Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2017

Contents

PART I

LESSONS

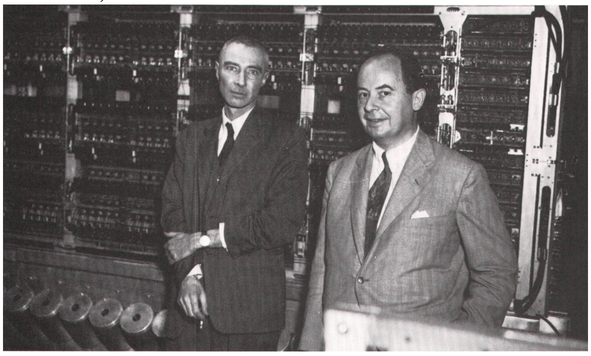
Chapter 1

Introduction

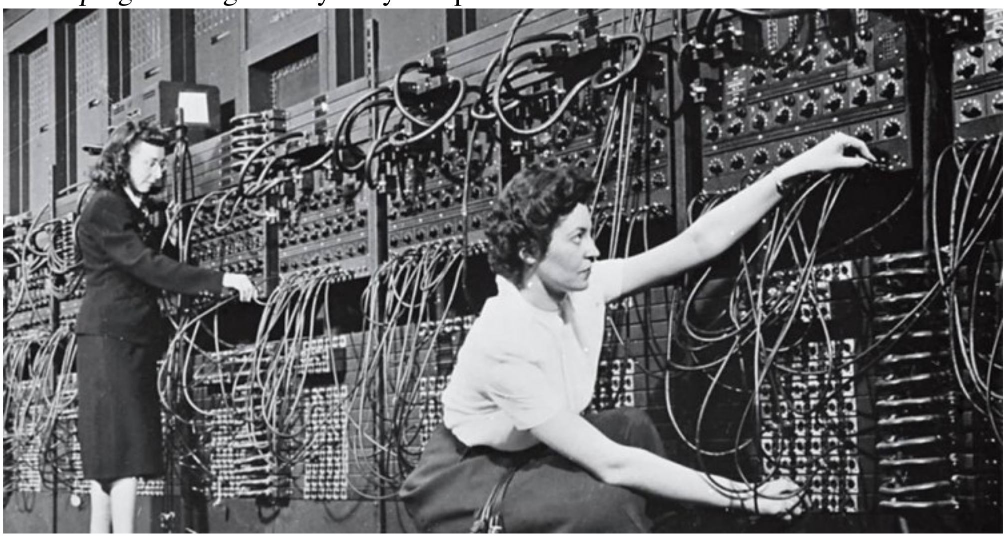
1.1 Programming and computational thinking

1.1.1 History

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



1.2. Hands-on programming Very early computers were hardwired



1.3. Program entry Later programs were written on punchcards

1.4. 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 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 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

- 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

- 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 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? 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?

> 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

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

1.13. 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 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 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*.
- *Microsoft Windows* users can use *putty* but it is probably a better solution to install a virtual environment such as *VMware* (`http://www.vmware.com/`) or *Virtualbox* (`https://www.virtualbox.org/`).

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

- 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 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 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

- 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 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 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 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 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 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 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 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 You have already seen `cout`:

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

3.13. 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 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

- 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

- Testing: `== != < > <= >=`
- Not, and, or: `! && ||`
- Bitwise: `& | ^`
- Shortcut operators:

```
if ( x>=0 && sqrt(x)<5 ) {}
```

3.17. 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. Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

$$C = (F - 32) \cdot 5/9, \qquad F = 9/5\,C + 32$$

Can you use Unix pipes to make one accept the output of the other?

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

$$n_1 \mod n_2$$

(without using the `%` modulus operator!)
Print out the value you get.
- Also print out the result from using the modulus operator: `%`.

3.5 Conditionals

3.18. 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;
}
```

- The 'else' part is optional
- You can leave out braces in case of single statement.

3.19. Complicated conditionals Chain:

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

Nest:

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

3.20. 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 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.

3.5.1 Further practice

The website `http://www.codeforwin.in/2015/05/if-else-programming-practice.html` lists the following exercises for conditional:

1. Write a C program to find maximum between two numbers.
2. Write a C program to find maximum between three numbers.
3. Write a C program to check whether a number is even or odd.
4. Write a C program to check whether a year is leap year or not.
5. Write a C program to check whether a number is negative, positive or zero.
6. Write a C program to check whether a number is divisible by 5 and 11 or not.
7. Write a C program to count total number of notes in given amount.
8. Write a C program to check whether a character is alphabet or not.
9. Write a C program to input any alphabet and check whether it is vowel or consonant.
10. Write a C program to input any character and check whether it is alphabet, digit or special character.
11. Write a C program to check whether a character is uppercase or lowercase alphabet.
12. Write a C program to input week number and print week day.
13. Write a C program to input month number and print number of days in that month.
14. Write a C program to input angles of a triangle and check whether triangle is valid or not.
15. Write a C program to input all sides of a triangle and check whether triangle is valid or not.
16. Write a C program to check whether the triangle is equilateral, isosceles or scalene triangle.
17. Write a C program to find all roots of a quadratic equation.
18. Write a C program to calculate profit or loss.

Chapter 4

Looping

4.1 Basic 'for' statement

4.1. 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 involving var
  cout << "The square of " << var << " is " << var*var << endl;
}
```

C difference: Use compiler flag `-std=c99`.

Exercise 4.1. Read an integer value, and print 'Hello world' that many times.

