Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2017

Contents

PART I

INTRODUCTION

Chapter 1

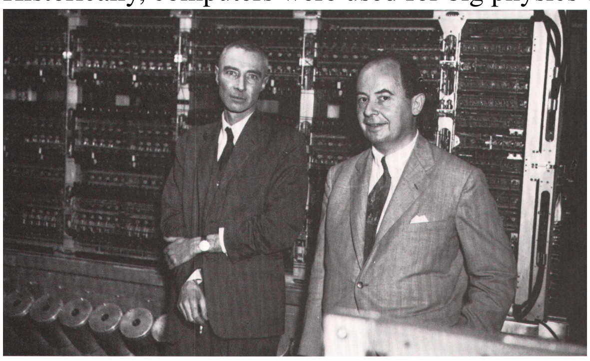
Introduction

1.1 Programming and computational thinking

1.1.1 History

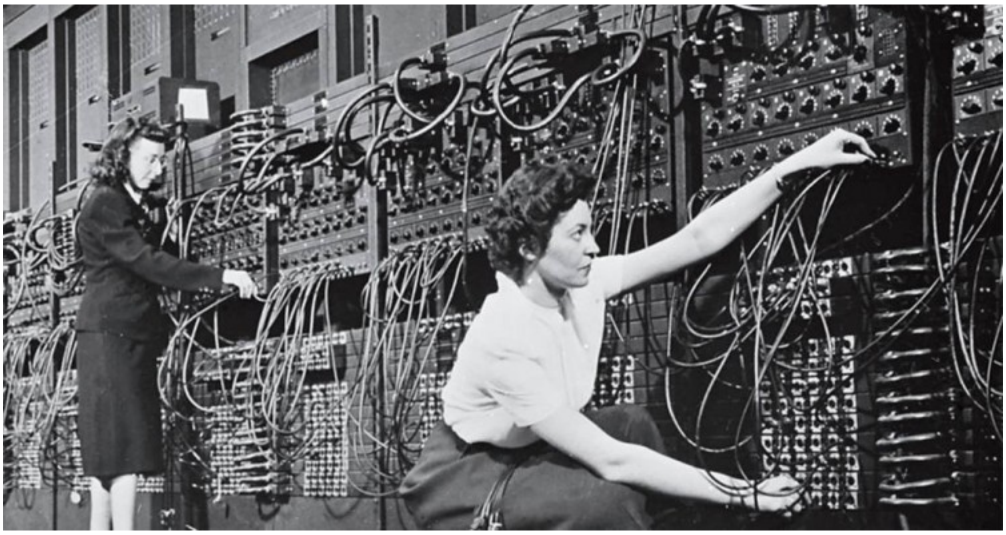
Earliest computers

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



Hands-on programming

Very early computers were hardwired



Program entry

Later programs were written on punchcards

The first programming language

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

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

Programming is not simple

Programs can get pretty big:

Margaret Hamilton, director of the Software Engineering Division, the MIT Instrumentation Laboratory, which developed on-board flight software for the Apollo space program.

It's not just translating formulas anymore.

Translating ideas to computer code: computational thinking.

Computational thinking: elevator scheduling

Mathematical thinking:

- Number of people per day, speed of elevator ⇒ yes, it is possible to get everyone to the right floor.
- Distribution of people arriving *etc.* ⇒ average wait time.

Sufficient condition ≠ existence proof.

Computational thinking: actual design of a solution

- Elevator scheduling: someone at ground level presses the button, there are cars on floors 5 and 10; which one do you send down?

Coming up with a strategy takes creativity!

Exercise 1.1. A straightforward calculation is the simplest example of an algorithm.
Calculate how many schools for hair dressers the US can sustain. Identify the relevant factors, estimate their sizes, and perform the calculation.

Exercise 1.2. Algorithms are usually not uniquely determined. There can be cleverness involved.
Four self-driving cars arrive simultaenously at an all-way-stop intersection. Come up with an algorithm that a car can follow to safely cross the intersection. If you can come up with more than one algorithm, what happens two cars using different algorithms meet each other?

Computation and complexity

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.

What is the average search time in the two cases?

Having a correct solution is not enough!

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.

A program has layers of abstractions.

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

A program contains structures that support the algorithm. You may have to design them yourself.

1.1.3 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

Advanced programmers know that hardware influences the speed of execution (see TACC's ISTC course).

1.1.4 Algorithms

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

Program steps

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

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

There are many programming languages, and not every language is equally suited for every purpose. In this book you will learn C++ and Fortran, because they are particularly good for scientific computing. And by ‘good’, we mean

- They can express the sorts of problems you want to tackle in scientific computing, and
- they execute your program efficiently.

There are other languages that may not be as convenient or efficient in expressing scientific problems. For instance, *python* is a popular language, but typically not the first choice if you’re writing a scientific program. As an illustration, here is simple sorting algorithm, coded in both C++ and python.

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]:
      swaptmp = numbers[j+1]
      numbers[j+1] = numbers[j]
      numbers[j] = swaptmp
```

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

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

But this ignores one thing: the sorting algorithm we just implemented is not actually a terribly good one, and in fact python has a better one built-in.

The right language is not all

Python with quicksort algorithm:

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

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

So that is another consideration when choosing a language: is there a language that already comes with the tools you need. This means that your application may dictate the choice of language. If you’re stuck with one language, don’t reinvent the wheel! If someone has already coded it or it’s part of the language, don’t redo it yourself.

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 if you're going to be doing some computational science 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*.

2.1.1 Language support in your editor

The author of this book is very much in favour of the *emacs* editor. The main reason is its support for programming languages. Most of the time it will detect what language a file is written in, based on the file extension:

- `cxx,cpp,cc` for C++, and
- `f90,F90` for Fortran.

If your editor somehow doesn't detect the language, you can add a line at the top of the file:

```
// -*- c++ -*-
```

for C++ mode, and

```
! -*- f90 -*-
```

for Fortran mode.

Main advantages are automatic indentation (C++ and Fortran) and supplying block end statements (Fortran). The editor will also apply 'syntax colouring' to indicate the difference between keywords and variables.

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 source 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.)

Here is the workflow for program development

1. You think about how to solve your program
2. You write code using an editor. That gives you a source file.
3. You compile your code. That gives you an executable.
   Oh, make that: you try to compile, because there will probably be compiler errors: places where you sin against the language syntax.
4. You run your code. Chances are it will not do exactly what you intended, so you go back to the editing step.

Chapter 3

Teachers guide

This book was written for a one-semester introductory programming course at The University of Texas at Austin, aimed primarily at students in the physical and engineering sciences. Thus, examples and exercises are as much as possible scientifically motivated. This target audience also explains the inclusion of Fortran.

This book is not encyclopedic. Rather than teaching each topic in its full glory, the author has taken a 'good practices' approach, where students learn enough of each topic to become a competent programmer. This serves to keep this book at a manageable length, and to minimize class lecture time, emphasizing lab exercises instead.

Even then, there is more material here than can be covered and practiced in one semester. If only C++ is taught, it is probably possible to cover the whole of Part II; for the case where both C++ and Fortran are taught, we have a suggested timeline below.

3.1 Justification

The chapters of Part II and Part III are presented in suggested teaching order. Here we briefly justify our (non-standard) sequencing of topics and outline a timetable for material to be covered in one semester. Most notably, Object-Oriented programming is covered before arrays and pointers come very late, if at all.

There are several thoughts behind this. For one, dynamic arrays in C++ are most easily realized through the `std::vector` mechanism, which requires an understanding of classes. The same goes for `std::string`.

Secondly, in the traditional approach, OOP is taught late, if at all, in the course. We consider OOP to be an important notion in program design, and central to C++, rather than an embellishment on the traditional C mechanisms.

3.2 Timeline for a C++/F03 course

As remarked above, this book is based on a course that teaches both C++ and Fortran2003. Here we give the timeline used, including some of the assigned exercises.

For a one semester course of slightly over three months, two months would be spent on C++ (see table 3.1), after which a month is enough to explain Fortran. Remaining time will go to exams and elective topics.

lesson#	date	Topic	Exercises prime	geom	infect
1	1/18	Statements and expressions	38.1, 38.2		
2	1/24	Conditionals	38.3		
3	1/26	Control structures			
4	1/31	Looping	38.4, 38.5, 38.7		
5		continue			
6	2/06	Functions	38.6		
7		continue			
8	2/12	I/O (lecture 8)			
9	2/19	Structs	38.8		
10	2/23	Objects	38.9, 38.11	40.3	
11		continue			
12	2/28	has-a relation		40.6, 40.7, 40.1, 40.8	
13	3/02	inheritance		40.9, 40.10	
14	3/07	Arrays			**??**, 39.3, 39.4, 39.5
15		continue			
16	3/23	Strings			

Table 3.1: Two-month lesson plan for C++

3.2.1 Project-based teaching

To an extent it is inevitable that students will do a number of exercises that are not connected to any earlier or later ones. However, to give some continuity, we have given some programming projects that students gradually build towards.

prime Prime number testing, culminating in prime number sequence objects, and testing a corollary of the Goldbach conjecture.

geom Geometry related concepts; this is mostly an exercise in object-oriented programming.

infect The spreading of infectuous diseases; these are exercises in basic array programming.

Rather than including the project exercises in the didactic sections, each section of these projects list the prerequisite basic sections.

3.2.2 Fortran or advanced topics

After two months of grounding in OOP programming in C++, the Fortran lectures and exercises reprise this sequence, letting the students do the same exercises in Fortran that they did in C++. However, array mechanisms in Fortran warrant a separate lecture.

If the course focuses solely on C++, the third month can be devoted to

- templates,
- exceptions,

- namespaces,
- multiple inheritance,
- the cpp preprocessor,
- closures.

PART II

C++

Chapter 4

Basic elements of C++

4.1 From the ground up: Compiling C++

In this chapter and the next you are going to learn the C++ language. But first we need some externalia: how do you deal with any program?

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 the `g++` (or `icpc`) compiler, with output `myprogram`, on `myprogram.cxx`'.

So let's do an example.

This is a minimal program:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

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.

Exercise 4.1. Make a program file with the above lines, compile it and run it.

As you may have guessed, this program produces absolutely no output.

Here is a statement that at least produces some output:

Code:

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

Output:

```
./hello
Hello world!
```

Exercise 4.2. Make a program source file that contains the 'hello world' statement, compile it and run it. Think about where the statement goes.
(Did you indent the 'hello world' line? Did you editor help you with the indentation?)

File names

File names can have extensions: the part after the dot.

- `program.cxx` or `program.cc` are typical extensions for C++ sources.
- `program.cpp` is sometimes used, but your instructor does not like that.
- `program` without extension usually indicates an *executable*.

4.2 Statements

Each programming language has its own (very precise!) rules for what can go in a source file. Globally we can say that a program contains instructions for the computer to execute, and these instructions take the form of a bunch of 'statements'. Here are some of the rules on statements; you will learn them in more detail as you go through this book.

Program statements

- A program contains statements, each terminated by a semicolon.
- 'Curly braces' can enclose multiple statements.
- A statement can correspond to some action when the program is executed.
- Some statements are definitions, of data or of possible actions.
- 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 4.3. Take the 'hello world' program you wrote above, 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.

Fixed elements

You see that certain parts of your program are inviolable:

- There are *keywords* such as `return` or `cout`; you can not change their definition.
- Curly braces and parentheses need to be matched.
- There has to be a `main` keyword.
- The `iostream` and `namespace` are usually needed.

Exercise 4.4. 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 the computation $1/3$,

with the same `cout` statement?

4.3 Variables

A program could not do much without storing data: input data, temporary data for intermediate results, and final results. Data is stored in *variables*, which have

- a name, so that can refer to them,
- a *datatype*, and
- a value.

Think of a variable as a labeled placed in memory.

- The variable is defined in a *variable declaration*,
- which can include an *variable initialization*.
- After a variable is defined, and given a value, it can be used,
- or given a (new) value in a *variable assignment*.

Typical variable lifetime

```
int i,j; // declaration
i = 5; // set a value
i = 6; // set a new value
j = i+1; // use the value of i
i = 8; // change the value of i
       // but this doesn't affect j:
       // it is still 7.
```

4.3.1 Variable declarations

A variable is defined once in a *variable declaration*, but it can be given a (new) value multiple times. It is not an error to use a variable that has not been given a value, but it may lead to strange behaviour at runtime, since the variable may contain random memory contents.

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 matters whether you use upper or lowercase: the language is *case sensitive*.

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;
```

Where do declarations go?

