A Quick Introduction to C++ Programming

Julien Tierny

Abstract These are brief notes for a quick introduction to C++ programming. For conciseness, only the key concepts are presented. For further readings, see [1].

1 Basics

1.1 Programming

- Program: sequence of instructions (cooking recipe);
- Interpreted VS Compiled programs:
  - Compiled program: the code is transformed into a set of low level instructions that the CPU can directly understand (binary file);
  - Interpreted program: the program is analyzed, interpreted and transformed on the fly;
  - Examples:
    - Interpreted: Bash, Perl, Python, Matlab, Java, Ruby, etc;
    - Compiled: C, C++, ObjectiveC, Fortran, etc.
  - Comparison: Compiled program are **MUCH** faster.

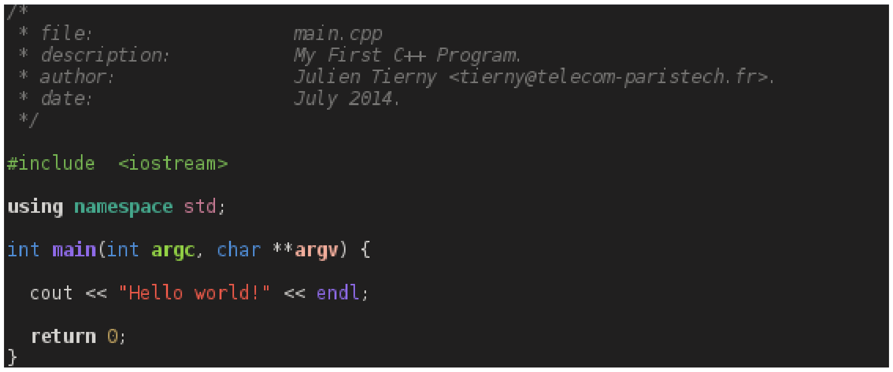
1.2 The C++ language

- Invented in 1982 by Bjarne Stroustrup;
- Successor of the C language (1969, Dennis Ritchie);

Julien Tierny
CNRS LIP6, Sorbonne Universites; CNRS LTCI, Telecom ParisTech; France.
e-mail: `julien.tierny@lip6.fr`

- Pros:
  - Fast compiled language;
  - General purpose programming language;
  - Highly portable (pretty much any processor);
  - Object-oriented: well suited for large programs;
  - **WIDELY** used (lots of additional libraries);
- Cons:
  - Often considered as a difficult language to learn.

1.3 My First C++ program



```
/*
 * file:                main.cpp
 * description:         My First C++ Program.
 * author:              Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                July 2014.
 */

#include  <iostream>

using namespace std;

int main(int argc, char **argv) {

  cout << "Hello world!" << endl;

  return 0;
}
```

Fig. 1 A simple C++ program that prints “Hello world”.

- Edit a file called `main.cpp`;
- It must at least include:
  - `int main(int argc, char **argv){`
  - `  // Some code here`
  - `  return 0;`
  - `}`
- In general, one line = one instruction;
- Each instruction line **MUST** end with the character “`;`”;
- Comments (text which is not executed) must be preceded by “`//`”;

2 Setting up a C++ program

2.1 *Compilation*

- C++ source codes are translated into binary executable files through a process called *Compilation*;
- The output binary file contains low level instructions that are directly executable by the CPU;
- In our exercises, the program in charge of the compilation, called a *compiler*, will be `gcc`.

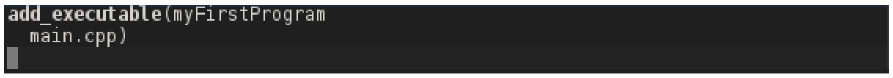
2.2 *Libraries*

- It is possible to call from your C++ program external functionalities that have been programmed by a third-party independently;
- External functionalities are implemented in "*libraries*", which can be understood as *program building blocks*;
- Libraries are C++ functions that have been compiled. Independently, they cannot be executed. They need to be called by a C++ program to be executed.
- A C++ program that makes use of a library needs, after compilation into a binary file, to be *linked* to the binary files of the libraries it uses. The linking process is done by programs such as `ld`.
- An example of library (that we will use in our exercises) is the Visualization Tool Kit (VTK, http://www.vtk.org), which already implements many interesting visualization features.
- To use an external library from a C++ program:
  - One needs to include a *header file* (`*.h`) which formally describes (from a C++ perspectives) the functionalities of the library;
  - In Figure 1, such an inclusion happens with the line `#include <iostream>`.

2.3 *Building Instructions*

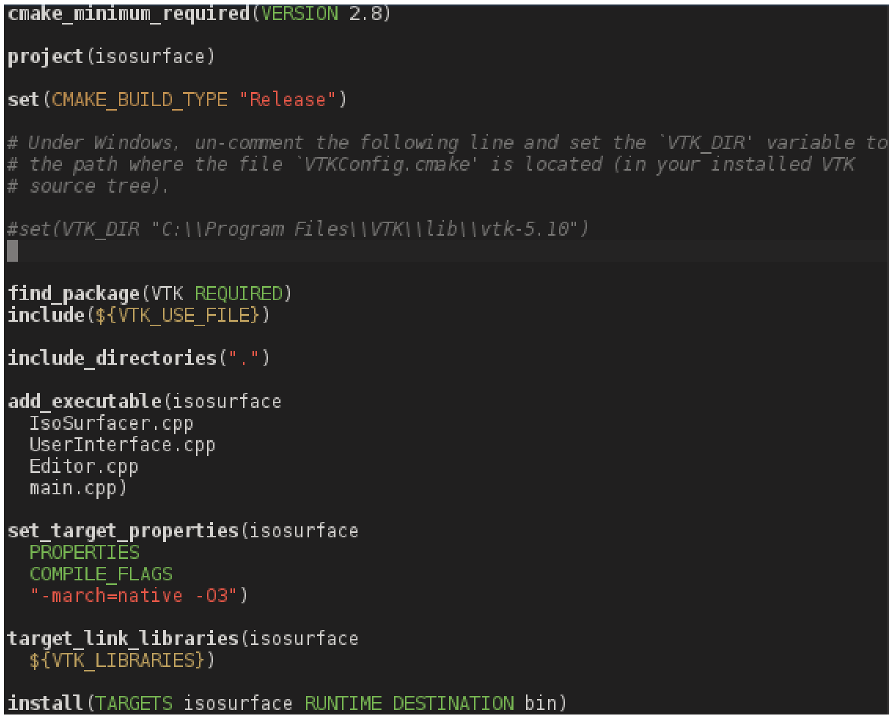
- In general, a C++ program is made of a large number of source (`*.cpp`) and header (`*.h`) files and links to several libraries.
- Then compiling a code and linking it to its libraries with low level programs such as `gcc` and `ld` can be a very tedious task for a developer.
- Moreover, depending on the operating systems (Windows, Linux, MacOs, etc.), libraries can be installed in different paths, which challenges even more the compilation of a given program;

- To simplify the compilation of C++ programs and to make this process easily portable (to be able to easily compile a program on Windows and Linux), compilation instruction systems have been developed. In our exercise, we will use `cmake` (http://www.cmake.org).
- For each C++ program, a file called `CMakeLists.txt` needs to be edited;



```
add_executable(myFirstProgram
  main.cpp)
```

Fig. 2 `CMakeLists.txt` file for building the program shown in Figure 1.



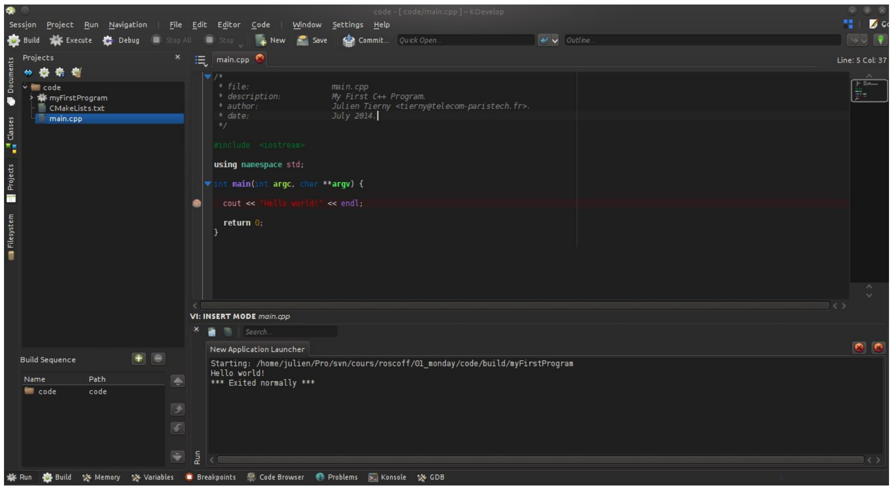
```
cmake_minimum_required(VERSION 2.8)

project(isosurface)

set(CMAKE_BUILD_TYPE "Release")

# Under Windows, un-comment the following line and set the `VTK_DIR' variable to
# the path where the file `VTKConfig.cmake' is located (in your installed VTK
# source tree).

#set(VTK_DIR "C:\\Program Files\\VTK\\lib\\vtk-5.10")


find_package(VTK REQUIRED)
include(${VTK_USE_FILE})

include_directories(".")

add_executable(isosurface
  IsoSurfacer.cpp
  UserInterface.cpp
  Editor.cpp
  main.cpp)

set_target_properties(isosurface
  PROPERTIES
  COMPILE_FLAGS
  "-march=native -O3")

target_link_libraries(isosurface
  ${VTK_LIBRARIES})

install(TARGETS isosurface RUNTIME DESTINATION bin)
```

Fig. 3 `CMakeLists.txt` file for building one of the programs of the exercises.

