LINMA2710 – Scientific Computing

P.-A. Absil

February 6, 2023

Contents

Chapter 1

Brief Introduction to C++ for Scientific Computing

In the context of this course, *scientific computing* refers to the application of computer simulations to solve scientific problems, with an emphasis on partial differential equations.

Widespread languages for scientific computing include Fortran, C, C++, Java, Maple, Matlab, and Python. Matlab is used in several courses of the Louvain School of Engineering. In the present course, the focus will be on C++ and, more secondarily, Python.

The reader of this chapter is assumed to have a fairly good knowledge of Matlab/Octave or Python NumPy. This knowledge is not essential to grasp the gist of these notes, but it is useful here and there.

1.1 Resources

The C++ official standard can be obtained from `https://isocpp.org/std/the-standard`. The official version is not freely available, but drafts are. The long (more than 1000 pages) document "Standard for Programming Language C++" is not intended for teaching; rather, it is meant for people writing C++ compilers and standard library implementations. Some more reader-friendly resources are listed at `https://isocpp.org/get-started`. The main sources of information for the present notes have been [PW12] and `http://www.cplusplus.com/doc/tutorial/`. An overview of the recent evolution of the language can be found in [Str20].

1.2 Why C++?

- Prevalence: Some of the most important scientific computing toolkits/libraries/environments are written in C++, for general purposes (Sandia's Trilinos), PDEs (Argonne's PETSc (mainly C), ExaHyPE, FEniCS), multi-physics simulations (preCICE), optimization (IPOPT 3.0

and beyond, g2o) linear algebra (Blaze, Ginkgo, SLATE, ChASE), machine learning (MLPACK), computational fluid dynamics (OpenFOAM, OpenLB), finite elements (libMesh,FreeFem++, MFEM, deal.II), simulation of dynamical systems (Siconos), networks (NetworKit), signal processing (FINUFFT)... See for example `https://en.wikipedia.org/w/index.php?title=List_of_numerical_libraries`.

- Speed: C++ can be much faster than interpreted languages such as Matlab. As an example, see the table towards the end of the technical report at `http://www.math.fsu.edu/~whuang2/papers/ROPTLIB.htm`, where a timing comparison between C++, Matlab and Python is reported for a joint matrix diagonalization problem. When the matrices are large, the BLAS[1] calls dominate, and timings are fairly similar. However, when the matrices are small, C++ becomes around 20 times faster! In this course, you will get to conduct experiments that produce similar findings.

- Code reuse: More modern, user-friendly and equally capable languages may exist, but there is currently a large C++ code base that will not go away soon.

- Influence: Several languages are modeled (at least in part) on C/C++, e.g., AMPL, Java, and PHP. See `https://en.wikipedia.org/wiki/List_of_C-family_programming_languages` for a list, keeping in mind that the notion of a language influencing another is somewhat loose.

- Lingua franca: C++ often serves as a common language between programmers who work in different languages.

1.3 What is C++?

C++ is a programming language. It differs from Matlab in several respects, notably:

- C++ is a more general-purpose programming language. For example, several of the most popular video games are programmed (at least in part) in C++. See also the list of applications of C++ compiled by Bjarne Stroustrup (the creator of C++): `https://www.stroustrup.com/applications.html`.

- C++ is a typed language. All variables must be declared explicitly before being used, and a function definition must specify the type of the parameters. As we will see, this has several consequences: function overloading, need for templates...

- C++ compilers and development environments are freely available.

[1]BLAS are Basic Linear Algebra Subprograms. They are called by Matlab, Python (at least if a suitable distribution is used, such as Anaconda or Canopy), and C++ to perform operations such as matrix multiplications.

1.4 Integrated Development Environments

Programming in C++ can be considerably facilitated by a suitable integrated development environment (IDE) or a source-code editor. Popular choices include Eclipse (primarily for developing Java applications but may also be used for many other languages), Visual Studio Code (source-code editor made by Microsoft for Windows, Linux and macOS), Atom (source-code editor developed by GitHub), Code::Blocks, CLion and Emacs.

The rest of this section focuses on Visual Studio Code (VS Code) and Eclipse. You do not have to install both; you can choose just one of them. You can even install neither of them if you have a preference for another alternative. For example, you may find that Code::Blocks has a lower entrance barrier; feel free to use it instead. However, Visual Studio Code and Eclipse seem to be better at spotting syntax errors, and to have more active communities. On macOS, you may prefer Xcode.

Exercise 1

Build and run a "Hello World" program in your chosen IDE. For VS Code and Eclipse, instructions are given in the rest of this section.

1.4.1 Installing and using Visual Studio Code on Ubuntu Linux

Here is how the VS Code installation can be performed in Ubuntu (tested on Ubuntu 18.04.6 LTS but a similar procedure should work with several other Linux distributions). A basic familiarity with Linux and the command line is likely to be necessary. To get the command line in plain Ubuntu, click the Ubuntu logo and type *terminal*. In the present document, the command line prompt will be denoted by

```
\$
```

To install VS code, follow the instructions at `https://code.visualstudio.com/docs/setup/linux`. This may lead you to download a package and run commands similar to

```
\$ sudo apt install ./code_1.63.2-1639562499_amd64.deb
\$ code
```

As an outcome, the VS Code window should appear.

Check if `g++` (the command that calls the GNU C++ compiler) is installed:

```
\$ g++ --version
```

If not, install it:

```
\$ sudo apt install g++
```

Then follow the instructions at `https://code.visualstudio.com/docs/languages/cpp`. They will let you install the C/C++ extension of VS Code and guide you to your first "Hello World" program.

1.4.2 Installing and using Visual Studio Code on Microsoft Windows 10

This section is based on `https://code.visualstudio.com/docs/languages/cpp` and several related VS Code help pages, where more detailed information can be found.

First install Visual Studio Code, as follows. Download the (stable) User Installer from `https://code.visualstudio.com/docs/languages/cpp`. Run the .exe file. Accept default choices. If the default color theme does not suit you, change it with File > Preferences > Color Theme.

In VS Code, install the C/C++ extension, as follows. Select the Extensions view icon on the Activity bar (which should be on the left of your VS Code window) or use the keyboard shortcut Ctrl+Shift+X. Search for "C++". Select Install for the C/C++ extension. The installation takes a few seconds.

The rest of `https://code.visualstudio.com/docs/languages/cpp` should be easy (though lengthy) to follow. Here come a few tips. If you have a Beligian keyboard, the shortcut to open the Integrated Terminal in Visual Studio Code is Ctrl+Shift+ù. If the installation procedure for MinGW displays an error or stops prematurely, try again; the "Installing Files" step of the installation of MinGW should take several minutes. It is recommended to accept the default choice for directory names. If `g++ --version` does not work in the Integrated Terminal of VS Code, try first to restart VS Code. You can also try `g++ --version` in Window's PowerShell (Win-S and search for PowerShell), where you can also use `$Env:Path` to retrieve the PATH variable. To create the "Hello world" file, open a folder with File > Open Folder (you can also create a new folder as you usually do in Windows), then File > New File.

Once you are able to successfully build and run your "Hello World", it is recommended to go through `https://code.visualstudio.com/docs/cpp/config-mingw`.

1.4.3 VS Code tips and tricks

To create the "Hello world" file, open a folder with File > Open Folder (you can also create a new folder as you usually do in Linux), then File > New File.

Ctrl+Shift+P opens a list of shortcuts. If you have a Beligian keyboard, the default shortcut to open the Integrated Terminal may not work, but you can customize the shortcuts under File > Preferences > Keyboard Shortcuts.

VS Code is highly customizable. See `https://code.visualstudio.com/docs/getstarted/settings` for the setting editor and the settings file location.

If you want to work simultaneously with multiple project folders: File > Add Folder to Workspace...

In large projects, the ability to set and jump to bookmarks can be of great help. One possibility is to install the "Numbered Bookmarks" extension: Ctrl+Shift+X, search for "Numbered Bookmarks". To customize how Numbered Bookmarks navitage through all files, see `https://marketplace.`

visualstudio.com/items?itemName=alefragnani.numbered-bookmarks (and see above for the settings file location).

To run a .cpp file right away (without having to select the Terminal > Run Build Task command then go the the terminal): Select the Extensions view icon on the Activity bar (or Ctrl+Shift+X). Look for "Code Runner" and install it. Open a .cpp file. Click on the "play" button on the top right (or Ctrl+Alt+N) to run the code file of current active text editor.