4.2. 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(;;) ...`

4.3. Nested loops Traversing a matrix:

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

4.2 Looping until

4.4. 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) { ... }
```

4.5. 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;
}
```

4.6. Skip iteration

```
for (int var=low; var<N; var++) {
  statement;
  if (some_test) {
    statement;
    statement;
  }
}
```

Alternative:

```
for (int var=low; var<N; var++) {
  statement;
  if (!some_test) continue;
  statement;
  statement;
}
```

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

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

or

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

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

4.8. While syntax 1

```
cout << "Enter a positive number: " ;
cin >> invar;
while (invar>0) {
  cout << "Enter a positive number: " ;
  cin >> invar;
```

```
}
cout << "Sorry, " << invar << " is negative" << endl;
```

Problem: code duplication.

4.9. While syntax 2

```
do {
  cout << "Enter a positive number: " ;
  cin >> invar;
} while (invar>0);
cout << "Sorry, " << invar << " is negative" << endl;
```

More elegant.

Exercise 4.2. One bank account has 100 dollars and earns a 5 percent per year interest rate. Another account has 200 dollars but earns only 2 percent per year. In both cases the interest is deposited into the account.
After how many years will the amount of money in both accounts be the same?

4.3 Exercises

Exercise 4.3. Find all triples of integers u, v, w under 100 such that $u^2 + v^2 = w^2$. Make sure you omit duplicates of solutions you have already found.

Exercise 4.4. 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: every time you find a sequence that is longer than the previous maximum, print out the starting number.

Exercise 4.5. 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`.

Exercise 4.6. **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 $|x_- - x_+| < 10^{-10}$.

4.3.1 Further practice

The website http://www.codeforwin.in/2015/06/for-do-while-loop-programming-exercises.html lists the following exercises:

1. Write a C program to print all natural numbers from 1 to n. - using while loop
2. Write a C program to print all natural numbers in reverse (from n to 1). - using while loop
3. Write a C program to print all alphabets from a to z. - using while loop
4. Write a C program to print all even numbers between 1 to 100. - using while loop
5. Write a C program to print all odd number between 1 to 100.
6. Write a C program to print sum of all even numbers between 1 to n.
7. Write a C program to print sum of all odd numbers between 1 to n.
8. Write a C program to print table of any number.
9. Write a C program to enter any number and calculate sum of all natural numbers between 1 to n.
10. Write a C program to find first and last digit of any number.
11. Write a C program to count number of digits in any number.
12. Write a C program to calculate sum of digits of any number.
13. Write a C program to calculate product of digits of any number.
14. Write a C program to swap first and last digits of any number.
15. Write a C program to find sum of first and last digit of any number.
16. Write a C program to enter any number and print its reverse.
17. Write a C program to enter any number and check whether the number is palindrome or not.
18. Write a C program to find frequency of each digit in a given integer.
19. Write a C program to enter any number and print it in words.
20. Write a C program to print all ASCII character with their values.
21. Write a C program to find power of any number using for loop.
22. Write a C program to enter any number and print all factors of the number.
23. Write a C program to enter any number and calculate its factorial.
24. Write a C program to find HCF (GCD) of two numbers.
25. Write a C program to find LCM of two numbers.
26. Write a C program to check whether a number is Prime number or not.
27. Write a C program to check whether a number is Armstrong number or not.
28. Write a C program to check whether a number is Perfect number or not.
29. Write a C program to check whether a number is Strong number or not.
30. Write a C program to print all Prime numbers between 1 to n.
31. Write a C program to print all Armstrong numbers between 1 to n.
32. Write a C program to print all Perfect numbers between 1 to n.
33. Write a C program to print all Strong numbers between 1 to n.
34. Write a C program to enter any number and print its prime factors.
35. Write a C program to find sum of all prime numbers between 1 to n.
36. Write a C program to print Fibonacci series up to n terms.
37. Write a C program to find one's complement of a binary number.
38. Write a C program to find two's complement of a binary number.
39. Write a C program to convert Binary to Octal number system.
40. Write a C program to convert Binary to Decimal number system.
41. Write a C program to convert Binary to Hexadecimal number system.
42. Write a C program to convert Octal to Binary number system.

43. Write a C program to convert Octal to Decimal number system.
44. Write a C program to convert Octal to Hexadecimal number system.
45. Write a C program to convert Decimal to Binary number system.
46. Write a C program to convert Decimal to Octal number system.
47. Write a C program to convert Decimal to Hexadecimal number system.
48. Write a C program to convert Hexadecimal to Binary number system.
49. Write a C program to convert Hexadecimal to Octal number system.
50. Write a C program to convert Hexadecimal to Decimal number system.
51. Write a C program to print Pascal triangle upto n rows.

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.

5.1. Turn blocks of code into functions

- Code fragment with clear function:
- Turn into *subprogram*: function *definition.*
- Use by single line: function *call.*

5.2. Function definition and call

```
for (int i=0; i<N; i++) {
  cout << i;
  if (i%2==0)
    cout << " is even";
  else
    cout << " is odd";
  cout << endl;
}
```

```
void report_evenness(int n) {
  cout << i;
  if (i%2==0)
    cout << " is even";
  else
    cout << " is odd";
  cout << endl;
}
...
int main() {
  ...
  for (int i=0; i<N; i++)
    report_evenness(i);
}
```

5.3. Why functions?

- Easier to read
- Shorter code: reuse
- Maintainance and debugging

- 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.