- Two examples of `CMakeLists.txt` files are given:
  - Figure 2: This file contains the building instructions for the simple program shown in Figure 1. It specifies the name of the executable program file (`myFirstProgram`) and specifies which C++ source files should be considered for compilation.

  - Figure 3 shows the `CMakeLists.txt` file for a more complex program (used later in the exercices). In particular, with the `find_package` command, the paths where the VTK libraries files are installed are automatically identitied. Several `*.cpp` source files are considered and the instructions for the linking to the VTK library are set with the command `target_link_libraries`.
- Then under Linux for example, to compile the program, enter the directory containing the file `CMakeLists.txt` and enter following command (omit the `$` character):
  - `$ cmake . && make`

2.4 *Integrated Development Environment*



code - [code/main.cpp] – KDevelop
Session Project Run Navigation File Edit Editor Code Window Settings Help
Build Execute Debug Stop All Stop New Save Commit... Quick Open... Outline...
Projects
code
myFirstProgram
CMakeLists.txt
main.cpp
Line: 5 Col: 37
VI: INSERT MODE main.cpp
New Application Launcher
Build Sequence
Name Path
code code
Run Build Memory Variables Breakpoints Code Browser Problems Konsole GDB

```
/*
 * file:                main.cpp
 * description:         My First C++ Program.
 * author:              Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                July 2014.
 */

#include  <iostream>

using namespace std;

int main(int argc, char **argv) {

  cout << "Hello world!" << endl;

  return 0;
}
```

```
Starting: /home/julien/Pro/svn/cours/roscoff/01_monday/code/build/myFirstProgram
Hello world!
*** Exited normally ***
```

Fig. 4 Screenshot of the KDevelop IDE with the simple C++ program of Figure 1.

- Integrated Development Environments (IDE) are programs that enable program developers to easily write, debug and execute code.
- Popular examples of IDEs are: MicroSoft Visual Studio, XCode, Eclipse, Netbeans, KDevelop, etc.
- In our exercises, we will use the KDevelop IDE;
- A screenshot of KDevelop for the simple C++ program of Figure 1 is shown in Figure 4:
  - To compile your program, click on the ”*Build*“ button (top left);
  - To execute your program, click on the ”*Execute*“ button (top left);

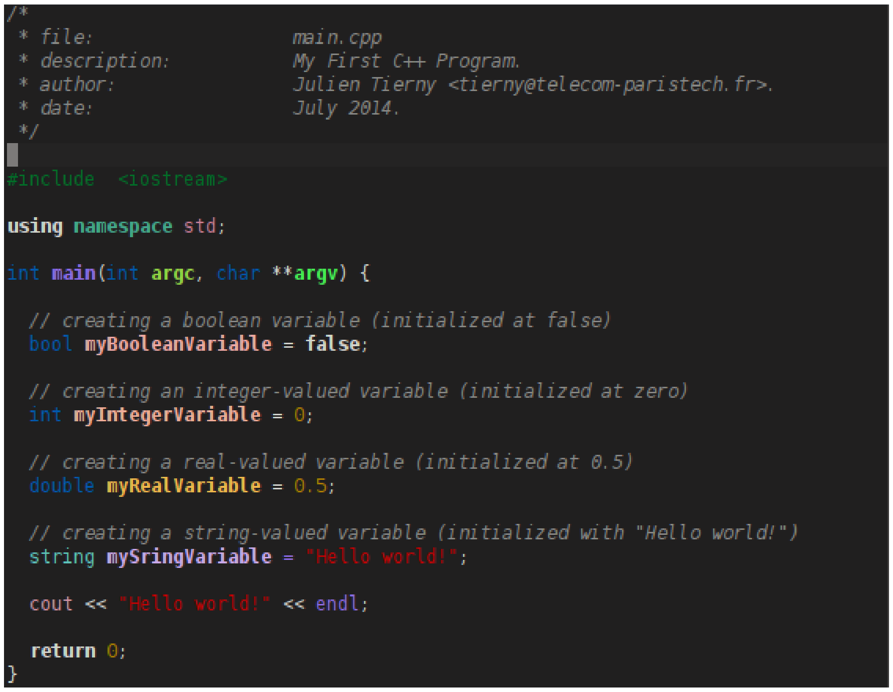
- The banner on the left shows the different C++ files (`*.cpp` and `*.h`) used in the program;
- The central banner consists of a text editor for editing source files (`*.cpp` and `*.h`):
  - This editor provides syntactic coloring as well as auto-completion;
- The bottom banner shows the result of the execution of the program ("*Hello world!*" is printed on the output);
- KDevelop also provides a debugging mode:
  - To execute your program in Debug mode, click on the "*Debug*" button (top left);
  - Debug mode enables you to pause (and later resume) the execution of your program at specific execution lines, called "*breakpoints*";
  - An example of breakpoint is given at the execution line colored in red (with a red marker on the left margin);
  - Pausing the execution of a program in Debug mode enables to inspect the values taken by the variables of a program at specific points of the execution and better understand the behavior of a program.

Important Note: In the exercises, a skeleton of code will be provided to you in KDevelop. This means you will not have to create your own `CMakeLists.txt` file. To compile your program, you'll only need to click on the "*Build*" button of KDevelop. Moreover, you will only be asked to complete portions of existing code (no start from scratch).

3 Basic Types and Operators

3.1 Variables

- To store some information inside a program, one needs to use *variables*;
- Variables are defined with a specific *type*, for example:
  - `bool` for booleans (variables that can only take the values `true` or `false`);
  - `int` for integer values;
  - `double` for real values (double floating point precision);
  - `string` for character string (words and sentences);
- Variables need to be declared with an instruction line, ending by "`;`" and starting by a type specification, as shown in Figure 5;
- It is a good practice to initialize variables with an initial value, as showcased in Figure 5;
- It is also a good practice to use **explicit** variable names. Another person should understand, when reading the name of your variable, what information it stores.



```
/*
 * file:                main.cpp
 * description:         My First C++ Program.
 * author:              Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                July 2014.
 */

#include  <iostream>

using namespace std;

int main(int argc, char **argv) {

  // creating a boolean variable (initialized at false)
  bool myBooleanVariable = false;

  // creating an integer-valued variable (initialized at zero)
  int myIntegerVariable = 0;

  // creating a real-valued variable (initialized at 0.5)
  double myRealVariable = 0.5;

  // creating a string-valued variable (initialized with "Hello world!")
  string mySringVariable = "Hello world!";

  cout << "Hello world!" << endl;

  return 0;
}
```

Fig. 5 Examples of variable definitions.

3.2 Operators

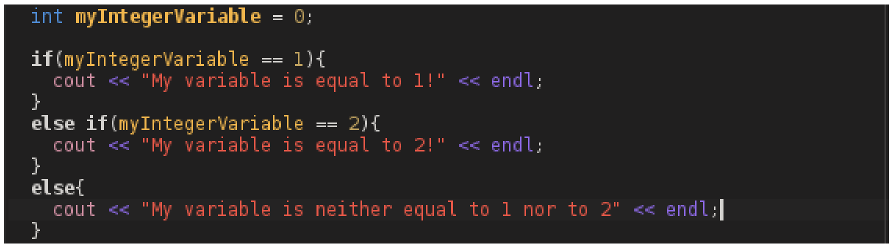
- Several operators exist to apply operations on variables;
- Of particular importance is the assignment operator =:
  - `int myIntegerVariable = 5;`

  will assign the value 5 to the variable `myIntegerVariable`;
- Common operators include +, −, ⋆, /;
- For example:
  ```
  int myVariable0 = 5;
  int myVariable1 = 2;
  int myVariable2 = 0;
  myVariable2 = myVariable0 * myVariable1;
  ```
  will assign the value 10 to `myVariable2` (5 * 2);
- Specific operators enable to *increment* or *decrement* the content of variables:
  - ++ increments the content of a variable;
  - −− decrements the content of a variable:
    ```
    int myIntegerVariable = 0;
    myIntegerVariable++;
    ```

```
// myIntegerVariable now contains 1
myIntegerVariable--;
// myIntegerVariable now contains 0 again
```

4 Control Structures

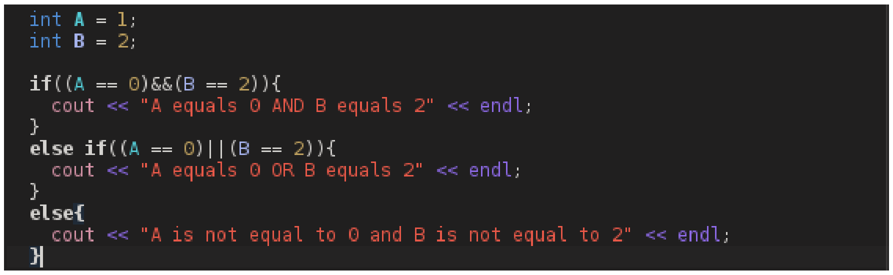
4.1 Conditions



```
int myIntegerVariable = 0;

if(myIntegerVariable == 1){
  cout << "My variable is equal to 1!" << endl;
}
else if(myIntegerVariable == 2){
  cout << "My variable is equal to 2!" << endl;
}
else{
  cout << "My variable is neither equal to 1 nor to 2" << endl;
}
```

Fig. 6 Example of usage of the `if` condition control structure.

