Advanced Programming

alberto.sartori@sissa.it

Compiler

- g++
- clang++
- c++

```
c++ -c filename.cc
```

Compile only up to the object file filename.o. Use `-s` if you want to generate only the assembler file.

```
c++ hello.o
```

Generate the executable.

The main is the only function that automatically has a return statement also if not explicitly specified.

The value returned by `main()` function is stored in a variable called `?`

```
echo $?
```

will show the value of the variable obtained from the main.

Hello World

```
std::cout << "Hello, World!" << std::endl;
```

```
#include <iostream>
```

At preprocessor time include (copypasting it) a header file to the code.

Built-in types

From C++11 you can initialize variables like:

```
int a{-1};
int a=-1;
```

The first method is recommended.

An **object** is a set of bytes, portion of memory. The binary values define its value that is interpreted by the code according to one type.

A **variable** is a named object.

```
unsigned int a{1};
```

If you want to store positive integers only.

```
int16_t a{};
```

If you want to specify the size of the integer variable.

```
std::string s{"Hello, World! \n"};
```

Is not a built-in type but very useful to create strings, you don't have to specify the type of the data that you're putting into the string.

```
const int cc{7};
constexpr double ce{cc * 8.1234 / M_PI}; // evaluated at compile-time
```

Double check that you're not modifying the value of the variable during the code. If you try to modify it you will create an error in the compiler. With expressions they're compiled at compiler time (from C++11). All the values in the expression should be also evaluated at compiler time.

Loops

- For
- While
- Do While (check the condition for the first time at the end of the first loop)

```
break;      //exit from the loop, skip other commands
continue;   //directly to the next iteration
```

Are keywords to stay or exit from a loop.

Watch out

- Pay attention to size of types, e.g, int occupies 32 bits, you can produce **overflow** errors if you're considering numbers that cannot be represented by the amount of memory of the specific type.
- Don't use `==` or `!=` to compare floating point numbers because of roundoff errors.

Pointers

```
double d = 9.9;
double &ref = d;  //alias
double *ptr = &d;
```

After `&ref` definition `ref` can be used instead of d, it's his **alias**.
Instead `*ptr` is a **pointer** to `d` adress in memory. It's value is directly the adress of `d`.

If I want to change the value of `ref` using an alias:

```
ref = 7.7;
```

Instead if I want to use the pointer (you modify immediately whatever you have to the pointed adress):

```
*ptr = 5.5;
```

Functions

```
ret_type function_name(int a, double b){
    function body;
    return //same ret_type variable
}
```

If no value has to be returned you can ignore the `return` statement and use `void` as `ret_type`.

```
auto vname = 6;
auto f6 = factorial(6);
```

Since C++11 you can define an `auto` variable, which type is defined by the compiler. In the example the type is the one defined by the auto `factorial()` function.

Since C++14 also `ret_type` for functions can be defined as `auto`.

The definition of a function can be divided into:

- **Declaration**: it has to be before the `main()`.

  ```
  int dwim(const int, const int);
  ```

- **Definition**

  ```
  int dwim(const int a, const int b) {
    std::cout << "int ------ ";
    return a + b;
  }
  ```

  In C++ (not C) we can do **function overloading**, calling functions with the same name, but with different variable types. The compiler will understand automatically which function should call.

Template

```
template <typename T>
T dwim(const T a, const T b);

[...]

template <typename T>
T dwim(const T a, const T b) {
  return a + b;
}
```

The template definition survive until the end of the statement. I can use the function so defined with all the types with which the function makes sense. The compiler generates the symbol for the function only for the types used in the code.

If I want to specify manually which type should be used:

```
std::cout << dwim<int>(a, b) << std::endl;
```

Variable by value or by reference

```
void swap(int a, int b)      // by value (in stack)
void swap(int *a, int *b)    // by pointer C style
void swap(int &a, int &b)    // by reference (in heap) C++ style
```

Variables in the **stack** are automatically deleted when they're out of scope (outside the brackets). Instead variables in the **heap**. Variables called as argument of functions are usually called in the stack and so changes that are happening inside the function are not considered outside the function itself.

Example: Swap function

```
void buggy_swap(int a, int b) {
  int tmp{b};
  b = a;
  a = tmp;
}

void c_swap(int* a, int* b) {
  int tmp{*b};
  *b = *a;
  *a = tmp;
}

void cxx_swap(int& a, int& b) {
  int tmp{b};
  b = a;
  a = tmp;
}

template <typename T>
void templ_swap(T& a, T& b) {
  T tmp{b};
  b = a;
  a = tmp;
}
```

By reference I don't need to use the uncomfortable pointers syntax as in C.

Arrays

Contiguous portion of memory, where each cell has the same type of the others.

```
int a[4];   // integer array of length 4
a[2] = 9;   // set 3rd element to 9
```

In C and C++ the index starts from 0.

```
for (std::size_t k{0}; k < 4; ++k)  //std::size_t type
    std::cout << "ai[" << k << "] = " << ai[k] << '\n';
```

We use `std::size_t` to avoid overflow or similar errors. It helps to document what you're doing. In this case `std::size_t` is an alias for `long long unsigned int`.

The syntax to define elements of an array is:

```
double ad[7]{1.1, 2.2, 3.3};  // the rest is set to 0
```

An array is implicitly converted to a pointer to its first element:

```
*ad = 9.9;  // set the first element to 9.9
```

Arrays are passed to function trough the pointer method so that their values can be modified by functions.

```
void surprise(double a[], const std::size_t n)
```

Dynamic arrays

Usually the array size must be known when you define the array.

```
int* da{new int[n]};  // allocated on the HEAP, or free-store
```

It allocates an array of this size without the need to know the length at compiler time.

```
delete[] da;
```

The portion of memory should be deleted manually at the end of the scope. Another good motivation of using heap is that usally the stack is a very small part of the memory and so it's easy to incur in stack overflow.