Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2017

Contents

PART I

LESSONS

Chapter 1

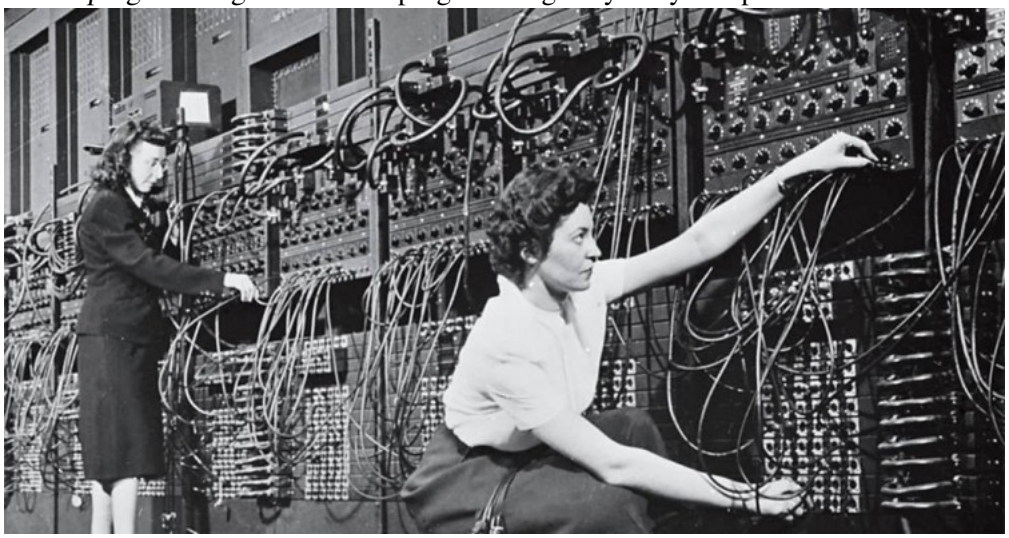
Introduction

1.1 Programming and computational thinking

1.1.1 History

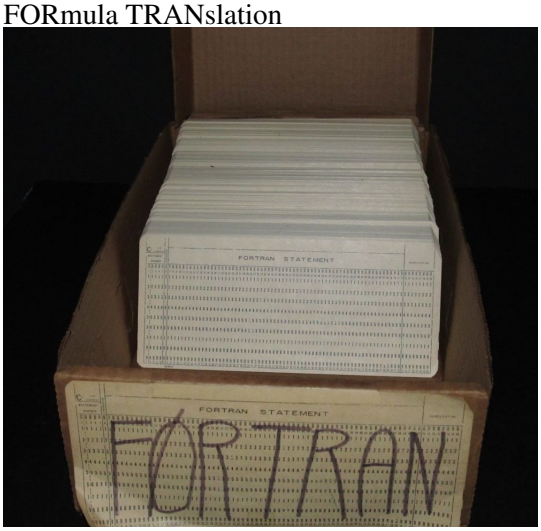
1.1. Earliest computers Earliest computers Historically, computers were used for big physics calculations, for instance, atom bomb calculations

1.2. Hands-on programming Hands-on programming Very early computers were hardwired



1.3. Program entry Program entry Later programs were written on punchcards

1.4. The first programming language The first programming language Initial programming was about translating the math formulas; after a while they made a language for that: FORmula TRANslation



1.5. Programming is everywhere Programming is everywhere Programming is used in many different ways these days.

- You can make your own commands in *Microsoft Word*.
- You can make apps for your *smartphone*.
- You can solve the riddles of the universe using big computers.

This course is aimed at people in the last category.

1.1.2 Computational thinking

1.6. Programming is not simple Programming is not simple Programs can get pretty big

It's not just translating formulas anymore.
Translating ideas to computer code: computational thinking.

1.7. Examples of computational thinking Examples of computational thinking

- Looking up a name in the phone book
  - start on page 1, then try page 2, et cetera
  - or start in the middle, continue with one of the halves.
- Elevator scheduling: someone at ground level presses the button, there are cars on floors 5 and 10; which one do you send down?

1.8. Abstraction Abstraction

- The elevator programmer probably thinks: 'if the button is pressed', not 'if the voltage on that wire is 5 Volt'.
- The Google car programmer probably writes: 'if the car before me slows down', not 'if I see the image of the car growing'.
- ... but probably another programmer had to write that translation.

1.9. Data abstraction Data abstraction What is the structure of the data in your program?

Stack: you can only get at the top item

Queue: items get added in the back, processed at the front

1.1.3 Hardware

1.10. Do you have to know much about hardware? Do you have to know much about hardware? Yes, it's there, but we don't think too much about it in this course.
https://youtu.be/JEpsKnWZrJ8

1.1.4 Algorithms

1.11. What is an algorithm? What is an algorithm?

> An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time
>
> [A. Levitin, Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2003]

The instructions are in some language:

- We will teach you C++ and Fortran;
- the compiler translates those languages to machine language
- Abstraction: a program often defines its own language that implements concepts of your application.

1.12. Program steps Program steps

- Simple instructions: arithmetic.
- Compicated instructions: control structures
  - conditionals
  - loops

1.13. Program data Program data

- Input and output data: to/from file, user input, screen output, graphics.
- Data during the program run:
  - Simple variables: character, integer, floating point
  - Arrays: indexed set of characters and such
  - Data structures: trees, queues
    - Defined by the user, specific for the application
    - Found in a library (big difference between C/C++!)