1.4.4 Installing and using Eclipse on Ubuntu Linux

Here is how the Eclipse installation can be performed in Ubuntu (tested on Ubuntu 20.04 but a similar procedure should work with several other Linux distributions). A basic familiarity with Linux and the command line is likely to be necessary. To get the command line in plain Ubuntu, click the Ubuntu logo and type *terminal.* In the present document, the command line prompt will be denoted by

```
\$
```

Get Eclipse IDE for C/C++ Developers from https://www.eclipse.org/downloads/packages/ and cd to your download director.

```
$ sudo tar xf eclipse-cpp-2021-06-R-linux-gtk-x86_64.tar.gz -C /opt
$ sudo ln -s /opt/eclipse/eclipse /usr/local/bin/
$ sudo apt install g++
$ sudo apt install make
```

Now you are ready to play around with Eclipse. Start it with

```
\$ eclipse
```

You should be able to fairly quickly[2] find out how to load and run a basic "Hello World" project.[3]

Fans of the Emacs editor will be glad to hear that Eclipse has a built-in key-binding configuration that emulates emacs-style key bindings. Open Window > Preferences, find "Keys", and choose the Emacs scheme. The emulation is not complete though. For example, the `C-x r` commands for register manipulation do not work. For this, see the *Emacs+* Eclipse extension.

[2]Getting started with Eclipse for C/C++ is not completely straightforward. However, if you are not able to find your way within about one hour to compile and run a "Hello World" project, then you probably do not have the basic computer skills required to take this course.

[3]Here are some hints for Eclipse 2021-06. New "hello world" project: File > New > C/C++ Project > C++ Managed Build > Hello World C++ Project > Cross GCC. In the Build Project sub-window, right click on your new project and choose Build Project. In the "Launch Configuration" drop down menu (around the top left of the Eclipse window): New Launch Configuration > Debug & C/C++ Application. Click on the "Run" button (around top left) and look at the output in the Console window. Now you can modify your code, hit the Build button (hammer icon), and hit the Run button (play icon).

1.4.5 Installing and using Eclipse on Microsoft Windows 10

First you will need a gcc compiler. There are several options; the suggested one is MinGW. To install it, you can follow the procedure at `https://code.visualstudio.com/docs/cpp/config-mingw`.

If Eclipse is already installed on your Windows machine, do as instructed at `https://eclipse.org/cdt/downloads.php`.

Otherwise, download Eclipse from `https://eclipse.org/downloads/` and run the installer (eclipse-inst-jre-win64.exe) as a user (double left click), not as administrator. Take note of the path where Eclipse is installed as you may have to find eclipse.exe therein and open it in order to use Eclipse.

The Eclipse installer may complain: "Java for Windows Missing". Install the latest Oracle JDK as instructed (jdk-8u121-windows-x64.exe). The installation may take a few seconds to start and may seem frozen for a while during the process. If it looks frozen, it is perhaps because it has opened a window below other windows that you will only notice in the window list. When the JDK is installed, run the Eclipse installer again.

In the Eclipse installer, choose "Eclipse IDE for C/C++ Developers". The installation may take a couple of minutes.

Start Eclipse. File > New > C/C++ Project > C++ Managed Build. Choose, e.g., "helloworld" as project name. Select "Hello World C++ Project" and "MinGW GCC". If nothing seems to happen, then click on the tiny icon on the left that displays "C/C++" when you hover over it. Right click on "helloworld" (not helloworld.cpp) in the Project Explorer (in the left of the Eclipse window). Choose "Build Project". Something should happen in the Console tab at the bottom of the Eclipse window, ending with "Build Finished. 0 errors, 0 warnings". Right click again on "helloworld", "Run as", "Local C/C++ Application". "!!!Hello World!!!" should appear in the Console tab. The hammer and "play" icons on the top left of the Eclipse window can serve as shortcuts.

1.5 Hello World!

Here is a basic "Hello World" example.

Listing 1.1: Hello World

```cpp
#include <iostream>
int main() {
        std::cout << "!!!Hello␣World!!!" << std::endl; // prints !!!Hello World
            !!!
}
```

To compile and run it without an IDE, first save it in a file, say `hello.cpp`. Then compile it with

```
\$ g++ -o hello hello.cpp
```

Then run the excutable

```
\$ ./hello
```

See also section 1.6 on makefiles.

1.5.1 Discussion

`#include` in Listing 1.1 is a directive. The line

```
#include <iostream>
```

is interpreted before the compilation of the program itself begins. It instructs the preprocessor to include the header `iostream`, which provides the objects `std:cout` and `std:endl`, among others. As part of the *C++ standard library*, these objects are part of the `std` namespace, hence their `std::` prefix.

Prepending `std::` can be cumbersome. Here is a remedy:

```
#include <iostream>
using namespace std;
int main() {
        cout << "!!!Hello␣World!!!" << endl; // prints !!!Hello World!!!
}
```

Blank lines have no effect on a program. They can be used to improve readability:

```
#include <iostream>

int main() {
        std::cout << "!!!Hello␣World!!!" << std::endl; // prints !!!Hello World
            !!!
}
```

`int main()` initiates the declaration of a function, called `main`. The execution of all C++ programs begins with the function called `main`, regardless on where it is located within the code. Here the output of `main()` is of type `int`, i.e., an integer. We could have inserted

```
return 0;
```

in the body of `main()`, but `main()` can also be left without a return value in which case it defaults to returning 0. This has no practical importance in our context.

The braces `{ }` delimitate the function's body.

The line

```
        std::cout << "!!!Hello␣World!!!" << std::endl; // prints !!!Hello World
            !!!
```

is a C++ statement. `std::cout` identifies the standard character output device. `<<` is the *insertion operator*: what follows it is inserted into `std::cout`. `"!!!Hello World!!!"` is the content inserted to the standard character output. The semicolon `;` marks the end of the statement. Finally, the part `// prints !!!Hello World!!!` is a comment. Commenting in C++ goes as follows:

```
// This is a single-line, ``C++-style'' comment
/* and this is a multi-line, ``C-style'' comment, which
goes on possibly over several lines. */
```

C++ is a compiled language. The command

```
\$ g++ -o hello hello.cpp
```

translates the computer program `hello.cpp` into the machine code `hello`, which is then executed with

```
\$ ./hello
```

(Just `hello`, without `./`, will also work if the current directory (`.`) is in your `$PATH` environment variable.)

1.5.2 Pitfalls

The next programs generate an error. Find out why. If you try them out in Eclipse, you will immediately realize the usefulness of an IDE.

```
#include <iostream>
int main() {
        std::cout << "!!!Hello␣World!!!" << std::endl // prints !!!Hello World
            !!!
}
```

```
#include <iostream>
int main() {
        cout << "!!!Hello␣World!!!" << endl; // prints !!!Hello World!!!
}
```

1.6 Makefiles

Makefiles are a convenient tool to (re)compile projects that consist of several files. Mastering makefiles is not necessary as long as an IDE (such as Eclipse, see Section 1.4) is available.[4] However, whereas an IDE is easy to install on your laptop, it may not be available or impractical to use on a remote computer, e.g., a CÉCI cluster. A compact tutorial on creating and using a makefile can be found in [PW12, §6.2.4].

1.7 Variables and types

C++ is *statically typed.* Each variable must be declared and its type must be specified.

Here is an example:

[4]Actually, Eclipse uses a (complicated) makefile under the hood. It can be found in your chosen workspace directory.

```
1  #include <iostream>
2  // #include <string>
3  // using namespace std;
4
5  int main() {
6    double a = 3.1; // declares a variable of type double with identifier a and
         initializes it to the value 3.1
7    double b (3.9); // likewise, with the "constructor initialization" syntax
8    double c {7.1}; // likewise, with the "uniform initialization" syntax
9    // double 2a = 3; is not allowed: identifiers must begin with a letter
10   int i = a; // narrowing cast
11   int j = b; // narrowing cast
12   double d = i; // widening cast
13   double a_div2_wrong = 1/2*a; // try to understand what is wrong here
14   std::string s="test";
15   double x; // declare x without specifying a value
16   double A = 3.2; // C++ is a case-sensitive language
17
18   std::cout << "a␣=␣" << a << std::endl;
19   std::cout << "b␣=␣" << b << std::endl;
20   std::cout << "c␣=␣" << c << std::endl;
21   std::cout << "i␣=␣" << i << std::endl;
22   std::cout << "j␣=␣" << j << std::endl;
23   std::cout << "d␣=␣" << d << std::endl;
24   std::cout << "a_div2_wrong␣=␣" << a_div2_wrong << std::endl;
25   std::cout << 1/2 << std::endl;
26   std::cout << "(int)a␣=␣" << (int)a << std::endl;
27   std::cout << "s␣=␣" << s << std::endl;
28   std::cout << "x␣=␣" << x << std::endl;
29   std::cout << "A␣=␣" << A << std::endl;
30 }
```

Here is the output:

```
a = 3.1
b = 3.9
c = 7.1
i = 3
j = 3
d = 3
a_div2_wrong = 0
0
(int)a = 3
s = test
x = 2.07436e-317
A = 3.2
```

In line 6, `double` is the *type*, `a` is the *identifier*, and `3.1` becomes the *value* of the variable.