5.4. Prime function Using a function, primality testing would look like:

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

5.5. Anatomy of a function definition

- Result type: what's computed. `void` if no result

- Name: make it descriptive.
- Arguments: zero or more.
  `int i,double x,double y`
- Body: any length.
- Return statement: usually at the end, but can be anywhere; the computed result.

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;
- zero or more *function parameters* or *function arguments*: the data that the function operates on; and the
- *function body*: the statements that make up the function; and
- a *return* statement. Which doesn't have to be last, by the way.

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

5.6. Program with function

```
#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;
}
```

5.7. Function call 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`.)

5.8. Functions without input, without return result

```
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;
  }
```

5.9. 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;
}
```

5.10. Functions with return result

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

5.11. Scope Function body is a *scope*: local variables.
No local functions.

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

Functions can not be nested: you can not define a function inside the body of another function.

5.2.1 Parameter passing

5.12. Mathematical type function Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function.
- *pass by value*

Exercise 5.1. Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you want to compute

$$x = \sqrt{y}.$$

This is equivalent to finding the zero of

$$f(x) = x^2 - y.$$

Newton's method does this by evaluating

$$x_{\text{next}} = x - f(x)/f'(x)$$

until the guess is accurate enough.

- Write functions `f(x,y)` and `deriv(x,y)`, and a function `newton_root` that uses `f` and `deriv` to iterate to some precision.
- Take an initial guess for `x`, not zero.
- As a stopping test, use $|\mathtt{f}(\mathtt{x},\mathtt{y})| < 10^{-5}$.
- Use `double` for all your variables.

5.13. Results other than through return Also good design:

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

5.14. Pass by reference example

```
bool can_read_value( int &value ) {
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status!=0;
}
...
if (!can_read_value(n))
  // if you can't read the value, set a default
  n = 10;
```

Exercise 5.2. 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
```

Exercise 5.3. Write a function that tests divisibility and returns a remainder:

```
int number,divisor,remainder;
// get the number and divisor from the user
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

5.2.2 Recursive functions

5.15. Recursion Functions are allowed to call themselves, which is known as *recursion*. You can define factorial as

$$F(n) = n \times F(n-1) \qquad \text{if } n > 1\text{, otherwise } 1$$

```
int factorial( int n ) {
  if (n==1)
    return 1;
  else
    return n*factorial(n-1);
}
```

Exercise 5.4. The sum of squares:

$$S_n = \sum_{n=1}^{N} n^2$$

can be defined recursively as

$$S_1 = 1, \qquad S_n = n^2 + S_{n-1}.$$

Write a recursive function that implements this second definition. Test it on numbers that are input by the user.
Then write a program that prints the first 100 sums of squares.

Exercise 5.5. Write a recursive function for computing Fibonacci numbers:

$$F_0 = 1, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

First write a program that computes F_n for a value n that is input by the user.
Then write a program that prints out a sequence of Fibonacci numbers; the user should input how many.

If you let your Fibonacci program print out each time it computes a value, you'll see that most values are computed several times. (Math question: how many times?) This is wasteful in running time. This problem is addressed in section 24.3.1.

5.2.3 Polymorphic functions

5.16. Polymorphic functions You can have multiple functions with the same name:

```
double sum(double a,double b) {
  return a+b; }
double sum(double a,double b,double c) {
  return a+b+c; }
```

Distinguished by input parameters: can not differ only in return type.

5.2.4 Default arguments

5.17. Default arguments Functions can have *default argument*(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
```

Any default argument(s) should come last in the parameter list.

5.2.5 Static variables

Variables in a function have *lexical scope* limited to that function. Normally they also have *dynamic scope* limited to the function execution: after the function finishes they completely disappear. (Class objects have their *destructor* called.)

There is an exception: a *static variable* persists between function invocations.

```
void fun() {
  static int remember;
}
```

For example

```
int onemore() {
  static int remember++; return remember;
}
int main() {
  for ( ... )
    cout << onemore() << end;
  return 0;
}
```

gives a stream of integers.

Exercise 5.6. The static variable in the `onemore` function is never initialized. Can you find a mechanism for doing so? Can you do it with a default argument to the function?

5.2.6 Prototypes and separate compilation

5.18. Forward declaration Problem:

```
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

leads to compiler error.
Use prototype for *forward declaration*:

```
int f(int); // prototype of f
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

See chapter 14 for prototypes.

Chapter 6

Structures

6.1 Why structures?

6.1. Bundling information Sometimes a number of variables belong logically together. For instance two doubles can be the x, y components of a vector.
This can be captured in the `struct` construct.

```
struct vector { double x; double y; } ;
```

(This can go in the main program or before it.)
Initialize:

```
struct vector { double x=0.; double y=0.; } ;
```

6.2 The basics of structures

6.2. Using structures Once you have defined a structure, you can make variables of that type. Setting and initializing them takes a new syntax:

```
struct vector p1,p2;

p1.x = 1.; p1.y = 2.;
p2 = {3.,4.};

p2 = p1;
```

6.3. Functions on structures You can pass a structure to a function:

```
double distance( struct vector p1,struct vector p2 ) {
  double d1 = p1.x-p2.x, d2 = p1.y-p2.y;
  return sqrt( d1*d1 + d2*d2 );
}
```

6.4. Returning structures You can return a structure from a function:

```
struct vector vector_add
      ( struct vector p1,struct vector p2 ) {
   struct vector p_add = {p1.x+p2.x,p1.y+p2.y};
```

```
      return p_add;
   };
```

(Something weird here with scopes: the explanation is that the returned value is copied.)

Exercise 6.1. Write a function `inner_product` that takes two `vector` structures and computes the inner product.