1.2 About the choice of language

1.14. Comparing two languages Comparing two languages Python vs C++ on bubblesort:

```
for i in range(n-1):
  for j in range(n-i-1):
    if numbers[j+1]<numbers[j]:
      swap = numbers[j+1]
      numbers[j+1] = numbers[j]
      numbers[j] = swap
```

```
for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (numbers[j+1]<numbers[j]) {
      int swap = numbers[j+1];
      numbers[j+1] = numbers[j];
      numbers[j] = swap;
    }
```

```
[] python bubblesort.py 5000
Elapsed time: 12.1030311584
[] ./bubblesort 5000
Elapsed time: 0.24121
```

1.15. The right language is not all The right language is not all Python with quicksort algorithm:

```
numpy.sort(numbers,kind='quicksort')

[] python arraysort.py 5000
Elapsed time: 0.00210881233215
```

1.3 Further reading

Tutorial, assignments: http://www.cppforschool.com/

Problems to practice: http://www.spoj.com/problems/classical/

Chapter 2

Warming up

2.1 Programming environment

Programming can be done in any number of ways. It is possible to use an Integrated Development Environment (IDE) such as *Xcode* or *Visual Studio*, but for the purposes of this course you should really learn a *Unix* variant.

- If you have a *Linux* computer, you are all set.
- If you have an *Apple* computer, it is easy to get you going. Install `XQuartz` and a *package manager* such as *homebrew* or *macports*.
- *Windows*Windows users can use *putty* but it is probably a better solution to install a virtual environment such as *VMware*.

Next, you should know a text editor. The two most common ones are *vi* and *emacs*.

2.2 Compiling

The word 'program' is ambiguous. Part of the time it means the *source code*: the text that you type, using a text editor. And part of the time it means the *executable*, a totally unreadable version of your code, that can be understood and executed by the computer. The process of turning your source code into an executable is called *compiling*, and it requires something called a *compiler*. (So who wrote the source code for the compiler? Good question.)

Let's say that

- you have a source code file `myprogram.cxx`,
- and you want an executable file called `myprogram`,
- and your compiler is *g++*, the C++ compiler of the *GNU* project. (If you have the Intel compilers, you will use *icpc* instead.)

To compile your program, you then type

```
g++ -o myprogram myprogram.cxx
```

On TACC machines, use the Intel compiler:

```
icpc -o myprogram myprogram.cxx
```

which you can verbalize as 'invoke `g++`, with output `myprogram`, on `myprogram.cxx`'.

So let's do an example.

This is a minimal program:

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

int main() {
  return 0;
}
```

1. The first two lines are magic, for now. Always include them.
2. The `main` line indicates where the program starts; between its opening and closing brace will be the *program statements*.
3. The `return` statement indicates successful completion of your program.

As you may have guessed, this program does absolutely nothing.

Here is a statement that at least produces some output:

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

Exercise 2.1. Make a program source file that contains the 'hello world' statement, compile it and run it. Think about where the statement goes.

Chapter 3

Basic elements of C++

3.1 Statements

3.1. Program statements Program statements

- A program contains statements, each terminated by a semicolon.
- 'Curly braces' can enclose multiple statements.
- A statement corresponds to some action when the program is executed.
- Comments are 'Note to self', short:

```
cout << "Hello world" << endl; // say hi!
```

  and arbitrary:

```
cout << /* we are now going
            to say hello
         */ "Hello!" << /* with newline */ endl;
```

Exercise 3.1. Take the 'hello world' program you wrote earlier, and duplicate the hello-line. Compile and run.
Does it make a difference whether you have the two hellos on the same line or on different lines?
Experiment with other changes to the layout of your source. Find at least one change that leads to a compiler error.

3.2. Fixed elements Fixed elements You see that certain parts of your program are inviolable:

- There are *keywords* such as `return` or `cout`.
- Curly braces and parentheses need to be matched.
- There has to be a `main` keyword.
- The `iostream` and `namespace` are usually needed.

Exercise 3.2. Experiment with the `cout` statement. Replace the string by a number or a mathematical expression. Can you guess how to print more than one thing, for instance the string `One third is` and the result of $1/3$, with the same `cout` statement?

3.2 Variables

3.3. Variable declarations Variable declarations Programs usually contain data, which is stored in a *variable*. A variable has

- a *datatype*,
- a name, and
- a value.

These are defined in a *variable declaration* and/or *variable assignment*.

3.4. Variable names Variable names

- A variable name has to start with a letter,
- can contains letters and digits, but not most special characters (except for the underscore).
- For letters it matter whether you use upper or lowercase: the language is *case sensitive*.

3.5. Declaration Declaration There are a couple of ways to make the connection between a name and a type. Here is a simple *variable declaration*, which establishes the name and the type:

```
int n;
float x;
int n1,n2;
double re_part,im_part;
```

Declarations can go pretty much anywhere in your program.

3.6. Datatypes Datatypes Variables come in different types;

- We call a variable of type `int`, `float`, `double` a *numerical variable*.
- For characters: `char`. Strings are complicated.
- You can make your own types. Later.

3.7. Assignment Assignment Once you have declared a variable, you need to establish a value. This is done in an *assignment* statement. After the above declarations, the following are legitimate assignments:

```
n = 3;
x = 1.5;
n1 = 7; n2 = n1 * 3;
```

You see that you can assign both a simple value or an *expression*; see section 3.4 for more detail.

3.8. Assignments Assignments A variable can be given a value more than once. You the following sequence of statements is a legitimate part of a program:

```
int n;
n = 3;
n = 2*n + 5;
n = 3*n + 7;
```

3.9. Special forms Special forms Update:

```
x = x+2; y = y/3;
// can be written as
x += 2; y /= 3;
```

Integer add/subtract one:

```
i++; j--;  /* same as: */ i=i+1; j=j-1;
```

Pre/post increment:

```
x = a[i++]; /* is */ x = a[i]; i++;
y = b[++i]; /* is */ i++; y = b[i];
```

3.10. Initialization Initialization You can also give a variable a value a in *variable initialization*. Confusingly, there are several ways of doing that. Here's two:

```
int n = 0;
double x = 5.3, y = 6.7;
double pi{3.14};
```

Exercise 3.3. Write a program that has several variables. Assign values either in an initialization or in an assignment. Print out the values.

3.11. Truth values Truth values So far you have seen integer and real variables. There are also *boolean values* which represent truth values. There are only two values: `true` and `false`.

```
bool found{false};
```

Exercise 3.4. Print out `true` and `false`. What do you get?

3.3 Input/Output, or I/O as we say

3.12. Terminal output Terminal output You have already seen `cout`:

```
float x = 5;
cout << "Here is the root: " << sqrt(x) << endl;
```

3.13. Terminal input Terminal input There is also a *cin*, which serves to take user input and put it in a numerical variable.

```
int i;
cin >> i;
```

However, this function is somewhat tricky.
http://www.cplusplus.com/forum/articles/6046/.

3.14. Better terminal input Better terminal input It is better to use `getline`. This returns a string, rather than a value, so you need to convert it with the following bit of magic:

```
#include <iostream>
#include <sstream>
using namespace std;
  std::string saymany;
  int howmany;

  cout << "How many times? ";
  getline( cin,saymany );
```

```
stringstream saidmany(saymany);
saidmany >> howmany;
```

You can not use `cin` and `getline` in the same program.

Exercise 3.5. Write a program that

- Displays the message `Type a number`,
- accepts an integer number from you (use `cin`),
- and then prints out three times that number plus one.

3.4 Expressions

3.15. Arithmetic expressions Arithmetic expressions

- Expression looks pretty much like in math.
  With integers: `2+3`
  with reals: `3.2/7`
- Use parentheses to group `25.1*(37+42/3.)`
- Careful with types.
- There is no 'power' operator: library functions. Needs a line
  ```
  #include <cmath>
  ```
- Modulus: `%`

3.16. Boolean expressions Boolean expressions

- Testing: `== != < > <= >=`
- Not, and, or: `! && ||`
- Bitwise: `& | ^`
- Shortcut operators:
  ```
  if ( x>=0 && sqrt(x)<5 ) {}
  ```

3.17. Conversion and casting Conversion and casting Real to integer: round down:

```
double x,y; x = .... ; y = .... ;
int i; i =  x+y:
```

Dangerous:

```
int i,j; i = ... ; j = ... ;
double x ; x = 1+i/j;
```

The fraction is executed as integer division. Do:

```
(double)i/j /* or */ (1.*i)/j
```

Exercise 3.6. Compute some arithmetic expressions and print them out. Experiment with conversions. Also boolean expressions.

Exercise 3.7. Write a program that ask for two integer numbers `n1,n2`.

- Assign the integer ratio n_1/n_2 to a variable.

- Can you use this variable to compute the modulus

$$n_1 \mod n_2$$

  Print out the value you get.
- Also print out the result from using the modulus operator:`%`.

3.5 Conditionals

3.18. If-then-else If-then-else A *conditional* is a test: 'if something is true, then do this, otherwise maybe do something else'. The C++ syntax is

```
if ( something ) {
  do something;
} else {
  do otherwise;
}
```

(Can leave out braces in case of single statement.)

3.19. Complicated conditionals Complicated conditionals Chain:

```
if ( something ) {
  ...
} else if ( something else ) {
  ...
}
```

Nest:

```
if ( something ) {
  if ( something else ) {
    ...
  } else {
    ...
  }
}
```

3.20. Switch Switch

```
switch (n) {
case 1 :
case 2 : cout << "very small" << endl;
  break;
case 3 : cout << "trinity" << endl;
  break;
default : cout << "large" << endl;
}
```

3.21. Local variables in conditionals Local variables in conditionals The curly brackets in a conditional allow you to define local variables:

```
if ( something ) {
  int i;
  .... do something with i
}
// the variable 'i' has gone away.
```

Exercise 3.8. Read in an integer. If it's a multiple of three print 'Fizz'; if it's a multiple of five print 'Buzz'. It it is a multiple of both three and five print 'FizzBuzz'. Otherwise print nothing.

3.6 Looping

3.22. Repeat statement Repeat statement Sometimes you need to repeat a statement a number of times. That's where the *loop* comes in. A loop has a counter, called a *loop variable*, which (usually) ranges from a lower bound to an upper bound.
Here is the syntax in the simplest case:

```
for (int var=low; var<upper; var++) {
  statements;
}
```

Exercise 3.9. Read an integer value, and print 'Hello world' that many times.

3.23. Loop syntax Loop syntax

- The loop variable can be defined outside the loop:

```
int var;
for (var=low; var<upper; var++) {
```

- The stopping test be any test; can even be empty.
- The increment can be a decrement or something like `var*=10`
- Any and all of initialization, test, increment can be empty:
  `for(;;) ...`

3.24. Nested loops Nested loops Traversing a matrix:

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    ...
```

3.6.1 Looping until

3.25. Indefinite looping Indefinite looping Sometimes you want to iterate some statements not a predetermined number of times, but until a certain condition is met. There are two ways to do this.
First of all, you can use a 'for' loop and leave the upperbound unspecified:

```
for (int var=low; ; var=var+1) { ... }
```

3.26. Break out of a loop Break out of a loop This loop would run forever, so you need a different way to end it. For this, use the `break` statement:

```
for (int var=low; ; var=var+1) {
  statement;
  if (some_test) break;
  statement;
}
```

3.27. While loop While loop The other possibility is a `while` loop, which repeats until a condition is met. Syntax:

```
while ( condition ) {
  statements;
}
```

The while loop does not have a counter or an update statement; if you need those, you have to create them yourself.

3.7 Exercises

Exercise 3.10. **Root finding.** For many functions f, finding their zeros, that is, the values x for which $f(x) = 0$, can not be done analytically. You then have to resort to numerical *root finding* schemes. In this exercise you will implement a simple scheme.
Suppose x_-, x_+ are such that

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values are of opposite sign. Then there is a zero in the interval (x_-, x_+). Try to find this zero by looking at the halfway point, and based on the function value there, considering the left or right interval.

- How do you find x_-, x_+? This is tricky in general; if you can find them in the interval $[-10^6, +10^6]$, halt the program.
- Finding the zero exactly may also be tricky or maybe impossible. Stop your program if $|x_- - x_+| < 10^{-10}$.

Exercise 3.11. Find all triples of integers u, v, w under 100 such that $u^2 + v^2 = w^2$. Make sure you get unique triples and leave out permutations of something you already found.

Exercise 3.12. The integer sequence

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd} \end{cases}$$

leads to the Collatz conjecture: no matter the starting guess u_1, the sequence $n \mapsto u_n$ will always terminate.
For $u_1 < 1000$ find the values that lead to the longest sequence.

Exercise 3.13. Large integers are often printed with a comma (US usage) or a period (European usage) between all triples of digits. Write a program that accepts an integer such as 2542981 from the user, and prints it as `2,542,981`.

Chapter 4

Arrays

4.1 Traditional arrays

Static allocation, initialization.

```
// basic/array.cxx
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << endl;
}
{
  int numbers[5]{5,4,3,2,1};
  cout << numbers[3] << endl;
}
{
  int numbers[5] = {2};
  cout << numbers[3] << endl;
}
```

Disadvantage: no test on bounds.

Exercise 4.1. Check whether an array is sorted.

Exercise 4.2. Find the maximum element in an array.
Also report at what index the maximum occurs.

Arrays can be passed to a subprogram, but the bound is unknown there.

```
// basic/array.cxx
void print_first_index( int ar[] ) {
  cout << "First index: " << ar[0] << endl;
}
  {
    int numbers[] = {1,4,2,5,6};
    print_first_index(numbers);
  }
```

Exercise 4.3. Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

Subprograms can alter array elements. This was not possible with scalar arguments.

4.2 Multi-dimensional arrays

Declaration, pass to subprogram.

```
// array/contig.cxx
void print12( int ar[][6] ) {
  cout << "Array[1][2]: " << ar[1][2] << endl;
  return;
}
    int array[5][6];
    array[1][2] = 3;
    print12(array);
```

Memory layout: multi-dimensional arrays are actually contiguous and linear.

```
// array/contig.cxx
void print06( int ar[][6] ) {
  cout << "Array[0][6]: " << ar[0][6] << endl;
  return;
}
    int array[5][6];
    array[1][0] = 35;
    print06(array);
```

4.3 Other allocation mechanisms

A declaration

```
float ar[500];
```

is local to the scope it is in. This has some problems:

- Allocated on the *stack*; may lead to stack overflow.
- Can not be used as a class member:

```
class thing {
private:
  double array[ ???? ];
public:
  thing(int n) {
    array[ n ] ???? this does not work
  }
}
```

- Can not be returned from subprogram:

```
void make_array( double array[],int n ) {
  double array[n] ???? does not work
}
int main() {
  ....
  make_array(array,100);
}
```

Use of *new* uses the equivalence of array and reference.

```
// array/arraynew.cxx
void make_array( int **new_array, int length ) {
  *new_array = new int[length];
}
  int *the_array;
  make_array(&the_array,10000);
```

Since this is not scoped, you have to free the memory yourself:

```
// array/arraynew.cxx
class with_array{
private:
  int *array;
  int array_length;
public:
  with_array(int size) {
    array_length = size;
    array = new int[size];
  };
  ~with_array() {
    delete array;
  };
};
  with_array thing_with_array(12000);
```

Notice how you have to remember the array length yourself? This is all much easier by using a `std::vector`. See http://www.cplusplus.com/articles/37Mf92yv/.

Deprecated use of *malloc*.

4.4 Vector class for arrays

Use of `std::vector`.

```
// array/arraystd.cxx
vector<double> my_array(array_length);
  my_array[i] = user_number;
```

```
  //examplesnippet stdvector
}
#if 0
//examplesnippet stdvector
my_array[array_length] = 0.123; // wrong but probably works
my_array.at(array_length) = 0.123; // runtime error
cout << "the array has " << my_array.size() << " elements" << endl;
```

Pre-allocated and dynamic; difference in cost.