Exercise 2

Explain why `a_div2_wrong = 0`. Check the value of `a` after the following initializations:

`int a = 1.1;`

`int a = 1.9;`

```
int a = -1.1;
int a = -1.9;
```

1.8 Control flow

Flow control statements allow to repeat segments of code, take decisions and bifurcate.

1.8.1 Selection statements: if, else if, else

Check this example:

```
#include <iostream>
// using namespace std;

int main() {
        int x = 100; // initialize integer variable x
        if (x == 100)
        {
                std::cout << "x␣is␣100" << std::endl;
        }
        else
        {
                std::cout << "x␣is␣not␣100" << std::endl;
        }
        if (x > 0)
          std::cout << "x␣is␣positive";
        else if (x < 0)
          std::cout << "x␣is␣negative";
        else
          std::cout << "x␣is␣0";
        return 0;
}
```

The syntax is `if (condition) statement1 elseif (condition) statement2 else statement3`. If there is more than a single statement per case, then they must be grouped into blocks enclosed with braces; for a single statement, braces are not necessary but the coding style may require them nevertheless (see Section 1.11).

Exercise 3

Try this after the code above:

```
        if (x=10)
        {
                std::cout << "Entered␣\"if\"␣body␣\n";
        }
        std::cout << "x␣=␣" << x << "\n";
```

What happened? Try something equivalent in Python.

Answer of exercise 3

`x=10` is an assignment, whereas `x==10` is a comparison. In Python, assignments in expressions are not allowed.

1.8.2 Another selection statement: switch

See [PW12, §2.5] or the relevant section in `http://www.cplusplus.com/doc/tutorial/control/`. While the `switch` statement is convenient in Matlab, it may be considered less so in C++ due to its peculiar syntax. Resorting to `if` and `else if` may be preferable.

1.8.3 Iteration statements

While and do-while

Examples:

```
        // WHILE LOOP
        int n = 0; // initialize integer variable n
        while (n<3) // while condition
        {  // begin body of while loop
                std::cout << n << ", "; // prints the value of n
                ++n;  // increases n by 1
        }  // end body of while loop
        std::cout << "while loop is over" << std::endl;

        //DO-WHILE LOOP
        std::string str; // declare variable str of type string
        do { // begin do-while loop
                std::cout << "Enter \"bye\" to stop" << std::endl;
                std::getline(std::cin,str); // store user input in str
        } while (str != "bye"); // end do-while loop
        std::cout << "do-while loop is over" << std::endl;
```

`\"` is an *escape sequence*: since the quotes are reserved to mark the end of the string, they cannot be represented directly in the string.

The do-while loop has no equivalent in Matlab and Python. But there are workarounds.

Exercise 4

Implement the do-while loop above in Python.

Answer of exercise 4

```
while True:
    str = input("Enter \"bye\" to stop: ")
    if str == "bye":
        break
```

For loops

Let us introduce `for` loops with a basic implementation of the composite Simpson rule for numerical integration (inspired from [TLNC10, Alg. 6.3]). The example uses a function; see Section 1.12 for details.

```cpp
#include <iostream> // for cout
#include <math.h> // for pow
using namespace std;

double f(double x) { // the function that we want to integrate
        return pow(x,2); // returns x^2
}

int main() {
        double a = 0; // left endpoint of the integration interval
        double b = 1; // right endpoint
        int n = 1e3;  // number of subintervals

        double s = 0;
        double h = (b-a)/n; // size of the subintervals
        double x_minus, x_plus, x;
        for (int i=1; i<=n; i++)
        {
                x_minus = a + (i-1)*h;
                x_plus = a + i*h;
                x = (x_minus + x_plus)/2;
                s = s + 1./6.*f(x_minus) + 4./6.*f(x) + 1./6.*f(x_plus);
        }
        s = h*s;

        cout << s << endl; // display result
}
```

Observe the dots in “`1./6.`”.

Exercise 5

What happens if the dots are removed? Explain what you observe. Hint: try displaying the result of `1/6` using `std::cout`.

Answer of exercise 5

The result of `1/6` is 0. This is an integer division.

Exercise 6

Implement the same algorithm in Matlab/Octave or Python and compare timings. What do you observe?

Hint: To time the execution of an executable, say `Simpson`, you can use

```
\$ time ./Simpson
```

You should observe that C++ is much faster. This stems from the fact that your C++ is *compiled* into machine code, whereas your Matlab script is *interpreted.*

Range-based for loops

The syntax is `for ( declaration : range ) statement;`, where `range` is a sequence of elements of type `declaration`.

Example:

```
    //RANGE-BASED "FOR" LOOP
    std::string str2 {"Hello!"}; // initializes variable str
    for (char c : str2) // the elements in a string are of type char
    {
        std::cout << "[" << c << "]";
    }
    std::cout << '\n';
```

The above piece of code requires C++11 (a version standard of C++ approved in 2011) or higher for the string variable initialization. This may already be the case by defaut with your C++ compiler. If not, here are instructions. In the Eclipse IDE, right-click the project and, in Properties -> C/C++ Build -> Settings -> Tool Settings -> GCC C++ Compiler -> Miscellaneous -> Other Flags, put `-std=c++11` at the end. In VS Code, insert `-std=c++11` in `args` in file `tasks.json`; more about `tasks.json` in `https://code.visualstudio.com/docs/cpp/config-mingw`.

In order to check which version standard of C++ your compiler is using, you can compile and execute a program with the following line:

```
std::cout << __cplusplus << std::endl;
```

If the value printed can be, e.g., 201103 (C++11), 201402 (C++14), 201703 (C++17).

Exercise 7

Using the material of Section 1.10, produce a similar example with an array instead of a string. Note that arrays, like strings, are ranges.

Answer of exercise 7

```
    int array[2] {8, 9};
    for (int i : array)
    {
        std::cout << "[" << i << "]";
    }
    std::cout << std::endl;
```

Exercise 8

Write a loop to find the smallest integer i such that $1 + 2^i \neq 1$ in double precision. The value 2^i for the obtained i is often termed the *machine precision* (in double precision).

1.8.4 Jump statements

See the relevant section in `http://www.cplusplus.com/doc/tutorial/control/`. Advice: avoid them as much as possible, and stay away from `goto` statements.

1.9 Pointers

Pointers are variables whose value is the address of another variable. One purpose of pointers is the dynamic allocation of memory to store arrays, as we will see in Section 1.12.3. Another purpose is to pass arguments to functions by reference an not by value; see also Section 1.12.1.

Here is an illustrative example.

```
int i = 7; // declares a variable of type int with identifier i and
    initializes it to the value 7
int j; // declares a variable of type int with identifier j
int * p; // declares pointer p that points to an integer
p = &i; // assigns the address of variable i to pointer p
j = *p; // assigns to variable j the value pointed to by p
i = 8; // assigns the value 8 to variable i
cout << "i␣=␣" << i << endl; // prints the value of variable i
cout << "*p␣=␣" << *p << endl; // prints the value pointed to by p
cout << "j␣=␣" << j << endl; // prints the value of variable j
cout << "p␣=␣" << p << endl; // prints the value of pointer p (the address of
    i)
cout << "&p␣=␣" << &p << endl; // prints the address of pointer p
```

The output differs between runs because the addresses are assigned at runtime. A possible output is

```
i = 8
*p = 8
j = 7
p = 0x7fffa8f42f38
&p = 0x7fffa8f42f40
```

We thus have the following situation:

Identifier	p	i
Type	`int *`	`int`
Address	`&p` = 0x7fffa8f42f40	`&i` = 0x7fffa8f42f38
Value	`p` = 0x7fffa8f42f38	`i = *p` = 8

Table 1.1: Illustration of pointer, dereference and address-of.

Observe that the value of pointer `p` is the address of variable `i`.

Note that the `*` symbol has two different meanings in the example above. In the declaration

```
int * p;
```

the asterisk `*` means that we declare a pointer. In the assignment

```
j = *p;
```

the asterisk is the *dereference operator*: `*p` is the value pointed by pointer `p`. Sticking to a well-chosen coding style may help avoid confusions; see Section 1.11.