Exercise 6.2. Write a 2×2 matrix class (that is, a structure storing 4 real numbers), and write a function `multiply` that multiplies a matrix times a vector.

Chapter 7

Classes and objects

7.1 What is an object?

7.1. Classes look a bit like objects

```
class Vector {
public:
  double x,y;
};

int main() {
  Vector p1;
  p1.x = 1.; p1.y = 2.;
```

We'll get to that '`public`' in a minute.

7.2. Class initialization and use Use a *constructor*:

```
class Vector {
public:
  double x,y;
  Vector( double userx,double usery ) {
    x = userx; y = usery;
  }
};

int main() {
  Vector p1(1.,2.);
```

7.3. Member initialization Other syntax for initialization:

```
class Vector {
public:
  double x,y;
  Vector( double userx,double usery ) : x(userx),y(usery) {
  }
};
```

7.4. Private data

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
  double x() { return vx; }; // 'accessor'
  double y() { return vy; };
};

int main() {
  Vector p1(1.,2.);
  cout << "p1 = " << p1.x() << "," << p1.y() << endl;
```

7.5. Functions on objects

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
  double length() { return sqrt(vx*vx + vy*vy); };
  double angle() { return 0.; /* something trig */; };
};

int main() {
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
```

We call such internal functions 'methods'

7.6. Methods that alter the object

```
class Vector {
  /* ... */
  void scaleby( double a ) {
    vx *= a; vy *= a; };
  /* ... */
};
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 has length " << p1.length() << endl;
```

7.7. Methods that create a new object

```
class Vector {
  /* ... */
  Vector scale( double a ) {
    return Vector( vx*a, vy*a ); };
  /* ... */
};
  /* ... */
  cout << "p1 has length " << p1.length() << endl;
  Vector p2 = p1.scale(2.);
  cout << "p2 has length " << p2.length() << endl;
```

7.8. Constructor

```
Vector p1(1.,2.), p2;
cout << "p1 has length " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 has length " << p2.length() << endl;
```

gives:

```
pointdefault.cxx: In function 'int main()':
pointdefault.cxx:32:21: error: no matching function for call to
                'Vector::Vector()'
   Vector p1(1.,2.), p2;
```

So:

```
Vector() {};
Vector( double x,double y ) {
  vx = x; vy = y;
};
```

7.2 Relations between classes

7.3 Inclusion relations between classes

7.9. Has-a relationship A class usually contains data members. These can be simple types or other classes. This allows you to make structured code.

```
class Course {
private:
  Person the_instructor;
  int year;
}
class Person {
  string name;
  ....
}
```

This is called the *has-a relation*.

At this time, do exercises in section 22.2.

7.10. Literal and figurative has-a Compare:

```
class Segment {
private:
  Point starting_point,ending_point;
}
  ...
  Segment somesegment;
  Point somepoint = somesegment.get_the_end_point();
```

Versus:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
}
```

Implementation vs API.

7.11. Polymorphism in constructors You have to decide what to store and what to derive, but you can construct two ways:

```
class Segment {
private:
 // up to you how to implement!
public:
  Segment( Point start,float length,float angle )
    { .... }
  Segment( Point start,Point end ) { ... }
```

Advantage: with a good API you can change your mind about the implementation!

At this time, do the exercises in section 22.3

7.4 Inheritance

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

```
class General {
protected: // note!
 int g;
public:
 void general_method() {};
};
class Special : public General {
public:
  void special_method() { g = ... };
};
```

7.13. Inheritance: derived classes 'Derived' class `Special` 'inherits' methods and data from 'base class' `General`:

```
int main() {
  Special special_object;
  special_object.general_method();
```

Data needs to be `protected`, not `private`, to be inheritable.

7.14. Constructors When you run the special case constructor, usually the general case needs to run too. By default the 'default constructor', but:

```
class General {
public:
  General( double x,double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
};
```

7.15. More

- Multiple inheritance: an X is-a A, but also is-a B.
  This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.

Chapter 8

Arrays

8.1 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 8.1. Check whether an array is sorted.

Exercise 8.2. 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 8.3. 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.

8.2 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);
```

8.3 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`.

8.4 Vector class for arrays

Syntax:

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

```
vector<type> name(size)
```

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.

A vector behaves like an array:

```
vector<double> x(25);
x[1] = 3.14;
cout << x[2];
```

There is a second way of accessing elements:

```
vector<double> x(5);
x[5] = 1.; // will probably work
x.at(5) = 1.; // runtime error!
```

Method `size` gives the size of the vector:

```
vector<char> words(37);
cout << words.size(); // will print 37
```

The vector class is a template class: the type that it uses (int, float) is not predetermined, but you can make a vector object out of whatever type you like.

8.4.1 Vector methods

- Get elements with `ar[3]` (zero-based indexing).
- Get elements, including bound checking, with `ar.at(3)`.
- Size: `ar.size()`.
- Other functions: `front`, `back`.

8.4.2 Vectors are dynamic

Use `push_back` to add elements at end.

```
vector<int> array(5);
array.push_back(35);
cout << array.size(); // is now 6 !
```

Other methods that change the size: `insert`, `erase`.

8.4.3 Vector assignment

The limitation that you couldn't create an array in an object still holds:

```
class witharray {
private:
  vector<int> the_array( ???? );
public:
  witharray( int n ) {
    thearray( ???? n ???? );
  }
}
```

The following mechanism works:

```
class witharray {
private:
  vector<int> the_array;
public:
  witharray( int n ) {
    thearray = vector<int>(n);
  }
}
```

You could read this as

- `vector<int> the_array` declares a int-vector variable, and
- `thearray = vector<int>(n)` assigns an array to it.

