The C++ programming language

— T.03 —

Summary:

- My first C++ program
- Overview of the C++ programming language
- Some differences between C and C++
- Classes
- Templates
- Exceptions
- Other stuff (not explained in this course)
- Exercises

[Remark: C++ is a very large and complex programming language (some say* that it is far to much complex; see the right-hand side image). For AED we will only need a relatively small subset of what it has to offer. The rest, although important, will not even be mentioned in these slides.]

* "When its 3 A.M., and you've been debugging for 12 hours, and you encounter a virtual static friend protected volatile templated function pointer, you want to go into hibernation and awake as a werewolf and then find the people who wrote the C++ standard and bring ruin to the things that they love." (Excerpt from The Night Watch, by James Mickens).

Recommended bibliography for this lecture:

- **Thinking in C++. Volume One: Introduction to Standard C++**, Bruce Eckel, second edition, Prentice Hall, 2000.
- **Thinking in C++. Volume Two: Practical Programming**, Bruce Eckel and Chuck Allison, Prentice Hall, 2003.
- **Online reference documentation about C++**
- **C++ Annotations**, Frank B. Brokken, 2015.
- **C++ Primer**, Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, fifth edition, Addison-Wesley, 2013.

My first C++ program

The "hello world" program:

```
1 // Hello world program
2
3 #include <iostream>
4
5 int main(void)
6 {
7   std::cout << "Hello world!\n";
8   return 0;
9 }
```

The line numbers on the left are not part of the code.

Explanation:

- Line 1 is a comment. The comment starts with `//` and ends at the end of the line; `/* ... */` comments are also allowed in C++.
- Line 3 instructs the compiler to replace that line by the contents of the file named `iostream` (one of the files of the C++ compiler's standard libraries); like the C compiler, the C++ compiler also uses a preprocessor.
- Lines 5 to 9 declare and define a function named `main`, which is the entry point of the program (the entry point of the program is always called `main`) and, in this case, takes no arguments and returns an integer.
- Lines 6 to 9 constitute the body of the function.
- Line 6 starts a block of code; code blocks start with a `{`.
- Line 7 sends the string `"Hello world!\n"` to the `cout` stream specified in the `std` namespace (the standard output stream).
- Line 8 forces a return from the main function with a return value of 0 (since `main` is the entry point of the program, this actually specifies the error code of the entire program; 0 means all is well, non-zero means some error occurred).
- Line 9 ends a block of code; code blocks end with a `}`.

Overview of the C++ programming language

The C++ programming language is a large and complex programming language. It is a superset of the C language (hence the name C++). Most programs written in C are also valid C++ programs. The C++ language has the following features, which are not present in the C language (this list is not exhaustive):

- It is possible to pass values by reference to a function, without making the use of pointers explicit.
- It is possible to define functions with the same name but with different lists of arguments (function overloading).
- The last arguments of a function may have default values (great if one wants to add a new parameter to a function without modifying the code already written that calls the function).
- One can create a data type and the functions that manipulate it (a `class`) in a much more elegant way. It is even possible to make the standard arithmetic operators, such as + and −, work with arguments of the new data type. It is possible to hide the internal details of the data type, so that we can change them without affecting the parts of the program that use the data type.
- One can create so-called name spaces to better control which symbols (such as variables and function names) are visible in each part of our code.
- It is possible to create templates of functions and classes (generic programming).
- It is possible to handle exceptions in a disciplined way (in C one would have to use `goto` statements or, in some cases, the ugly `setjump` and `longjump` functions)

It is customary to use the extension `.cpp` (meaning C-plus-plus) in the names of the source files of a C++ program (`.cc`, `.cxx`, `.c++`, and even `.C`, are also sometimes used).