The example above also uses the ampersand sign `&`, known as the *address-of operator*: `&p` is the address of variable `p`. Do not confuse the address of `p` (`&p`), the value of `p`, and (since `p` is a pointer) the value pointed to by `p` (`*p`). These are three different things, as the table above shows.

Exercise 9

Why is the following not correct?

```
int * q;
q = &q;
```

Can a pointer point to itself?

Answer of exercise 9

It is not correct because `q = &q` makes `q` point to a pointer, not to an int. However, a pointer can point to itself, using the following trick. (Source: http://stackoverflow.com/questions/2532102/can-a-pointer-ever-point-to-itself)

```
int ** q;
q = (int **)&q;
cout << "The␣address␣of␣q␣is␣" << &q << endl;
cout << "q␣holds␣the␣address␣" << q << endl;
cout << "The␣value␣at␣" << q << "␣is␣" << *q << endl; // the *q is why we made
     q an int **
```

The output is

```
The address of q is 0x7ffe67ae8c88
q holds the address 0x7ffe67ae8c88
The value at 0x7ffe67ae8c88 is 0x7ffe67ae8c88
```

Note that `&q` is actually of type `int ***`, hence it needs to be cast to type `int **` to be assigned to `q`.

1.10 Arrays

In contrast with MATLAB (an abbreviation for “matrix laboratory”), C++ has not been designed for matrix computations. Fortunately, there are fixes: an example of C++ class for linear algebra calculations can be found in [PW12, Ch. 10], and more sophisticated C++ template libraries are available (such as Armadillo and Eigen). Before considering such libraries, let us first have a look at the most basic way of defining a vector and a matrix in C++, namely arrays.

Check this out:

```
1  #include <iostream>
2  // using namespace std;
3
4  int main() {
5    double array1a[2]; // declares a vector of length 2
6    double array1b[4] = {5., 2.2, 3.3, 4.4}; // vector initialization
7    double array2a[2][3]; // declares a matrix of size 2-by-3
8    double array2b[2][3] = { {1., 2., 3.}, {2, 1, 3.} }; // matrix initialization
9    array1a[0] = 2.1; // indexing begins from zero
10   array1a[2] = 7.7; // this does not generate an error
11   array2a[1][2] = 1.3;
12   std::cout << "array1a[0]␣=␣" << array1a[0] << std::endl;
13   std::cout << "array1a[1]␣=␣" << array1a[1] << std::endl;
14   std::cout << "array1a[2]␣=␣" << array1a[2] << std::endl;
15   std::cout << "array1b[0]␣=␣" << array1b[0] << std::endl;
16   std::cout << "array2a[0][0]␣=␣" << array2a[0][0] << std::endl;
17   std::cout << "array2a[0][1]␣=␣" << array2a[0][1] << std::endl;
18   std::cout << "array2a[1][2]␣=␣" << array2a[1][2] << std::endl;
19   // Arrays are closely related to pointers, as we now illustrate:
20   std::cout << "array1b␣=␣" << array1b << std::endl;
21   std::cout << "&array1b[0]␣=␣" << &array1b[0] << std::endl;
22   std::cout << "&array1b[1]␣=␣" << &array1b[1] << std::endl;
23   std::cout << "&array1b[2]␣=␣" << &array1b[2] << std::endl;
24   std::cout << "array1b␣+␣1␣=␣&array1b[1]␣=␣" << array1b + 1 << std::endl;
25   std::cout << "*(array1b␣+␣1)␣=␣array1b[1]␣=␣" << *(array1b + 1) << std::endl; //
26   return 0;
27 }
```

The output is

```
array1a[0] = 2.1
array1a[1] = 3.11264e-317
array1a[2] = 7.7
array1b[0] = 7.7
array2a[0][0] = 4.94066e-324
array2a[0][1] = 6.953e-310
array2a[1][2] = 1.3
array1b = 0x7ffe582f9a20
&array1b[0] = 0x7ffe582f9a20
&array1b[1] = 0x7ffe582f9a28
&array1b[2] = 0x7ffe582f9a30
array1b + 1 = &array1b[1] = 0x7ffe582f9a28
*(array1b + 1) = array1b[1] = 2.2
```

Comments:

1. In contrast with Matlab, C++ uses zero-based numbering: see code line 9.

2. Array entries get a value even when they are not initialized.

3. The assignment `array1a[2] = 7.7;` has written a value outside `array1a`. This has affected `array1b`; observe the value of `array1b[0]`.

4. `array1b` (for example) is actually a pointer that points to the first element of the array.

5. The brackets are actually a deferencing operator known as *offset operator*, as the output `*(array1b + 1) = array1b[1] = 2.2` illustrates.

6. `*(array1b + 1)` is an example of pointer arithmetics. Only additions and subtractions are allowed.

7. The output suggests that the memory size of a `double` variable is 8 bytes.

8. Statements such as code line 5 require the size of the array (here 2) to be known at the coding stage. For dynamic memory allocation, see Section 1.12.3.

If you find that the absence of boundary checking is unacceptable, you can resort to a template library that does boundary checking, e.g., Armadillo (`http://arma.sourceforge.net/`), or you can create your own class as we will do in Section 1.16.2.

1.11 Coding style

C++ can be confusing. Here are examples:

1. In `blah bleh(int(x));` we create an instance (object) `bleh` of class `blah` by passing to the constructor the value of `x` cast into an integer. But `blah bleh(int x);` is the prototype of a function called `bleh` that takes an integer variable as input and returns a variable of type `blah`.

2. In Section 1.9, we have seen that the asterisk (*) takes two different meanings: in `int * p;` it means that we declare a pointer to an integer, and in `j = *p` we assign to variable `j` the value pointed to by `p`. Writing the pointer declaration as `int * p;` and not as the equally valid `int *p;` is a matter of style that may help avoid confusions.

Perhaps you can think of other examples.

Adhering to a well-chosen coding style may help avoid confusions. Popular standards are JSF (`http://www.stroustrup.com/JSF-AV-rules.pdf`) and the Google C++ Style Guide (`https://google.github.io/styleguide/cppguide.html`). See also [PW12, §6.6].

Once you have chosen a style, try to be as consistent as possible. This makes it easier to automate code modification.

Here are a few suggested rules (some of which will make sense only later in this course), inspired from [PW12, §6.6]:

1. Code indentation.

2. Long lines of code split across multiple lines.
3. Meaningful variable names without being too long.
4. Variables declared close to where they are used. (But we may want to avoid keeping constructing an object in a loop; constructing the object once and for all before the loop may be considerably more efficient.)
5. Pointer names begin with `p`, e.g., `p_return_result` or `pLastResult`.
6. Function (as well as class member function) names are in camel-case and the first word is a verb, e.g., `GetSize()`.
7. Names of arguments to functions are in camel-case but begin in lower-case, e.g., `firstDimension`.
8. Class data which are private or protected start with `m`, e.g., `mSize`. In principle, all class data is private; the way to access it is by member functions.
9. Class names are also in camel-case.
10. Comment the code profusely.

1.12 Functions

Whereas functions are indicated by a keyword in Matlab (`function`) and Python (`def`), there is no such keyword in C++. The syntax is

```
type name(type1 identifier1, type2 identifier2, ...) { statements }
```

where `type` is the *return type* (the type of the value returned by the function), `name` is the identifier of the function, `type1` is the type of the first *parameter*, `identifier1` is the identifier of the first parameter, and `statements` is the function's body.

Example:

```
#include <iostream>

int Add(int a, int b); // function prototype, convenient to define the
    function after main()

int main()
{
  int z;
  z = Add(5, 3); // calls function "Add" with arguments 5 and 3, and assigns
      returned value to z
  std::cout << "The␣result␣is␣" << z;
}

int Add(int a, int b) // function definition
{  // start of function body
  int r;
15    r = a + b;
16    return r;
17  } // end of function body
```

Pay attention to the "`;`" signs. Try forgetting one in the Eclipse IDE to see how useful an IDE can be.

To represent the absence of return value or parameter, use the special type `void`.

```
1  void PrintMessage(void)
2  {
3    std::cout << "I’m␣a␣function!";
4  }
5
6  int main()
7  {
8    PrintMessage();
9  }
```

Note that the return type of `main()` is `int`. Actually, `main()` returns the value zero when it ends normally.

Line 3 is a function *prototype* (or *declaration*). (It is also sometimes, including in these notes, called a function *signature*, though according to some authors a signature should not include the name of the parameters, nor of the function, and not even the return type.)

The function *definition* (or *implementation*) goes from line 12 to line 17.