Declarations can go pretty much anywhere in your program, but need to come before use of the variable.

Note: it is legal to define a variable before the main program
but that's not a good idea. Please only declare *inside* main
(or inside a function et cetera).

4.3.2 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.

4.3.3 Assignments

Setting a variable

```
i = 5;
```

means storing a value in the memory location. It is not defining a mathematical equality

let $i = 5$.

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;
```

Variable of the left-hand side gets value of the right-hand side.

You see that you can assign both a simple value or an *expression*.

Assignments

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

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

These are not math equations: variable on the lhs gets the value of the rhs.

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];
```

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};
```

Do not use uninitialized variables! Doing so is legal, but there is no guarantee about the initial value. Do not count on it being zero. . .

Exercise 4.5. Write a program that has several variables. Assign values either in an initialization or in an assignment. Print out the values.

4.3.4 Floating point variables

Mathematically, integers are a special case of real numbers. In a computer, integers are stored very differently from *floating point* numbers.

- Within a certain range, roughly $-2 \cdot 10^9, \ldots, 2 \cdot 10^9$, all integer values can be represented.
- On the other hand, not all real numbers have a floating point representation. For instance, since computer numbers are binary, $1/2$ is representable but $1/3$ is not.
- You can assign variables of one type to another, but this may lead to truncation (assigning a floating point number to an integer) or unexpected bits (assigning a single precision floating point number do a double precision).

Floating point constants

- Default: `double`
- Float: `3.14f` or `3.14F`
- Long double: `1.22l` or `1.22L`.

This prevents numerical accidents:

```
double x = 3.;
```

converts float to double, maybe introducing random bits.

Warning: floating point arithmetic

Floating point arithmetic is full of pitfalls.

- Don't count on `3*(1./3)` being exactly 1.
- Not even associative.

(See Eijkhout, Introduction to High Performance Computing, chapter 3.)

4.3.5 Boolean 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};
found = true;
```

Exercise 4.6. Print out `true` and `false`. What do you get?

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

A program typically produces output. For now we will only display output on the screen, but output to file is possible too. Regarding input, sometimes a program has all information for its computations, but it is also possible to base the computation on user input.

Terminal output

You have already seen `cout`:

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

Terminal input

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

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

There is also `getline`, which is more general.

Exercise 4.7. 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.

For more I/O, see chapter 13.

4.5 Expressions

The most basic step in computing is to form expressions such as sums, products, logical conjuctions, string appending. Expressions in programming languages for the most part look the way you would expect them to.

- Mathematical operators: `+ - /` and `*` for multiplication.
- C++ does not have a power operator (Fortran does).
- Integer modulus: `5%2`
- You can use parentheses: `5*(x+y)`. Use parentheses if you're not sure about the precedence rules for operators.
- 'Power' and various mathematical functions are realized through library calls.

Math library calls

Math function in `cmath`:

```
#include <cmath>
.....
x = pow(3,.5);
```

For squaring, usually better to write `x*x` than `pow(x,2)`.

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: `%`

4.5.1 Truth values

In addition to numerical types, there are truth values, `true` and `false`, with all the usual logical operators defined on them.

Logical expressions in C++ are evaluated using *shortcut operators*: you can write

```
x>=0 && sqrt(x)<2
```

If `x` is negative, the second part will never be evaluated because the 'and' conjunction can already be concluded to be false. Similarly, 'or' conjunctions will only be evaluated until the first true clause.

Boolean expressions

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

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

The 'true' and 'false' constants could strictly speaking be stored in a single bit. C++ does not do that, but there are bit operators that you can apply to, for instance, all the bits in an integer.

Bit operators

Bitwise: `& | ^`

4.5.2 Type conversions

Since a variable has one type, and will always be of that type, you may wonder what happens with

```
float x = 1.5;
int i;
i = x;
```

or

```
int i = 6;
float x;
x = i;
```

- Assigning a floating point value to an integer truncates the latter.
- Assigning an integer to a floating point variable fills it up with zeros after the decimal point.

Exercise 4.8.

- What happens when you assign a positive floating point value to an integer variable?
- What happens when you assign a negative floating point value to an integer variable?
- What happens when you assign a `float` to a `double`? Try various numbers for the original float. Can you explain the result? (Hint: think about the conversion between binary and decimal.)

The rules for type conversion in expressions are not entirely logical. Consider

```
float x; int i=5,j=2;
x = i/j;
```

This will give `2` and not `2.5`, because `i/j` is an integer expression and is therefore completely evaluated as such, giving `2` after truncation. The fact that it is ultimately assigned to a floating point variable does not cause it to be evaluated as a computation on floats.

You can force the expression to be computed in floating point numbers by writing

```
x = (1.*i)/j;
```

or any other mechanism that forces a conversion, without changing the result. Another mechanism is the *cast*; this will be discussed in section 23.2.

Exercise 4.9. Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

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

Check your program for the freezing and boiling point of water.
(Do you know the temperature where Celsius and Fahrenheit are the same?)
Can you use Unix pipes to make one accept the output of the other?

Exercise 4.10. 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:`%`.

Complex numbers exist, see section 22.2.

4.6 Library functions

Some functions, such as *abs* can be included through `cmath`:

```
#include <cmath>
using std::abs;
```

Others, such as *max*, are in the less common `algorithm`:

```
#include <algorithm>
using std::max;
```

4.7 Review questions

Exercise 4.11. What is the output of:

```
int m=32, n=17;
cout << n%m << endl;
```

Exercise 4.12. Given

```
int n;
```

write code that uses elementary mathematical operators to compute n-cubed: n^3. Do you get the correct result for all n? Explain.

Chapter 5

Conditionals

A program consisting of just a list of assignment and expressions would not be terribly versatile. At least you want to be able to say 'if $x > 0$, do one computation, otherwise compute something else', or 'until some test is true, iterate the following computations'. The mechanism for testing and choosing an action accordingly is called a *conditional*.

5.1 Conditionals

Here are some forms a conditional can take.

Single statement

```
if (x<0)
  x = -x;
```

Single statement and alternative:

```
if (x>=0)
  x = 1;
else
  x = -1;
```

Multiple statements:

```
if (x<0) {
  x = 2*x; y = y/2;
} else {
  x = 3*x; y = y/3;
}
```

Chaining conditionals (where the dots stand for omitted code):

```
if (x>0) {
  ....
} else if (x<0) {
  ....
} else {
```

```
        ....
    }
```

Nested conditionals:

```
    if (x>0) {
      if (y>0) {
        ....
      } else {
        ....
      }
    } else {
      ....
    }
```

- In that last example the outer curly brackets are optional. But it's safer to use them anyway.
- When you start nesting constructs, use indentation to make it clear which line is at which level. A good editor helps you with that.

Exercise 5.1. 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.

5.2 Switch statement

If you have a number of cases corresponding to specific integer values, there is the `switch` statement.

Switch statement example

Code:

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

Output:

```
echo "2" | ./switch
very small
```

5.3 Scopes

The true and false branch of a conditional can consist of a single statement, or of a block in curly brackets. Such a block creates a *scope* where you can define local variables.

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

5.4 Operators

Comparison and logical operators

Operator	meaning	example
`==`	equals	`x==y-1`
`!=`	not equals	`x*x*!=5`
`>`	greater	`y>x-1`
`>=`	greater or equal	`sqrt(y)>=7`
`<,<=`	less, less equal	
`&&,||`	and, or	`x<1 && x>0`
`!`	not	`!( x>1 && x<2 )`

Chapter 6

Looping

6.1 Basic 'for' statement

There are many cases where you want to repeat an operation or series of operations:

- A time-dependent numerical simulation executes a fixed number of steps, or until some stoping test.
- Recurrences:

$$x_{i+1} = f(x_i).$$

- Inspect or transformation every element of a database table.

(Fine point: the first two cases actually need to be performed in sequences, while the last one corresponds more to a mathematical 'forall' quantor.)

What you need is known as a *loop*: a number of statements that get repeated. The two types of loop statement in C++ are:

- the *for loop* which is typically associated with a pre-determined number of repetitions, and with the repeated statements being based on a counter of some sort; and
- the *while loop*, where the statements are repeated indefinitely until some condition is satisfied.

We will now consider the for loop; the while loop comes in section 6.2.

In the most common case, a for loop has a *loop counter*, ranging from some lower to some upper bound. And example showing the syntax for this simple case is:

```
for (int var=low; var<upper; var++) {
  // statements involving the loop variable:
  cout << "The square of " << var << " is " << var*var << endl;
}
```

The `for` line is called the *loop header*, and the statements between curly brackets the *loop body*.

Exercise 6.1. Read an integer value, and print 'Hello world' that many times.

The loop header has three components, all of which are optional.

- An initialization. This is usually a declaration and initialization of an integer *loop variable*. Using floating point values is less advisable.
- A stopping test, usually involving the loop variable. If you leave this out, you need a different mechanism for stopping the loop; see 6.2.

- An increment, often `i++` or spelled out `i=i+1`. You can let the loop count down by using `i--`.

Some variations on the simple case.