Be sure to `reserve`!

4.4.1 Matrix as vector of vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row); // check on that >> syntax!
```

This is not contiguous.

4.4.2 Matrix class based on vector

Make sure you've studied classes; section 5.4.

You can make a 'pretend' matrix by storing a long enough `vector` in an object:

```
// array/matrixclass.cxx
class matrix {
private:
  std::vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n) {
    this->m = m; this->n = n;
    the_matrix.reserve(m*n);
  };
  void set(int i,int j,double v) {
    the_matrix[ i*n +j ] = v;
  };
  double get(int i,int j) {
    return the_matrix[ i*n +j ];
  };
```

The syntax for `set` and `get` can be improved.

Exercise 4.4. Write a method `element` of type `double&`, so that you can write

```
A.element(2,3) = 7.24;
```

4.5 Exercises

Exercise 4.5. A knight on the chess board moves by going two steps horizontally or vertically, and one step either way in the orthogonal direction. Given a starting position, find a sequence of moves that brings a knight back to its starting position. Are there starting positions for which such a cycle doesn't exist?

Exercise 4.6. Put eight queens on a chessboard so that none threatens any other.

Exercise 4.7. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

Chapter 5

Scope, functions, classes

5.1 Scope

Global variables, local variables.

When you defined a function for primality testing, you placed it outside the main program, and the main program was able to use it. There are other things than functions that can be defined outside the main program, such as *global variables*.

Here is a program that uses a global variable:

```
int i=5;
int main() {
  i = i+3;
  cout << i << endl;
  return 0;
}
```

5.2 Functions

A *function* (or *subprogram*) is a way to abbreviate a block of code and replace it by a single line. Any program you can write with functions you can also write without. It will be just as fast, but maybe harder to read, it may be longer, and harder to debug.

- Functions make your code easier to read: you replace a block of code by a descriptive name.
- Your code may become shorter: if you find yourself writing the same block of code twice, you replace it by one function definition, and twice a single line *function call*. This is known as *code reuse*.
- Easier to debug: if you use the same (or roughly the same) block of code twice, and you find an error, you need to fix it twice.
- Maintainance: if a block occurs twice, and you change something in such a block once, you have to remember to change the other occurrences too.

If you know how to write a function, primality testing would look like:

```
bool isprime;
number = 13;
isprime = prime_test_function(number);
```

Loosely, a function takes input and computes some result which is then returned. Formally, a function consists of:

- *function result type*: you need to indicate the type of the result;
- name: you get to make this up;
- *function variables*: the
- *function body*: the statements that make up the function; and
- a *return* statement.

Functions are defined before the main program, and used in that program: Here is a program with a function that doubles its input:

```
#input <iostream>
using namespace std;

int twice_function(int n) {
  int twice_the_input = 2*n;
  return twice_the_input;
}

int main() {
  int number = 3;
  cout << "Twice three is: " << twice_function(number) << endl;
  return 0;
}
```

It's as if the function call

1. causes the function body to be executed, and
2. the function call is replaced by whatever you `return`.
3. (If the function does not return anything, for instance because it only prints output, you declare the return type to be *`void`*.)

Examples.

- Functions without input, without output.

```
void print_header() {
  cout << "****************" << endl;
  cout << "* Ouput        *" << endl;
  cout << "****************" << endl;
  }
int main() {
  print_header();
  cout << "The results for day 25:" << endl;
  // code that prints results ....
  return 0;
```

```
}
```

- Functions with input:

```
void print_header(int day) {
  cout << "****************" << endl;
  cout << "* Ouput        *" << endl;
  cout << "****************" << endl;
  cout << "The results for day " << day << ":" << endl;
  }
int main() {
  print_header(25);
  // code that prints results ....
  return 0;
}
```

- Functions with output:

```
#include <cmath>
double pi() {
  return 4*atan(1.0);
}
```

A function body defines a *scope*: the local variables of the function calculation are invisible to the calling program.

Functions can not be nested.

5.2.1 Parameter passing

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function.
- Mark parameters as `const`, just in case:

```
void print_header(const int day)
```

- *call by value*

Also good design:

- Return no function result,
- or return (0 is success, nonzero various informative statuses), and
- return other information by changing the parameters.

call by reference

```
int can_read_value( int &value ) {
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
```

```
  return file_status;
}
```

Exercise 5.1. Write a function `swap` of two parameters that exchanges the input values:

```
int i=2,j=3;
swap(i,j);
// now i==3 and j==2
```

5.2.2 Recursive functions

Fibonacci, what else?

5.3 Structures

A structure is an object that can contain multiple variables.

```
// basic/struct.cxx
struct gridpoint {
  int i,j; };
```

The elements of a structure are usually called *members*.

You can now use this structure type in a variable declaration, and then use the members of this structure variable.

```
// basic/struct.cxx
struct gridpoint onepoint;
onepoint.i = 1; onepoint.j = 2;
```

This includes passing the structure to a function:

```
// basic/struct.cxx
float distance( struct gridpoint point ) {
  return sqrt( point.i*point.i + point.j*point.j );
}
  float dist = distance(onepoint);
```

5.3.1 Prototypes and separate compilation

5.4 Classes and objects

Everything you have learned up till now was (more or less) part of the C language, the predecessor of C++. (The one clear exception was `cin`/`cout`.) You will now learn one of the major distinguishing features that makes C++ an 'object oriented language': *classes*class and *objects*object.

A program naturally contains objects: the combination of data and functions operating on that data.