1.12.1 Arguments passed by value and by reference

In C++, arguments are passed *by value*.

```
1  void SetTo2_fail(int i)
2  {
3          i = 2;
4  }
5
6  int main()
7  {
8    int a = 1;
9    SetTo2_fail(a);
10   std::cout << "a␣=␣" << a << "\n";
11 }
```

The outcome is `a = 1`. In line 9, the function is passed the *value* of `a`, namely `1`. Variable `a` itself is *not* passed to the function. Variable `a` is thus not modified.

If we want the function to modify variable `a`, then we must pass the *address* of `a` to the function. Here is a way:

```
1  void SetPointedTo2(int * p_i)
2  {
3          *p_i = 2; // stores 2 in memory with address p_i
4  }
5
```

```
int main()
{
  int c = 1; // initializes int variable c
  int * p_c; // declares pointer p_c that points to int
  p_c = &c; // p_c stores the address of c
  SetPointedTo2(p_c);
  std::cout << "c␣=␣" << c << "\n";
}
```

An alternative to using pointers is to use *reference variables*, as follows:

```
void SetTo2(int& i)
{
        i = 2;
}

int main()
{
  int b = 1;
  SetTo2(b);
  std::cout << "b␣=␣" << b << "\n";
}
```

The ampersand (`&`) in the declaration `void SetTo2(int& i)` indicates that the variable is passed *by reference*. In the call `SetTo2(b);`, the variable `b`—not its value—is passed to the function and becomes known as `i` in the function.

The ampersand (`&`) has thus two meanings: "address-of" (as in `&a`, see Section 1.9), and "reference to" (as in `int&`). (As we saw in Section 1.9, `*` has also two meanings.)

For the sake of space and time efficiency, we may want to pass by reference a variable that must not be modified. To guarantee this, the parameters can be qualified as constant, using the `const` keyword. Here is a (fairly pointless but simple) example:

```
int AddByRef(const int& a, const int& b)
{
        int r;
        r = a + b;
        // a++; // this would generate an error
        return r;
}
```

The many subtleties of the `const` keyword are discussed in `http://duramecho.com/ComputerInformation/WhyHowCppConst.html`.

Exercise 10

It is possible to declare a reference to a variable, like this: `int& ref_to_a = a;`. Try to modify `ref_to_a` and `a`. What happens?

Does `int& ref_to_int = 6;` work? Why (not)?

1.12.2 Default values in parameters

Like in Python, functions parameters can be given default values in C++.

```cpp
int AddDefault(int a, int b=2, int c=3)
{
        return a+b+c;
}

int main()
{
  std::cout << "AddDefault(4,␣1)␣returns␣" << AddDefault(4, 1) << "\n";
  std::cout << "AddDefault(4,␣{},␣2)␣returns␣" << AddDefault(4, {}, 2) << "\n";
}
```

The output is

```
AddDefault(4, 1) returns 8
AddDefault(4, {}, 2) returns 6
```

1.12.3 Passing and returning arrays

The next example, drawn from [PW12, §5.2.2], shows a function that dynamically allocates memory for a matrix.

```cpp
// Function to allocate memory for a matrix dynamically
double ** AllocateMatrixMemory(int numRows, int numCols)
{
        double** matrix; // "matrix" is an array of pointers to double, hence "**"
        matrix = new double* [numRows]; // allocate memory for an array of pointers (to double) of length numRows
        // Actually, the variable matrix contains the address of the pointer matrix[0]
        for (int i=0; i<numRows; i++)
        {
                matrix[i] = new double [numCols]; // allocate memory for an array of double of length numCols
                // Actually, the variable matrix[i] contains the address of matrix[i][0]
        }
        return matrix;
}

// Function to free memory of a matrix
void FreeMatrixMemory(int numRows, double** matrix)
{
        for (int i=0; i<numRows; i++)
        {
                delete[] matrix[i]; // note that numCols is not needed
        }
        delete[] matrix;
}

int main()
{
  // Dynamic memory allocation example:
  double** A;
  A = AllocateMatrixMemory(5, 3);
  A[0][1] = 2.0;
  A[4][2] = 4.0;
  FreeMatrixMemory(5, A); // it is crucial to delete dynamically-allocated
      memory
}
```

Exercise 11

Write a function that prints all the elements of a 2-dimensional array. Hint: you will need to pass the dimensions of the array.

Exercise 12

Modify `AllocateMatrixMemory` and `FreeMatrixMemory` for 3-dimensional arrays.

1.12.4 Passing a function to a function

A function can be passed to a function. This is quite easy in Matlab and Python. In C++, the matter is a bit more intricate since the language is typed. Here is an example:

```
int addition (int a, int b)
{ return (a+b); }
int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main()
{
  int (*p_function) (int a, int b); // declare a pointer to a function with
      two int parameters that returns an int
  p_function = &addition; // the function pointed to by p_function is addition
  // p_function = addition; // this works too
  std::cout << (*p_function)(1, 2) << std::endl;
  std::cout << p_function(1, 2) << std::endl; // equivalent to previous line
  std::cout << operation(1, 2, p_function) << std::endl;
  std::cout << operation(1, 2, &addition) << std::endl; // same
  std::cout << operation(1, 2, addition) << std::endl; // equivalent
}
```

Exercise 13

Write a function that implements Newton's method.

Answer of exercise 13

See [PW12, §5.7].

Exercise 14

Write a function that implements the bisection method.

Exercise 15

Do the last exercise in an object-oriented way. (For this you will need the material of Section 1.14.)

1.12.5 Overloaded functions

In the example below, drawn from `http://www.cplusplus.com/doc/tutorial/functions2/`, the two `operate` functions have different parameter types. This makes them different functions with the same name. Hence `operate(a, b)` behaves differently according to the type of `a` and `b`.

```
int operate (int a, int b)
{
  return (a*b);
}
double operate (double a, double b)
{
  return (a/b);
}

int main()
{
  // Function overloading example:
  int n=3,m=4;
  double x=3.0,y=4.0;
  std::cout << "operate(n,␣m)␣=␣" << operate(n, m) << '\n';
  std::cout << "operate(x,␣y)␣=␣" << operate(x, y) << '\n';
}
```

1.12.6 A first look at templates: function templates

The example above can be rewritten as follows using templates.

```
#include <iostream>
#include <typeinfo> // for typeid

template<class T>
T OperateT (T a, T b)
{
        int intvar;
        if (typeid(a).name() == typeid(intvar).name())
                return (a*b);
        else
                return (a/b);
}

int main()
{
  std::cout << "OperateT<int>(3,␣4)␣=␣" << OperateT<int>(3, 4) << '\n';
  std::cout << "Deduced␣type:␣OperateT(3,␣4)␣=␣" << OperateT(3, 4) << '\n';
  std::cout << "Deduced␣type:␣OperateT(3.,␣4.)␣=␣" << OperateT(3., 4.) << '\n'
      ;
}
```

The output is

```
OperateT<int>(3, 4) = 12
Deduced type: OperateT(3, 4) = 12
Deduced type: OperateT(3., 4.) = 0.75
```

1.12.7 Namespaces

Namespaces can be useful to avoid conflicts in function names. See `http://www.cplusplus.com/doc/tutorial/namespaces/` for details. Observe the syntax `Namespace_name::function_name`, where `::` is the scope operator.

1.13 Structures

In C++, a `struct` is like a `class` (see Section 1.14) but with different default access level.

It is however uncommon to define member functions for a `struct`. Without member functions, a `struct` in C++ is quite similar to a structure array in Matlab.

We will not use structures in these notes. See, e.g., `http://www.cplusplus.com/doc/tutorial/structures/` for more information on structures.

1.14 Classes

When Bjarne Stroustrup began developing C++ in 1979, he called it "C with Classes". (In 1983, "C with Classes" was renamed to "C++", "++" being the increment operator in C.) Thus classes, hence object-oriented programming, are arguably the key difference between the C and C++ languages.

A *class* is a structure that can contain members variables—also called data members—and also member functions.

Here is an example, inspired from `http://www.cplusplus.com/doc/tutorial/classes/`.