- The loop variable can be defined outside the loop:

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

  but it's cleaner to make it local. However:

```
int var;
..... code that sets var .....
for ( ; var<upper; var++) {
  ...
}
```

- The stopping test can be omitted

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

  if the loops ends in some other way. You'll see this later.
- The stopping test doesn't need to be an upper bound:

```
for (int var=high-1; var>=low; var--) { ... }
```

- Here are a couple of popular increments:
  - `i++` for a loop that counts forward;
  - `i--` for a loop that counts backward;
  - `i+=2` to cover only odd or even numbers, depending on where you started;
  - `i*=10` to cover powers of ten.

Quite often, the loop body will contain another loop. For instance, you may want to iterate over all elements in a matrix. Both loops will have their own unique loop variable.

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

Exercise 6.2. Write an i, j loop that prints out all pairs with

$$1 \leq i, j \leq 10, \quad j \leq i.$$

Output one line for each `i` value.
Same, but

$$1 \leq i, j \leq 10, \quad |i - j| < 2.$$

6.2 Looping until

The basic for loop looks pretty deterministic: a loop variable ranges through a more-or-less prescribes set of values. This is appopriate for looping over the elements of an array, but not if you are coding some process that needs to run until some dynamically determined condition is satisfied. In this section you will see some ways of coding such cases.

First of all, the stopping test in the 'for' loop is optional:

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

So how do you end such a loop? For that you use the `break` statement. If the execution encounters this statement, it will continue with the first statement after the loop.

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

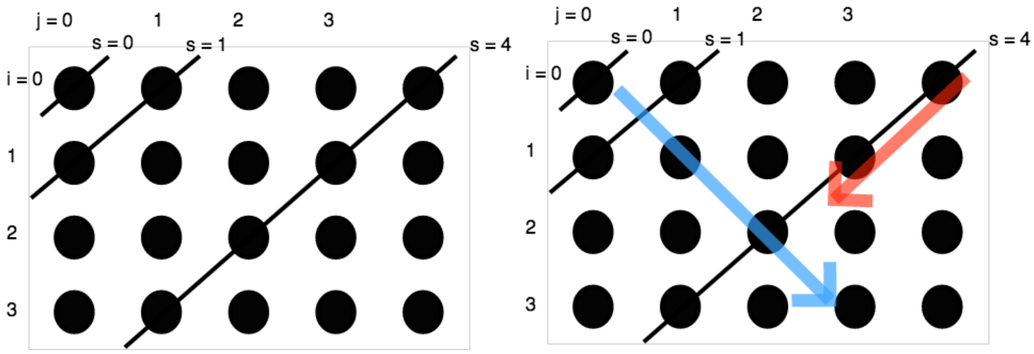


Figure 6.1: Traversal of i, j range by $i + j = c$

For the following exercise, see figure **??**.

Exercise 6.3. Write a double loop over $0 \leq i, j < 10$ that prints the first pair where the product of indices satisfies $i \cdot j > N$, where N is a number your read in. A good test case is $N = 40$.
Can you traverse the i, j indices such that they first enumerate all pairs $i + j = 1$, then $i + j = 2$, then $i + j = 3$ et cetera? Again, you should report the first pair i, j for which $i \cdot j > N$.
Write a program that prints out both pairs, each on a separate line, with the numbers separated with a comma, for instance `8,5`.

Another mechanism to alter the control flow in a loop is the `continue` statement. If this is encountered, execution skips to the start of the next iteration.

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;
}
```

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.

The two while loop variants can be described as ‘pre-test’ and ‘post-test’. The choice between them entirely depends on context. Here is an example in which the second syntax is more appropriate.

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.

While syntax 2

```
do {
  cout << "Enter a positive number: " ;
  cin >> invar;
```

```
} while (invar>0);
cout << "Sorry, " << invar << " is negative" << endl;
```

More elegant.

Exercise 6.4. 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 the first account be more than in the second?

6.3 Exercises

Exercise 6.5. 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 6.6. 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 at 1.

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \cdots$$

Try all starting values $u_1 = 1, \ldots, 1000$ to 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 6.7. 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 6.8. **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 $|x_- - x_+| < 10^{-10}$.

6.3.1 Further practice

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

Chapter 7

Functions

A *function* (or *subprogram*) is a way to abbreviate a block of code and replace it by a single line.

- Find a block of code that has a clearly identifiable function.
- Turn this into a function: the function definition will contain that block, plus a header and (maybe) a return statement.
- The function definition is placed before the main program.
- The function is called by its name.

Example

The code for an odd/even test

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

becomes

```
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);
}
```

Code becomes more readable: introduce application terminology.

Code reuse

Repeated code:

```
float s = 0;
for (int i=0; i<nx; i++)
  s += abs(x[i]);
cout << "Inf norm x: " << s << endl;
s = 0;
for (int i=0; i<ny; i++)
  s += abs(y[i]);
cout << "Inf norm y: " << s << endl;
```

becomes:

```
int InfNorm( float a[],int n) {
  float s = 0;
  for (int i=0; i<n; i++)
    s += abs(a[i]);
  return s;
}
int main() {
  ... // stuff
  cout << "Inf norm x: " << InfNorm(x,nx)
  cout << "Inf norm y: " << InfNorm(y,ny)
```

Code becomes shorter, easier to maintain.
(Don't worry about array stuff in this example)

7.1 Function definition and call

In the *function definition*:

- The keyword `void` indicates that the function does not give any results back to the main program.
- The name `report_evenness` is picked by you.
- The parenthetical `(int n)` is called the 'argument list' or 'parameter list': it says that the function takes an `int` as input. For purposes of the function, the int will have the name `n`, but this is not necessarily the same as the name in the main program.
- The 'body' of the function, the code that is going to be executed, is enclosed in curly brackets.

The *function call* consists of

- The name of the function, and
- In between parentheses, the value of the input argument.

In the previous example, the function had an input, and performed some screen output. To have a function that takes part in the computation of your program, you would write something like:

```
int my_computation(int i) {
   return i+3;
}
...
// in the main:
int result;
result = my_computation(5);
```

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

Program with function

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

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

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

Functions can be motivated as making your code more structured and intelligible. The source where you use the function call becomes shorter, and the function name makes the code more descriptive. This is sometimes called ‘self-documenting code’.

Sometimes introducing a function can be motivated from a point of *code reuse*: if the same block of code appears in two places in your source, your replace this by one function definition, and two (single line) function calls. The two occurences of the function code do not have to be identical:

Code reuse

```
double x,y, v,w;
y = ...... computation from x .....
w = ...... same computation, but from v .....
```

can be replaced by

```
double computation(double in) {
  return .... computation from ‘in’ ....
}

y = computation(x);
w = computation(v);
```

A final argument for using functions is code maintainability:

- 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.
- Localization: any variables that only serve the calculation in the function now have a limited *scope*.

```
void print_mod(int n,int d) {
  int m = n%d;
  cout << "The modulus of " << n << " and " << d
       << " is " << m << endl;
```

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. The function body is a *scope*: it can have local variables. (You can not nest function definitions.)
- a *return* statement. Which doesn't have to be the last statement, by the way.

The function can then be used in the main program, or in another function:

Function call

The function call

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

Functions without input, without return result

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

Functions with return result

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

The `atan` is a *standard function*

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.

Exercise 7.1. 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

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

Newton's method does this by evaluating

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

until the guess is accurate enough.

- Write functions `f(x,y)` and `deriv(x,y)`, that compute $f_y(x)$ and $f'_y(x)$.
- Read a value y and iterate until $|\mathtt{f}(\mathtt{x},\mathtt{y})| < 10^{-5}$. Print x.
- Second part: write a function `newton_root` that computes $\sqrt{y}$.

7.2 Parameter passing

C++ functions resemble mathematical functions: you have seen that a function can have an input and an output. In fact, they can have multiple inputs. The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*:

- A function has one result, which is returned through a return statement. The function call then looks like

  ```
  y = f(x1,x2,x3);
  ```

- The definition of the C++ parameter passing mechanism says that input arguments are copied to the function, meaning that they don't change in the calling program:

Code:

```
double f( double x ) {
  x = x*x;
  return x;
}
  /* ... */
  number = 5.1;
  cout << "Input starts as: "
       << number << endl;
  other = f(number);
  cout << "Input var is now: "
       << number << endl;
  cout << "Output var is: "
       << other << endl;
```

Output:

```
Input starts as: 5.1
Input var is now: 5.1
Output var is: 26.01
```

We say that the input argument is *passed by value*. In this example, the function parameter `x` acts as a local variable in the function, and it is initialized with a copy fo the value of `number` in the main program.

Having only one output is a limitation on functions. Therefore there is a mechanism for altering the input parameters and returning (possibly multiple) results that way. To do this, you use an ampersand for the parameter in the function definition:

Parameter passing by reference

```
void f(int &i) {
  i = /* some expression */ ;
};
int main() {
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

Using the ampersand, the parameter is *passed by reference*: instead of copying the value, the function receives a *reference*: information where the variable is stored in memory.

We also the following terminology for function parameters:

- *input* parameters: passed by value, so that it only functions as input to the function, and no result is output through this parameter;
- *output* parameters: passed by reference so that they return an 'output' value to the program.
- *throughput* parameters: these are passed by reference, and they have an initial value when the function is called. In C++, unlike Fortran, there is no real separate syntax for these.

Pass by reference example 1

```
void f( int &i ) {
  i = 5;
}
int main() {
  int var = 0;
  f(var);
  cout << var << endl;
```

Compare the difference with leaving out the reference.

As an example, consider a function that tries to read a value from a file. With anything file-related, you always have to worry about the case of the file not existing and such. So our function return:

- a boolean value to indicate whether the read succeeded, and
- the actual value if the read succeeded.

The following is a common idiom, where the success value is returned through the `return` statement, and the value through a parameter.

Pass by reference example 2

```
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;
}

int main() {
  int n;
  if (!can_read_value(n))
    // if you can't read the value, set a default
    n = 10;
```

Exercise 7.2. Write a function `swapij` of two parameters that exchanges the input values:

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

Exercise 7.3. 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;
```

7.3 Recursive functions

In mathematics, sequences are often recursively defined. For instance, the sequence of factorials $n \mapsto f_n \equiv n!$ can be defined as

$$f_0 = 1, \qquad \forall_{n>0}\colon f_n = n \times f_{n-1}.$$

Instead of using a subscript, we write an argument in parentheses

$$F(n) = n \times F(n-1) \qquad \text{if } n > 0\text{, otherwise } 1$$

and now it looks like a C++ function:

```
int factorial(int n)
```

is the function header of a factorial function. So what would the function body be? We need a `return` statement, and what we return should be $n \times F(n-1)$:

```
int factorial(int n) {
  return n*factorial(n-1);
} // almost correct, but not quite
```

So what happens if you write

```
int f3; f3 = factorial(3);
```

Well,

- The expression `factorial(3)` calls the `factorial` function, substituting `3` for the argument `n`.
- The return statement returns `n*factorial(n-1)`, in this case `3*factorial(2)`.
- But what is `factorial(2)`? Evaluating that expression means that the `factorial` function is called again, but now with `n` equal to 2.
- Evaluating `factorial(2)` returns `2*factorial(1)`,...
- ... which returns `1*factorial(0)`,...
- ... which return ...
- Uh oh. We forgot to include the case where `n` is zero. Let's fix that:

```
int factorial(int n) {
  if (n==0)
    return 1;
  else
    return n*factorial(n-1);
}
```

- Now `factorial(0)` is 1, so `factorial(1)` is `1*factorial(0)`, which is 1,...
- ... so `factorial(2)` is 2, and `factorial(3)` is 6.

Exercise 7.4. 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 7.5. 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 43.3.1.

7.4 More function topics

7.4.1 Default arguments

Default arguments

Functions can have *default argument*(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

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

7.4.2 Polymorphic functions

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.

7.5 Review questions

Exercise 7.6. Suppose a function

```
bool f(int);
```

is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which `f` is true.

Exercise 7.7. Suppose a function

```
bool f(int);
```

is given, which is true for some negative input value. Write a main program that finds the (negative) input with smallest absolute value for which `f` is true.

Chapter 8

Scope

8.1 Scope rules

The concept of *scope* answers the question 'when is the binding between a name (read: variable) and the internal entity valid'.

8.1.1 Lexical scope

C++, like Fortran and most other modern languages, uses *lexical scope* rule. This means that you can textually determine what a variable name refers to.