Some differences between C and C++ (part 1)

C++ allows arguments to be passed by reference. This is accomplished by placing an & before the argument name. In the following code, we show on the left how this has to be done in C and on the right we show how this can be done in C++ (we may also use the left version, but the right one is more elegant, as there are no explicit pointer dereferences):

```
// C; called as follows: swap(&var1,&var2);
void swap(int *x,int *y)
{
  int tmp = *x;
  *x = *y:
  *y = tmp;
}
```

```
// C++; called as follows: swap(var1,var2);
void swap(int &x,int &y)
{
  int tmp = x;
  x = y:
  y = tmp;
}
```

C++ allows functions with the same name but different argument lists to coexist. For example, the following code is valid in C++:

```
int    square(int    x) { return x * x; }
double square(double x) { return x * x; }
```

The code `square(1)` will call the first function, because its argument is an `int`, and the code `square(1.0)` will call the second function, because its argument is a `double`. It is, however, illegal to have two functions with the same name and with the same argument list (same number of arguments and same types), but with different return types. For example, the function

```
double square(int x) { return double(x) * double(x); }  // double(x) does the same as (double)x
```

cannot coexist with the first function defined above.

Some differences between C and C++ (part 2)

C++ allows the specification of default values for the last arguments of a function. This is done by providing initializations (with the default values) in the argument list of the function. It is actually better to put the initializations in the function prototype, as in the following example:

```
int f(int x,int y = 2,int z = 3); // function prototype (usually placed in a header file)

int f(int x,int y,int z) // actual definition of the function
{
  return x + 2 * y + 3 * z;
}
```

In this case we not only can call the `f` function with three arguments as usual, but we can also call it with two (the third, `z`, will get the value **3**, as specified in the function prototype), and with one (the second and third will get, respectively, the values **2** and **3**). [**Warning:** this feature of the C++ language should be used with extreme care when the function is also overloaded.]

C++ allows the definition of variables in almost any place inside a compound statement. In (old) C, that is only allowed at the beginning of a compound statement. The following code is valid in C++ (it is also valid in modern C dialects):

```
int i;                      // i defined here
for(int j = 0;j < 10;j++)   // j defined here; it will cease to exist at the end of the for cycle
{
  i = 2 * j;
  int k = i + 2 * j;        // k defined here (definition after a statement is allowed)
  cout << k;                // print the value of k
}
```

Some differences between C and C++ (part 3)

In C++ it is possible to control the visibility of symbols by placing them is a name space. In the following code we define two name spaces and put in each of them a global variable and a function (same names but different name spaces; of course this is not recommended but sometimes it cannot be avoided):

```
namespace NEW
{
  static int t_bytes;
  int f(int x) { return 2 * x; }
}
namespace OLD
{
  static int t_bytes;
  int f(int x) { return 3 * x; }
}
```

To get a specific variable or function, place the name of the name space followed by `::` before the variable or function name. For example, `NEW::t_bytes` is the `t_bytes` of the `NEW` name space. It is also possible to say

```
using NEW::t_bytes;
```

and from that point on `t_bytes` will be synonymous with `NEW::t_bytes` (or course, for this to work no symbol with the name `t_bytes` can already exist in the current name space). It is also possible to import all symbols from a name space, thus making all of them available without the `name_space::` prefix. For example,

```
using namespace OLD;
```

will make `t_bytes` a synonym of `OLD::t_bytes` and `f` a synonym of `OLD::f`.

The `std` name space is reserved for standard library functions.

Some differences between C and C++ (part 4)

It is possible to call C functions from a C++ program (the calling conventions are the same, as are the fundamental data types), so that is a question of using the proper function names. This is actually a problem that has to be solved, because the function name that the compiler uses internally is not simply the name of the function: it has also to encode the types of its arguments (this has to be done because a function may be overloaded). The internal names are said to be "mangled." C functions do not have mangled names, so the compiler has to be told to use (or generate) unmangled function names. The following example shows how this is done:

```
extern "C" int f(int x);
extern "C"
{
  int g(int x);
  int h(int x);
}
```

Type casts, in C++, although they can be done just as in C, should be done in the form of a function call, as illustrated in the following code:

```
int i = (int)1.0;  // a C-style cast   (try not to use)
int j = int(1.0);  // a C++-style cast (use)
```

This allows the compiler to better check if the cast makes sense.