Listing 1.2: Classes

```
#include <iostream>
// using namespace std;

class Rectangle // Rectangle is the identifier of the class
{ // begin body of class declaration
public: // begin public section
  Rectangle(); // declare default constructor
  Rectangle(int,int); // declare specialized constructor
```

```
 9    void SetWidth(int); // declare prototype of member function SetWidth
10    int GetArea(void) {return (mWidth*mHeight);} // declare and define GetArea
          member function
11  private: // begin private section
12    int mWidth, mHeight; // declare data members
13  }; // note the semicolon ";" at the end of the class declaration
14
15  Rectangle::Rectangle() // define default constructor
16  {
17    mWidth = 5;
18    mHeight = 5;
19  }
20
21  Rectangle::Rectangle(int a, int b) // define specialized constructor
22  {
23    mWidth = a;
24    mHeight = b;
25  }
26
27  void Rectangle::SetWidth(int a) // define member function SetWidth
28  {
29    mWidth = a;
30  }
31
32  int main()
33  {
34    Rectangle rect (3,4); // declare object rect of class Rectangle, passing
          integers 3 and 4 to the constructor
35    Rectangle rectb; // declare object rectb of class Rectangle using the
          default constructor
36    rectb.SetWidth(4); // call member function SetWidth of object rectb
37    std::cout << "rect␣area:␣" << rect.GetArea() << std::endl; // call member
          function GetArea of object rect
38    std::cout << "rectb␣area:␣" << rectb.GetArea() << std::endl;
39    return 0;
40  }
```

An instance of a class is termed an *object*. For example, in line 34, object `rect` of class `Rectangle` is constructed. Hence `Rectangle` is the *class name* and `rect` is the *object name*.

Line 6 is an *access specifier*. There are three possible access specifiers: `private` (default), `protected`, and `public`. Line 6 is the start of the public section of the class. Public members are accessible from anywhere where the object is visible. See, e.g., [PW12, §6.2.5] for details. More about this in Section 1.19.

Line 11 is the start of the private section in the declaration of class `Rectangle`. Private members are only accessible within the members of the same class. For example, the member function `GetArea` uses the private members `mWidth` and `mHeight` (line 10), but `rect.mWidth` in `main()` would not work. We try to stick to a coding convention where the identifiers of private members start with `m`.

Line 7 declares the *default constructor*. Whereas in Python the contructor is called `__init__`, in C++ the constructor has the same name as the class (here `Rectangle`) without any return type.

The default constructor is defined from line 15. Observe the *scope operator* `::`. Here `Rectangle::Rectangle()` indicates that we are defining the member function `Rectangle()` of class `Rectangle`.

The default constructor is used in line 35 where an object of class `Rectangle` is declared without argument.

Line 8 declares a *specialized constructor*. It is defined from line 21 and used in line 34 to declare an object of class `Rectangle` where the data members are assigned the specified values.

The member function `SetWidth` is defined in line 27. Observe again the scope operator `::`. The member function `SetWidth` is called in line 36 with object `rectb`.

Exercise 16

Add a member function of your choice to class `Rectangle`.

1.15 Pointers to classes

We have seen in Section 1.9 that pointers are variables that store the address of another variable. It is also possible to declare a pointer to an instance of a class (i.e., an object). There is then a convenient syntax to access class members of the object pointed to: in the example below, `p_rect->GetArea()` is equivalent to `(*p_rect).GetArea()`. (In the example below, class `Rectangle` is assumed to be as written in Listing 1.2.)

```cpp
int main()
{
  Rectangle * p_rect; // declare pointer p_rect to an object of class Rectangle
  p_rect = new Rectangle (5,6); // memory allocation and initialization of object pointed to by p_rect
  std::cout << "*p_rect␣area:␣" << (*p_rect).GetArea() << std::endl;
  std::cout << "*p_rect␣area:␣" << p_rect->GetArea() << std::endl;
  delete p_rect; // remember: always write a "delete" statement to match a "new" statement
}
```

The *arrow operator* `->` means "de-reference [the object specified on the left of the arrow] and access the member [specified on the right of the arrow]".

Exercise 17

Out of curiosity, let's print the value of the pointer to `Rectangle` object `p_rect`. Compare it with the value of a pointer to an integer.

Answer of exercise 17

On my computer, the value of a pointer to an object looks like `0xa50030`, whereas the value of a pointer to an integer looks like `0x7fffe215f09c`.

1.16 Operator overloading

C++ (like Python) uses zero-based numbering, as we saw in Section 1.10 on arrays. This can be confusing for users (in particular Matlab users) who are familiar with one-based numbering. In this section, we will see a technique, termed *operator overloading*, that makes it possible to redefine the `()` operator to access array elements with one-based numbering.

Here is a first example where the binary `+` operator is overloaded:

```cpp
#include <iostream>

class CVector // define class CVector
{  // begin body of CVector class definition
public:
  int x,y; // declare two member variables
  CVector() {}; // default constructor (needed because the automatically generated default constructor is only available if no other constructors have been provided, but see next line)
  CVector(int a,int b) : x(a), y(b) {} // declare and define specialized constructor using member initialization
  CVector operator+(const CVector&) const; // declare operator "+" overloading. The first keyword "const" signals that we want the argument to remain constant. The second keyword "const" signals that we want the instance of the class from which this member function is called to remain constant.
};  // end body of CVector class definition

CVector CVector::operator+ (const CVector& param) const // define member function "operator+", to which an object (locally labelled "param") of class CVector is passed by reference, and which returns an object of class CVector
{
  CVector temp; // create object "temp" of class "CVector" using default constructor
  temp.x = x + param.x; // "x" refers to the member variable "x" of the object on which member function "operator+" is called
  temp.y = y + param.y;
  // x = 10; // trying to modify the read-only class instance - not allowed by the compiler due to the 2nd keyword "const" above
  return temp; // return the expected object of class CVector
}

int main()
{
  CVector u (3,1); // declare object u of class CVector using the specialized constructor
  CVector v (1,2);
  CVector result1, result2;
  result1 = u.operator+(v); // operator+ is just like a normal member function of class CVector
  result2 = u + v; // but it can be called with this simpler syntax, with the same outcome
  std::cout << result1.x << ',' << result1.y << '\n';
  std::cout << result2.x << ',' << result2.y << '\n';
  std::cout << u.x << ',' << u.y << '\n';
}
```

1.16.1 Overloading the assignment operator

Let us now overload the assignment operator `=`. This is an occasion to introduce the keyword `this`, which is a pointer that points to the current object.

(The “equivalent” of `this` in Python is `self`. However, `self` is used in Python more profusely than `this` in C++.)

Listing 1.3: Overloading =

```cpp
#include <iostream>

class CVector // define class CVector
{  // begin body of CVector class definition
public:
  int x,y; // declare two member variables
  CVector() {}; // default constructor (needed because the automatically generated default constructor is only available if no other constructors have been provided, but see next line)
  CVector(int a,int b) : x(a), y(b) {} // declare and define specialized constructor using member initialization
  CVector& operator=(const CVector& param);
};  // end body of CVector class definition

CVector& CVector::operator=(const CVector& param)
{
  x = param.x;
  y = param.y;
  return *this;
}

int main()
{
  CVector u (3,1); // declare object u of class CVector using the specialized constructor
  CVector r, s;
  r.operator=(u);
  r = u; // equivalent to previous line
  s.operator=(r.operator=(u));
  s = r = u; // equivalent to previous line
  std::cout << "r.x,␣r.y␣=␣" << r.x << ",␣" << r.y << '\n';
  std::cout << "s.x,␣s.y␣=␣" << s.x << ",␣" << s.y << '\n';
}
```

Observe that `*this` (where `*` is the dereference operator that we have already seen in Section 1.9) is the object pointed by `this`, i.e., the current object. Moreover, `operator=` returns by reference (not by value) in view of the ampersand `&` in the return type `CVector&`. Consequently, the member function `operator=` returns a reference to the current object.

Exercise 18

Why not simply define `operator=` as follows?

```
void CVector::operator=(const CVector& param)
{
  x = param.x;
  y = param.y;
}
```

Actually, the compiler accepts this definition. But someting in Listing 1.3 no longer works. What? Why? Try it out. (This fairly fundamental technicality is apparently seldom pointed out in textbooks.)

Answer of exercise 18

The chain `s = r = u;` no longer works.

Exercise 19

What is the difference between the copy constructor and the assignment operator?

Answer of exercise 19

The copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.

