TP C++ - 1

Thierry Géraud, Edwin Carlinet

Contents

1 Language introduction

C++ was designed ten years after the birth of *C* by Bjarne Stroustrup in the 80s when he worked at Bell Labs. Stroustrup wanted a language both *fast* and *high-level* enough for efficient, large software development. Thats why he chose *C*, and he added high-level features inspired from *Simula* (1967), the first object-oriented programming language, to it.

The current standard is *C++14*, but we will only use features from the *C++11*. It includes many lovely features: a powerful object model, template programming, the very useful *C++* Standard Library, etc. This is roughly the outline of this workshop.

So, what are the best reasons to prefer *C++* over *C*?

- Safer code with exception handling and a better control on data
- Speed up thanks to more static evaluations
- Easier translation of abstract concepts in code
- Generic programming

- Ready-to-use algorithms and data structures

Nevertheless, you must be warned about the complexity of *C++*. The core language is much more complicated than the one of *C*. We advise you to **test** every behavior that would seem strange to you and not to hesitate to ask your questions.

2 General points

2.1 Files

Classes are to be declared into *.hh* files (*headers*) and the implementation of their methods must be written into *.cc* files.

2.2 Compiling

We will use GCC.

```
g++ -Wall -Wextra -Werror -pedantic -std=c++11 -o test test.cc
```

3 Input/Output

3.1 Standard streams

Operators `<<` and `>>` are used to read and write on *streams*. The `<<` operator, also called *insertion operator*, is used to write data on a stream, whereas the `>>` operator, or *extraction operator*, is used to read data from an input stream. These two operators both return the stream they worked on. It is then possible to chain input/output operations successively onto a single stream.

There are four standard I/O instances defined into the standard library:

- `cin`: standard input stream of the program
- `cout`: standard output stream of the program
- `cerr`: error output stream, used for error messages
- `clog`: logging output stream; write on `cerr` with bufferisation.

```cpp
#include <iostream>

int main()
{
    int i = 0;
    // Read an integer:
    std::cin >> i;
    // Display this integer and the following one:
    std::cout << i << " " << i + 1 << std::endl;
}
```

3.2 File streams

File streams are defined in the header `fstream`. There are two types of streams:

- `ifstream` (input file stream) to open a file for reading,
- `ofstream` (output file stream) to open a file for writing.

Constructors of those types take a filename as argument. Yet, `<<` and `>>` are used to read or write from such streams. When reading, `<<` stores the next token (word) in a string.

To test if a stream is still valid (e.g. we have not reached the end of file), the stream object itself can be used as a boolean expression in a test.

```
// Exemple: copying a file word by word replacing
// spaces and "\n" by commas

#include <fstream>
#include <string>


int main()
{
  std::ifstream in("input.txt");
  std::ofstream out("output.txt");
  std::string word;

  while (in >> word)
    out << word << ",";

  in.close();
  out.close();
}
```

See `http://en.cppreference.com/w/cpp/io/basic_ifstream` and `http://en.cppreference.com/w/cpp/io/basic_ofstream` for more file stream manipulations.

3.3 Exercises

1. Write the program `wordcount` that counts the number of words in a file and outputs: `File` *`filename`* `has ## words.`

   ```
   $ echo "word1 word2" > input.txt
   $ ./wordcount input.txt
   File input.txt has 2 words.
   ```

2. Write the program `rot13` that reads the standard input and outputs the text where each letter of each word has been rotated by 13 in the alphabet. Each word outputted will be followed by a new line.

   ```
   $ echo "abcd word" | ./rot13
   nopq
   jbeq
   ```

4 References

References are a new way to pass arguments to a function. Indeed, when passing arguments to a function, it is possible to pass them by:

Copy The value is copied in the function, which only manipulates a clone of the original argument. Passing a plain object is equivalent to passing plain structure.

Address The variable's address is passed as argument to the function, which directly uses the value of the original variable.

Reference A reference to the object is passed to the function. A reference can be manipulated as a plain object but has the properties of a pointer. This means that if the reference is modified inside the function body, the modification also applies to the original object.

```
void foo(int  x, // by copy
         int* y, // by pointer
         int& z) // by reference
{
  x = 1;
  *y = 2;
  z = 3;
}


int main()
{
  int a = 0, b = 0, c = 0;
  foo(a, &b, c);
  std::cout << a << " " << b << " " << c << "\n" // a = 0, b = 2, c = 3;
}
```

4.1 Const reference vs copies

Even if the function is not subject to modify the value passed as argument, it may more convenient to use references instead of passing by value (e.g. to avoid the copy of a large object). Then, use the *const* qualifier to specify that the value will not be modified.

```
void foo_byval(std::string x) // the string is locally copied
{
}

void foo_byref(const std::string& x) // the string is not copied and guaranteed
                                     // to not be modified
{
}

void foo_byref2(std::string& x) // the string is not copied but can be modified
{
}

int main()
{
  std::string s = "my_string";

  foo_byval(s); // OK
  foo_byref(s); // OK
  foo_byref2(s); // OK

  foo_byval("temporary string"); // OK std::string(const char*) constructor is called
  foo_byref("temporary string"); // STILL OK std::string(const char*) constructor is called
  foo_byref2("temporary string"); // FAIL a temporary cannot be bound to a non-const reference
}
```

4.2 Exercices

1. Write the function `swap` that exchanges the values of two integers.
2. Write an equivalent function using C++ strings and references.

```
void
foo(const char* x, const char* y, char** out)
{
  int a = strlen(x);
  int b = strlen(y);
  *out = malloc((a + b + 1) * sizeof(char));
  int i = 0;
  while (*x != '\0')
    (*out)[i++] = *x++;
  while (*y != '\0')
    (*out)[i++] = *y++;
  *out[i] = '\0';
}
```

5 OOP

5.1 Object-oriented programming

Object-oriented programming (OOP, usually called *object programming*) is a way to structure your application by grouping data and data processing within a single entity, the *object*. In object-oriented programming, systems are entirely constituted of entities (the objects) that possess characteristics defined by their type.

