CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

```
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7
Instruction 8
Instruction 9
Instruction 10
Instruction 11
...
```

Again

Check a condition

```
while (condition) {
   instructions
}
```

Executed at least once

```
do {
   instructions
} while (condition) ;
```

```
while (condition) {
   instructions
}
```

Not much used

Loops are also identical to Java.

In Java you usually declare the index in the `for ()`; not in traditional C, where usually all variables are declared at the beginning of a function (Java behaviour is allowed in C99, which borrows it from Java).

```
for (initialization; condition; increment) {
   instructions
}
```

NOT declaration, except in C99

```
for (int i = 0; i < 100; i++) {
   ...
}
```

What happens if we type =, not ==?

```
int go_on = 1;

while (go_on = 1) {
   printf("Enter 0 to exit the loop\n");
   scanf("%d", &go_on);
}
```

re-assignment

The same problem happens with `while` loops as well. This is a good way to have an infinite loop.

Clever Trick

```
if (constant == variable) {
    ...
}
```

A good habit to take is to compare constant to variable instead of variable to constant. As you cannot assign a value to a constant, the compiler will complain if you forget one equal sign. Unfortunately it will not solve the problem of comparing two variables.

Command line parameters

```
int main(int argc, char *argv[]) {

}
```

Number of elements in argv[]

No `.length` for arrays

Like Java, C allows passing parameters on the command line. `main()` takes TWO parameters, the first one telling how many elements the array of strings contains. Note that `argc` is always at least 1, in C the first parameter contains the name of the program (as it was typed).

Error Management in C

There are NO exceptions.

If you want to be on the safe side when programming, you should assume that users are complete morons who cannot read instructions nor type properly, and that user input can really be anything. In the same way, interaction with anything outside your program may file – files may not be here, they maybe corrupted, networks and databases may be down, remote machines may have crashed ... Brace for the worst.

Error management:
almost ALL functions return a value

```
int scanf(const char *restrict format, ...);
```

Number of items assigned

It returns the number of items assigned because you can scan several variables at once.

scanf_example.c

```
#include <stdio.h>

int main() {
    int n;

    printf("Enter an integer: ");
    scanf("%d", &n);
    printf("The square of %d is %d\n", n, n * n);
    return 0;
}
```

Not initialized

That is the naively optimistic way to write a C program

If the user mistypes something, there is no error, no crash, no exception – the program goes on and displays garbage.

```
$ ./scanf_example
Enter an integer: 3
The square of 3 is 9
$./scanf_example
Enter an integer: A
The square of 32767 is 1073676289
$
```

uh?

Initialization

NEVER assume anything else than "not initialized means rubbish", because it often does. A program that runs fine on one computer and crashes or displays rubbish on another computer isn't a good program.

Some C compilers initialize bytes to 0.

Others don't.

You never can tell.

scanf_example.c

```
#include <stdio.h>

int main() {
    int n;

    printf("Enter an integer: ");
    if (scanf("%d", &n) == 1) {
      printf("The square of %d is %d\n", n, n * n);
    } else {
      printf("An integer was expected.\n");
    }
    return 0;
}
```

You have to test that you have indeed read an integer value

Then your program behaves correctly and is far "user-friendlier".

```
$./scanf_example
Enter an integer: A
An integer was expected.
$
```

There is a way to check that everything matches expectations, but it' s more a development tool for programmers.

Brutal Error Management

```
#include <assert.h>
```

Crash program if false

```
assert(condition);
```

Useful for debugging

Very risky in production

You don't need to check the outcome of every function (nobody checks what printf() returns) but every time there is a risk a function fails, it must be checked, and an "exit route" defined if something goes wrong.

a LOT of code is devoted to ERROR CHECKING

No exceptions in C

The "exit route" usually needs thinking. Some errors are recoverable, some are not. In any case, the user must get a feeling that your program is in control.

What must we do?

ERROR CHECKING

Stop the program?

Give another chance?

Use a safe default instead?

Let's see a few important functions. On Unix systems, help is obtained with the man (short for manual) command. As some C functions bear the same name as Unix commands, you sometimes have to say "man 3" (C functions are in section 3, Unix commands in section 1)

Built-in functions (a short selection)

Documentation:

```
man function name
man 3 function name
```

As C has no classes, no objects and no methods, C functions are similar to static functions in Java, and aren't attached to any particular variable.

C function :

Like class (static) function in Java

IMPORTANT

In C (NOT in C++) functions must uniquely be identified by their NAME

Although the return type, as well as the number and type of parameters are checked by compilers to catch mistakes early, function names must be unique in C. There is no way in C to overload a function with a function with the same name and different parameters.

