
```

Preprocess

Compile

Link

When you compile a C program there are in fact three stages:

first the preprocessor reads your program, replaces #include commands by the text of the various files, and substitutes what is

#defined. Then the resulting code is transformed into an "object file" (.o file - unrelated to Java or C++ objects), instructions understandable by the computer processor. Finally, the "object file" is "linked" with the standard functions that you use but didn't write to generate the program that you can run.

"Understandable" language



The build process of C program is significantly different from what happens with Java. With java, javac turns your program into a .class file that you cannot read but that the computer cannot read directly either. It's the Java Virtual Machine (JVM) that can make sense of the .class and really executes the program. C removes the middleman – and is faster as a result (if well written).

Binary code the computer can run

This summarizes the process. Each step can be performed independently.

"Understandable" language



pre-processing
compiling



Binary code

Binary code
from "libraries"



linking

Binary code the computer can run

A compiled language is **faster** and uses **fewer resources**.

On the other hand, a virtual machine provides a better safety net (you can crash a machine by running a C program, I have done it) and allows you to dynamically observe or change things (a feature known as "reflection"), which is impossible with a compiled language. It's speed versus comfort.

Let's improve the code ...

The example program we've seen isn't very well written. From a professional standpoint, it has one main issue: you should never "hardcode" (that means typing a number value) the size of an array or string in a program. What if one day we need to make an array bigger? We'll have to look for every place where we refer to the size, and if we miss one the program risks crashing. This is what the preprocessor `#define` statements are made for. You can set the value at one single place at the beginning of your program, the preprocessor will replace it (it plays the same role as Java `final` values).

```

#include <stdio.h>
#define INPUT_LEN    80

int main() {
    char input[INPUT_LEN];
    float km;

    printf("Please enter a number of kilometers: ");
    fgets(input, INPUT_LEN, stdin);
    sscanf(input, "%f", &km);
    printf("%f kilometers = %f miles\n", km, km * 0.62);
    return 0;
}

```

to_miles.c

This isn't perfect yet (we trust users to type a numerical value ...) but a step in the right direction.

Any C program has the same structure as this small example. No class! Variables are declared at the beginning of functions.

```

#include <stdio.h>
#define INPUT_LEN    80

int main() {
    char input[INPUT_LEN];
    float km;

    printf("Please enter a number of kilometers: ");
    fgets(input, INPUT_LEN, stdin);
    sscanf(input, "%f", &km);
    printf("%f kilometers = %f miles\n", km, km * 0.62);
    return 0;
}

```

to_miles.c

Header files, constant definitions, and global variables

Main function body

Variable declarations

input

Processing/output

Using mathematical functions

Using mathematical functions in a program sheds some additional light on how the program is built.

log_sample.c

```
#include <stdio.h>

#define INPUT_LEN    80

int main() {
    char input[INPUT_LEN];
    double val;

    printf("Please enter a value: ");
    fgets(input, INPUT_LEN, stdin);
    sscanf(input, "%lf", &val);
    printf("Log value: %lf\n", log(val));
    return 0;
}
```

`%lf` is a format place-marker for double values ("long float")

If I write a program such as this one, I won't be able to obtain an executable program, and it's interesting to see what happens when I compile it.

```
$ gcc log_sample.c -o log_sample
log_sample.c: In function 'main':
log_sample.c:12:31: warning: incompatible
implicit declaration of built-in function
'log' [enabled by default]
    printf("Log value: %lf\n", log(val));
                              ^
/tmp/ccOUKA3J.o: In function 'main':
log_sample.c(.text+0x57): undefined reference
to 'log'
collect2: error: ld returned 1 exit status
$
```

I'm getting a warning and an error, and they are generated at different steps.

"Understandable" language



The warning comes during compiling. When C encounters a function that it doesn't know, compiling doesn't fail as it would in Java. It simply assumes that the function is an int function (a common occurrence in C: many functions return an integer success or failure code). However, the function is used as if it were returning a double (which it does). Hence a warning.

```

#include <stdio.h>
#include <math.h>
#define INPUT_LEN 80

int main() {
    char input[INPUT_LEN];
    double val;

    printf("Please enter a value: ");
    fgets(input, INPUT_LEN, stdin);
    sscanf(input, "%lf", &val);
    printf("Log value: %lf\n", log(val));
    return 0;
}

```

log_sample.c

To remove the warning, we must include `math.h` that contains prototypes for all mathematical functions and says that `log()` returns a double. Then it becomes consistent.

```
$ gcc log_sample.c -o log_sample
```

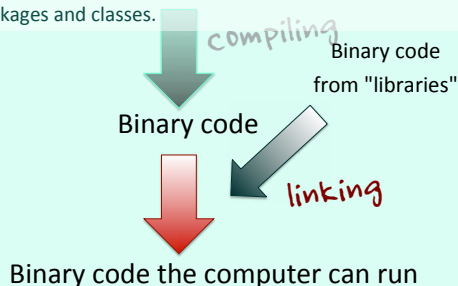
```