A class is kind of a model declaring common properties for a set of objects (its *instances*). It is a user-defined type which specifies:

- the type of the class members
- a set of functions (called *methods*) handling the object members
- special methods called accessors and mutators (*getters* and *setters*)

5.2 Class structure

A class is structured as follows:

```
class ClassName
{
public:
    ClassName(); // Constructor
    ~ClassName(); // Destructor

    // Methods

private:
    // Attributes
};  // <---   [!] don't forget the semicolon [!]
```

A class contains *methods* and *attributes*. A *method* is a function linked to a class and used to access the object's data, or to modify the behavior of the object by modifying its attributes.

Obviously, classes can have attributes which are themselves objects.

5.3 Method definition

There are *two* ways to define a method. Inside the class body:

```
// Header file (.hh)
class ClassName
{
public:
    int myMethod()
    {
        return 0;
    }
};
```

Outside the class body

```
// Header File (.hh)
class MyClass
{
public:
    int myMethod();
};

// Implementation file (.cc)
int MyClass::myMethod()
{
    return 0;
}
```

5.4 Visibility

It is possible to prevent access to attributes or methods to any function outside the class. This is called the class *visibility*. To enable it, use one of the following keywords:

public Free access: methods and attributes declared as *public* in the class are directly accessible from the outside of the class. The set composed of public attributes and methods is called the class **interface**.

private Access is restricted to internal methods: *private* members can only be accessed and modified by methods of the class they belong to. This is also used to hide attributes and methods to the outside of the class. For example, if *obj1* and *obj2* are instances of the class *A* and *obj3* an instance of the class *B*, *obj1* can access its own private members and those of *obj2*, but not those of *obj3*.

5.5 Accessor / Mutator (getter/setter)

At each method call, the compiler implicitly passes a pointer to the object as argument. This argument is the first one in the argument list, even if it is invisible to the programmer. Still, it can be accessed within the body of the method. Its name is *this*. *this* is a constant pointer: it cannot be changed. It is used to access attributes of the class but can be omitted if there is no ambiguity.

```
class ThisIsEasy
{
public:
    int getAttr()
```

```
    {
        return this->attr;
    }

    void setAttr(int x)
    {
        // Some sanity check about x e.g. assert(x > 0)
        this->attr = x;
    }

private:
    int attr_;
};
```

5.6 Constant object

In C, the *const* keyword is used to *constify* a variables content. The compiler is then able to reject further attempts to modify this content and can produce some optimization.

The C++ extends this concept to the class: it is possible, in C++, to define constant objects. Usually, the compiler can easily identify forbidden operations when working on variables. However, when manipulating objects, things are not as easy, since operations are usually performed by methods. This means that the user has to point out, amongst all methods, which ones are allowed to work on constant objects. We will specify it by using the 'const' keyword in method declarations.

In the previous example, the attribute accessor should have been *const* as it does not modify any attribute (neither return it by non-const reference).

```
class ThisIsEasy
{
public:
    int getAttr() const
    {
        return this->attr;
    }

    void setAttr(int x)
    {
        // Some sanity check about x e.g. assert(x > 0)
        this->attr = x;
    }

private:
    int attr_;
};
```

5.7 Object constructor

A constructor is used to initialize the attributes of an object.

The constructor of the *Class_name* class is the method that will be called at the creation of an object of type *Class_name*. A constructor is defined like any other method, except that it has neither return value nor return type.

As youve seen before for functions and methods, you can also *overload* constructors. Its useful when you want to perform specific initializations, depending on the informations you have when you instantiate the class.

This example shows you how to overload a constructor:

```cpp
#include <string>

class Person
{
public:
    Person(const std::string& name, const std::string& date_of_birth)
        : name_(name) // initialization of the name\_
    {
        // generate age from date\_of\_birth
        // age\_ = ...;
    }

    Person(const std::string& name, unsigned int age)
        : name_(name)
        , age_(age)
    {}

private:
    std::string name_;
    unsigned int age_;
};
```

Then we could use these two constructors like this:

```cpp
Person p1("Toto", "21/02/1942"); // use the first constructor
Person p2("Titi", 42); // use the second one
```

5.8 Exercices

Design a class *Stack* using a single linked list implementation.

Suppose the following typedef:

```cpp
typedef int stack_content_t;

struct node_type {
  stack_content_t val;
  node_type*      next;
};
```

The class will provide the following functionality:

- `Stack()` Default construction leads to an empty stack.
- `void push(const stack_content_t& x)` push an element on top.
- `pop()` remove the top of the stack
- `const stack_content_t& top()` returns a reference to the top of the constant stack.
- `stack_content_t& top()` returns a reference to the top of the mutable stack.
- `void flush()` remove all the elements
- `unsigned size()` return the size of the stack (**in constant time**).
- `bool empty()` return true if the stack is empty.

The stack should also be streamable, i.e., supports a display on a stream. Finally, add every pre/post conditions to your methods (e.g. we should ensure that the stack is not empty before accessing the top element).

```
int main()
{
  Stack s;
  s.push(10);
  s.push(20);
  std::cout << s << std::endl; // affiche [ 20 10 ]
  s.pop();
  std::cout << s << std::endl; // affiche [ 10 ]
  s.flush(); // release memory
}
```