```
class object_with_an_int {
private:
  int stored_integer;
public:
  // constructor: code that makes the object
  object_with_an_int( int the_int ) {
    stored_integer = the_int;
  };
  void increase() {
    stored_integer++;
  };
  void print() {
    cout << "We are storing: " << stored_integer << endl;
  };
};
  object_with_an_int has_an_int(5);
  has_an_int.increase();
  has_an_int.print();
```

This looks a little like a structure, but a structure only had data elements, and no functions. Also note that unlike declaring a structure, a class definition does not itself make any data: it defines the structure of objects of that class. The line in the main program is what creates an object with the structure as defined in the class definition.

You can have more than one version of a function. This is known as *polymorphism*.

```
void increase() {
  stored_integer++;
};
void increase(int by) {
  stored_integer += by;
};
object_with_an_int has_an_int(5);
has_an_int.increase();
has_an_int.increase(5);
has_an_int.print();
```

You can even redefine arithmetic operators such as `+*/%`.

5.5 Exercises

Exercise 5.2. Write a fraction class which stores non-integer numbers as a numerator/denominator pair. Implement addition and multiplication functions. Make sure to simplify fractions!
Can you print fractions so that $5/3$ is displayed as `1~2/3`?

Chapter 6

References and addresses

6.1 Reference

Passing a variable to a routine passes the value; in the routine, the variable is local.

```
// basic/arraypass.cxx
void change_scalar(int i) { i += 1; }
```

You can indicate that this is unintended:

```
// basic/arraypass.cxx
/* This does not compile:
   void change_const_scalar(const int i) { i += 1; }
*/
```

If you do want to make the change visible in the *calling environment*, use a reference:

```
// basic/arraypass.cxx
void change_scalar_by_reference(int &i) { i += 1; }
```

There is no change to the calling program. (Some people find this bad, that you can see from the use of a function whether it passes *by reference* or *by value*.)

Arrays are always pass by reference:

```
// basic/arraypass.cxx
void change_array_location( int ar[], int i ) { ar[i] += 1; }
  int numbers[5];
  numbers[2] = 3.;
  change_array_location(numbers,2);
```

The old-style way of doing things:

```
// basic/arraypass.cxx
void change_scalar_old_style(int *i) { *i += 1; }
  number = 3;
  change_scalar_old_style(&number);
```

Chapter 7

Polymorphism

7.1 The basic idea

Sometimes you want to have the same function name for two slightly different purposes. C++ allows you to define the same function twice, as long as their parameters are different enough.

For instance, here is the same ‘sum’ function defined for both integers and reals:

Chapter 8

Memory

8.1 Memory and scope

If a variable goes *out of scope*, its memory is deallocated.

Deallocating objects is slightly more complicated: an *object destructor* is called.

```
// basic/destructor.cxx
class SomeObject {
public:
  SomeObject() { cout << "calling the constructor" << endl; };
  ~SomeObject() { cout << "calling the destructor" << endl; };
};
  cout << "Before the nested scope" << endl;
  {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
  }
  cout << "After the nested scope" << endl;
```

gives:

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Chapter 9

Prototypes

9.1 Prototypes for functions

Suppose you have a function

```
int tester(int x) {
  if ( something with x ) {
    return one value;
  else
    return other value;
  fi
}
```

and a line in your main program

```
int t = tester(1,2);
```

then the compiler can give an error: the function expects one argument and you supply two. If your program has a line

```
int t  = tester(5.27);
```

then the compiler can give a warning about the type mismatch. (Why is this not an actual error?)

The minimal information the compiler needs about the function `tester` is that it takes an `int` input and gives an `int` as output. This is described in the function *prototype*:

```
int tester(int);
```

Prototypes are useful if you spread your program over multiple files. You would put your functions in one file:

```
// file: def.cxx
int tester(int x) {
  .....
}
```

and the main program in another:

```
// file : main.cxx
int tester(int);

int main() {
  int t = tester(...);
  return 0;
}
```

Or you could use your function in multiple files and you would have to write it only once.

Even better than writing the prototype every time you need the function is to have a *header file*:

```
// file: def.h
int tester(int);
```

The definitions file would include this:

```
// file: def.cxx
#include "def.h"
int tester(int x) {
  .....
}
```

and so does the main program

```
// file : main.cxx
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

Having a header file is an important safety measure:

- Suppose you change your function definition, adding a parameter:
- The compiler will complain when you compile the definitions file;
- So you change the prototype in the header file;
- Now the compiler will complain about the main program, so you edit that too.

By the way, why does that compiler even recompile the main program, even though it was not changed? Well, that's because you used a *makefile*. See the tutorial.

9.2 Global variables

9.3 Prototypes for class methods

9.4 Header files and templates

The use of *templates* often make separate compilation impossible: in order to compile the templated definitions the compiler needs to know with what types they will be used.

Chapter 10

Efficiency

10.1 Order of complexity

10.1.1 Time complexity

Exercise 10.1. For each number n from 1 to 100, print the sum of all numbers 1 through n.