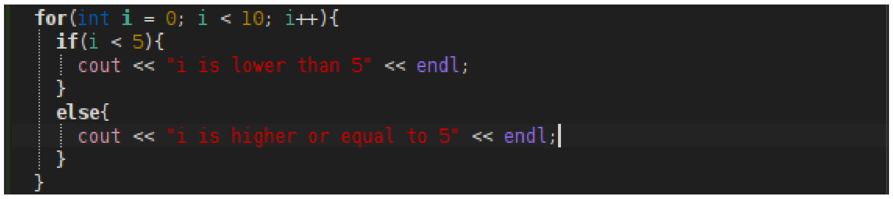
- In a program, some instructions may need to be executed only if certain conditions are fulfilled;
- To verify if a condition is fulfilled, one needs to use the control structure `if`;
- Figure 6 shows a basic example of usage of the `if` condition control structure:
  - Note the { } characters, which delimit the instructions that should be executed if the condition is fulfilled;
  - Note the presence of parenthesis to delimit conditions.
- Several condition operators can be used in conjunction with a `if` condition:
  - == checks if two expressions are equal (cf. Figure 6);
  - != checks if two expressions are different;
  - < checks if one expression is lower than the other;
  - <= checks if one expression is lower or equal to the other;
  - > checks if one expression is higher than the other;
  - >= checks if one expression is higher or equal to the other.
- Several conditions can be checked simultaneously (as showcased in Figure 7):
  - `&&`: Both conditions need to be fulfilled (AND);
  - `||`: At least one condition needs to be fulfilled (OR).



```
int A = 1;
int B = 2;

if((A == 0)&&(B == 2)){
  cout << "A equals 0 AND B equals 2" << endl;
}
else if((A == 0)||(B == 2)){
  cout << "A equals 0 OR B equals 2" << endl;
}
else{
  cout << "A is not equal to 0 and B is not equal to 2" << endl;
}
```

Fig. 7 `if` control structure with multiple conditions.

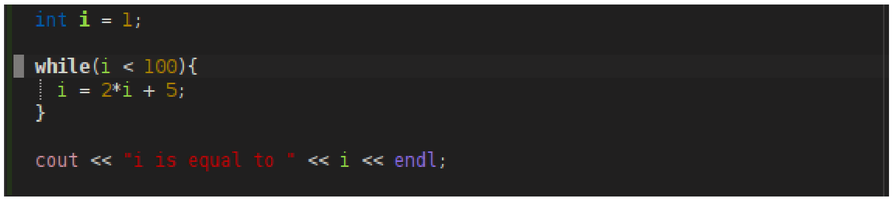
4.2 *Iterations*



```
for(int i = 0; i < 10; i++){
  if(i < 5){
    cout << "i is lower than 5" << endl;
  }
  else{
    cout << "i is higher or equal to 5" << endl;
  }
}
```

Fig. 8 Example of simple `for` loop: the program iterates from 0 to 9 (included) and prints out a specific message in the first 5 iterations and another in the last 5 iterations.

- Within a program, it is often useful to iterate over a set of elements, to apply a specific processing to each item;
- Several control structures exist for iterating over a set:
  - `for` loops:
    - This control structure enables to iterate over a sequence and to apply specific instructions at each iteration;
    - An example is given in Figure 8;
    - The specification of the starting, stopping and iteration conditions of a `for` loop is done as follows (see Figure 8):
      - First the starting condition needs to be specified (`int i = 0`). Note that an iteration variable (here `i`) can be declared on the fly;
      - Second, the stopping condition needs to specified (`i < 10`);
      - Last, the iteration condition needs to be specified (`i++`);
      - Conditions are separated by ”`;`“.
  - `while` loop:
    - This control structure enables to *repeat* some instructions, while a specific condition is fulfilled, as shown in Figure 9;



```
int i = 1;

while(i < 100){
  i = 2*i + 5;
}

cout << "i is equal to " << i << endl;
```

Fig. 9 Example of `while` loop.

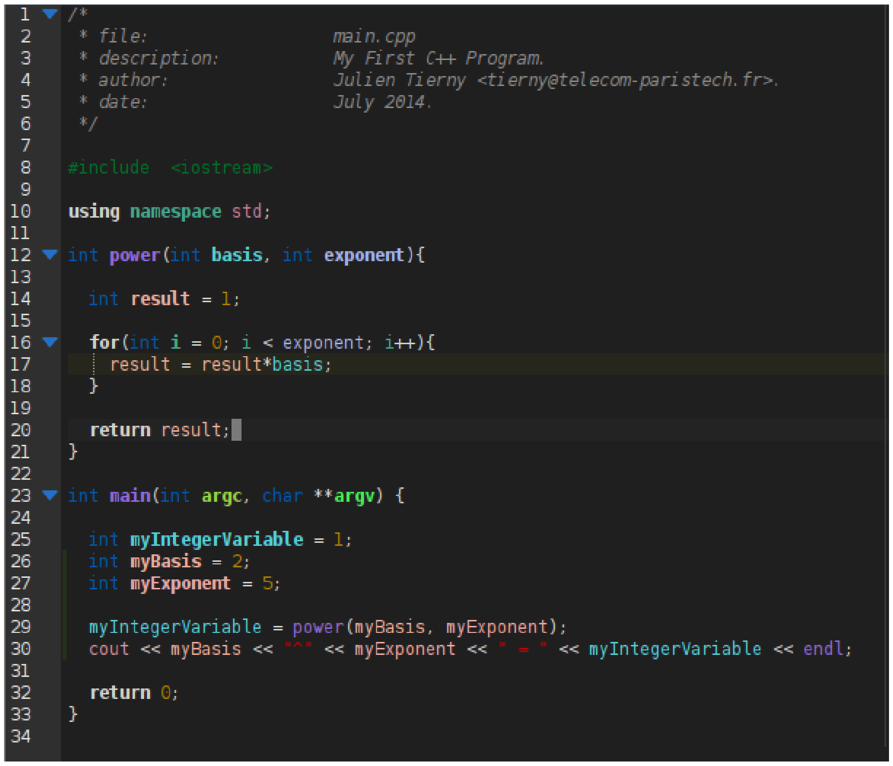
· In contrast to `for` loops, `while` loops are especially useful when clear stopping conditions can be specified although the number of iterations is not known a priori.

5 Functions

- In order to make a C++ source code easier to read and more re-usable, one should organize code sections into *functions*;
- Functions are sets of instructions that can be called several times within a program;
- Functions can take arguments and return a result. Both arguments and result type must be specified in the function declaration;
- Figure 10 shows an example of function implementation and usage.
  - The specification of the function (return type, name, arguments) happens line 12;
  - The instructions related to the function are listed from line 14 to 20;
  - Notice the presence of the keyword `return`, which is meant to return the result to the calling context;
  - Notice the characters { }, which delimit the instruction lines dedicated to the function;
  - Line 29 shows a usage example of the function.
- Note that arguments (in our example two integer variables called `basis` and `exponent`) are *local* variables, which only live during the execution of the function. They cannot be reached from outside the function (see the **demo**).

6 Addressing

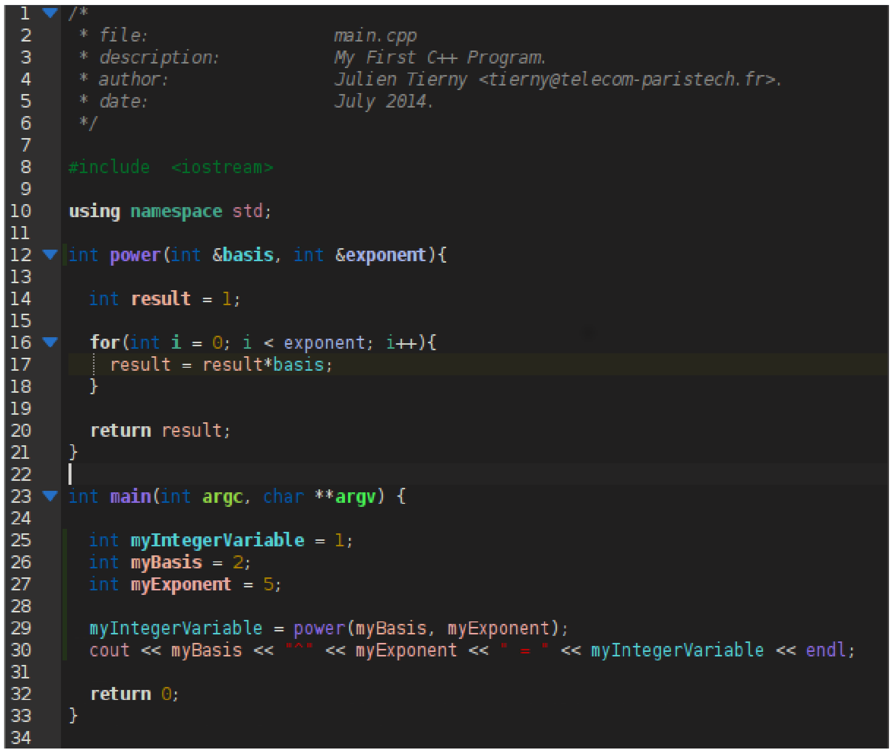
- In many scenarios, it may be useful not to manipulate variables, but addresses to variables;
- For instance, in the example of Figure 10:



```
 1  /*
 2   * file:                    main.cpp
 3   * description:             My First C++ Program.
 4   * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 5   * date:                    July 2014.
 6   */
 7
 8  #include  <iostream>
 9
10  using namespace std;
11
12  int power(int basis, int exponent){
13
14    int result = 1;
15
16    for(int i = 0; i < exponent; i++){
17      result = result*basis;
18    }
19
20    return result;
21  }
22
23  int main(int argc, char **argv) {
24
25    int myIntegerVariable = 1;
26    int myBasis = 2;
27    int myExponent = 5;
28
29    myIntegerVariable = power(myBasis, myExponent);
30    cout << myBasis << "^" << myExponent << " = " << myIntegerVariable << endl;
31
32    return 0;
33  }
34
```

Fig. 10 Example of function implementation (from line 12 to 21) and usage (line 29).