However, technically, it actually does the following:

- The class object initially has a zero-size vector;
- the expression `vector<int>(n)` creates an anonymous vector of size `n`;
- which is then assigned to the variable `the_array`,
- so now you have an object with a vector of size `n` internally.

8.4.4 Vectors and functions

8.4.4.1 Vector as function return

You can have a vector as return type of a function:

```
vector<int> make_array(int n) {
  vector<int> x(n);
  x[0] = n;
  return x;
}
  /* ... */
  x1 = make_array(10);
```

8.4.4.2 Pass vector to function

You can pass a vector to a function:

```
void print0( vector v ) {
  cout << v[0] << endl;
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

```
void set0( vector<float> &v,float x ) {
  v[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

This means you have to pass by reference:

Exercise 8.4. Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 99;
vec<float> values = random_floats(length);
sort(values);
```

(This creates a vector of random values of a specified length, and then sorts it.)

8.4.5 Dynamic size of vector

Writing

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

to add elements to the vector, dynamically resizing it.

8.4.6 Timing

Different ways of acessing a vector can have drastically different timing cost.

You can push elements into a vector:

```
vector<int> flex;
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
vector<int> stat(LENGTH);
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  stat.at(i) = i;
```

or with subscript:

```
vector<int> stat(LENGTH);
stat[0] = 0.;
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  stat[i] = i;
```

You can also use `new` to allocate:

```
int *stat = new int[LENGTH];
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  stat[i] = i;
```

Timings are partly predictable, partly surprising:

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

The increased time for `new` is a mystery.

8.5 Wrapping a vector in an object

You may want to a create objects that contain a vector, for instance because you want to add some methods.

```
class printable {
private:
  vector<int> values;
public:
  printable(int n) {
    values = vector<int>(n);
  };
```

```
    string stringed() {
      string p("");
      for (int i=0; i<values.size(); i++)
        p += to_string(values[i])+" ";
      return p;
    };
  };
```

Unfortunately this means you may have to recreate some methods:

```
  int &at(int i) {
    return values.at(i);
  };
```

8.6 Multi-dimensional cases

8.6.1 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.

8.6.2 Matrix class based on vector

Make sure you've studied classes; section **??**.

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

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

8.7 Exercises

Exercise 8.6. Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

Exercise 8.7. 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 8.8. Put eight queens on a chessboard so that none threatens any other.

Exercise 8.9. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

Chapter 9

Strings

9.1 Basic string stuff

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable, or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Strings can be *concatenated*:

```
txt = txt1+txt2;
txt += txt3;
```

You can query the *size*:

```
int txtlen = txt.size();
```

You can get individual characters by using a subscript:

```
cout << "The second character is <<" << txt[1] << ">>" << endl;
```

Indexing is zero-based.

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

`http://en.cppreference.com/w/cpp/string/basic_string`

Exercise 9.1. Write a function to convert an integer to a string: the input `205` should give `two hundred fifteen`, et cetera.

Exercise 9.2. Write a pattern matcher, where a period `.` matches any one character, and `x*` matches any number of ‘`x`’ characters.
For example:

- The string `abc` matches `a.c` but `abbc` doesn’t.
- The string `abbc` matches `ab*c`, as does `ac`, but `abzbc` doesn’t.

9.2 Conversion

`to_string`

9.3 `iostream`

Chapter 10

Input/output

10.1 Formatted output

10.1. Default output Normally, output of numbers takes up precisely the space that it needs:

```
cout << "Unformatted:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << endl;
cout << endl;
```

10.2. Output

```
Unformatted:
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

10.3. Reserve space You can specify the number of positions, and the output is right aligned in that space by default:

```
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << setw(6) << i << endl;
cout << endl;
```

(Only applies to immediately following number)

10.4. Output

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
```

```
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

10.5. Padding character Normally, padding is done with spaces, but you can specify other characters:

```
cout << "Padding:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
       << left << setfill('.') << setw(6) << i << endl;
```

Note: single quotes denote characters, double quotes denote strings.
Note: many of these output modifiers need

```
#include <iomanip>
```

10.6. Output

```
Padding:
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

10.7. Left alignment Instead of right alignment you can do left:

```
cout << "Padding:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
       << left << setfill('.') << setw(6) << i << endl;
cout << endl;
```

10.8. Output

```
Padding:
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
```

```
Number: 10000000
Number: 100000000
```

10.9. Number base Finally, you can print in different number bases than 10:

```
cout << "Base 16:" << endl;
cout << setbase(16) << setfill(' ');
for (int i=0; i<16; i++) {
  for (int j=0; j<16; j++)
    cout << i*16+j << " " ;
  cout << endl;
}
cout << endl;
```

10.10. Output

```
Base 16:
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
etc
```

Exercise 10.1. Make the above output more nicely formatted:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
```

Exercise 10.2. Use integer output to print fixed point numbers aligned on the decimal:

```
  1.345
 23.789
456.1234
```

Use four spaces for both the integer and fractional part.