1.16.2 Overloading operator()

We now come to the purpose announced at the start of this section, namely overloading `operator()` to access array elements with one-based numbering instead of the confusing-for-Matlab-users zero-based numbering. An example, which you should read and understand, is given in [PW12, Ch. 10]. Here is a simplified example.

```
#include <iostream>
#include <cassert> // for the assert function
// using namespace std;

class Vector
{
private:
  double* mData; // private member variable
  int mSize; // likewise

public:
  Vector(int size); // specialized constructor
  ~Vector(); // destructor, needed for the "delete" that matches the "new" in
       the definition of the constructor
  int GetSize() const; // member function that returns an int and leaves the
       class instance invariant
  double& operator()(int i); // overloading operator()
  friend int length(const Vector& v); // friend function "length", see below
       how it is defined and used
};

Vector::Vector(int size) // definition of the specialized constructor
```

```
20 {
21   mData = new double[size]; // memory allocation for an array of "double" of
         length "size".
22   mSize = size;
23 }
24
25 Vector::~Vector() // definition of the destructor of class Vector
26 {
27   delete[] mData; // here comes the "delete" required by the "new" above
28 }
29
30 int Vector::GetSize() const {return mSize;}
31
32 double& Vector::operator()(int i)
33 {
34   assert(i >= 1); // assert will abort the program if its argument is not
         satisfied
35   assert(i <= mSize);
36   return mData[i-1]; // returns by reference in view of the return type double
         &
37 }
38
39 int length(const Vector& v) {return v.mSize;} // definition of friend function
        "length"
40
41 int main() {
42   Vector v(2); // declare object v of class Vector using specialized
         constructor
43   // Vector v{2}; // same as above, with uniform initialization syntax (C++11
         needed)
44   std::cout << "v(2)␣=␣" << v(2) << '\n';
45   v(1) = 3; // possible since Vector::operator() returns by reference
46   v(2) = 4;
47   // v(0) = 6; // likewise for the 2nd assert
48   // v(3) = 5; // would make the program abort due to the 1st assert
49   std::cout << "v(2)␣=␣" << v(2) << '\n';
50   std::cout << "v.GetSize()␣=␣" << v.GetSize() << '\n';
51   std::cout << "length(v)␣=␣" << length(v) << '\n';
52
53   return 0;
54 }
```

The imitation of Matlab is pushed further with the function `length`. It is declared as a `friend` function in class `Vector` on line 16, and it is defined on line 16. Observe that it is not preceded by `Vector::` in the definition, and that it has access to the private members of the class (`Vector`) in which it is defined. Here the way to guarantee that the instance of class `Vector` is not affected by `length` is to prepend the keyword `const` before the parameter `Vector& v`; see lines 16 and 39.

Exercise 20

Implement a member function of class `Vector` that computes the inner product, with the following signature:

`double Vector::GetInnerProdWith(const Vector& v) const`.

Implement a function that does likewise, with the following signature:
`double GetInnerProdOf(const Vector& v, const Vector& w)`

Overload the operator `*` to compute the inner product.

Hint: you may want to create a member function `Read` such that `v.Read(i)` returns `mData[i]` by value while leaving object `v` constant. Is it possible to complete the task without such a function?

Answer of exercise 20

`const` is infectious, as mentioned in `http://duramecho.com/ComputerInformation/WhyHowCppConst.html`.

Exercise 21

Let `int i = 0`. Try `const int * p_i = &i`, `int * const p_i = &i`, and `const int * const p_i = &i`. What do they allow you to do with `i`, `*p_i`, and `p_i`?

Exercise 22

In Python, implement a function that computes the inner product with a basic “for” loop. Compare timings between C++ and Python for the computation of the inner product of two large vectors.

1.17 Header files

It may be convenient to separate interface from implementation. The interface goes into *header files*, with extension `.hpp` or `.h`. The implementation goes into *source files*, with extension `.cpp`.

Let us do it for the `Vector` class that we defined above.

Listing 1.4: vector_s.hpp

```
#ifndef VECTOR_S_HPP_ // include guard
#define VECTOR_S_HPP_ // include guard

class Vector
{
private:
  double* mData; // private member variable
  int mSize; // likewise

public:
  Vector(int size); // specialized constructor
  ~Vector(); // destructor
  int GetSize() const; // member function that returns an int and leaves the
      class instance invariant
  double& operator()(int i); // overloading operator()
  friend int length(const Vector& v); // friend function "length"
};
```

```
17
18 int length(const Vector& v); // prototype signature of "length" friend function
19
20 #endif /* VECTOR_S_HPP_ */ // include guard
```

The C++ compiler should include the same file only once. However, it is easy to write code where a same header file is included multiple times. This happens, in particular, when we include two files that each include the same file. *Include guards* are a technique to prevent this. In listing 1.4, lines 1, 2 and 20 form the include guard. It is recommended to always guard your header files. (The IDE Eclipse, for example, does it automatically when you create a new header file.)

Listing 1.5: vector_s.cpp

```cpp
#include <iostream>
#include <cassert> // for the assert function
#include "vector_s.hpp"

Vector::Vector(int size) // definition of the specialized constructor
{
  mData = new double[size]; // memory allocation for an array of "double" of length "size".
  mSize = size;
}

Vector::~Vector() // definition of the destructor of class Vector
{
  delete[] mData; // here comes the "delete" required by the "new" above
}

int Vector::GetSize() const {return mSize;}

double& Vector::operator()(int i)
{
  assert(i >= 1); // assert will abort the program if its argument is not satisfied
  assert(i <= mSize);
  return mData[i-1]; // returns by reference in view of the return type double&
}

int length(const Vector& v) {return v.mSize;} // definition of friend function "length"
```

We can now use class `Vector`, e.g., as follows.

Listing 1.6: vector_simple_w_header.cpp

```cpp
#include <iostream>
#include <cassert>
#include "vector_s.hpp"

int main() {
  Vector v(2); // declare object v of class Vector using specialized constructor
  // Vector v{2}; // same as above, with uniform initialization syntax (C++11
      needed)
  std::cout << "v(2)␣=␣" << v(2) << '\n';
  v(1) = 3; // possible since Vector::operator() returns by reference
  v(2) = 4;
  // v(0) = 6; // would make the program abort due to the 1st assert
  // v(3) = 5; // likewise for the 2nd assert
  std::cout << "v(2)␣=␣" << v(2) << '\n';
  std::cout << "v.GetSize()␣=␣" << v.GetSize() << '\n';
  std::cout << "length(v)␣=␣" << length(v) << '\n';

  return 0;
}
```

Now, compile as follows:

```
\$ g++ -o vector_simple_w_header vector_simple_w_header.cpp vector_s.cpp
```

If you are using VS Code: Ctrl+K Ctrl+O; choose the folder in which your cpp files are located; Terminal > Configure Default Build Task; choose g++. It creates a tasks.json file. In that file, replace `"${file}"` by `"${fileDirname}/*.cpp"` (on Linux) or `"${fileDirname}\\**.cpp"` (on Windows 10). You can now build with Ctrl+Shift+B.

Exercise 23

Create the three files presented in this section. Update them with the member functions `Read` and `GetInnerProdWith` that you have created in Exercise 20.

1.18 File input and output

Initializing a vector of size 2 is easy, but things may get tedious when the size gets large. We will now see how to read the values from a file and write values to a file.

Here is a basic example.

```
#include <iostream> // for std::cout
#include <fstream> // for std::fstream
#include <cassert> // for assert

int main()
{
   std::fstream myfile; // declare myfile as instance of class std::fstream

   // *** With basic arrays:
   myfile.open("file.txt", std::ios::in | std::ios::out); // open "file.txt"
       for both input and output operations
   assert(myfile.is_open()); // better to check, otherwise the code may run
       without error writing nothing
   double a[3] = {1.1, 2./7., 3.4}, b[3]; // declaration and initialization of
        double array a, declaration of double array b

14   myfile.setf(std::ios::scientific); // choose scientific format for writing to myfile
15   myfile.setf(std::ios::showpos); // choose to write "+" sign before positive numbers
16   myfile.precision(10); // write 10 digits after the decimal point (this meaning applies to the scientific format)
17
18   for(int i = 0; i < 3; i++)
19   {
20     myfile << a[i] << '\n'; // write elements of a to myfile
21   }
22
23   myfile.seekp(0, std::ios::beg); // set the "put position" of myfile to an offset 0 counted from the beginning of the stream
24   for (int i = 0; i < 3; i++) {
25      myfile >> b[i]; // read double elements of myfile to b
26      std::cout << a[i] << "␣" << b[i] << std::endl; // print on the screen
27   }
28   std::cout << std::endl; // skip one line on the screen
29
30   myfile.close(); // close myfile
31 }
```

Exercise 24

Adapt the code above to use `Vector` objects instead of plain arrays. Write the elements to file in a row instead of a column.

The code above demonstrates the C++ way of writing to / reading from a file. In line 23, the "p" in `seekp` refers to the *put position*. To set the *get position*, it would be `seekg`. For more information, see, e.g., `http://www.cplusplus.com/doc/tutorial/files/`.

1.19 Inheritance between classes

Say you are involved in a big collaborative project that makes a heavy use of class `Vector` (Listing 1.4). You would need to compute the Hadamard (i.e., element-wise) product of two `Vector` objects, but there is no such method. What can you do? One possibility is to fork the code and modify the `Vector` class.