```
int main() {
  int i;
  if ( something ) {
    int j;
    // code with i and j
  }
  int k;
  // code with i and k
}
```

- The lexical scope of the variables `i,k` is the main program including any blocks in it, such as the conditional, from the point of definition onward. You can think that the variable in memory is created when the program execution reaches that statement, and after that it can be refered to by that name.
- The lexical scope of `j` is limited to the true branch of the conditional. The integer quantity is only created if the true branch is executed, and you can refer to it during that execution. After execution leaves the conditional, the name ceases to exist, and so does the integer in memory.

8.1.2 Shadowing

Scope can be limited by an occurrence of a variable by the same name:

```
int main() {
  int i = 3;
  if ( something ) {
    int i = 5;
```

```
    }
    cout << i << endl; // gives 3
    if ( something ) {
      float i = 1.2;
    }
    cout << i << endl; // again 3
  }
```

The first variable `i` has lexical scope of the whole program, minus the two conditionals. While its *lifetime* is the whole program, it is unreachable in places because it is *shadowed* by the variables `i` in the conditionals.

8.1.3 Lifetime versus reachability

The use of functions introduces a complication in the lexical scope story: a variable can be present in memory, but may not be textually accessible:

```
void f() {
  ...
}
int main() {
  int i;
  f();
  cout << i;
}
```

During the execution of `f`, the variable `i` is present in memory, and it is unaltered after the execution of `f`, but it is not accessible.

A special case of this is recursion:

```
void f(int i) {
  int j = i;
  if (i<100)
    f(i+1);
}
```

Now each incarnation of `f` has a local variable `i`; during a recursive call the outer `i` is still alive, but it is inaccessible.

8.1.4 Scope subtleties

8.1.4.1 Mutual recursion

If you have two functions `f,g` that call each other, you need *forward declaration*. There is also forward declaration of *classes*.

8.1.4.2 Closures

We don't have lambdas or *closuresclosure* yet in this book.

8.2 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 8.1. 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?

8.3 Review questions

Exercise 8.2. Is this a valid program?

```
void f() { i = 1; }
int main() {
  int i=2;
  f();
  return 0;
}
```

If yes, what does it do; if no, why not?

Exercise 8.3. What is the output of:

```
#include <iostream>
using namespace std;
int main() {
```

```
  int i=5;
  if (true) { i = 6; }
  cout << i << endl;
  return 0;
}
```

Exercise 8.4. What is the output of:

```
#include <iostream>
using namespace std;
int main() {
  int i=5;
  if (true) { int i = 6; }
  cout << i << endl;
  return 0;
}
```

Exercise 8.5. What is the output of:

```
#include <iostream>
using namespace std;
int main() {
  int i=2;
  i += /* 5;
  i += */ 6;
  cout << i << endl;
  return 0;
}
```

Chapter 9

Structures

9.1 Why structures?

You have seen the basic datatypes in section 4.3.3. These are enough to program whatever you want, but it would be nice if the language had some datatypes that are more abstract, closer to the terms in which you think about your application. For instance, if you are programming something to do with geometry, you had rather talk about points than explicitly having to manipulate their coordinates.

Structures are a first way to define your own datatypes. A `struct` acts like a datatype for which you choose the name. A `struct` contains other datatypes; these can be elementary, or other structs.

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

The elements of a structure are also called *members*. You can give them an initial value:

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

9.2 The basics of structures

How to use structures

1. Declare what is in your structure;
2. Make some structures;
3. Use them.

```
// declaration of the struct
struct AStructName { int num; double val; }
int main() {
  // declaration of struct variables
  AStructName mystruct1,mystruct2;
  .... code that uses your structures ....
}
```

Struct initialization

```
struct vector_a { double x; double y; } ;
// needs compiler option: -std=c++11
struct vector_b { double x=0; double y=0; } ;

int main() {

  // initialization when you create the variable:
  struct vector_a x_a = {1.5,2.6};
  // initialization done in the structure definition:
  struct vector_b x_b;
  // ILLEGAL:
  // x_b = {3.7, 4.8};
  x_b.x = 3.7; x_b.y = 4.8;
  //end snippet

  return 0;
}
```

Using structures

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

Code:

```
int main() {

  struct vector p1,p2;

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

  p2 = p1;
  cout << "p2: " << p2.x << "," << p2.y << endl;
```

Output:

```
./point
p2: 1,2
```

Period syntax: ‘apostrophe-s’.

Note: if you use initializations in the `struct` definition, you can not use the brace-assignment.

Functions on structures

You can pass a structure to a function:

Code:

```
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 );
}
  /* ... */
  struct vector p1 = { 1.,1. };
  cout << "Displacement x,y?" << endl;
  double dx,dy; cin >> dx >> dy;
  cout << "dx=" << dx << ", dy=" << dy << endl;
  struct vector p2 = { p1.x+dx,p1.y+dy };
  cout << "Distance: " << distance(p1,p2) << endl;
```

Output:

```
( echo 5 ; echo 12 ) | ./pointfun
Displacement x,y?
dx=5, dy=12
Distance: 13
```

Returning structures

You can return a structure from a function:

Code:

```
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;
};
  /* ... */
  p3 = vector_add(p1,p2);
  cout << "Added: " <<
    p3.x << "," << p3.y << endl;
```

Output:

```
./pointadd
Added: 5,6
```

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

Exercise 9.1. Write a function `inner_product` that takes two `vector` structures and computes the inner product.

Exercise 9.2. 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.
Can you make a matrix structure that is based on the vector structure, for instance using it to store the matrix columns?

Passing structures by reference

Prevent copying cost by passing by reference, use `const` to prevent changes:

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

Chapter 10

Classes and objects

10.1 What is an object?

You learned about `structs` (chapter 9) as a way of abstracting from the elementary data types. The elements of a structure were called its members.

You also saw that it is possible to write functions that work on structures. Since these functions are really tied to the definition of the `struct`, shouldn't there be a way to make that tie explicitly?

That's what an object is:

- An object is like a structure in that it has data members.
- An object has *methods* which are the functions that operate on that object.

C++ does not actually have a 'object' keyword; instead you define a class with the `class` which describes all the objects of that class.

First of all, you can make an object look pretty much like a structure:

Code:

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

int main() {
  Vector p1;
  p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.
  cout << "sum of components: " << p1.x+p1.y  << endl;
```

Output:

```
./pointstruct
sum of components: 3
```

- There are data members. We will get to the `public` in a minute.
- You make an object of that class by using the class name as the datatype.
- The data members can be accessed with the period.

10.1.1 Constructor

Next we'll look at a syntax for creating class objects that is new. If you create an object, you actually call a function that has the same name as the class: the *constructor*. By default there is a constructor which has no arguments, and does nothing. A constructor can for instance be used to initialize data members:

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double userx,double usery ) {
    vx = userx; vy = usery;
  }

Vector p1(1.,2.);
```

10.1.2 Interface versus implementation

In the example above, the data members of the `Vector` class were declared `public`, meaning that they are accessable from the calling (main) program. While this is initially convenient for coding, it is a bad idea in the long term. For a variety of reasons it is good practice to separate interface and implementation of a class.

Example of accessor functions

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double userx,double usery ) {
    vx = userx; vy = usery;
  }

double x() { return vx; };
double y() { return vy; };
```

Interface versus implementation

- Implementation: data members, keep `private`,
- Interface: `public` functions to get/set data.
- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

Private access gone wrong

```
class SumIsOne {
public:
  float x,y;
  SumIsOne( double xx ) { x = xx; y = 1-x; };
}
int main() {
  SumIsOne pointfive(.5);
  pointfive.y = .6;
```

```
}
```

10.1.3 Initialization

Member default values

Class members can have default values, just like ordinary variables:

```
class Point {
private:
  float x=3., y=.14;
private:
  // et cetera
}
```

Each object will have its members initialized to these values.

Member initialization

Other syntax for initialization:

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

If the data members follow a `public` directive, code outside the class can access the data members, both for getting and setting their values. This may be convenient for coding, but it's not a clean coding style. It's better to make data members `private`, and use *accessor* functions to get and set values.

Private data

```
class Vector {
private: // recommended!
  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.);
```

10.1.4 Accessor functions

Remember the lines:

```
private:
  double vx,vy;
```

This implies that vx,vy are not accessible from anything outside the object. In order to change them we need functions, which we call functions:

Accessor for setting private data

Class methods:

```
void setx( double newx ) { vx = newx; };
void sety( double newy ) { vy = newy; };
p1.setx(3.12);
/* ILLEGAL: p1.x() = 5; */
cout << "P1's x=" << p1.x() << endl;
```

With the accessors, you have just seen a first example of a class *method*: a function that is only defined for objects of that class, and that have access to the private data of that object. Let's now look at more meaningful methods. For instance, for the Vector class you can define functions such as length and angle.

Code:

```
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;
```

Output:

```
./pointfunc
p1 has length 2.23607
```

By making these functions public, and the data members private,, you define an Application Programmer Interface (API) for the class:

- You are defining operations for that class; they are the only way to access the data of the object.
- The methods can use the data of the object, or alter it.
- The actual data of the object is not accessible from outside the object.

So far you have seen methods that use the data members of an object to return some quantity. It is also possible to alter the members. For instance, you may want to scale a vector by some amount:

Code:

```
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;
```

Output:

```
./pointscaleby
p1 has length 2.23607
p1 has length 4.47214
```

The methods you have seen so far only returned elementary datatypes. It is also possible to return an object, even from the same class. For instance, instead of scaling the members of a vector object, you could create a new object based on the scaled members:

Code:

```
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;
```

Output:

```
./pointscale
p1 has length 2.23607
p2 has length 4.47214
```

10.1.5 Default constructor

One of the more powerful ideas in C++ is that there can be more than one constructor. You will often be faced with this whether you want or not. The following code looks plausible:

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

but it will give an error message during compilation. The reason is that

```
Vector p;
```

calls the default constructor. Now that you have defined your own constructor, the default constructor no longer exists. So you need to define it explicitly:

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

10.1.6 Accessors

It is a good idea to keep data members private, and use accessor functions.