10.2 Floating point output

10.11. Floating point precision Use `setprecision` to set the number of digits before and after decimal point:

```
x = 1.234567;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x << endl;
  x *= 10;
}
```

10.12. Output

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

(Notice the rounding)

10.13. Fixed point precision Fixed precision applies to fractional part:

```
cout << "Fixed precision applies to fractional part:" << endl;
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x << endl;
  x *= 10;
}
```

10.14. Output

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

10.15. Aligned fixed point output Combine width and precision:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4) << x << endl;
  x *= 10;
}
```

10.16. Output

```
    1.2346
   12.3457
  123.4567
```

```
 1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

10.17. Scientific notation

```
cout << "Combine width and precision:" << endl;
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4) << x << endl;
  x *= 10;
}
```

10.18. Output

```
Combine width and precision:
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

10.3 Saving and restoring settings

```
ios::fmtflags old_settings = cout.flags();
```

```
cout.flags(old_settings);
```

```
int old_precision = cout.precision();
```

```
cout.precision(old_precision);
```

10.4 File output

10.19. Text output to file

```
#include <fstream>
```

Use:

```
ofstream file_out;
file_out.open("fio_example.out");
/* ... */
file_out << number << endl;
file_out.close();
```

Chapter 11

References and addresses

11.1 Reference

This section contains further facts about parameter passing. Make sure you study section 5.2.1 first.

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, since you can not 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 12

Polymorphism

12.1 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 13

Memory

13.1 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.

```
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces
.. deleting 0 spaces.. deleting 0 spaces.. deleting 0 spacesclass SomeObjec
.. deleting 0 spaces.. deleting 0 spaces.. deleting 0 spacesclass SomeObjec
.. deleting 0 spaces.. deleting 0 spaces.. deleting 0 spacesclass SomeObjec
.. deleting 0 spaces.. deleting 0 spaces.. deleting 0 spacesclass SomeObjec
public:
public:
public:
public:
  SomeObject() { cout << "calling the constructor" << endl; };
  SomeObject() { cout << "calling the constructor" << endl; };
  SomeObject() { cout << "calling the constructor" << endl; };
  SomeObject() { cout << "calling the constructor" << endl; };
  ~SomeObject() { cout << "calling the destructor" << endl; };
  ~SomeObject() { cout << "calling the destructor" << endl; };
```

```
    ~SomeObject() { cout << "calling the destructor" << endl; };
    ~SomeObject() { cout << "calling the destructor" << endl; };
  };
  };
  };
  };
    cout << "Before the nested scope" << endl;
    cout << "Before the nested scope" << endl;
    cout << "Before the nested scope" << endl;
    cout << "Before the nested scope" << endl;
    {
    {
    {
    {
      SomeObject obj;
      SomeObject obj;
      SomeObject obj;
      SomeObject obj;
      cout << "Inside the nested scope" << endl;
      cout << "Inside the nested scope" << endl;
      cout << "Inside the nested scope" << endl;
      cout << "Inside the nested scope" << endl;
    }
    }
    }
    }
    cout << "After the nested scope" << endl;
    cout << "After the nested scope" << endl;
    cout << "After 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 14

Prototypes

14.1 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);
```

A first use of prototypes is *forward declaration*:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

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.

14.1.1 Header files

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.

14.1.2 C and C++ headers

You have seen the following syntaxes for including header files:

```
#include <header.h>
#include "header.h"
```

The first is typically used for system files, with the second typically for files in your own project. There are some header files that come from the C standard library such as `math.h`; the idiomatic way of including them in C++ is

```
#include <cmath>
```

14.2 Global variables

If you have a variable that you want known everywhere, you can make it global:

```
int processnumber;
int main() {
  processnumber = // some system call
};
```

It is then defined in functions defined in your program file.

If your program has multiple files, you should not put '`int processnumber`' in the other files, because that would create a new variable, that is only known to the functions in that file. Instead use:

```
extern int processnumber;
```

which says that the global variable `processnumber` is defined in some other file.

What happens if you put that variable in a *header file*? Since the *preprocessor* acts as if the header is textually inserted, this again leads to a separate global variable per file. The solution then is more complicated:

```
//file: header.h
#ifndef HEADER_H
#define HEADER_H
#ifndef EXTERN
#define EXTERN extern
#fi
EXTERN int processnumber
#fi

//file: aux.cc
#include "header.h"

//file: main.cc
#define EXTERN
```

```
#include "header.h"
```

This also prevents recursive inclusion of header files.

14.3 Prototypes for class methods

Header file:

```
class something {
public:
  double somedo(vector);
};
```

Implementation file:

```
double something::somedo(vector v) {
   .... something with v ....
};
```

14.4 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.

14.5 Namespaces and header files

Never put `using namespace` in a header file.

Chapter 15

Efficiency

15.1 Order of complexity

15.1.1 Time complexity

Exercise 15.1. 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 15.2. How many operations, as a function of n, are performed in these two solutions?

15.1.2 Space complexity

Exercise 15.3. 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 15.4. How much space do the two solutions require?

Chapter 16

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.

16.1 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.

16.2 Parametrized macros

Instead of simple text substitution, you can have *parametrized preprocessor macros*

```
#define CHECK_FOR_ERROR(i) if (i!=0) return i
...
ierr = some_function(a,b,c); CHECK_FOR_ERROR(ierr);
```

When you introduce parameters, it's a good idea to use lots of parentheses:

```
// the next definition is bad!
#define MULTIPLY(a,b) a*b
...
x = MULTIPLY(1+2,3+4);
```

Better

```
#define MULTIPLY(a,b) (a)*(b)
...
x = MULTIPLY(1+2,3+4);
```

Another popular use of macros is to simulate multi-dimensional indexing:

```
#define INDEX2D(i,j,n) (i)*(n)+j
...
double array[m,n];
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    array[ INDEX2D(i,j,n) ] = ...
```

Exercise 16.1. Write a macro that simulates 1-based indexing:

```
#define INDEX2D1BASED(i,j,n)  ????
...
double array[m,n];
for (int i=1; i<=m; i++)
  for (int j=n; j<=n; j++)
    array[ INDEX2D1BASED(i,j,n) ] = ...
```

16.3 Conditionals

There are a couple of *preprocessor conditions*.

16.3.1 Check on a value

The `#if` macro tests on nonzero. A common application is to temporarily remove code from compilation:

```
#if 0
  bunch of code that needs to
  be disabled
#endif
```

16.3.2 Check for macros

The `#ifdef` test tests for a macro being defined. Conversely, `#ifndef` tests for a macro not being defined. For instance,

```
#ifndef N
#define N 100
#fi
```

Why would a macro already be defined? Well you can do that on the compile line:

```
icpc -c file.cc -DN=500
```

Another application for this test is in preventing recursive inclusion of header files; see section 14.2.

Chapter 17

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 8 you saw how you could create an array of ints as `vector<int>` and of doubles as `vector<double>`. Here you will learn the mechanism for that.

17.1. Templated type name Basically, you want the type name to be a variable. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...
```

17.2. Example: function Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 17.1. Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.
Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

```
float float_eps;
epsilon(float_eps);
cout << "For float, epsilon is " << float_eps << endl;

double double_eps;
epsilon(double_eps);
cout << "For double, epsilon is " << double_eps << endl;
```

17.1 Templating over non-types

THESE EXAMPLES ARE NOT GOOD.

See: https://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Temp

17.3. Templating a value Templeting over integral types, not double.

The templated quantity is a value:

```
template<int s>
std::vector<int> svector(s);
  /* ... */
  svector(3) threevector;
  cout << threevector.size();
```

Exercise 17.2. Write a class that contains an array. The length of the array should be templated.

Chapter 18

Error handling

18.1 General discussion

Sources or errors:

- Array indexing. See section 8.4.
- Null pointers
- Division by zero and other numerical errors.

Guarding against errors.

- Check preconditions.
- Catch results.
- Check postconditions.

Error reporting:

- Message
- Total abort
- Exception

Assertions:

```
#include <cassert>
...
assert( bool )
```

assertions are omitted with optimization

Function return values

18.2 Exception handling

```
void do_something() {
  if ( oops )
    throw(5);
}

throw {
  do_something();
```

```
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

Sophisticated:

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

Multiple types of exceptions

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std:;string c ) {
  // handle string exception
}
```

Catch all exceptions:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

- Functions can define what exceptions they throw:
  ```
  void func() throw( MyError, std::string );
  void funk() throw();
  ```
- Predefined exceptions: `bad_alloc`, `bad_exception`
- An exception handler can throw an ecxeption; to rethrow the same exception use ‘`throw;`’ without arguments.
- Exceptions delete all stack data, but not `new` data.

Chapter 19

Standard Template Library

The C++ language has a *Standard Template Library* (STL), which contains functionality that is considered standard, but that is actualy implemented in terms of already existing language mechanisms. The STL is enormous, so we just highlight a couple of parts.

You have already seen arrays (chapter 8) and strings (chapter 9).

19.1 Streams

You have used statements like:

```
cout << "My value is: " << myvalue << endl;
```

How does this work? The 'double less' is an operator with a left operand that is a stream, and a right operand for which output is defined; the result of this operator is again a stream. Recursively, this means you can chain any number of applications of << together.

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
class container {
  /* ... */
  int value() const {
  /* ... */
  };
  /* ... */
std::ostream &operator<<(std::ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
  /* ... */
  container eye(5);
  cout << eye << endl;
```

19.2 About the ‘using’ keyword

Only use this internally, not in header files that the user sees.

Chapter 20

Table of exercises

20.1 cplusplus

http://www.cplusplus.com/forum/articles/12974/

Dungeon crawl.

20.2 world best learning center

http://www.worldbestlearningcenter.com/index_files/cpp-tutorial-variables_datatypes_exercises.htm

PART II

PROJECTS

Chapter 21

Prime numbers

21.1 Preliminaries

Assuming you have learned about

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

21.2 Arithmetic

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

Exercise 21.1. Read two integers into two variables, and print their sum, product, quotient, modulus.

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

Exercise 21.2. 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.

21.3 Conditionals

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

Exercise 21.3. 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.

21.4 Looping

Control structures such as loops; section 4.1.

Exercise 21.4. Read an integer and determine whether it is prime by testing for the smaller numbers whether they are a divisor of that number.

Print a final message

```
Your number is prime
```

or

```
Your number is not prime: it is divisible by ....
```

where you report just one found factor.

Exercise 21.5. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

21.5 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 21.6. 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);
```

Read the number in, and print the value of the boolean.

21.6 While loops

Before doing this section, make sure you study section 4.2.

Exercise 21.7. Take the prime number testing program, and modify it to read in how many prime numbers you want to print. Print that many successive primes. Keep a variable `number_of_primes_found` that is increased whenever a new prime is found.

21.7 Global variables: optional

Before doing this section, make sure you study section 14.1.

Exercise 21.8. Use global variables to rewrite exercise 21.7. 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.

21.8 Structures

Before doing this section, make sure you study section 7.1, 11.1.

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

Exercise 21.9. Rewrite the exercise that found a predetermined number of primes, putting the `number_of_primes_found` and `last_number_tested` variables in a structure. Your main program should now look like:

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

21.9 Classes and objects

Before doing this section, make sure you study section **??**.

