Crash course in C++

beOI Training

Table of contents

Hello world!

```cpp
// This is a comment
#include <bits/stdc++.h> // imports STL library
using namespace std; // avoids writing "std::"

// Function declaration: type name() { [...] }
int main() {
    // All instructions end with ";"
    cout << "Hello World!" << endl;
    // endl = new line
}
```

- The function **int** main() is called when starting the program.
- **int** means that main() returns an integer, but main() automatically returns 0 (historical reasons).
- The direction of << indicates that the words are sent to cout, the "standard output".

Compiling and runing

C++ programs first have to be "translated" into a language that the computer can understand.

Compiling (through the command line)

```
g++ -std=c++11 -Wall -Wextra -Wshadow hello.cpp
```

- g++: compiler name
- -std=c++11: used C++ version
- -Wall -Wextra -Wshadow: activates many useful warnings that help identify bugs
- hello.cpp: source code

Running: ./a.out

- ./: current folder
- a.out: name of the executable generated by g++.

Variables and operations

We must specify a type for every variable

```
int i = 5; // integers in range [-2^31, 2^31[
double j = 5.4; // number with commas
bool b = true; // boolean (true or false)
char ch = 'D'; // single character
string s = "abcd"; // multiple characters

// We can also initialize the variables later
int k,l; // declare multiple variables
         // of the same type

k = i + 2;
l = 7 / 3; // integer division => l = 2
s += ch; // appending a character
j /= 3; // dividing j by 3
i++; l--; // adding 1, subtracting 1
```

Conditionals

```
int age;
cout << "How old are you? ";
cin >> age; // reading input
if (age < 18)
    cout << "You are underage." << endl;
else if (age <= 120)
    cout << "You are an adult." << endl;
else {
    int a=3, b=4, c=5;
    bool rectangle = (a*a + b*b == c*c);
    if (rectangle && !(a == 0 || b == 0))
        cout << "Hypotenuse = " << c << endl;
}
```

- Curly braces {} are optional if writing a single line in the if/else/loop
- The direction of >> indicates that the integer comes from cin, the "standard input".

Loops

```
// Print numbers from 1 to 5
for (int i=1; i<=5; i++)
    cout << i << endl;

string s; // initially empty
while (s != "yes") { // != means "not equal"
    cout << "Do you like programming? ";
    cin >> s;
}
```

- **for** (;;) loops have 3 parts:
  - Initialisation: initialise one or more variables
  - Condition: the loops stops when the condition becomes false
  - Incrementation: executed at the end of each interation

Functions

```
// Must specify return and parameters type
int square(int x) {
    return x*x;
}
void sayHello(string s) { // void = return nothing
    cout << "Hello " << s << endl;
}
int main() {
    int y = square(4); // y = 16
    sayHello("Victor");
}
```

- Can't be nested and always placed *before* their call. Otherwise, you can declare them like this: **void** sayHello(string s); and implement them later on.
- When a function is not **void**, all executions have to end with a **return**.

Arrays

All elements of an array must have the same type.

```
int maxi(int tab[], int n) { // [] is alywas after
    int ma = 0;                  // the name
    for (int i=0; i<n; i++)
        ma = max(ma, tab[i]);
    return ma; // returns the maximum from tab
}
int main() {
    int a[5], b[4][3]; // 4 rows and 3 columns
    for (int i=0; i<5; i++)
        cin >> a[i];
    cout << maxi(a, 5) << endl;
}
```

- The first element has index [0].
- The size can't be modified.
- An array doesn't know its size! We have to pass it separately when calling a function.

Table of contents

STL and containers

The standard library (STL) contains many useful structures and functions.

A very useful structure is the vector<>:

```
vector<int> v(3,-1); // 3 elements set to -1
v.push_back(7); // adding 7 to the vector
v[1] = 5; // access it like an array
for (int i : v) // iterate over elements
    cout << i << endl; // prints -1, 5, -1, 7
```

- We specify the element type between "<>": <int>.
- Structures have constructors that initialise them (here (3,−1)) an many methods (here .push_back()).
- More info about containers: `http://en.cppreference.com/w/cpp/container`

STL algorithms

STL also has many algorithms ready to use:

```
vector<int> v{4,-1,3,2}; // initialise a vector
sort(v.begin(), v.end()); // sort a vector
swap(v[0], v[1]); // swap the contents
reverse(v.begin(), v.end()); // reverse the vector

for (int i : v)
    cout << v << endl; // print 4, 3, -1, 2
```

- And many others: copy, binary search, matching, selecting the i-the smallest element, ...
- More info about STL algorithms: `http://en.cppreference.com/w/cpp/algorithm`

Table of contents

Disclaimer

Beware: it gets hard from now on.