In C++, use `nullptr` instead of `NULL`. It serves the same purpose but its use is safer, because in C++ `NULL` is defined to be the constant `0` (and not a pointer to `void` with the value 0 as it is in C).

Some differences between C and C++ (part 5)

Memory can be allocated with the `new` operator and deallocated with the `delete` operator. When used to allocate an instance of a class `new` calls automatically its constructor. Likewise, `delete` calls its destructor. The argument of the `new` operator is a data type. Its return value is a pointer to that type. Note, however, that as in C when one specifies an array what one gets is a pointer to its first element (the constructor of the element type is called for each one of elements of the array). The argument of the `delete` operator should be a pointer received from the `new` operator; if it is not all hell can break loose. The `delete` operator does not have a return value. For array types, the operator `delete[]` calls the destructor for each of the elements of the array (the `delete` operator calls the destructor only for the first element).

The following example shows how `new` and `delete` can be used.

```
int *p_i = new int;              // get memory to an integer
*p_i = 3;                        // give it the value 3
delete p_i;                      // free its memory
p_i = new int(10);               // get memory to another integer at initialize it with the value 10
double *p_d = new double[100];   // get memory for an array of 100 doubles
delete[] p_d;                    // free its memory
class abc;                       // class fully declared elsewhere
abc *pc = new abc;               // pointer to an instance of class abc
```

If there is not enough memory the `new` operator throws a `std::bad_alloc` exception.

Classes (part 1)

Roughly, a C++ class is the combination of a C structure with a set of functions that manipulate the structure. It is a great way to compartmentalize our code, safely hiding the details of how the structure and associated functions are actually implemented. A class is declared just like a structure, but with some extra ingredients:

- while the members of structures have to be data types, members of classes may also be functions.
- when an instance (an object) of a class is created, a constructor member function is called to initialize the data fields of the instance.
- when an instance of the class is destroyed, a destructor member function is called to do any necessary cleanup work.
- some of the members of the class, be they data fields or functions, can be made public, i.e., visible to the entire program, or they can be made private, i.e., visible only by the code that implements the class.
- some data fields may act like global variables, existing only one instance of them irrespective of the number of instances of the class that were created (in C one would have to use a separate global variable to get the same effect; in C++ it is an integral part of the class).

A class with name `CLASS_NAME` is declared as follows (the order of the public and private parts is arbitrary):

```
class CLASS_NAME
{
  private:  // private members part
    // put declarations (of functions) and definitions (of functions or data fields) here
  public:   // public members part
    // put declarations (of functions) and definitions (of functions or data fields) here
};
```

We may have several private and public parts. Class member functions may be defined outside of the class declaration.

Classes (part 2)

The following example presents the code of a very simple class:

```
class dot
{
  private:
    static int n_dots;  // counts the number of dots created
    double d_x;         // x coordinate
    double d_y;         // y coordinate
  public:
    dot(double x,double y) { n_dots++; d_x = x; d_y = y; } // constructor
    ~dot(void) { n_dots--; }                               // destructor
    double x(void) { return d_x; }                         // definition
    double y(void);                                        // declaration (prototype)
    int number_of_dots(void) { return n_dots; }            // definition
};
double dot::y(void) { return d_y; }                        // definition
int main(void)
{
  dot d(0.0,0.0);    // create a dot; almost the same as "dot d = dot(0.0,0.0);"
  double x = d.x();  // get the x coordinate of d
  // more code
}
```

Note that the name of the constructor function is the name of the class and that the name of the destructor is the name of the class preceded by ~. Note also that inside member functions the names of the data members of the class can be used without reference to the class instance (see discussion of the `this` pointer in the next slide).

Classes (part 3)

When a member function of a class is called it receives an extra hidden argument named `this`. This argument is a pointer to the memory area that holds the data of the class instance that is being used. For example, the member function `dot::y` of the previous slide, shown on the left hand side of the following code, is conceptually transformed by the compiler into the code shown on the right hand side:

```
// C++ code
double dot::y(void)
{
  return d_y;
}
```

```
// possible C implementation
double dot_y_implementation(dot * const this)
{
  return this->d_y;
}
```

We can use the `this` pointer in our code (in non static member functions!) without declaring it.