IMPORTANT

In C (NOT in C++) functions must uniquely be identified by their NAME

There is a mechanism for defining functions that take a variable number of parameters

Variable numbers of parameters of undetermined types are allowed (think of `printf()`). Not used very often, but sometimes useful.

IMPORTANT

In C (NOT in C++) functions must uniquely be identified by their NAME

Functions cannot be nested.

You cannot in C create a local function inside another function. All functions live at the same level.

IMPORTANT

In C (NOT in C++) functions must uniquely be identified by their NAME

Functions cannot be nested.

Functions must be declared before being used.

This is linked to how the C compiler processes a .c file. It does a single pass over it, reading code from beginning to end.

A compiler tries to catch programming errors early.

The goal of the compiler is to warn you of potential bugs even before you first run your program. An important task in bug-chasing is to ensure that you are passing to a function the parameter it expects, and assigning its result to a variable of a suitable type. If you use the function BEFORE it is defined, the compiler (which contrarily to `javac` doesn't first survey the whole program) will ignore parameters and assume that the function returns an `int`.

```
type func1(...) {
}
type func2(...) {
}
type func3(...) {
    func1();
}
...
type funcn(...) {
}
```

You have two options:
1) You define every function before you call it, so that the compiler knows it when you use it. Functions that are last called will come first, and `main()` will be the last function in the file (if it contains a `main()` function).

```
int main(int argc, char *argv[]) {
}
```

2) You put at the beginning of the file function prototypes that are just return type, name and parameters (no function body), so that the compiler knows how the functions should be used. Everything else (including actual function definition) can follow in any order.

Or (better) use prototypes.

```
double my_func(double x, int n);
double my_func(double x, int n) {
  ...
}
```

```
#include
#include "func.h"
#define
```

func.h

Function prototypes

```
code of functions
main()
```

order doesn't matter

Usually you put all the prototypes of functions that can be called externally into your own header file, doing what C does with its own built-in functions.

```
#include <filename.h>
```

Looks for filename.h at "well known places" (eg /usr/include on Linux, C:\Program Files\MSXML x.x\inc on Windows)

```
#include "filename.h"
```

Looks for filename.h in the current directory

You must have noticed I have used double quotes instead of angle brackets. Angle brackets are used for "system" header files, which the compiler knows where to find. Double quotes are used for your own files, and unless you specify special flags for compiling they are looked for in the current directory.

Some functions (like the mathematical functions) require both to include a special header file and to link with a special library. Functions that you use most are in the C standard library and only need a header file.

Two categories:

Functions that require header file AND external library

Math functions

Functions that only require a header file

Input/Output functions

C Standard Library

First of all let's define the limits of this presentation.

1. I'll only cover an important subset.
2. Other important functions will be seen later.

C functions are far less numerous than Java methods ...

Functions return a value

Sometimes it's the result you want. ⇨ Math functions

Sometimes it indicates success/failure and you may choose not to check it. ⇨ input/output functions

With printf() if something goes wrong it will be easy to spot ... No need to code something special.

```
int printf(const char * restrict format, ...);
```

Returns the number of characters printed or a negative value if an error occurs.

In practice nobody really cares and the return value is usually ignored.

As in Java you always have in C three default "streams"

Input/Output functions

Three default "streams"

stdin 

stdout 

stderr 

BUT

Stream redirection

```
$ my_program < input_file

$ my_program > output_file
```

You can substitute a text file to either the standard input (reading from the file instead of the keyboard) or the standard output (wtiting to the file instead of the screen)

`stderr` ?

BUT

Unix (Linux) "pipe"

The "pipe" sign is the vertical bar.

```
$ my_program1 | my_program2
          stdout ⇨ stdin
   stderr          stderr
```

You can also turn the output of one program into the input of another program. This is very useful when organizing workflows.

Interestingly, most "char" functions actually work with "int" (4-byte) variables that are truncated when assigned to a "char" (1-byte) variable.

Input/Output functions

character input/output

operates on int variables

```
c = getchar()
c = fgetc(stdin)
```

EOF

```
putchar(c)
fputc(c, stdout)
```

For reading lines of text, use `fgets()` that loads data into an array the maximum size of which is provided. `fgets()` also loads end-of-line characters. DON'T USE `gets()`. If you read more than what the array can store, you'll corrupt memory.

Input/Output functions

line input/output

```
char *fgets(char *str, int size, stdin)
```

\n also read

\0 appended

NULL

```
gets(str)
```

DON'T use

Functions for writing unformatted output are quite simple.

Input/Output functions

line input/output

```
int fputs(char *str, stream)
int puts(str)
```

stdout

appends \n to the output

> 0 if OK

EOF if error

Reminders

```
scanf(format, &var)
```

Formatted output

```
printf(format, ...)
```

stdout

```
fprintf(stream, format, ...)
```

The `scanf()` format tells how to parse text, and the `(f)printf()` format how to render program variables.