There are several possible solutions to this exercise. Let's assume you don't know the formula for the sum of the numbers $1 \ldots n$. You can have a solution that keeps a running sum, and a solution with an inner loop.

Exercise 10.2. How many operations, as a function of n, are performed in these two solutions?

10.1.2 Space complexity

Exercise 10.3. Read numbers that the user inputs; when the user inputs zero or negative, stop reading. Add up all the positive numbers and print their average.

This exercise can be solved by storing the numbers in a `std::vector`, but one can also keep a running sum and count.

Exercise 10.4. How much space do the two solutions require?

Chapter 11

Preprocessor

In your source files you have seen lines starting with a hash sign, like

```
#include <iostream>
```

Such lines are interpreted by the *C preprocessor*.

Your source file is transformed to another source file, in a source-to-source translation stage, and only that second file is actually compiled by the *compiler*. In the case of an `#include` statement, the preprocessing stage takes form of literaly inserting another file, here a *header file* into your source.

There are more sophisticated uses of the preprocessor.

11.1 Textual substitution

Suppose your program has a number of arrays and loop bounds that are all identical. To make sure the same number is used, you can create a variable, and pass that to routines as necessary.

```
void dosomething(int n) {
  for (int i=0; i<n; i++) ....
}

int main() {
  int n=100000;

  double array[n];

  dosomething(n);
```

You can also use a *preprocessor macro*:

```
#define N 100000
void dosomething() {
  for (int i=0; i<N; i++) ....
}

int main() {
```

```
double array[N];

dosomething();
```

It is traditional to use all uppercase for such macros.

11.2 Parametrized macros

Simulate 2D indexing in 1D array.

11.3 Conditionals

Comment out code.

Test whether macro is set. In particular header file.

PART II

PROJECTS

Chapter 12

Prime numbers

12.1 Preliminaries

Assuming you have learned about

- statements, section 3.1
- variables, section 3.2
- I/O, section 3.3

12.2 Arithmetic

Before doing this section, make sure you study section 3.4.

Exercise 12.1. Read two integers into two variables, and print their sum, product, quotient, modulus.

A less common operator is the modulo operator `%`.

Exercise 12.2. Read two numbers and print out their modulus. Two ways:

- use the `cout` function to print the expression, or
- assign the expression to a variable, and print that variable.

12.3 Conditionals

Before doing this section, make sure you study section 3.5.

Exercise 12.3. Read two numbers and print a message like

```
3 is a divisor of 9
```

if the first is an exact divisor of the second, and another message

```
4 is not a divisor of 9
```

if it is not.

12.4 Looping

Control structures such as loops; section 3.6.

Exercise 12.4. Read an integer, and test for all smaller numbers whether they are a divisor of that number. If there is a divisor, print out that number.
Print a final message

```
Your number is prime
```

or

```
Your number is not prime
```

Exercise 12.5. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

12.5 Functions

Before doing this section, make sure you study section 5.2.

Above you wrote several lines of code to test whether a number was prime.

Exercise 12.6. Write a function that takes an integer input, and return a boolean corresponding to whether the input was prime.

```
bool isprime;
isprime = prime_test_function(13);
```

12.6 While loops

Before doing this section, make sure you study section 3.6.1.

Exercise 12.7. Take the prime number exercise, and modify it to read in how many prime numbers you want to print. Keep a variable `number_of_primes_found` that is increased whenever a new prime is found.

Exercise 12.8. Take the previous exercise and rewrite it using a 'while' loop.

12.7 Global variables: optional

Before doing this section, make sure you study section 9.1.

Exercise 12.9. Use global variables to rewrite exercise 12.7. Your main program should exactly be this:

```
int main() {
  int nprimes;
  cout << "How many primes do you want? " << endl;
  cin >> nprimes;
```

```
  while (numberfound<nprimes) {
    int number = nextprime();
    cout << "Number " << number << " is prime" << endl;
  }

  return 0;
}
```

The trick here is to write the function `nextprime` uses the remembered global information, calculates the next prime, and returns it.

12.8 Structures

Before doing this section, make sure you study section 5.3, 6.1.

A `struct` functions to bundle together a number of data item. We only discuss this as a preliminary to classes.

Exercise 12.10. Rewrite exercise 12.7 to put the `numberfound` and `currentnumber` variables in a structure. Your main program should now look like:

```
// primes/5primesbystruct.cxx
struct primesequence sequence;
while (sequence.numberfound<nprimes) {
  int number = nextprime(sequence);
  cout << "Number " << number << " is prime" << endl;
}
```

You also need to use the structure in the `nextprime` exercise.

12.9 Classes and objects

Before doing this section, make sure you study section 5.4.

In exercise 12.10 you made a structure that contains the data for a primesequence, and you have separate functions that operate on that structure or on its members.

Exercise 12.11. Write a class `primesequence` that contains the members of the structure, and the functions `nextprime`, `isprime`. The function `nextprime` does not need the structure as argument, because the structure members are in the class, and therefore global to that function.

Your main program should look as follows:

```
primesequence sequence;
while (sequence.numberfound<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << endl;
}
```

In the previous exercise you defined the `primesequence` class, and you made one object of that class:

```
primesequence sequence;
```

But you can make multiple sequences, that all have their own internal data and are therefore independent of each other.

Exercise 12.12. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes. Write a program to test this for the even numbers up to 2 million.

12.10 Arrays

Another algorithm for finding prime numbers is the *Eratosthenes sieve*. It goes like this.