Use accessor functions!

```
class PositiveNumber { /* ... */ }
class Point {
private:
  // data members
public:
  Point( float x,float y ) { /* ... */ };
  Point( PositiveNumber r,float theta ) { /* ... */ };
  float get_x() { /* ... */ };
  float get_y() { /* ... */ };
  float get_r() { /* ... */ };
  float get_theta() { /* ... */ };
};
```

Functionality is independent of implementation.

The remainder of this section is advanced material. Make sure you have studied section 14.3.

It is a good idea to make the data in an object private, to control outside access to it.

- Sometimes this private data is auxiliary, and there is no reason for outside access.
- Sometimes you do want outside access, but you want to precisely control by whom.

Accessor functions:

```
class thing {
private:
 float x;
public:
 float get_x() { return x; };
 void set_x(float v) { x = v; }
};
```

This has advantages:

- You can print out any time you get/set the value; great for debugging
- You can catch specific values: if x is always supposed to be positive, print an error (throw an exception) if nonpositive.

Better accessor:

```
class thing {
private:
 float x;
public:
 float &the_x() { return x; };
};
int main () {
  thing t;
  t.the_x() = 5;
  cout << t.the_x();
}
```

The function `the_x` returns a reference to the internal variable `x`.

Setting members through accessor

Code:

```
class SomeObject {
private:
  float x=0.;
public:
  SomeObject( float v ) : x(v) {};
  float &xvalue() { return x; };
};

int main() {
  SomeObject myobject(1.);
  cout << "Object member initially :"
       << myobject.xvalue() << endl;
  myobject.xvalue() = 3.;
  cout << "Object member updated   :"
       << myobject.xvalue() << endl;
```

Output:

```
Object member initially :1
Object member updated   :3
```

If the internal variable is something indexable:

```
class thing {
private:
  vector<float> x;
public:
  operator[](int i) { return x[i]; };
};
```

You define the subscript operator `[]` for the object, in terms of indexing of the private vector.

10.1.7 Methods

Methods can be

- private, because they are only used internally;

- public, because they should be usable from outside a class object, for instance in the main program;
- protected, because they should be usable in derived classes (see section 10.3.1).

You can have multiple methods with the same name, as long as they can be distinguished by their argument types. This is known as *overloading*.

10.1.8 Operator overloading

Instead of writing

```
myobject.plus(anotherobject)
```

you can actually redefine the + operator so that

```
myobject + anotherobject
```

is legal.

The syntax for this is

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

```
class Point {
private:
  float x,y;
public:
  Point operator*(float factor) {
    return Point(factor*x,factor*y);
  };
};
```

See section 14.4 for redefining the parentheses and square brackets.

10.1.9 Copy constructor

Just like the default constructor which is defined if you don't define an explicit constructor, there is an implicitly defined *copy constructor*. This constructor is invoked whenever you do an obvious copy:

```
my_object x,y; // regular or default constructor
x = y;         // copy constructor
```

Usually the copy constructor that is implicitly defined does the right thing: it copies all data members. If you want to define your own copy constructor, you need to know its prototype. There are a couple of possibilities; see for instance:

```
class has_int {
private:
  int mine{1};
```

```
public:
  has_int(int v) { mine = v; };
  has_int(has_int &other) {
    cout << "(calling copy constructor)" << endl;
    mine = other.mine; };
};
```

10.1.10 Destructor

Just there is a constructor routine to create an object, there is a *destructor* to destroy the object. As with the case of the default constructor, there is a default destructor, which you can replace with your own.

A destructor can be useful if your object contains dynamically created data: you want to use the destructor to dispose of that dynamic data to prevent a *memory leak*.

The destructor is typically called without you noticing it. For instance, any statically created object is destroyed when the control flow leaves its scope.

Example:

Code:

```
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;
```

Output:

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

Exercise 10.1. Write a class

```
class HasInt {
private:
  int mydata;
public:
  HasInt(int v) { /* initialize */ };
  ...
}
```

used as

```
{ HasInt v(5);
  v.set(6)
}
```

which gives output

```
**** creating object with 5 ****
**** setting object to 6 ****
**** object destroyed after 1 update ****
```

10.2 Inclusion relations between classes

The data members of an object can be of elementary datatypes, or they can be objects. For instance, if you write software to manage courses, each `Course` object will likely have a `Person` object, corresponding to the teacher.

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

Designing objects with relations between them is a great mechanism for writing structured code, as it makes the objects in code behave like objects in the real world. The relation where one object contains another, is called a *has-a relation* between classes.

At this time, do exercises in section 40.3.

Sometimes a class can behave as if it includes an object of another class, while not actually doing so. For instance, a line segment can be defined from two points

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

or from one point, and a distance and angle:

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

In both cases the code using the object is written as if the segment object contains two points. This illustrates how object-oriented programming can decouple the API of classes from their actual implementation.

Related to this decoupling, a class can also have two very different constructors.

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

Depending on how you actually implement the class, the one constructor will simply store the defining data, and the other will do some conversion from the given data to the actually stored data.

This is another strength of object-oriented programming: you can change your mind about the implementation of a class without having to change the program that uses the class.

At this time, do the exercises in section 40.3

10.3 Inheritance

In addition to the has-a relation, there is the *is-a relation*, also called *inheritance*. Here one class is a special case of another class. Typically the object of the *derived class* (the special case) then also inherits the data and methods of the *base class* (the general case).

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

How do you define a derived class?

- You need to indicate what the base class is:
  ```
  class Special : public General { .... }
  ```
- The base class needs to declare its data members as `protected`: this is similar to private, except that they are visible to derived classes
- The methods of the derived class can then refer to data of the base class;
- Any method defined for the base class is available as a method for a derived class object.

The derived class has its own constructor, with the same name as the class name, but when it is invoked, it also calls the constructor of the base class. This can be the default constructor, but often you want to call the base constructor explicitly, with parameters that are describing how the special case relates to the base case. Example:

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

10.3.1 Methods of base and derived classes

Overriding methods

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class *override* a base class method:

```
class Base {
public:
  virtual f() { ... };
};
class Deriv : public Base {
public:
  virtual f() override { ... };
};
```

10.3.2 Virtual methods

Base vs derived methods

- Method defined in base class: can be used in any derived class.
- Method define in derived class: can only be used in that particular derived class.
- Method defined both in base and derived class, marked `override`: derived class method replaces (or extends) base class method.
- Virtual method: base class only declares that a routine has to exist, but does not give base implementation.
  A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual.
  You can not make abstract objects.

Abstract classes

Special syntax for *abstract method*:

```
class Base {
public:
  virtual void f() = 0;
};
class Deriv {
public:
  virtual void f() { ... };
};
```

Example: using virtual class

```
class VirtualVector {
private:
public:
  virtual void setlinear(float) = 0;
  virtual float operator[](int) = 0;
};
```

Suppose `DenseVector` derives from `VirtualVector`:

```
DenseVector v(5);
v.setlinear(7.2);
cout << v[3] << endl;
```

Implementation

```
class DenseVector : VirtualVector {
private:
  vector<float> values;
public:
  DenseVector( int size ) {
    values = vector<float>(size,0);
  };
  void setlinear( float v ) {
    for (int i=0; i<values.size(); i++)
      values[i] = i*v;
  };
  float operator[](int i) {
    return values.at(i);
  };
};
```

10.3.3 Advanced topics in inheritance

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 11

Arrays

11.1 Static arrays

An *array* is an indexed data structure, that for each index stores an integer, floating point number, character, object,... In scientific applications, arrays often correspond to vectors and matrices, potentialy of quite large size. (If you know about Finite Element Methods (FEMs), you know that vectors can have sizes in the millions or beyond.)

To define an array you need to declare its size, and you need to give it its contents. Those actions don't necessarily have to occur together. (And the contents can later change, as with any other variable.) However, if you know the array size and contents already before you run your code, you can create the whole array in one statement. There is more than one syntax for doing so.

```
{
  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;
}
```

As you see in this example, if `a` is an array, and `i` and integer, then `a[i]` is the i'th element.

- An array element `a[i]` can be used to get the value of an array element, or it can occur in the left-hand side of an assignment to set the value.
- The *array index* (or *array subscript*) `i` starts numbering at zero.
- Therefore, if an array has n elements, its last element has index `n-1`.
- If you try to get an array elements outside the bounds of the array, your program may give a runtime error, but that does not necessarily happen. You could just get some random value.
- An array does not contain the information about its size: you have to store that in variable.

11.1.1 Ranging over an array

If you need to consider all the elements in an array, there are two cases. First:

Range over elements

You can write a *range-based for* loop, which considers the elements as a collection.

```
for ( float e : array )
  // statement about element with value e
for (auto e : array)
  // same, with type deduced by compiler
```

Code:

```
int numbers[] = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output:

```
Max: 6 (should be 6)
```

(You can spell out `for (float e : array)` but `auto` is quite convenient.)

Indexing the elements

You can write an *indexed for* loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
  // statement about index i
```

Code:

```
int numbers[] = {1,4,2,6,5};
int tmp_idx = 0;
int tmp_max = numbers[tmp_idx];
for (int i=0; i<5; i++) {
  int v = numbers[i];
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << endl;
```

Output:

```
Max: 6 at index: 3
```

Exercise 11.1. Find the element with maximum absolute value in an array. Use:

```
int numbers[] = {1,-4,2,-6,5};
```

Which mechanism do you use for traversing the array?
Hint:

```
#include <cmath>
..
absx = abs(x);
```

Exercise 11.2. Find the location of the first negative element in an array.
Which mechanism do you use?

Exercise 11.3. Check whether an array is sorted.

11.2 Vector class for arrays

Statically allocated arrays are enough for many purposes. However, as pointed out above, they have some disadvantages. In this section we will look at one way of dynamically creating arrays: through the STL `vector`.

This takes a new syntax:

Vector definition

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

vector<type> name(size);
vector<type> name(size,value);
```

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.
- `value` is the uniform initial value of all elements.

Vector elements

In a number of ways, vector behaves like an array:

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

Ranging over a vector

```
for ( auto e : my_vector)
  cout << e;
```

`e` is a copy of the array element.

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output:

```
3.3
```

Ranging over a vector by reference

To set array elements, make `e` a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output:

```
6.6
```

Vector initialization

You can initialize a vector with much the same syntax as an array:

```
vector<int> odd_array{1,3,5,7,9};
vector<int> even_array = {0,2,4,6,8};
```

(This syntax requires compilation with the `-std=c++11` option.)

Vector initialization'

There is a syntax for initializing a vector with a constant:

```
vector<float> x(25,3.15);
```

which gives a vector of size 25, with all elements initialized to 3.15.

Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v[2] = 3.5;
vcopy = v;
cout << vcopy[2] << endl;
```

Output:

```
./vectorcopy
3.5
```

11.2.1 Vector methods

A `vector` is an object. Let's take a look at some of the methods that are defined for it.