Classification of characters

```
#include <ctype.h>
```

Validate data

Analyze text

ctype.h is a header file that you'll include often. It contains functions (and macros) to test characters. This is something you often do, for (for instance) trimming spaces from user input.

```
#include <ctype.h>

int issomething(int c)
```

Note

Returns zero if the character isn't a "something"

Returns something different from zero if it is a "something"

```
if (isdigit(c)){
   // c is one of '0', '1', '2', '3', '4',
   //             '5', '6', '7', '8', '9'
} else {
   // Not a digit
}
```

```
#include <ctype.h>
```

Doesn't work with Chinese characters ...

```
isalnum()  { isdigit()
           { isalpha()  { islower()
                        { isupper()
isspace()
ispunct()
isprint()
```

Just the main ones ...

alnum = alphabetical or numeric
punct = punctuation
print = printable

```
#include <ctype.h>

int tolower(int c);

int toupper(int c);
```

There isn't in C any function to change the case of a full string (it's very easy to write). The functions that are provided only operate against a single character.

String manipulation

```
#include <string.h>

int strlen(char *string);
```

Starts from the pointer passed and moves on, counting chars until one encounters \0.

C string management is rather primitive.

Basic copy/concat functions are notoriously unsafe.

```
#include <string.h>

char *strcpy (char *dest, char *src)
```

Starts at `dest` and copies what starts at `src` until `\0` is met.

```
char *strcat(char *dest, const char *src)
```

Starts at `dest` and moves on until one encounters `\0`, then copies what starts at `src` until `\0` is met.

NO BOUNDARY CHECKING


```
#include <string.h>

char *strcpy (char *dest, char *src)
char *strncpy(char *string1, char *string2, int n)

char *strcat(char *dest, const char *src)
char *strncat(char *dest, const char *src, int n)
```

Limits to n characters at most

There are safe versions that you should use 99% of the time (at least).

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

Compare only the n first characters. Not the same meaning as in `strncpy()`

```
int strcmp(char *string1, char *string2);

int strncmp(char *string1, char *string2, int n);
```

Returns zero if equal

< 0 if string1 comes alphabetically before string2

> 0 if the reverse is true

Both `strcmp()` and `strncmp()` are safe if strings are terminated with `\0`.

```
#include <string.h>

int strcmp(char *string1, char *string2);

int strncmp(char *string1, char *string2, int n);
```

VARIANTS

```
int strcasecmp(char *string1, char *string2);

int strncasecmp(char *string1, char *string2, int n);
```

Ignore case

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

Search strings

```
char *strchr(char *string, int c);
```

NULL if nothing found (special pointer value)

```
char str[10]
```

s	u	s	T	c	\0				

strchr(str, 'T') Looking for T

str = address of first letter (first element)

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

Search strings

```
char *strchr(char *string, int c);

char *strrchr(char *string, int c);

char *strstr(char *string, char  *substring);
```

Searches from the end

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

An interesting but weird function:

```
char *strtok(char *string, char *separators);
```

First call

blahblahblah		blahblah		blah		blahblah

There is in C a weird function that allows you to "tokenize" a string (it means retrieving pieces separated by special characters). The first time you call it with the string.

#include <string.h>

An interesting but weird function:

```
char *strtok(char *string, char *separators);
```

Call with "NULL" after

blahblahblah	blahblah	blah	blahblah

You call then again and again with NULL until it returns NULL. For reasons beyond what you have seen so far this function has some issues; you can use it in lab projects for now but a more complicated function called `strsep()` is recommended instead.

What about Chinese?

Java chars can store Chinese characters; C chars, which are only one byte, cannot encode more than 256 different values, which isn't enough for Chinese.
Nevertheless, you can perfectly input, process and output Chinese characters in a C program.

#include <wchar.h> ← Added in 1995

wchar_t is a type that extends char and handles multi-byte characters.

wide char

= character stored on several bytes

wchar_t

```
fwprintf()
fgetws()
wcslen()
wcscpy()
```

+ conversion

Prefixing (outside quotes) a string or character by L warns the compiler that we are dealing with wide chars.

What do they look like?

wide char

```
L"Wide-char string"
L'\x3b1'   α
L'南'
```

As multiple encodings exist, you should first in your program call the `setlocale()` function to specify (among other things) the encoding that is used.

How are they encoded?

wide char

Territory

```
setlocale(LC_ALL, "zh_CN.UTF-8");
```

`L'\x3b1'`

Language

Character encoding

"Language" is for error messages when a translation exists, "Territory" deals with details such as which day of the week is the first one.