– The arguments of the function (`basis` and `exponent`) are *local variables*: they are created when the execution flow enters the function (line 12) and they are destroyed when it leaves the function (line 21);
– Thus, the variables store *copies* of the initial values stored in the variables `myBasis` and `myExponent`;
– Sometimes, function arguments can be of a much complex nature than a single integer and their storage may require a lot of memory;
– To overcome this issue (for instance, when passing arguments to a function), one can only provide the address in memory of the variable, instead of creating a full copy of it (which will consequently save a lot of memory);
– Variable addresses can be handled in two ways in C++:
  1. with *References*;
  2. with *Pointers*;



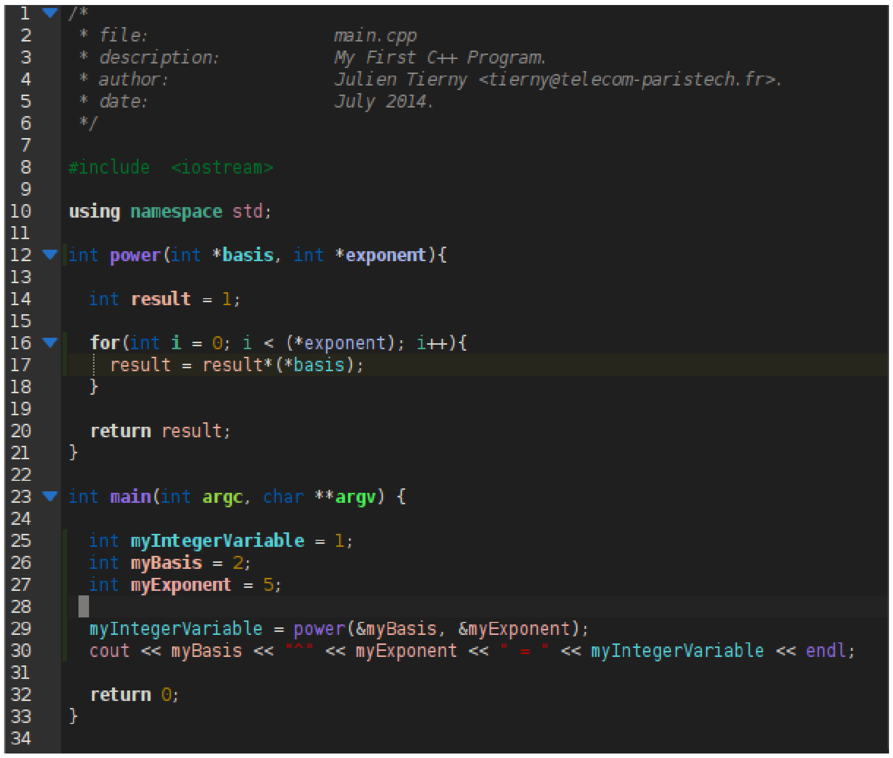
```
1  /*
2   * file:                   main.cpp
3   * description:            My First C++ Program.
4   * author:                 Julien Tierny <tierny@telecom-paristech.fr>.
5   * date:                   July 2014.
6   */
7
8  #include  <iostream>
9
10 using namespace std;
11
12 int power(int &basis, int &exponent){
13
14   int result = 1;
15
16   for(int i = 0; i < exponent; i++){
17     result = result*basis;
18   }
19
20   return result;
21 }
22
23 int main(int argc, char **argv) {
24
25   int myIntegerVariable = 1;
26   int myBasis = 2;
27   int myExponent = 5;
28
29   myIntegerVariable = power(myBasis, myExponent);
30   cout << myBasis << "^" << myExponent << " = " << myIntegerVariable << endl;
31
32   return 0;
33 }
34
```

Fig. 11 Example of Figure 10, but with *references* for the handling of the arguments of the function.

6.1 References

- References are the simplest way to handle variable addresses in C++;
- Figure 11 shows the example of Figure 10, but with references to variables instead of full copies:
  - To declare a reference, one needs to add the character `&` in front of the variable name (see line 12);
  - To use a reference to a variable, one proceeds exactly ”as-if“ the variable was a full copy. In other words, using a reference to a variable looks exactly the same as if one was using the variable itself (see the content of the function from lines 13 to 21 which has not change).
  - Internally, using a reference as shown in Figure 11 will avoid the creation of new variables and full copies of the content of the arguments.
- In general, unless a more subtle addressing is needed, it is a good practice to only pass arguments to functions with references.

6.2 Pointers



```
1  /*
2   * file:                  main.cpp
3   * description:           My First C++ Program.
4   * author:                Julien Tierny <tierny@telecom-paristech.fr>.
5   * date:                  July 2014.
6   */
7
8  #include <iostream>
9
10 using namespace std;
11
12 int power(int *basis, int *exponent){
13
14   int result = 1;
15
16   for(int i = 0; i < (*exponent); i++){
17     result = result*(*basis);
18   }
19
20   return result;
21 }
22
23 int main(int argc, char **argv) {
24
25   int myIntegerVariable = 1;
26   int myBasis = 2;
27   int myExponent = 5;
28
29   myIntegerVariable = power(&myBasis, &myExponent);
30   cout << myBasis << "^" << myExponent << " = " << myIntegerVariable << endl;
31
32   return 0;
33 }
34
```

Fig. 12 Example of Figure 10, but with *pointers* for the handling of the arguments of the function.

- References are the easiest way to address variables;
- However, references have a few restrictions which prevent a more subtle variable addressing;
- For an advanced addressing, one needs to use *Pointers*:
  - Figure 12 shows the example of Figure 10, but with pointers to variables instead of full copies:
    - Pointers can be understood as a special type of variable, which stores the address in memory of another variable of a specific type;
    - To **declare** a pointer, one needs to add the character $\star$ in front of the variable name (see line 12);
    - To **use** a pointer to a variable, one needs to add the character $\star$ in front of the variable name, each time, to explicitly refer not the pointer itself, but to the variable it points to (see lines 16 and 17);

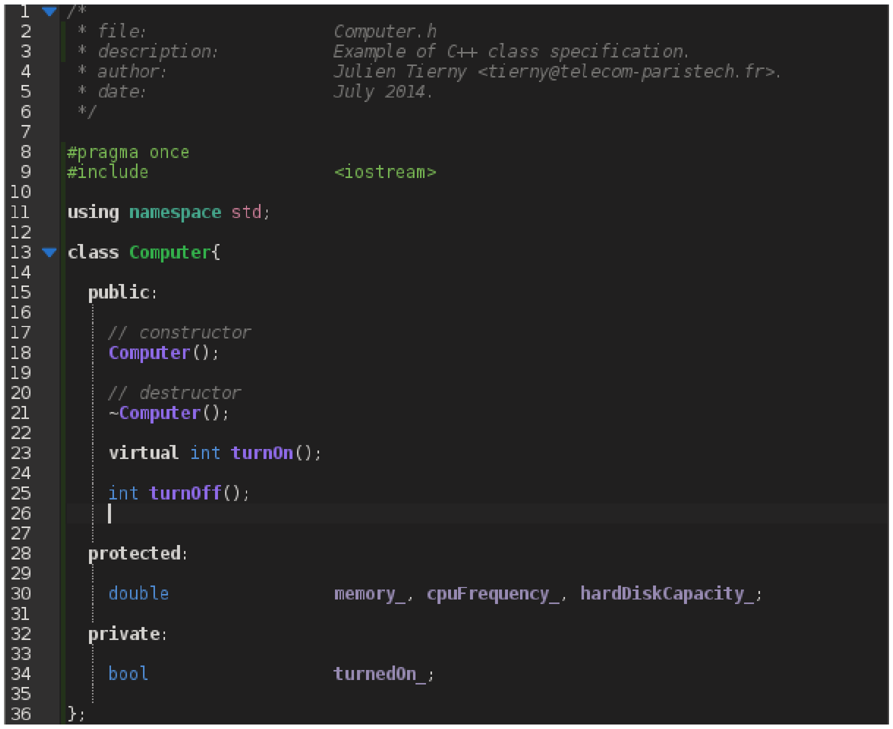
- To make the code more readable, it is often a good practice to use parentheses too (see lines 16 and 17);
- To **assign** the address of a variable to a pointer, one needs to use the character `&` in front of the variable name:
  - Line 29, the addresses in memory of the variable `myBasis` and `myExponent` are passed as argument to the function, by using this extra character.