But nothing is too hard for you :)

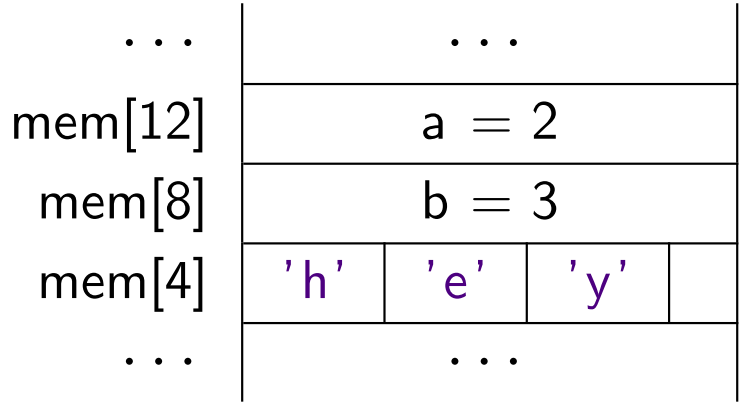
Memory and addresses

The memory of a PC is like a big array, filled with compartments for memorising the variables.

Program

```
...
int a = 2, b = 3;
char ch[]{'h','e','y'};
...
```

RAM Memory



- Each compartment can contain a byte (8 bits), and a variable can spread over a few compartments (**int** 4, **double** 8)
- Every compartment has an *address* (4, 8, 12, ...).
- The address lets us access the variable

Pointers

- A pointer is a variable that contains the address of another variable.
- We can read or modify a variable using its pointer.

Programme

```
int a = 2;
int *b = &a; // type = (int *)
cout << b << endl;  // 12
cout << *b << endl; // 2
*b = 5; // modifies a, pas b
cout << a << endl;  // 5
```

RAM Memory

...	...
mem[12]	a = 2
mem[8]	b = 12
...	...

- &a = "index of a in mem[]"
- *b = mem[b] (we can write there)

Passing a pointer

- When we pass a variable to a function, it is copied (passing *a value*).
- *If we modify it, it doesn't change the original variable.*

Passing a value

```
void add3(int a) {
    a += 3;
}   // a = copy of i
int main() {
    int i=2;
    add3(i);
    cout << i << endl;
}   // prints 2
```

Passing a pointer

```
void add3(int *p) {
    *p += 3;
}   // "mem[p] += 3"
int main() {
    int i=2;
    add3(&i);
    cout << i << endl;
}   // prints 5
```

On the right side, the address `p` itself is passed by value, *but we modify the value of* `i` *directly from memory using* `*p`.

References

References let you disguise a pointer as a normal variable (you use it without having to write & and $*$).

```
int a = 2;
int &b = a; // type = (int &)
cout << b << endl;  // 2
b = 5; // modifies a through b
cout << a << ' ' << b << endl;  // 5 5
```

Differences with pointers:

- A reference can only be linked to a single variable (and can't be changed afterwards).
- `b` is just like another name for `a`, it's an alias.
- We can't access the address of `a`.
- The & used to declare references **doesn't have anything to do** with the & of pointers.

Passing a reference

When we pass a variable as *a reference*, it is not copied, it is the "same variable".

Passing a value

```
void add3(int a) {
    a += 3;
}   // a = copy of i
int main() {
    int i=2;
    add3(i);
    cout << i << endl;
}   // prints 2
```

Passing a reference

```
void add3(int &a) {
    a += 3;
}   // a = alias of i
int main() {
    int i=2;
    add3(i);
    cout << i << endl;
}   // prints 5
```

- The only syntax difference is the & in front of the variable name.
- On the right side, since i and a are 2 names for one same variable, when we modify a, i also gets modified.
- We avoid copying the variable $\Rightarrow$ much faster.