First of all, there is a way of accessing vector elements through the `at` method. It has the big advantage that it does *bounds checking*.

Vector indexing

Your choice: fast but unsafe, or slower but safe

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

Safer, but also slower; see below.

The method `size` gives the size of the vector:

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

The existence of this method means that you don't have to remember the size of a vector: it has that information internally.

> *Advanced note:* 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.

11.2.2 Vectors are dynamic

There is an important difference between vectors and arrays: a vector can be grown or shrunk after its creation. Use the `push_back` method to add elements at the end:

Dynamic extension

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

This is not a good way of creating arrays: if you know the size, it is better to reserve the vector at the correct size.

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

11.2.3 Vector assignment

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

Can you make a class around a vector?

Vector needs to be created with the object:

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

Create and assign

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.

11.2.4 Vectors and functions

11.2.4.1 Vector as function return

Vector as function return

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

Code:

```
vector<int> make_vector(int n) {
  vector<int> x(n);
  x[0] = n;
  return x;
}
  /* ... */
  vector<int> x1 = make_vector(10); // "auto" also possible!
  cout << "x1 size: " << x1.size() << endl;
  cout << "zero element check: " << x1[0] << endl;
```

Output:

```
./vectorreturn
x1 size: 10
zero element check: 10
```

11.2.4.2 Pass vector to function

Vector as function argument

You can pass a vector to a function:

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

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

Vector pass by value example

Code:

```
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;
```

Output:

```
./vectorpassnot
3.5
```

Vector pass by reference

If you want to alter the vector, you have to pass by reference:

Code:

```
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;
```

Output:

```
./vectorpassref
4.6
```

Exercise 11.4. Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_floats(length);
sort(values);
```

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

11.2.5 Dynamic size of vector

Writing

Dynamic size extending

```
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. Since this requires operating system actions, it will probably be inefficient.

11.2.6 Timing

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

Vector extension

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;
```

Vector extension

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:

Timing

```
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.

So do you use at for safety or [] for speed? Well, you could use at during development of the code, and insert

```
#define at(x) operator[](x)
```

for production.

11.3 Use of new

Before doing this section, make sure you study section 16.2.1.

There is a dynamic allocation mechanism that is much inspired by memory management in C. Don't use this as your first choice.

Use of new uses the equivalence of array and reference.

```
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:

```
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/`.

The `new` mechanism is a cleaner variant of `malloc`, which was the dynamic allocation mechanism in C. It is still available, but should not be used.

11.4 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);
};
```

11.5 Old-style arrays

Static arrays are really an abuse of the equivalence of arrays and addresses of the C programming language. This appears for instance in parameter passing mechanisms.

11.5.1 Arrays and subprograms

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

```
void array_set( double ar[],int idx,double val) {
  ar[idx] = val;
}
  array_set(array,1,3.5);
```

Exercise 11.5. 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.

11.6 Multi-dimensional cases

11.6.1 Matrix as vector of vectors

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

This is not contiguous.

11.6.2 Matrix class based on vector

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

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

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

11.6.3 A matrix class

Matrix class

```
class matrix {
private:
  int rows,cols;
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    rows = m; cols = n;
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
};
```

Matrix class'

Better idea:

```
elements = vector<double>(rows*cols);
...
void get(int i,int j) {
  return elements.at(i*cols+j);
}
```

Exercise 11.7. Add methods such as `transpose`, `scale` to your matrix class.
Implement matrix-matrix multiplication.

11.7 Multi-dimensional arrays

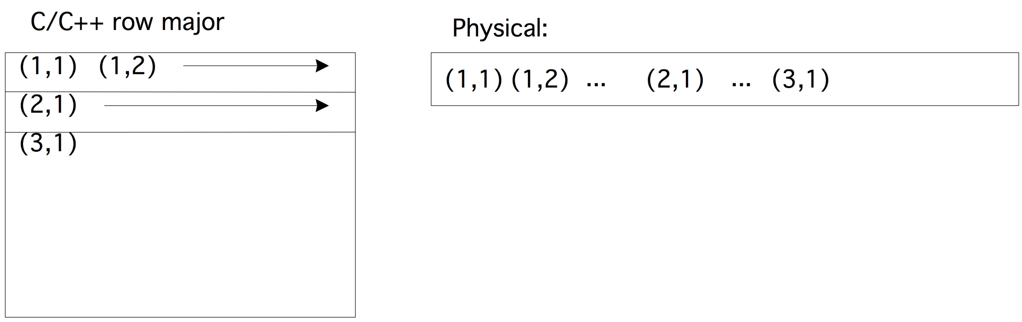
Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:

```
float matrix[15][25];

for (int i=0; i<15; i++)
  for (int j=0; j<25; j++)
    // something with matrix[i][j]
```

Passing a multi-dimensional array to a function, only the first dimension can be left unspecified:

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



11.7.1 Memory layout

Puzzling aspects of arrays, such as which dimensions need to be specified and which not in a function call, can be understood by considering how arrays are stored in memory. The question then is how a two-dimensional (or higher dimensional) array is mapped to memory, which is linear.

- A one-dimensional array is stored in contiguous memory.

- A two-dimensional array is also stored contiguously, with first the first row, then the second, et cetera.
- Higher dimensional arrays continue this notion, with contiguous blocks of the highest so many dimensions.

As a result of this, indexing beyond the end of a row, brings you to the start of the next row:

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

We can now also understand how arrays are passed to functions:

- The only information passed to a function is the address of the first element of the array;
- In order to be able to find location of the second row (and third, et cetera), the subprogram needs to know the length of each row.
- In the higher dimensional case, the subprogram needs to know the size of all dimensions except for the first one.

11.8 Dynamically allocated arrays

(To properly understand this section, you also need to read section 16.2.1.)

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);
  }
```

We will now look at several strategies for dynamically creating arrays.

11.9 Exercises

Exercise 11.8. 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.

Pascal's triangle

Pascal's triangle contains binomial coefficients:

```
Row    1:                         1
Row    2:                       1   1
Row    3:                     1   2   1
Row    4:                   1   3   3   1
Row    5:                 1   4   6   4   1
Row    6:               1   5  10  10   5   1
Row    7:             1   6  15  20  15   6   1
Row    8:           1   7  21  35  35  21   7   1
Row    9:         1   8  28  56  70  56  28   8   1
Row   10:       1   9  36  84 126 126  84  36   9   1
```

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

The coefficients can easily be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} \end{cases}$$

Exercise 11.9.

- Write a class `pascal` so that `pascal(n)` is the object containing n rows of the above coefficients. Write a method `get(i,j)` that returns the (i,j) coefficient.
- Write a method `print` that prints the above display.
- Write a method `print(int m)` that prints a star if the coefficient modulo m is nonzero, and a space otherwise.

```
        *
       * *
      *   *
     * * * *
    *       *
```

```
    * *         * *
   *   *       *   *
  * * * * * * * *
 *               *
* *               * *
```

- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
  Bonus: when you have that code working, optimize your code to use precisely enough space for the coefficients.

Exercise 11.10. 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 11.11. Put eight queens on a chessboard so that none threatens any other.

Exercise 11.12. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

Chapter 12

Strings

12.1 Characters

Characters and ints

- Type `char`;
- represents ‘7-bit ASCII’: printable and (some) unprintable characters.
- Single quotes: `char c = 'a'`
- Equivalent to (short) integer: `'x'-'a'` is distance `a--x`

Exercise 12.1. Write a program that accepts an integer $0 \cdots 26$ and prints the so-manieth letter of the alphabet.
Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

12.2 Basic string stuff

String declaration

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

// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

String creation

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable (use `-std=c++11`), or assign it dynamically:

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

Concatenation

Strings can be *concatenated*:

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

String is like vector

You can query the *size*:

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

or use subscripts:

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

More vector methods

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

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

Exercise 12.2. Write a function to print out the digits of a number: `156` should print `one five six`. Use a vector or array of strings, containing the names of the digits.
Start by writing a program that reads a single digit and prints its name.
For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

Exercise 12.3. Write a function to convert an integer to a string: the input `205` should give `two hundred fifteen`, et cetera.

Exercise 12.4. 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.

12.3 Conversion

`to_string`

Chapter 13

Input/output

13.1 Formatted output

Formatted output

- `cout` uses default formatting
- Possible: pad a number, use limited precision, format as hex/base2, etc
- Many of these output modifiers need

  ```
  #include <iomanip>
  ```

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

Code:

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

Output:

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

Reserve space

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

Code:

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

Output:

```
Set width:
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Padding character

Normally, padding is done with spaces, but you can specify other characters:

Code:

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

Output:

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

Note: single quotes denote characters, double quotes denote strings.

Left alignment

Instead of right alignment you can do left:

Code:

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

Output:

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

Number base

Finally, you can print in different number bases than 10:

Code:

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

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
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Exercise 13.1. 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
etc
```

Exercise 13.2. 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.

Hexadecimal

Hex output is useful for pointers (chapter 16):

```
    int i;
    cout << "address of i, decimal: "
 << (long)&i << endl;
    cout << "address if i, hex     : "
 << std::hex << &i << endl;
```

Back to decimal:

```
cout << hex << i << dec << j;
```

13.2 Floating point output

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;
}
```

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)

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;
}
```

Output

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

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;
}
```

Output

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

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;
}
```

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
```

13.3 Saving and restoring settings

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

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

13.4 File output

Text output to file

Streams are general: work the same for console out and file out.

```
#include <fstream>
```

Use:

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

Binary output

```
ofstream file_out;
file_out.open("fio_binary.out",ios::binary);
/* ... */
file_out.write( (char*)(&number),4);
```

13.4.1 Output your own classes

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

13.5 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>
using std::cin;
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;
  /* ... */
  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.

More info: http://www.cplusplus.com/forum/articles/6046/.

13.5.1 Input streams

Test, mostly for file streams: `is_eof` `is_open`

Chapter 14

References and addresses

14.1 Reference

This section contains further facts about parameter passing. Make sure you study section 7.2 first.

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

```
void change_scalar(int i) { i += 1; }
```

You can indicate that this is unintended:

```
/* 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:

```
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:

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

This is due to the fact that an array is really a pointer to memory; see 16.2.1. On the other hand, a `std::vector` is copied if you do not pass a reference.

The old-style way of doing things:

```
void change_scalar_old_style(int *i) { *i += 1; }
  number = 3;
  change_scalar_old_style(&number);
```

14.2 Pass by reference

Reference: change argument

```
void f( int &i ) { i += 1; };
int main() {
  int i = 2;
  f(i); // makes it 3
```

Reference: save on copying

```
class BigDude {
private:
   vector<double> array(5000000);
}
int main() {
   BigDude big;
   f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) { .... };
```

14.3 Reference to class members

Here is the naive way of returning a class member:

```
class Object {
private:
  SomeType thing;
public:
  SomeType get_thing() {
    return thing; };
};
```

The problem here is that the return statement makes a copy of `thing`, which can be expensive. Instead, it is better to return the member by *reference*:

```
  SomeType &get_thing() {
    return thing; };
```

The problem with this solution is that the calling program can now alter the private member. To prevent that, use a *const reference*:

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: " << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: " << an_int.int_to_get() << endl;
```

Output of this fragment:

```
Contained int is now: 4
Contained int is now: 17
```

See section 23.1.1.