7 Object Oriented Programming

- Object-oriented programming is the main concept that differentiates the language C++ from its predecessor C;
- The object-oriented programming paradigm enables developers to create new types (called *classes*) which contain both variables and functions altogether;
- The instances of a class (called *objects* instead of variables) thus contain their own information, and actions on this information can be triggered by calling the object's functions.
- Object-oriented programming also includes many features (such as inheritance and polymorphism) which enables to design highly re-usable code.

7.1 Header file (.h)*

- Each new class must be specified through a specific process;
- Usually, it is a good practice to write the specification of each class in a separate header file (`*.h`);
- Figure 13 shows an example of class specification:
  - A header file should start with the pre-processor instruction `#pragma once` (line 8);
  - The specification of a class starts with the keyword `class`, followed by its name (here `Computer`, line 13);
  - The remainder of the specification is written in between the characters { };
  - The specification of a class is divided into 3 sections:
    - `public:` This section specifies functions and variables that are accessible to everyone;
    - `protected:` This section specifies functions and variables that are only accessible to the class itself and other inheriting (derived) classes;
    - `private:` This section specifies functions and variables that are only accessible to the class itself.
  - Two special functions should always be implemented:

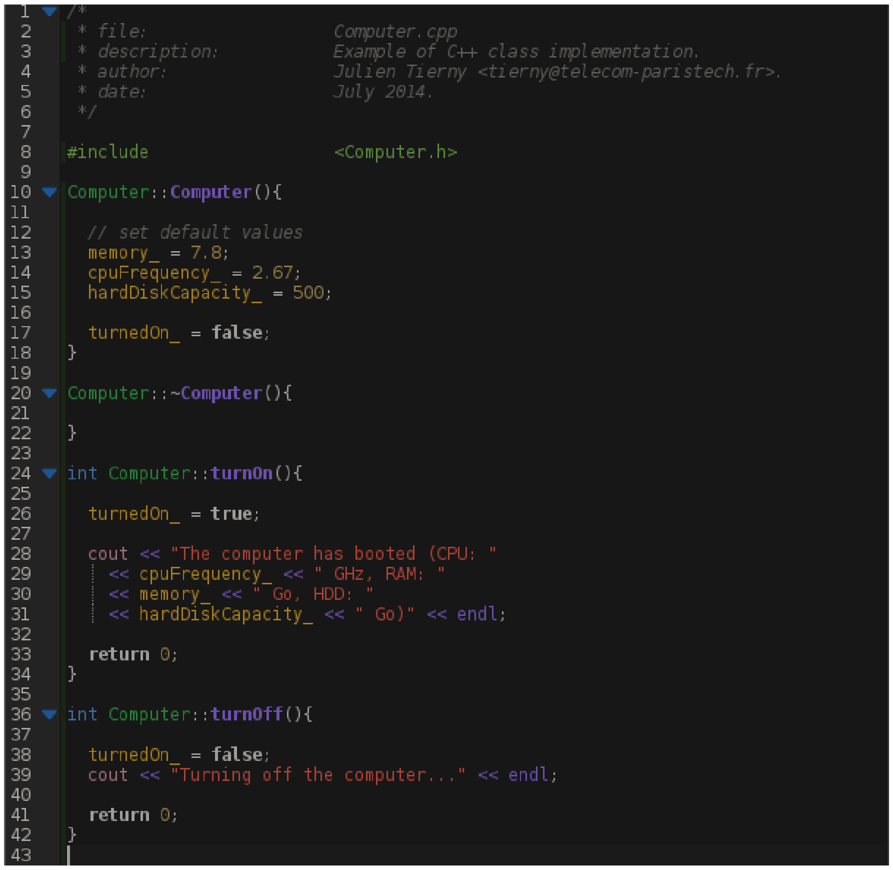


```
 1  /*
 2   * file:                    Computer.h
 3   * description:             Example of C++ class specification.
 4   * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 5   * date:                    July 2014.
 6   */
 7
 8  #pragma once
 9  #include                    <iostream>
10
11  using namespace std;
12
13  class Computer{
14
15    public:
16
17      // constructor
18      Computer();
19
20      // destructor
21      ~Computer();
22
23      virtual int turnOn();
24
25      int turnOff();
26
27
28    protected:
29
30      double                  memory_, cpuFrequency_, hardDiskCapacity_;
31
32    private:
33
34      bool                    turnedOn_;
35
36  };
```

Fig. 13 Specification of a class in a header file.

  · The constructor (specified line 18): this is the function that is called by the system when an object of the class is created. Usually, one sets default parameters in this function and allocate memory if needed. The name of the constructor always corresponds to that of the class;
  · The destructor (specified line 21): this is the function that is called by the system when an object of the class is destroyed. Usually, one frees the memory dynamically allocated to the object in this function. The name of the destructor always corresponds to that of the class, preceded by the character ”~“;

– In our example, a few variables are defined (these can be public, protected or private). A good practice is to name them with a distinctive character (to easily distinguish them from local variables later on). In this example, the character ”_“ is used at the end of the variable name;



```
1  /*
2   * file:                    Computer.cpp
3   * description:             Example of C++ class implementation.
4   * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
5   * date:                    July 2014.
6   */
7
8  #include                    <Computer.h>
9
10 Computer::Computer(){
11
12   // set default values
13   memory_ = 7.8;
14   cpuFrequency_ = 2.67;
15   hardDiskCapacity_ = 500;
16
17   turnedOn_ = false;
18 }
19
20 Computer::~Computer(){
21
22 }
23
24 int Computer::turnOn(){
25
26   turnedOn_ = true;
27
28   cout << "The computer has booted (CPU: "
29     << cpuFrequency_ << " GHz, RAM: "
30     << memory_ << " Go, HDD: "
31     << hardDiskCapacity_ << " Go)" << endl;
32
33   return 0;
34 }
35
36 int Computer::turnOff(){
37
38   turnedOn_ = false;
39   cout << "Turning off the computer..." << endl;
40
41   return 0;
42 }
43
```

Fig. 14 Implementation of the class specified in Figure 13.

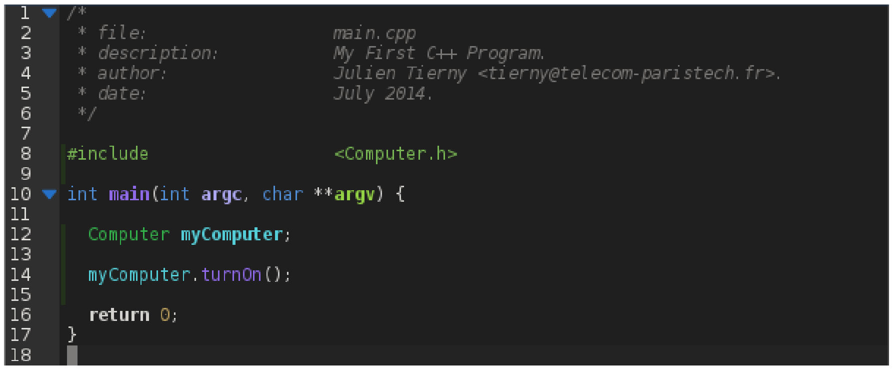
7.2 *Implementation file (*.cpp)*

- Classes are *implemented* in `*.cpp` files:
  - This file contains the implementation of the functions which have been declared in the corresponding `*.h` file.
- An example of class implementation is given in Figure 14:
  - A class implementation file should start with the following pre-processor instruction `#include <MyClassSpecification.h>` (here line 8) in order for the compiler to load the class specification in memory;
  - Then, the remainder of the class is meant to implement the functions defined in the header file:
    - · Each function implementation should be formatted as follows:

    · Return type (if any);
    · Class name followed by "`::`";
    · Function name;
    · Parenthesis (with arguments inside, if any);
    · Brackets ("{" and "}") with the function instructions in between;
    · If the function has a return type, then it should return a value with the keyword `return` (line 39 for instance).

- Note that neither the constructor nor the destructor have return types.