/tmp/cctyizTa.o: In function 'main':
log_sample.c(.text+0x57): undefined reference
to 'log'
collect2: error: ld returned 1 exit status
$

```

The warning is gone but we still have the error. The compiler created an intermediate "object file" in the `/tmp` directory (a work directory automatically cleaned-up when the system reboots), but it failed to build the program because it couldn't find the `log` function, and the linker (a program called `ld`) returned 1 (instead of 0) to indicate an error. Mathematical functions aren't part of the "standard" library like `printf` or `sscanf` and you need to tell the linker where to pull the code from.

This is where it failed. By default the linker searches for code in the "C standard library" called `libc.so` or `libc.a` on a Unix-like system (like Java looks for classes in the `java.lang` package). But you can link with a lot of libraries as in java you can import a lot of packages and classes.



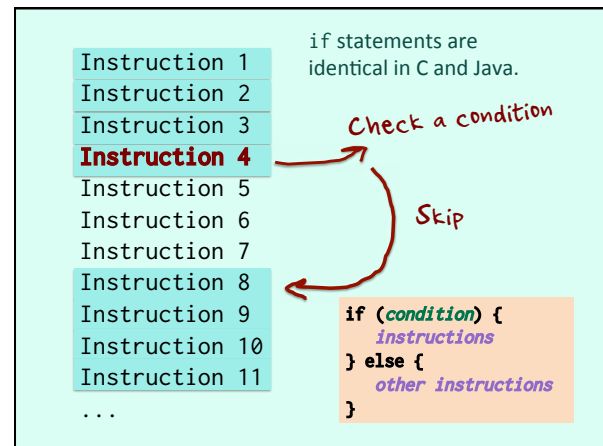
```
$ gcc log_sample.c -o log_sample -lm
$
```

Mathematical functions are located in a library called `libm.so` (the extension may vary). Documentation for functions tell where to find them. To indicate the name of the library to the linker, we need to add to our command the `-lm` flag, which is short for "link with `libm`". Libraries all have a name that starts with "lib" and therefore (because C developers like shortcuts) the name doesn't need to be given in full. If you also use functions from `libblablah`, you'll add `-lblablah` to your command line.

Go fetch missing functions in libm.so

Flow Control

By and large, in C flow control is similar to Java (or more precisely, flow control in Java is similar to C, as Java comes from C)



But there is no
boolean data
type in C?

So, what is the data type of a
condition?



But there is no
boolean data
type in C?

```

int a = 36;

if (a) {
    printf("Not zero\n");
} else {
    printf("Zero\n");
}

```

I have already hinted at
it:
integer values (char,
short, int, long) are
used. Comparison
operators return
integer values.

Use integers instead.
0 = false,
other = true.

Curly brackets not mandatory when conditions applies to a single instruction.
Better to keep them.

The curly brackets that I have used in the previous example are not mandatory (they are only mandatory for blocks that contain several instructions). It's safer to always use curly brackets, just in case we want to add more instructions to the block one day.

Inequality



Numbers OK



'a' > 'Z'

Comparison works as expected (and of course equality is ==, and inequality is !=). Beware with chars that lowercase a is greater than uppercase Z.

ASCII TABLE OF CHARACTERS																																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~			
0	1	2	3	4	5	6	7	8	9																							
!	"	#	\$	%	&	'	()	*	+	,	-	.	:	;	<	=	>	?@	[\]	^	_	`	a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~									
A	B	C	D	E	F	G	H	I	J	K	L																					

The reason is that the comparison is made on the numerical value of chars, and that in the ASCII code uppercase comes before lower case (the table above is an extended ASCII coding with values between 128 and 256 suitable for Western European countries; other encodings may be used elsewhere). Only the 127 first values are common to all encodings).



What happens if we type =, not ==?

c = b = a = 2;

```
if (a = 0) {
    printf("True\n");
} else {
    printf("False\n");
}
```

← ALWAYS

Combining conditions

&& means "and"**||** means "or"**!** means "not" (opposite)

You combine conditions with the same logical operators that you know from Java. The same rules of precedence exist ("not" applied before "and" applied before "or"), and a generous usage of parentheses is advised.

You may encounter **&** and **|** as "bit operators" (system programming). Related, but different!

Beware that single **&** and single **|** also exist but are bit operators.


There is a very interesting trap in C that doesn't exist in Java, and it has caught every C programmer at least once (usually more than once, but with experience you find the mistake faster). An assignment, as in Java, returns the value that was assigned, which allows to initialize several variables with the same value on the same line. If you type too fast and type a single **=** instead of a **==**, it's valid C (although as it's also one of the most famous mistakes you can make, some compilers will spot it and ask for a double pair of parentheses to confirm that it's really what you want). As a result, instead of testing equality, you are testing the result of the assignment, and it will be false if you assign 0 and true if you assign anything else. You can spend hours looking for the mistake without seeing it.

Same thing with **else if** statements.

Alternate option to nesting conditions

```
if (condition) {
    instructions
} else if (condition) {
    instructions
} else if (condition) {
    instructions
} else if (condition) {
    instructions
} else {
    instructions
}
```

Checked in
sequence



switch comes from C as well.

```
switch (expression) {
```

↑
char or integer
expression **ONLY**

Doesn't work with float/double

Doesn't work with arrays,
including strings

However, in Java you can apply switch to a String. You cannot in C. The expression must be an integer expression.

```
switch (expression) {  
    case value1: ← Single character  
                  or integer constant  
        instructions  
        break; ← You usually want  
                this, otherwise you  
                fall through to the  
                next case  
    case value2:  
        ...  
    default: ← When everything  
              else fails  
        instructions  
        break;
```

} As in Java, break exits the switch; without break, you fall through..