14.4 Reference to array members

You can define various operator, such as `+-*/` arithmetic operators, to act on classes, with your own provided implementation; see section 10.1.8. You can also define the parentheses and square brackets operators, so make your object look like a function or an array respectively.

These mechanisms can also be used to provide safe acess to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
class vector10 {
private:
  int array[10];
public:
  /* ... */
  int operator()(int i) {
    return array[i];
  };
  int operator[](int i) {
    return array[i];
  };
};
```

```
    /* ... */
    vector10 v;
    cout << v(3) << endl;
    cout << v[2] << endl;
    /* compilation error: v(3) = -2; */
```

Note that `return array[i]` will return a copy of the array element, so it is not possible to write

```
myobject[5] = 6;
```

For this we need to return a reference to the array element:

```
    int& operator[](int i) {
      return array[i];
    };
};
    /* ... */
    cout << v[2] << endl;
    v[2] = -2;
    cout << v[2] << endl;
```

Another reason for wanting to return a reference is to prevent the *copy of the return result* that is induced by the `return` statement. In this case, you may not want to be able to alter the object contents, so you can return a *const reference*:

```
    const int& operator[](int i) {
      return array[i];
    };
};
    /* ... */
    cout << v[2] << endl;
    /* compilation error: v[2] = -2; */
```

14.5 rvalue references

See the chapter about obscure stuff; section 23.3.3.

Chapter 15

Memory

15.1 Memory and scope

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

Deallocating objects is slightly more complicated: a *destructor* is called.

```
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 16

Pointers

16.1 What is a pointer

The term pointer is used to denote a reference to a quantity. The reason that people like to use C++ and C as high performance languages is that pointers are actually memory addresses. So you're programming 'close to the bare metal' and are in fargoing control over what your program does.

16.2 Pointers and addresses, C style

You have learned about variables, and maybe you have a mental concept of variables as 'named memory locations'. That is not too far of: while you are in the (dynamic) scope of a variable, it corresponds to a fixed memory location.

Exercise 16.1. When does a variable not always correspond to the same location in memory?

There is a mechanism of finding the actual address of a variable: you prefix its name by an ampersand. This address is integer-valued, but its range is actually greater than of the `int` type.

Memory addresses

If you have an

```
int i;
```

then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation. C style:

Code:

```
int i;
printf("address of i: %ld\n",
(long)(&i));
printf(" same in hex: %lx\n",
(long)(&i));
```

Output:

```
address of i: 140732748506796
 same in hex: 7ffee57ba6ac
```

and C++:

Code:

```
    int i;
    cout << "address of i, decimal: "
 << (long)&i << endl;
    cout << "address if i, hex    : "
 << std::hex << &i << endl;
```

Output:

```
address of i, decimal: 140732735583916
address if i, hex    : 0x7ffee4b676ac
```

You could just print out the address of a variable, which is sometimes useful for debugging. If you want to store the address, you need to create a variable of the appropriate type. This is done by taking a type and affixing a star to it.

Address types

The type of ‘`&i`’ is `int*`, pronounced ‘int-star’,
or more formally: ‘pointer-to-int’.

You can create variables of this type:

```
int i;
int* addr = &i;
```

Now if you have have a pointer that refers to an int:

```
int i;
int *iaddr = &i;
```

you can use (for instance print) that pointer, which gives you the address of the variable. If you want the value of the variable that the pointer points to, you need to *dereference* it.

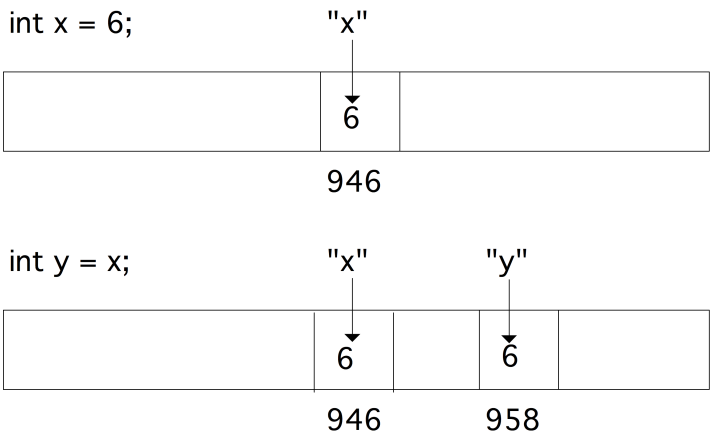
Dereferencing

Using `*addr` ‘dereferences’ the pointer: gives the thing it points to;
the value of what is in the memory location.

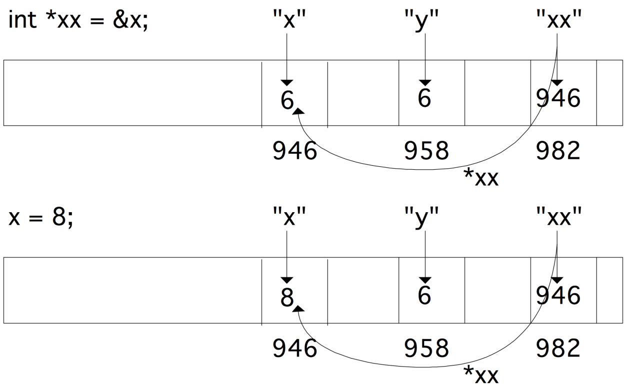
```
int i;
int* addr = &i;
i = 5;
cout << *addr;
i = 6;
cout << *addr;
```

This will print `5` and `6`:

illustration



illustration



- `addr` is the address of `i`.
- You set `i` to 5; nothing changes about `addr`. This has the effect of writing `5` in the memory location of `i`.
- The first `cout` line dereferences `addr`, that is, looks up what is in that memory location.
- Next you change `i` to `6`, that is, you write `6` in its memory location.
- The second `cout` looks in the same memory location as before, and now finds `6`.

The syntax for declaring a pointer-to-sometype allows for a small variation, which indicates the two way you can interpret such a declaration.

Star stuff

Equivalent:

- `int* addr`: `addr` is an int-star, or
- `int *addr`: `*addr` is an int.

The notion `int* addr` is equivalent to `int *addr`, and semantically they are also the same: you could say that `addr` is an int-star, or you could say that `*addr` is an int.

16.2.1 Arrays and pointers

In section 11.1 you saw the treatment of static arrays in C++. Examples such as:

```
void array_set( double ar[],int idx,double val) {
  ar[idx] = val;
}
  array_set(array,1,3.5);
```

show that, even though an parameters are normally passed by value, that is through copying, array parameters can be altered. The reason for this is that there is no actual array type, and what is passed is a pointer to the first element of the array. So arrays are still passed by value, just not the 'value of the array', but the value of its location.

So you could pass an array like this:

```
void array_set_star( double *ar,int idx,double val) {
  ar[idx] = val;
}
  array_set_star(array,2,4.2);
```

Array and pointer equivalence

Array and memory locations are largely the same:

```
double array[5];
double *addr_of_second = &(array[1]);
array = (11,22,33,44,55);
cout << *addr_of_second;
```

Dynamic allocation

`new` gives a something-star:

```
double *x;
x = new double[27];
```

(Actually, C uses `malloc`, but that looks similar.)

16.2.2 Pointer arithmetic

Pointer arithmetic

pointer arithmetic uses the size of the objects it points at:

```
double *addr_of_element = array;
cout << *addr_of_element;
addr_of_element = addr_of_element+1;
cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

Exercise 16.2. Write a subroutine that sets the i-th element of an array, but using pointer arithmetic: the routine should not contain any square brackets.

16.2.3 Multi-dimensional arrays

Multi-dimensional arrays

After

```
double x[10][20];
```

a row `x[3]` is a `double*`, so is `x` a `double**`?

Was it created as:

```
double **x = new double*[10];
for (int i=0; i<10; i++)
  x[i] = new double[20];
```

No: multi-d arrays are contiguous.

16.2.4 Parameter passing

C++ pass by reference

C++ style functions that alter their arguments:

```
void inc(int &i) { i += 1; }
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
  return 0;
}
```

C-style pass by reference

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
void inc(int *i) { *i += 1; }
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

Now the function gets an argument that is a memory address: `i` is an int-star. It then increases `*i`, which is an int variable, by one.

Exercise 16.3. Write another version of the `swap` function:

```
void swapij( /* something with i and j */ {
  /* your code */
}
```

```
int main() {
  int i=1,j=2;
  swapij( /* something with i and j */ );
  cout << "check that i is 2: " << i << endl;
  cout << "check that j is 1: " << i << endl;
  return 0;
}
```

Hint: write C++ code, then insert stars where needed.

16.2.5 Allocation

In section 11.1 you learned how to create arrays that are local to a scope:

Problem with static arrays

```
if ( something ) {
  double ar[25];
} else {
  double ar[26];
}
ar[0] = // there is no array!
```

The array `ar` is created depending on if the condition is true, but after the conditional it disappears again. The mechanism of using `new` (section 11.8) allows you to allocate storage that transcends its scope:

Declaration and allocation

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

Memory leak1

```
void func() {
  double *array = new double[large_number];
  // code that uses array
}
int main() {
  func();
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- ⇒ *memory leak*.

Memory leak2

```
for (int i=0; i<large_num; i++) {
  double *array = new double[1000];
  // code that uses array
}
```

Every iteration reserves memory, which is never released: another *memory leak*.

Your code will run out of memory!

De-allocation

Memory allocated with `new` does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
delete(array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

16.2.5.1 Malloc

The keywords `new` and `delete` are in the spirit of C style programming, but don't exist in C. Instead, you use `malloc`, which creates a memory area with a size expressed in bytes. Use the function `sizeof` to translate from types to bytes:

Allocation in C

```
int n;
double *array;
array = malloc( n*sizeof(double) );
if (!array)
  // allocation failed!
```

16.2.5.2 Allocation in a function

The mechanism of creating memory, and assigning it to a 'star' variable can be used to allocate data in a function and return it from the function.