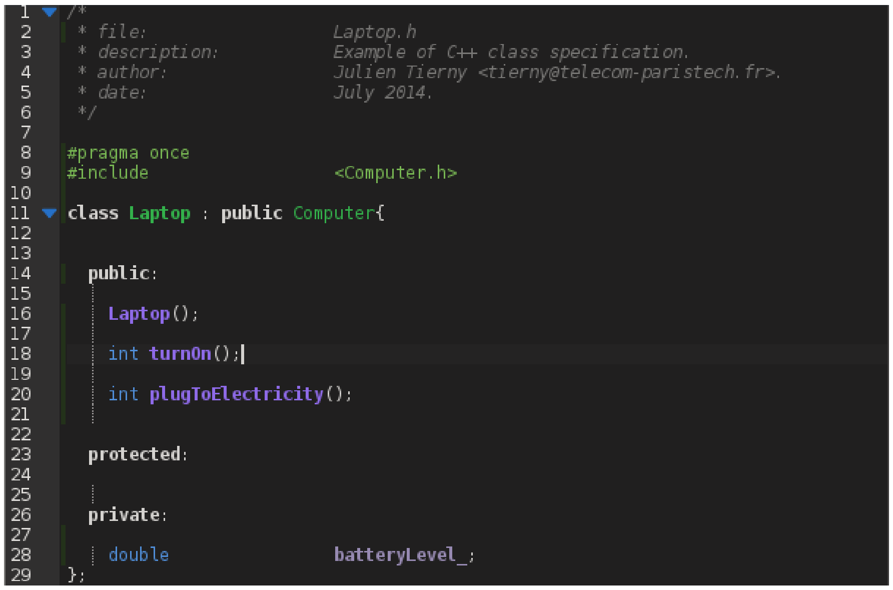
7.3 Usage



```
 1  /*
 2   * file:                    main.cpp
 3   * description:             My First C++ Program.
 4   * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 5   * date:                    July 2014.
 6   */
 7
 8  #include                    <Computer.h>
 9
10  int main(int argc, char **argv) {
11
12    Computer myComputer;
13
14    myComputer.turnOn();
15
16    return 0;
17  }
18
```

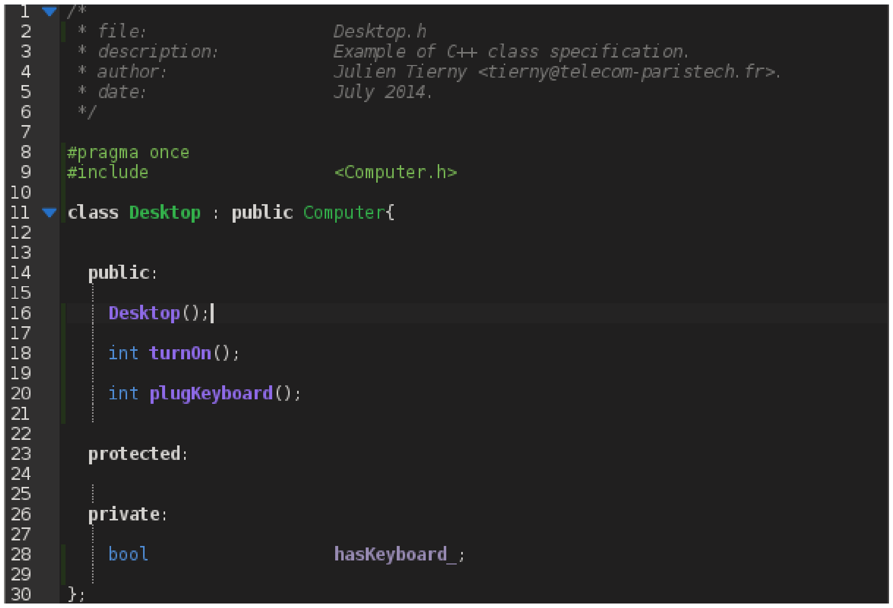
Fig. 15 Usage example for the class specified and implemented in Figures 13 and 14 respectively.

- An example of class usage is given in Figure 15:
  - To use a class, one first needs to indicate to the compiler where this class is specified (in which header file). This is done with the pre-processor command `#include <MyClassSpecification.h>` (line 8 in the example);
  - Then, one can declare an object as any other variable (line 12) by specifying its class (type) and a variable name (here `myComputer`);
  - Finally, to call the functions of the object or access its variables, one needs to write the name of the object, followed by the "`.`" character, followed by the name of the variable or the name of the function (with parenthesis and arguments if any, line 14).



```
/*
 * file:                    Laptop.h
 * description:             Example of C++ class specification.
 * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                    July 2014.
 */

#pragma once
#include                    <Computer.h>

class Laptop : public Computer{


  public:

    Laptop();

    int turnOn();

    int plugToElectricity();


  protected:


  private:

    double                  batteryLevel_;
};
```

(a)



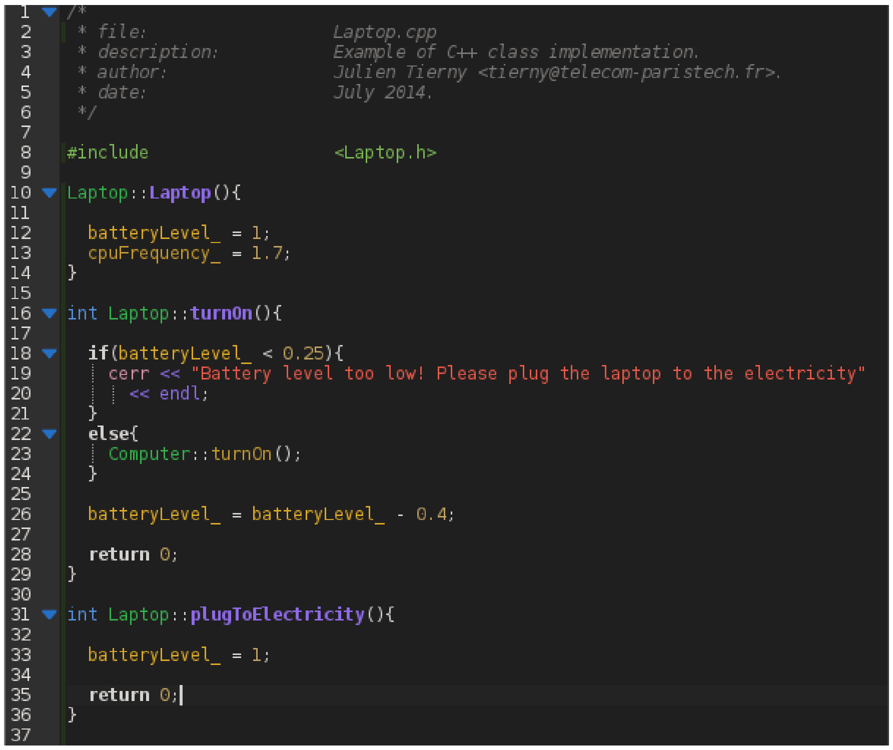
```
/*
 * file:                    Desktop.h
 * description:             Example of C++ class specification.
 * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                    July 2014.
 */

#pragma once
#include                    <Computer.h>

class Desktop : public Computer{


  public:

    Desktop();

    int turnOn();

    int plugKeyboard();


  protected:


  private:

    bool                    hasKeyboard_;

};
```

(b)

Fig. 16 Examples of classes that specialize the inherited class of Figure 13.