A cleaner way is to resort to *class inheritance*, where a *derived class* inherits the members of a *base class*, on top of which it can add its own members.

```
#include <iostream>
#include <cassert>
#include "vector_s.hpp"

// Class inheritance:
class VectorWithHadamard : public Vector
{
public:
  VectorWithHadamard(int i) : Vector(i) {} // derived specialized constructor
  VectorWithHadamard GetHadamardWith(VectorWithHadamard& v); // declaration of member function to compute Hadamard product
};

VectorWithHadamard VectorWithHadamard::GetHadamardWith(VectorWithHadamard& v)
{
  VectorWithHadamard output(GetSize());
  for (int i = 1; i <= v.GetSize(); i++)
  {
    output(i) = this->operator()(i) * v(i); // This works
    //output(i) = operator()(i) * v(i); // This works too
    //output(i) = mData[i-1] * v(i); // This does not work. Why?
  }
  return output;
}

// The following function works too, but the drawback is that it is not encapsulated in the class.
VectorWithHadamard GetHadamardOf(VectorWithHadamard& u, VectorWithHadamard& v)
{
  VectorWithHadamard output(u.GetSize());
  for (int i = 1; i <= v.GetSize(); i++)
  {
    output(i) = u(i) * v(i); // This works
  }
  return output;
}

int main() {
  VectorWithHadamard uH(2), vH(2), wH(2);
  uH(2) = 3;
  vH(2) = 4;
  wH = uH.GetHadamardWith(vH);
  // wH = GetHadamardOf(uH, vH); // This works too.
  std::cout << "wH␣=␣" << wH(1) << ",␣" << wH(2) << "\n";
  return 0;
}
```

The three access levels are `public`, `protected`, and `private`, ordered by increasing level of restriction. C++ is conservative, in the sense that when a class is inherited with a certain access level, this access level becomes the most permissive access level of all the inherited members. For example, if a class is inherited as `protected`, then all public members of the base class are inherited as `protected` in the derived class. Moreover, the private members of the base class are inherited as *hidden*.

The members of a class can access the public, protected, and private members, but not the hidden members. Non-members can only access the public members.

A consequence of all this is that the members of the derived class can access the public members and protected members inherited from the base class, but they cannot access the private members of the base class since they are *hidden*.

Note however that it may nevertheless be possible for a member function of a derived class to read or modify hidden member variables, provided that some public or protected member functions of the base class are available to perform these operations.

That being said, it is most common to inherit classes as `public`, trusting that the access levels have been chosen adequately in the base class and that it is thus not necessary to force a lower accessibility.

Further information about access privileges for derived classes can be found in [PW12, §7.3].

Exercise 25

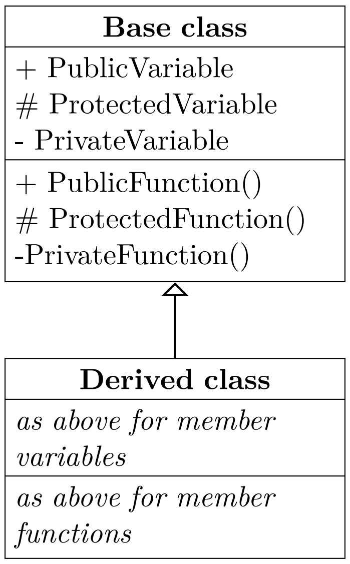
Answer the question in line 20 of the listing above.

It is possible for a derived class to inherit from more than one base class. The syntax is as follows:

```
class DerivedClass: public BaseClass1, public BaseClass2;
```

See `http://www.cplusplus.com/doc/tutorial/inheritance/` for more information.

Relationships between classes can be represented in schematic form with a *class diagram* in the *Unified Modelling Language (UML)* format, as follows:



The class diagram above was drawn using the tikz template at `http://www.texample.net/tikz/examples/class-diagram/`.

Exercise 26

Draw a class diagram for classes `Vector` and `VectorWithHadamard` in the Unified Modelling Language (UML) format. Ressources: [PW12, §7.2] or https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/.

1.20 Polymorphism

Pointers to the base class make it possible to view an object of a derived class as an object of the base class. Here is an example that uses the base class `Vector` and the derived class `VectorWithHadamard` introduced above.

```
  // Pointers to base class:
  VectorWithHadamard tH(2), sH(2); // declare object tH of class
      VectorWithHadamard using specialized constructor
  tH(2) = 1.1; sH(2) = 1.4;
  Vector * p_tH = &tH; // This works: tH is an object of class
      VectorWithHadamard derived from Vector
  std::cout << "p_tH->GetSize() = " << p_tH->GetSize() << "\n"; // This works
  //std::cout << p_tH->GetHadamardWith(uH); // This does not work, because
      p_tH is a pointer to Vector, not to VectorWithHadamard: only the members
       inherited from the base class (Vector) can be accessed.
  //VectorWithHadamard * p_v = &v; // Error. Conversion from
      VectorWithHadamard* to Vector* is possible, but not the other way round.
```

In the code above, `p_tH->GetSize()` uses the `GetSize` member function of class `Vector`, because `p_tH` was declared as a pointer to `Vector`.

Is it possible to let the derived class `VectorWithHadamard` redefine a member, say `GetInnerProdWith`, and have this redefined `GetInnerProdWith` be used by `p_tH->GetInnerProdWith()`? Yes, but then `GetInnerProdWith` must be declared as `virtual` in the base class. Here is a basic example:

```
#include <iostream>

class Base
{
public:
  virtual void DoSmth();
};

void Base::DoSmth()
{
  std::cout << "DoSmth in Base \n";
}

class Derived : public Base
{
public:
  void DoSmth();
};

void Derived::DoSmth()
{
  std::cout << "DoSmth in Derived \n";
}

int main() {
  Base B;
  Derived D;
  std::cout << "B.DoSmth():␣";
  B.DoSmth();
  std::cout << "D.DoSmth():␣";
  D.DoSmth();
  Base * pB_D = &D;
  std::cout << "pB_D->DoSmth():␣";
  pB_D->DoSmth();
  Derived * pD_D = &D;
  std::cout << "pD_D->DoSmth():␣";
  pD_D->DoSmth();
}
```

Exercise 27

Run the code above and interpret the output. Remove the `virtual` keyword, run it again, check and interpret the differences.

Exercise 28

In class `Vector`, insert the `virtual` keyword before the declaration of the member function `GetInnerProdWith`. In class `VectorWithHadamard`, redefine `GetInnerProdWith` to something else (for example, the inner product in a nonstandard metric). Check that the following computes your redefined inner product:

```
  VectorWithHadamard tH(2); // declare object tH of class VectorWithHadamard
      using specialized constructor
  tH(2) = 1.1;
  Vector * p_tH = &tH; // This works: tH is an object of class
      VectorWithHadamard derived from Vector
  // Demonstrating virtual member function GetInnerProdWith:
  std::cout << p_tH->GetInnerProdWith(tH) << "\n";
```

1.21 Other topics

Here are some fairly fundamental C++ topics that have been omitted (or almost omitted) in the present notes.

- We have defined a class named `Vector` in Section 1.14, but there is also the standard class `std::vector`. These are arrays that can change in size dynamically. See `http://www.cplusplus.com/reference/vector/vector/`, `http://www.cplusplus.com/reference/vector/vector/begin/`, etc.
- Abstract classes, where virtual members without definition (termed *pure virtual members*) are indicated with `=0`; see [PW12, §7.6].
- Templates; see [PW12, Ch. 8].

- Name visibility and scopes; see `http://www.cplusplus.com/doc/tutorial/namespaces/`.
- Global variables; see [PW12, §5.1] to know what they are, but it is recommended not to use them.
- Errors and exceptions: see [PW12, Ch. 9].
- C-style input and output with `#include <stdio.h>`. See in particular the `printf` function, documented at `http://www.cplusplus.com/reference/cstdio/printf/` (check the example first).
- Input arguments to a program; see [PW12, §3.4].
- Macro definitions using `#define`; see `http://www.cplusplus.com/doc/tutorial/preprocessor/`.
- Compiler flags; see [PW12, §1.3.3].

Bibliography

[PW12] Joe Pitt-Francis and Jonathan Whiteley. *Guide to Scientific Computing in C++*. Springer-Verlag, London, 2012. Freely available online from UCLouvain. `doi:10.1007/978-1-4471-2736-9`.

[Str20] Bjarne Stroustrup. Thriving in a crowded and changing world: C++ 2006–2020. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. `doi:10.1145/3386320`.

[TLNC10] Aslak Tveito, Hans Petter Langtangen, Bjørn Frederik Nielsen, and Xing Cai. *Elements of Scientific Computing*. Springer-Verlag, Berlin Heidelberg, 2010. Freely available online from UCLouvain. `doi:10.1007/978-3-642-11299-7`.