The syntax used to access `struct` data fields is also used to access `class` data members and `class` member functions. For example, if `d_x` had been made `public` in the previous slide, we could have used it as follows:

```
dot d;
d.d_x = 3.0; // set the d_x field of d to 3
```

Since it was made `private` that is not allowed, and we need to provide member functions to set and get its value (if we want to make that data member available to the rest of the program). Calling a member function is done in the same way. For example,

```
double y = d.y();
```

will call the public member function `dot::y` with the `this` pointer set to `&d`.

Classes (part 4)

The following example shows how to

- use arguments and attributes with the same name (the `this` pointer points to the object);
- how to define and use an arithmetic operator (in this case the + operator); and
- how to use I/O streams in the new class (because the left-hand side argument of the << arithmetic operator has a different type, we place the relevant code outside of the class body.

```
using namespace std;

class dot
{
  private:
    double x;
    double y;
  public:
    dot(double x,double y) { this->x = x; this->y = y; }
    ~dot(void) { }
    double get_x(void) { return x; };
    double get_y(void) { return y; };
    void set_x(double x) { this->x = x; };
    void set_y(double y) { this->y = y; };
    dot operator + (dot &d) { return dot(this->x + d.x,this->y + d.y); }
};
std::ostream & operator << (std::ostream &os,dot &d)
{
  return os << "(" << d.get_x() << "," << d.get_y() << ")\n";
}
```

Templates

Templates are a way to write code in a generic way, without specifying beforehand the data types or other parameters that will be used in a data structure or function. The idea is to write code once, and to use it many times. One writes a template for the code, keeping some data types, and possibly other parameters, unspecified. For a function, this is done as in the following example:

```
template <typename T> T f(T x)
{
  return T(7) * x;  // multiply x by 7; 7 is cast to type T (it must be possible to do that)
}
```

Here, the function template has one generic type named `T` (it can have more), and describes a family of functions, named `f`, whose purpose it to multiply its argument by 7. (Of course this could also be done in a far simpler way, but our purpose here is the describe how a template works.) To use the template to define an actual function, do as follows:

```
int    i = f<int>(3);     // i = 7 * 3
double d = f<double>(5.0) // d = 7.0 * 5.0
```

Class templates are done similarly (here using two generic types):

```
template <typename T1,typename T2> class XYZ
{
  private:
    T1 a_member_variable_of_type_T1;
    T2 a_member_variable_of_type_T2;
    // ...
};
```

and used similarly: `XYZ<int,double> a_variable_of_class_XYZ_int_double;`.

Exceptions

In a program, one possible way to deal with an unexpected case (such as trying to compute the square root of a negative number, when that was thought not to be possible to happen) is just to terminate it. In mission critical applications that is not desirable. What one needs is a way to handle gracefully the unexpected condition (after all, it may have been the result of memory corruption due to a very rare cosmic ray, and not the fault of the program). C++ implements a mechanism (try–catch) that can do that. The idea is to surround the program area we want to protect with a "safety net," that catches these unexpected events. We put our normal code in a `try` block, and we put the recovery code in one or more `catch` blocks. Exceptions (the unexpected events) are signaled by "throwing" an exception, using a `throw` statement. The following example will make things clear:

```
double sqrt(double x)
{
  if(x < 0.0) throw 0;  // throw an integer exception with the value 0
  return sqrt(x);
}
int main(void)
{
  try
  {
    cout << sqrt(-1.0) << endl;
  }
  catch(int i)
  {
    cout << "integer exception number " << i << " caught" << endl;
    exit(1);
  }
}
```

C++ stuff not covered by these class notes

Incomplete list of important things not covered in these lecture notes (C++ is a programming language with a extremely rich set of features):

- multiple inheritance (among other things, the `virtual` and `friend` keywords)
- lambda expressions
- the standard template library (STL)
- the Boost library