UNICODE

4-byte (provision for 6) codepoint

Draft proposal August 1988

Dealing with all the characters, current and past, that are used in the world is the goal of Unicode, which associates to every letter, sign or character a unique "codepoint".

UNICODE

One thing that is important (and a bit difficult to understand) is that the "codepoint" is not the ultimate encoding. Computers uses a lot of storage. If we were using 4 (or 6) bytes to store Latin characters that fit on one, volumes would be multiplied by as much in Western environments. In the same way, Chinese characters can fit on two bytes. So, depending on which characters you use most, you use an encoding that minimizes the number of bytes you use. As long as there is a formula to transform to and from codepoints, your encoding is compatible with Unicode.

UNICODE

http://unicode.org/

http://www.unicode.org/Public/UNIDATA/Blocks.txt

Blocks Ranges of values

CJK

Unicode codepoints are organized in "blocks". Chinese characters are found in blocks called CJK (for Chinese, Japanese and Korean). If you know which blocks you are using, you can encode codepoints on fewer bytes.

ISO8859 (same story with Windows character sets)

1 byte

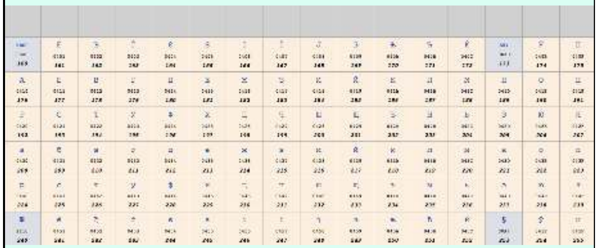
Before Unicode, the International Standards Organization (ISO) defined several one-byte encodings with only one half of values common to all encodings.

0->127 same as ASCII on 7 bits

128->255 "localized"

negative if signed char

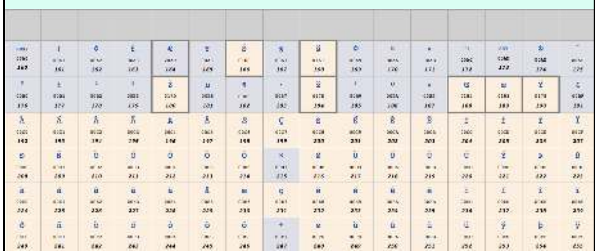
ISO8859 (same story with Windows character sets)



ISO8859-5

For people using the Cyrillic (Russian) alphabet.

ISO8859 (same story with Windows character sets)



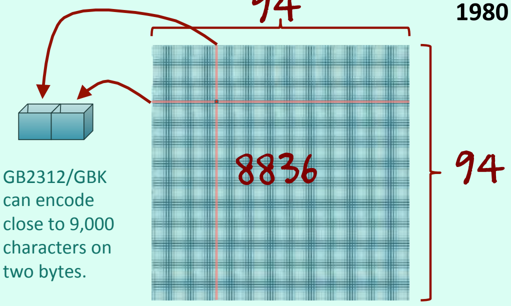
ISO8859-15

European countries.

And for Chinese?

The Chinese had to devise their own system, as one byte cannot encode all common characters. In fact, several systems were needed for simplified and traditional Chinese.

GB2312 Guojia Biaozhun 国家标准 **1980**



GB2312/GBK can encode close to 9,000 characters on two bytes.

Succeeded by GBK

These systems reach their limits when you want to mix in ONE text characters from different cultures (you can say "from here I'm using something different" but it's messy). This is why Unicode defined a universal system.

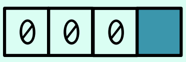
Hard to mix anything with anything

Dictionary? Side-by-side translation?

How can we represent Unicode?

Dumb approach 1 byte → 4 bytes

UTF-32 Encoding

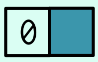


How can we represent Unicode?

Most characters 2 bytes

4 bytes for weird cases

UTF-16



How can we represent Unicode?

BIG PROBLEM:

Not having to reencode everything that preexisted (program text!)

In UTF-8 (most used on the web) 1 to four characters can be used. The first bit(s) of the first byte tells how many bytes compose the character. Continuation bytes always have the left-most bit set to 1.

UTF-8

- 1 byte for basic Latin
- 2 bytes for Europe, Middle East
- 3 bytes for Asia
- 4 bytes for weird languages

It's not as space-efficient as specialized encoding, but it is compatible with most of what pre-existed, the source code of computer programs in particular.

UTF-8

As soon as the leftmost bit is 1, you are in a multi-byte character. You can recognize a bit-pattern using a "bit-mask" and bit operations (such as &). A little wild but not difficult ...

0							

1	1	0					
1	0						

1	1	1	0				
1	0						
1	0						

1	1	1	1	0			
1	0						
1	0						
1	0						