1. You take a range of integers, starting at 2.
2. Now look at the first number. That's a prime number.
3. Scratch out all of its multiples.
4. Find the next number that's not scratched out; since that's not a multiple of a previous number, it must be a prime number. Report it, and go back to the previous step.

The new mechanism you need for this is the data structure for storing all the integers.

```
int N = 1000;
vector<int> integers(N);
```

Exercise 12.13. Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers.

Chapter 13

Geometry

This uses the material in section 5.4.

13.1 Point class

A class can contain elementary data. In this section you will make a `Point` class that models cartesian coordinates and functions defined on coordinates.

Exercise 13.1. Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

and a function `distance` so that if `p,q` are `Point` objects, the call

```
p.distance(q)
```

computes the distance.

Exercise 13.2. Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

Exercise 13.3. Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 4.4. Can you make a constructor where you do not specify the space dimension explicitly?

13.2 Using one class in another

Exercise 13.4. Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point &input_p2 );
LinearFunction( Point &input_p1,Point &input_p2 );
```

and a function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

13.3 Has-a relation

Objects of one class can also contain objects of another. This is called the *has-a relation*.

Suppose you want to write a `Rectangle` class, which could have methods such as `float Rectangle::area()` or `bool Rectangle::contains(Point)`. Since rectangle has four corners, you could store four `Point` objects in each `Rectangle` object. However, there is redundancy there: you only need three points to infer the fourth. Let's consider the case of a rectangle with sides that are horizontal and vertical; then you need only two points.

Exercise 13.5. Make a class `Rectangle` with two constructors:

```
Rectangle(Point bl,Point tr);
Rectangle(Point bl,float w,float h);
```

and functions

```
float area(); float width(); float height();
```

Let the `Rectangle` object store two `Point` objects.
Then rewrite your exercise so that the `Rectangle` stores only one point (say, lower left), plus the width and height.

The previous exercise illustrates an important point: for well designed classes you can change the implementation (for instance motivated by efficiency) while the program that uses the class does not change.

13.4 Is-a relationship

You can have classes where an object of one class is a special case of the other class. You declare that as

```
class special_case : public general_case
```

When you run the special case constructor, usually the general case needs to run too.

- If you define the constructor as

  ```
  special_case(....) { ... };
  ```

  it will call the default constructor `general_case()`. You can also call other constructors:

  ```
  special_case( int i ) : general_case(i,i) { ... }
  ```

Exercise 13.6. Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

Exercise 13.7. Advanced. Go back to section 13.1. Add methods `slope` and `intercept` to your `LinearFunction` class. You can implement this by computing these quantities from the defining points and storing them.

Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

Chapter 14

PageRank

14.1 Basic ideas

Assuming you have learned about arrays 4, in particular the use of `std::vector`.

The web can be represented as a matrix W of size N, the number of web pages, where $w_{ij} = 1$ if page i has a link to page j and zero otherwise. However, this representation is only conceptual; if you actually stored this matrix it would be gigantic and largely full of zeros. Therefore we use a *sparse matrix*: we store only the pairs (i, j) for which $w_{ij} \neq 0$. (In this case we can get away with storing only the indices; in a general sparse matrix you also need to store the actual w_{ij} value.)

Exercise 14.1. Store the sparse matrix representing the web as a

```
vector< vector<bool> >
```

structure.

1. At first, assume that the number of web pages is given and reserve the outer vector. Read in values for nonzero indices and add those to the matrix structure.
2. Then, assume that the number of pages is not pre-determined. Read in indices; now you need to create rows as they are needed. Suppose the requested indices are

   ```
   5,1
   3,5
   1,3
   ```

   Since your structure has only three rows, you also need to remeber their row numbers.

Now we want to model the behaviour of a person clicking on links.

PART III

REFERENCE

Chapter 15

Programming strategies

15.1 Programming: top-down versus bottom up

The exercises in chapter 12 were in order of increasing complexity. You can imagine writing a program that way, which is formally known as *bottom-up* programming.

However, to write a sophisticated program this way you really need to have an overall conception of the structure of the whole program.

Maybe it makes more sense to go about it the other way: start with the highest level description and gradually refine it to the lowest level building blocks. This is known as *top-down* programming.

https://www.cs.fsu.edu/~myers/c++/notes/stepwise.html

Example:

Run a simulation

becomes

Run a simulation:
    Set up data and parameters
    Until convergence:
        Do a time step

becomes

Run a simulation:
    Set up data and parameters:
        Allocate data structures
        Set all values
    Until convergence:
        Do a time step:
            Calculate Jacobian
            Compute time step
            Update

You could do these refinement steps on paper and wind up with the finished program, but every step that is refined could also be a subprogram.

We already did some top-down programming, when the prime number exercises asked you to write functions and classes to implement a given program structure; see for instance exercise 12.11.

A problem with top-down programming is that you can not start testing until you have made your way down to the basic bits of code. With bottom-up it's easier to start testing. Which brings us to...

15.2 Testing

If you write your program modularly, it is easy (or at least: easier) to test the components without having to wait for an all-or-nothing test of the whole program. In an extreme form of this you would write your code by *test-driven development*: for each part of the program you would first write the test that it would satisfy.

In a more moderate approach you would use *unit testing*: you write a test for each program bit, from the lowest to the highest level.

And always remember the old truism that 'by testing you can only prove the presence of errors, never the absence.

Chapter 16

Index