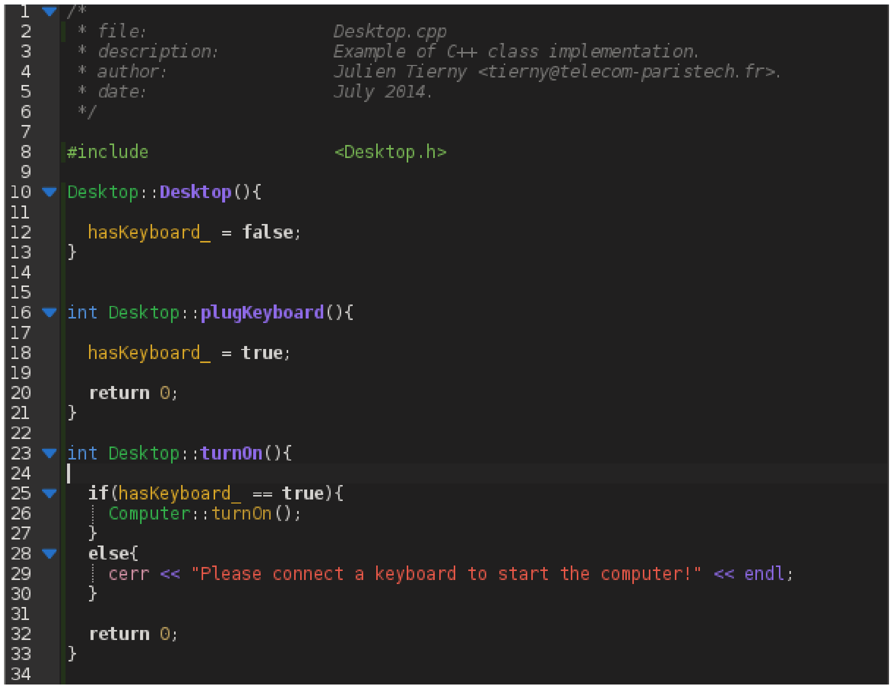
7.4 Inheritance

- One of the key-concepts of object-oriented programming is the notion of inheritance;
- This enables to easily define new classes, that are derived from others without having to re-write all of the common features.
- Figures 16(a) and 16(b) provide examples of classes that inherit from another:
  - The key idea of inheritance is that a given class (for instance the `Computer` class) can be *specialized*:
    - One can derive a new class (here `Laptop`) that specializes the description and behavior of the parent class:
      - In particular, a laptop is a special type of computer that has a battery (hence the new private variable `batteryLevel_`);
    - Thanks to the mechanism of inheritance, one does not need to re-write all the features of the parent class (`Computer`) but only those which are *specific* to the child class (here `Laptop` or `Desktop`).
  - To specify a class that inherits from another, one should proceed as follows (see Figure 16(a)):
    - One needs to include the header file of the parent class (line 9);
    - In the class definition, one needs to specify the inheritance, for example by adding the following keywords line 11: "`: public Computer`";
    - Then, one can add new variables (line 28), new functions (line 20) or replace functions that already exist in the parent class (line 18).
- Figures 17(a) and 17(b) provide two examples of implementations of the inheriting classes specified in Figures 16(a) and 16(b):
  - The implementation of an inherited class happens exactly the same way as an ordinary class;
  - Only the new and modified functions need to be implemented. For instance, the function `turnOff()` is the same for a laptop or a desktop computer. Hence, there is no need to re-implement this function in the inherited classes;
  - In these examples, the inherited class are specialized as follows:
    - `Laptop`:
      - A laptop is special computer with specific CPU and memory characteristics (see the constructor, line 10 to 14);
      - A laptop is equipped with a battery, whose level decreases after usage (see line 26);
      - A laptop requires to have a certain level of battery in order to boot (lines 18 to 24);
    - `Desktop`:
      - A Desktop is a special type of computer which does not have a battery;
      - Usually, a keyboard is required to be plugged in for the desktop computer to boot (line 25).



```
/*
 * file:                    Laptop.cpp
 * description:             Example of C++ class implementation.
 * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                    July 2014.
 */

#include                    <Laptop.h>

Laptop::Laptop(){

  batteryLevel_ = 1;
  cpuFrequency_ = 1.7;
}

int Laptop::turnOn(){

  if(batteryLevel_ < 0.25){
    cerr << "Battery level too low! Please plug the laptop to the electricity"
      << endl;
  }
  else{
    Computer::turnOn();
  }

  batteryLevel_ = batteryLevel_ - 0.4;

  return 0;
}

int Laptop::plugToElectricity(){

  batteryLevel_ = 1;

  return 0;
}
```

(a)



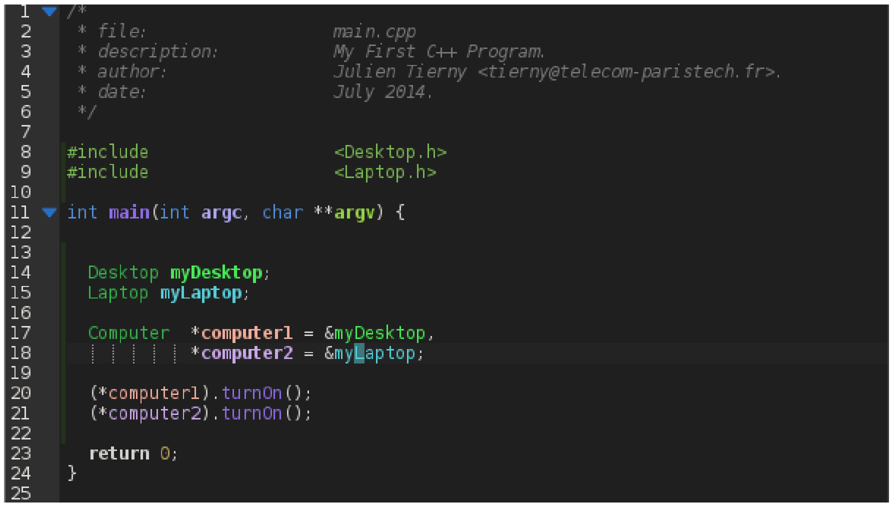
```
/*
 * file:                    Desktop.cpp
 * description:             Example of C++ class implementation.
 * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                    July 2014.
 */

#include                    <Desktop.h>

Desktop::Desktop(){

  hasKeyboard_ = false;
}


int Desktop::plugKeyboard(){

  hasKeyboard_ = true;

  return 0;
}

int Desktop::turnOn(){

  if(hasKeyboard_ == true){
    Computer::turnOn();
  }
  else{
    cerr << "Please connect a keyboard to start the computer!" << endl;
  }

  return 0;
}
```

(b)

Fig. 17 Examples of implementation of inherited classes.

- Note that, laptops and desktops have specific booting procedures (functions `turnOn()`):
  - The laptop checks for the battery level prior to booting;
  - The desktop checks for the presence of a keyboard prior to booting;
  - After these verifications, both computers boot the same way. Hence, both implementations call the `turnOn()` function of the parent class (`Computer`). In C++, calling a parent class's function is done as follows: `ParentClassName::function()` (see line 23 of Figure 17(a) and line 26 of Figure 17(b)).

7.5 Polymorphism



```
 1  /*
 2   * file:                 main.cpp
 3   * description:          My First C++ Program.
 4   * author:               Julien Tierny <tierny@telecom-paristech.fr>.
 5   * date:                 July 2014.
 6   */
 7
 8  #include                 <Desktop.h>
 9  #include                 <Laptop.h>
10
11  int main(int argc, char **argv) {
12
13
14    Desktop myDesktop;
15    Laptop myLaptop;
16
17    Computer  *computer1 = &myDesktop,
18              *computer2 = &myLaptop;
19
20    (*computer1).turnOn();
21    (*computer2).turnOn();
22
23    return 0;
24  }
25
```

Fig. 18 Example of polymorphism usage.

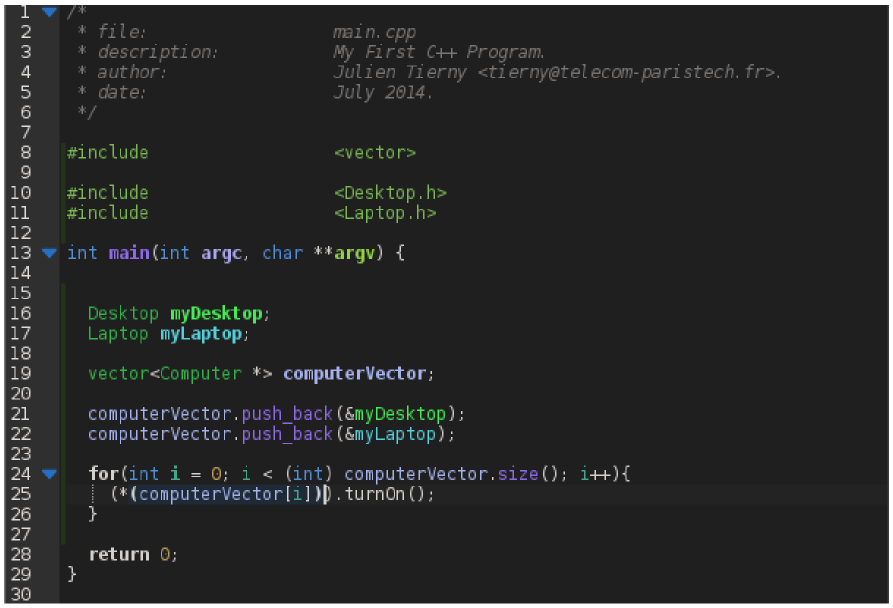
- Another appealing aspect of object-oriented programming is *Polymorphism*, which provides additional mechanisms to easily write general-purpose code;
- Figure 18 shows an example of polymorphism example:
  - A `Desktop` and a `Laptop` object are declared (line 14 and 15);
  - Then `Computer` pointers are declared and initialized on the addresses of the laptop and of the desktop (lines 17 and 18);
  - Thanks to polymorphism, when calling the function `turnOn()` on these two `Computer` objects, the system will automatically trigger the appropriate functions:

- line 20: the function `Desktop::turnOn()` will be automatically called;
- line 21: the function `Laptop::turnOn()` will be automatically called;

– Polymorphism is extremely useful in the development of large software, since it enables to process in a consistent manner several types of objects derived from a common class, while still automatically handling the specificities of the classes (here the booting procedure);

– Note that to trigger this polymorphic behavior on the function `turnOn()`, it has to be declared as a `virtual` function in the parent class (line 23 of Figure 13). This means that the implementation of this function might be specialized in the children classes.

8 Standard Template Library



```
/*
 * file:                    main.cpp
 * description:             My First C++ Program.
 * author:                  Julien Tierny <tierny@telecom-paristech.fr>.
 * date:                    July 2014.
 */

#include                    <vector>

#include                    <Desktop.h>
#include                    <Laptop.h>

int main(int argc, char **argv) {


  Desktop myDesktop;
  Laptop myLaptop;

  vector<Computer *> computerVector;

  computerVector.push_back(&myDesktop);
  computerVector.push_back(&myLaptop);

  for(int i = 0; i < (int) computerVector.size(); i++){
    (*(computerVector[i])).turnOn();
  }

  return 0;
}

```

Fig. 19 Example of usage of a `vector` container for the example given in Figure 18.