In exercise 21.9 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 21.10. 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 21.11. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to 20 million. Make an outer loop over the even numbers. In each iteration, make a `primesequence` object to generate p values. Then, for each p, make a second `primesequence` object to generate q values, and test with these.
For each even number, print out how it is the sum of two primes. If multiple possibilities exist, only print the first one you find.

21.10 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 21.12. 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 22

Geometry

This uses the material in section 7.

22.1 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 22.1. Make class `Point` with a constructor

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

Write a method `distance_to_origin` that returns a `float`.
Write a method `printout` that uses `cout` to display the point.
Write a function `distance` so that if `p,q` are `Point` objects,

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

computes the distance.

Exercise 22.2. Make a default constructor for the point class:

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

which you can use as:

```
Point p;
```

Exercise 22.3. Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 8.4. Can you make a constructor where you do not specify the space dimension explicitly?

22.2 Using one class in another

Exercise 22.4. Make a class `LinearFunction` with a constructors:

```
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;
```

Exercise 22.5. Make a class `LinearFunction` with two constructors:

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

where the first stands for a line through the origin.
Implement again the `evaluate` function so that

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

22.3 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.

22.1. Axi-parallel rectangle class Intended API:

```
float Rectangle::area();
```

It would be convenient to store width and height; for

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 22.6. Make a class `Rectangle` (sides parallel to axes) 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.

22.4 Is-a relationship

Read section 7.4.

Exercise 22.7. Take your code where a `Rectangle` was defined from one point, width, and height.
Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.
First ask yourself: what should the constructor of a `Square` look like?

Exercise 22.8. Revisit the `LinearFunction` class. Add methods `slope` and `intercept`.
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 23

PageRank

23.1 Basic ideas

Assuming you have learned about arrays 8, 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 23.1. 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

ALGORITHMS AND DATA STRUCTURES

Chapter 24

Tiniest of introductions to algorithms and data structures

24.1 Data structures

This really goes beyond this book.

- Arrays
- Lists
- Trees
- Graphs / DAGs
- Hashes

24.2 Algorithms

This *really* **really** goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

24.2.1 Sorting

An array a of length n is sorted if

$$\forall_{i<n-1}\colon a_i \leq a_{i+1}.$$

A simple sorting algorithm suggests itself immediately: if i is such that $a_i > a_{i+1}$, then reverse the i and $i+1$ locations in the array.

(Why is the array argument passed by reference?)

24.3 Programming techniques

24.3.1 Memoization

In section 5.2.2 you saw some examples of recursion. The factorial example could be written in a loop, and there are both arguments for and against doing so.

The Fibonacci example is more subtle: it can not immediately be converted to an iterative formulation, but there is a clear need for eliminating some waste that comes with the simple recursive formulation. The technique we can use for this is known as *memoization*: store intermediate results to prevent them from being recomputed.

Here is an outline.

```
int fibonacci(int n) {
  std::vector<int> fibo_values(n);
  for (int i=0; i<n; i++)
    fibo_values[i] = 0;
  fibonacci_memoized(fibo_values,n-1);
  return fibo_values[n-1];
}
int fibonacci_memoized( std::vector<int> &values, int top ) {
  int minus1 = top-1, minus2 = top-2;
  if (top<2)
    return 1;
  if (values[minus1]==0)
    values[minus1] = fibonacci_memoized(values,minus1);
  if (values[minus2]==0)
    values[minus2] = fibonacci_memoized(values,minus2);
  values[top] = values[minus1]+values[minus2];
  //cout << "set f(" << top << ") to " << values[top] << endl;
  return values[top];
}
//codesnipet end

int main() {
  int fibo_n;
  cout << "What number? ";
  cin >> fibo_n;
  cout << "Fibo(" << fibo_n << ") = " << fibonacci(fibo_n) << endl;

  return 0;
}
```

PART IV

REFERENCE

Chapter 25

Programming strategies

25.1 Programming: top-down versus bottom up

The exercises in chapter 21 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 21.10.

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…

25.1.1 Worked out example

Take a look at exercise 4.4. We will solve this in steps.

1. State the problem:
```
// find the longest sequence
```

2. Refine by introducing a loop
```
// find the longest sequence:

// Try all starting points
// If it gives a longer sequence report
```

3. Introduce the actual loop:
```
// Try all starting points
for (int starting=2; starting<1000; starting++)
// If it gives a longer sequence report
```

4. Record the length:
```
// Try all starting points
int maximum_length=-1;
for (int starting=2; starting<1000; starting++) {
  // If the sequence gives a longer sequence report:
  int length=0;
  // compute the sequence
  if (length>maximum_length) {
    // Report this sequence as the longest
  }
}
```

5.
```
// Try all starting points
int maximum_length=-1;
for (int starting=2; starting<1000; starting++) {
  // If the sequence gives a longer sequence report:
  int length=0;
  // compute the sequence
  int current=starting;
  while (current!=1) {
    // update current value
    length++;
  }
  if (length>maximum_length) {
    // Report this sequence as the longest
  }
```

```
}
```

25.2 Coding style

After you write your code there is the issue of *code maintainance*: you may in the future have to update your code or fix something. You may even have to fix someone else's code or someone will have to work on your code. So it's a good idea to code cleanly.

Naming Use meaningful variable names: `record_number` instead `rn` or `n`. This is sometimes called 'self-documenting code'.

Comments Insert comments to explain non-trivial parts of code.

Reuse Do not write the same bit of code twice: use macros, functions, classes.

25.3 Documentation

Take a look at Doxygen.

25.4 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 26

Index