- The Standard Template Library (STL) is a standard C++ library which provides many interesting features on top of the original C++ language;
- It provides ready-to-use generic basic classes that are extremely useful;
- Among its key features, the STL provides many basic algorithms (for instance, the sorting of a list of values) as well as *containers*;

- Containers are very useful classes that are simply meant to contain other classes;
- Figure 19 provides an example of usage of such a container:
  - In particular, the `vector` container is used:
    - A vector is simply a dynamic array of objects;
    - To use the vector container, one first needs to include its corresponding header file (line 8);
    - To define a vector, one needs to use the following syntax: `vector<MyClass> myVector;`
    - Then, one can add elements to this array. For instance with the function `push_back()` which adds an item to the end of the array;
    - In this example, we are building an array of pointers to computers;
    - Then, one can simply iterate over the array to boot each of these computers. This is done with a `for` loop (line 24), by iterating from 0 to the size of the array (which is given by the function `size()`);
    - To access the i^{th} element of a vector, one proceeds as follows: `myVector[i]` (by using the "`[`" and "`]`" characters);
- Other examples of containers provided by the STL include:
  - `pair`: for pairs only of objects;
  - `queue`: for priority queues of objects;
  - `set`: for dynamically sorted lists of objects;
  - `map`: for an efficient implementation of a mapping from one set of objects to another;
  - etc.
- The STL library is extremely well documented, so do not hesitate to check its documentation online, both to have a view of the list of classes it provides and to understand how to use their functions:
  - http://www.cplusplus.com/reference/

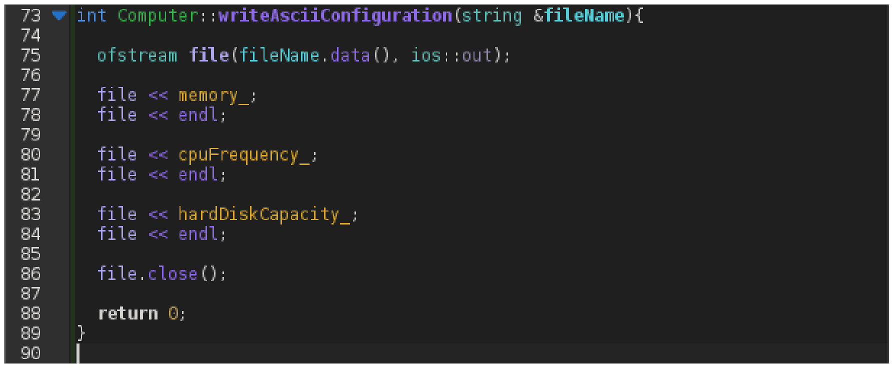
9 Input/Output Streams

- The reading of information from the hard drive disk and the writing to the disk is achieved in C++ with the notion of *stream*;
- A stream is special class of object that can receive (or generate) a flow of data;
- Two specific examples of streams have been used so far in the examples of this document:
  - `cout`: this symbolic stream represents the console output of the program. It has been used in our examples to print a message in the console (as off Figure 1);
  - `cerr`: this symbolic stream represents the errors sent to the calling console (used in Figure 17(a) line 19 for instance);

  – To use these specific streams, one needs to include the header `iostream` (see Figure 1, top).

- To receive data from a stream and to store it into a variable, one needs to use the operator >>;
- To send data from a variable to a stream, one needs to use the operator <<.

9.1 ASCII file streams

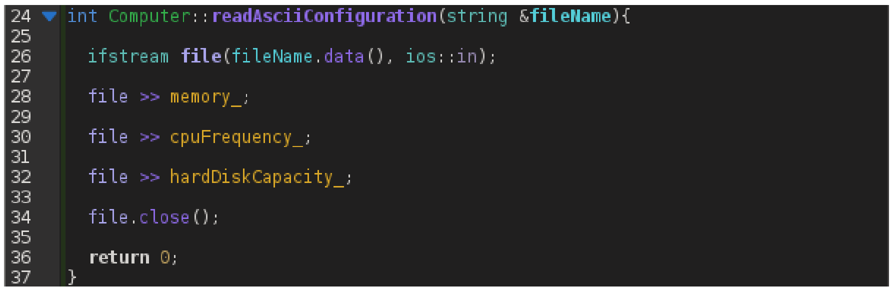


```
73 int Computer::writeAsciiConfiguration(string &fileName){
74
75   ofstream file(fileName.data(), ios::out);
76
77   file << memory_;
78   file << endl;
79
80   file << cpuFrequency_;
81   file << endl;
82
83   file << hardDiskCapacity_;
84   file << endl;
85
86   file.close();
87
88   return 0;
89 }
90
```

Fig. 20 Example of usage of output file streams to store information into a text file.

- To use input/output file streams, one needs to include the corresponding header: `#include <fstream>`;
- Figure 20 shows an example that makes use of file streams to store some information into a text file (ASCII, that can be read with any text editor):
  – In this example, a new function (`writeAsciiConfiguration`) has been added to the class `Computer`;
  – This function saves the configuration of the computer (CPU frequency, RAM memory and HDD capacity) to a text file, as follows:
    · One first needs to declare an object of the class `ofstream` ("*output file stream*", line 75). This declaration must be initialized with the following arguments:
      · Path to the file to write (`fileName.data()`);
      · Access mode (here writing in ASCII: `ios::out`).
    · Then from lines 77 to 84, the content of the variables of the `Computer` class are sent to the stream with the operator <<;
    · Note that the end of line character (`endl`) is also sent to delimit each variable in the output text file;

· Finally, the stream needs to be closed (line 86).



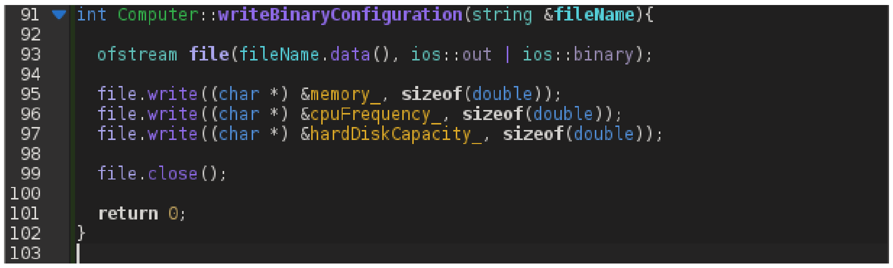
```
24 int Computer::readAsciiConfiguration(string &fileName){
25
26   ifstream file(fileName.data(), ios::in);
27
28   file >> memory_;
29
30   file >> cpuFrequency_;
31
32   file >> hardDiskCapacity_;
33
34   file.close();
35
36   return 0;
37 }
```

Fig. 21 Example of usage of the input file streams to read information from a text file.

- Figure 21 shows an example that makes use of file streams to read some information from a text file:
  - In this example, a new function (`readAsciiConfiguration`) has been added to the class `Computer`;
  - This function loads the configuration of the computer from a text file, as follows:
    - · One first needs to declare an object of the class `ifstream` ("*input file stream*", line 26). This declaration must be initialized with the following arguments:
      - · Path to the file to read (`fileName.data()`);
      - · Access mode (here reading in ASCII: `ios::in`).
    - · Then, from line 28 to 32, the content of the file is read and stored into the corresponding variables with the operator >>;
    - · Finally, the stream needs to be closed (line 34).

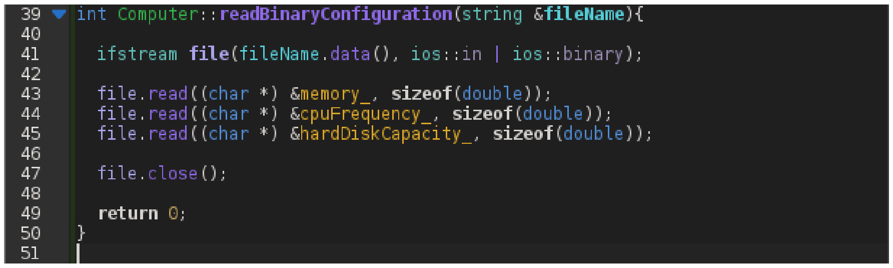
9.2 *Binary file streams*

- One can also use these file streams to handle *binary* files. In contrast to ASCII files, which can only support text information (thus requiring an implicit conversion, and possibly lack of precision), binary files enable to store the exact content of the variable (as it is represented in memory) to a file;
- The usage of binary file streams is highly similar to that of ASCII file streams;
- The following differences should be noted:
  - One should initialize the file streams in binary mode by adding the keywords "`| ios::binary`" to the access mode parameter (line 93 of Figure22(a) and line 41 of Figure22(b));



```
91  int Computer::writeBinaryConfiguration(string &fileName){
92
93    ofstream file(fileName.data(), ios::out | ios::binary);
94
95    file.write((char *) &memory_, sizeof(double));
96    file.write((char *) &cpuFrequency_, sizeof(double));
97    file.write((char *) &hardDiskCapacity_, sizeof(double));
98
99    file.close();
100
101   return 0;
102 }
103
```

(a)



```
39  int Computer::readBinaryConfiguration(string &fileName){
40
41    ifstream file(fileName.data(), ios::in | ios::binary);
42
43    file.read((char *) &memory_, sizeof(double));
44    file.read((char *) &cpuFrequency_, sizeof(double));
45    file.read((char *) &hardDiskCapacity_, sizeof(double));
46
47    file.close();
48
49    return 0;
50  }
51
```

(b)

Fig. 22 Example of usage of binary file streams for the examples shown in Figures 20 and 21.

– It is preferable to use specific functions for the reading/writing of binary information:
  · `write` (line 95 to 97 of Figure 22(a)): this function takes the following arguments:
    · A pointer to the variable which will send the information;
    · The number of bytes that should be written to file. In our example, we write variables one by one. Thus each call to the function writes the size of a double (which is given by `sizeof(double)`).
  · `read` (line 43 to 45 of Figure 22(b)): this function takes the same arguments as the function `write`.

References

1. B. Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.