Allocation in a function

```
void make_array( double **a, int n ) {
  *a = new double[n];
}
int main() {
  double *array;
  make_array(&array,17);
}
```

Note that this requires a 'double-star' or 'star-star' argument:

- The variable `a` will contain an array, so it needs to be of type `double*`;
- but it needs to be passed by reference to the function, making the argument type `double**`;
- inside the function you then assign the new storage to the `double*` variable, which is `*a`.

Tricky, I know.

16.2.6 Memory leaks

Pointers can lead to a problem called *memory leaking*: there is memory that you have reserved, but you have lost the ability to access it.

In this example:

```
double *array = new double[100];
// ...
array = new double[105];
```

memory is allocated twice. The memory that was allocated first is never release, because in the intervening code another pointer to it may have been set. However, if that doesn't happen, the memory is both allocated, and unreachable. That's what memory leaks are about.

16.3 Safer pointers in C++

Section 16.2.6 showed how how memory can become unreachable. The *C++11* standard has mechanisms that can help solve this problem[1].

A 'shared pointer' is a pointer that keeps count of how many times the object is pointed to. If one of these pointers starts pointing elsewhere, the shared pointer decreases this 'reference count'. If the reference count reaches zero, the object is deallocated or destroyed.

```
#include <memory>

auto array = std::shared_ptr<double>( new double[100] );
```

As an illustration, let's take a class that reports its construction and destruction:

```
class thing {
public:
  thing() { cout << "calling constructor\n"; };
  ~thing() { cout << "calling destructor\n"; };
};
```

and let's create a pointer and overwrite it:

1. A mechanism along these lines already existed in the 'weak pointer', but it was all but unusable.

Code:

```
cout << "set pointer1" << endl;
auto thing_ptr1 =
  shared_ptr<thing>( new thing );
cout << "overwrite pointer" << endl;
thing_ptr1 = nullptr;
```

Output:

```
set pointer1
calling constructor
overwrite pointer
calling destructor
```

You see that overwriting the pointer does not cause a *memory leak*, as would happen with C-style pointers, but causes destruction of the object it points to.

Next we overwrite a pointer, but after having copied it. Now the destructor will not be called until all references to the object – both the original pointer and the copy – have been overwritten.

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
  shared_ptr<thing>( new thing );
cout << "set pointer3 by copy" << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2" << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3" << endl;
thing_ptr3 = nullptr;
```

Output:

```
set pointer2
calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
calling destructor
```

Chapter 17

Prototypes

17.1 Prototypes for functions

In most of the programs you have written in this course, you put any functions or classes above the main program, so that the compiler could inspect the definition before it encountered the use. However, the compiler does not actually need the whole definition, say of a function: it is enough to know its name, the types of the input parameters, and the return type.

Such a minimal specification of a function is known as function *prototype*; for instance

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

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 and the main program in another:

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

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

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

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

17.1.1 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(float);
```

The header file gets included both in the definitions file and the main program:

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

```
// 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, changing its return type:
- 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.

It is necessary to include the header file in the main program. It is not strictly necessary to include it in the definitions file, but doing so means that you catch potential errors: if you change the function definitions, but forget to update the header file, this is caught by the compiler.

Remark 1 *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.*

Remark 2 *Header files were able to catch more errors in C than they do in C++. With polymorphism of functions, it is no longer an error to have*

```
// header.h
int somefunction(int);
```

and

```
#include "header.h"

int somefunction( float x ) { .... }
```

17.1.2 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>
```

17.2 Global variables

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

```
int processnumber;
void f() {
  ... 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.

17.3 Prototypes for class methods

Header file:

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

Implementation file:

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

17.4 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.

17.5 Namespaces and header files

Never put `using namespace` in a header file.

Chapter 18

Namespaces

18.1 Solving name conflicts

In section 11.2 you saw that the C++ library comes with a *vector* class, that implements dynamic arrays. You say

```
std::vector<int> bunch_of_ints;
```

and you have an object that can store a bunch of ints. And if you use such vectors often, you can save yourself some typing by having

```
using namespace std;
```

somewhere high up in your file, and write

```
vector<int> bunch_of_ints;
```

in the rest of the file.

But what if you are writing a geometry package, which includes a vector class? Is there confusion with the Standard Template Library (STL) vector class? There would be if it weren't for the phenomenon *namespace*, which acts as as disambiguating prefix for classes, functions, variables.

You have already seen namespaces in action when you wrote `std::vector`: the '`std`' is the name of the namespace. You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
|
```

so that you can write

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

18.1.1 Namespace header files

If your namespace is going to be used in more than one program, you want to have it in a separate source file, with an accompanying header file:

```
#include "geolib.h"
using namespace geometry;
```

The header would contain the normal function and class headers, but now inside a named namespace:

```
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

and the implementation file would have the implementations, in a namespace of the same name:

```
namespace geometry {
  point::point( double x,double y ) {
      xcoord = x; ycoord = y; };
  double point::x() { return xcoord; }; // 'accessor'
  double point::y() { return ycoord; };
  vector::vector( point from,point to) {
    this->from = from; this->to = to;
  };
```

```
  double vector::size() {
    double
      dx = to.x()-from.x(), dy = to.y()-from.y();
    return sqrt( dx*dx + dy*dy );
  };
}
```

18.2 Best practices

The problem with

```
using namespace std;
```

is that it may pull in unwanted functions. For instance:

This compiles, but should not:

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

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

This gives an error:

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

It is a good idea to pull in functions explicitly:

```
#include <iostream>
using std::cout;
```

In particular, one should never use the indiscriminate

```
using namespace std;
```

in a header file. Anyone using the header would have no idea what functions are suddenly defined.

Chapter 19

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.

19.1 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.

19.2 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 19.1. 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) ] = ...
```

19.3 Conditionals

There are a couple of *preprocessor conditions*.

19.3.1 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
```

19.3.2 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 17.2.

Chapter 20

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 11 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.

Templated type name

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

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

20.1 Templated functions

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 20.1. 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;
```

20.2 Templated classes

The most common use of templates is probably to define templated classes. You have in fact seen this mechanism in action: the STL contains in effect

```
template<typename T>
class vector {
private:
  // data definitions omitted
public:
  T at(int i) { /* return element i */ };
  int size() { /* return size of data */ };
  // much more
}
```

20.3 Templating over non-types

THESE EXAMPLES ARE NOT GOOD.

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

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

Chapter 21

Error handling

21.1 General discussion

Sources or errors:

- Array indexing. See section 11.2.
- 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

21.2 Exception handling

Throwing an *exception* is one way of signalling an error or unexpected behaviour:

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

It now becomes possible to detect this unexpected behaviour by *catching* the exception:

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

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class: 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;
}
```

You can multiple `catch` statements to catch different types of errors:

```
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 22

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 11),
- strings (chapter 12),
- streams (chapter 13).

22.1 Containers

Vectors (section 11.2) and strings (chapter 12) are special cases of a STL *container*. Methods such as `push_back` and `insert` apply to all containers.

22.1.1 Iterators

The container class has a subclass *iterator* that can be used to iterate through all elements of a container.

```
for (vector::iterator element=myvector.begin();
               element!=myvector.end(); elements++) {
  // do something with element
}
```

You would hope that, if `myvector` is a vector of `int`, `element` would be an int, but it is actually a pointer-to-int; section 16.2. So you could write

```
for (vector::iterator elt=myvector.begin();
               elt!=myvector.end(); elt++) {
  int element = *elt;
  // do something with element
}
```

This looks cumbersome, and you can at least simplify it by letting C++ deduce the type:

```
for (auto elt=myvector.begin(); ...... ) {
 .....
}
```

In the C++11/14 standard the iterator notation has been simplified to *range-based* iteration:

```
for ( int element : myvector ) {
 ...
}
```

22.2 Complex numbers

```
#include <complex>
complex<float> f;
f.re = 1.; f.im = 2.;

complex<double> d(1.,3.);
```

Math operator like `+`, `*` are defined, as are math functions.

22.3 About the 'using' keyword

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

Chapter 23

Obscure stuff

23.1 Const

23.1.1 Const arguments

Function arguments marked `const` can not be altered by the function code. The following segment gives a compilation error:

```
void f(const int i) {
  i++;
}
```

The use of const arguments is one way of protecting you against yourself. If an argument is conceptually supposed to stay constant, the compiler will catch it if you mistakenly try to change it.

23.1.2 Const references

A more sophisticated use of const is the *const reference*:

```
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.2 to return changed values to the calling environment. The const keyword negates that possibility of changing the parameter.

But there is a second reason for using references. Parameters are passed by value, which means that they are copied, and that includes big objects such as `std::vector`. Using a reference to pass a vector is much less costly in both time and space, but then there is the possibility of changes to the vector propagating back to the calling environment.

Marking a vector argument as const allows *compiler optimization*. Assume the function `f` as above, used like this:

```
std::vector<double> v(n);
for ( .... ) {
  f(v);
  y = v[0];
  ... y ... // code that uses y
}
```

Since the function call does not alter the vector, y is invariant in the loop iterations, and the compiler changes this code internally to

```
std::vector<double> v(n);
int saved_y = v[0];
for ( .... ) {
  f(v);
  ... saved_y ... // code that uses y
}
```

Consider a class that has methods that return an internal member by reference, once as const reference and once not:

Code:

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: " << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: " << an_int.int_to_get() << endl;
```

Output:

```
Contained int is now: 4
Contained int is now: 17
```

We can make visible the difference between pass by value and pass by const-reference if we define a class where the *copy constructor* explicitly reports itself:

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) { mine = v; };
  has_int(has_int &other) {
    cout << "(calling copy constructor)" << endl;
    mine = other.mine; };
};
```

Now if we define two functions, with the two parameter passing mechanisms, we see that passing by value invokes the copy constructor, and passing by const reference does not:

Code:

```
void f_with_copy(has_int other) {
  cout << "function with copy" << endl; };
void f_with_ref(const has_int &other) {
  cout << "function with ref" << endl; };
  /* ... */
  cout << "Calling f with copy..." << endl;
  f_with_copy(an_int);

  cout << "Calling f with ref..." << endl;
  f_with_ref(an_int);
```

Output:

```
Demonstrating the copy constructor...
(calling copy constructor)
Calling f with copy...
(calling copy constructor)
function with copy
Calling f with ref...
function with ref
... done
```

23.1.3 Const methods

We can distinguish two types of methods: those that alter internal data members of the object, and those that don't. The ones that don't can be marked `const`:

```
class Things {
private:
  int i;
public:
  int get() const { return i; }
  int inc() { return i++; }
}
```

While this is in no way required, it can be helpful in two ways:

- It will catch mismatches between the prototype and definition of the method. For instance,

  ```
  class Things {
  private:
    int var;
  public:
    int f(int ivar,int c) const {
      return var+c; // typo: should be 'ivar'
    }
  }
  ```

  Here, the use of `var` was a typo, should have been `ivar`. Since the method is marked `const`, the compiler will generate an error.
- It allows the compiler to optimize your code. For instance:

  ```
  class Things {
  public:
    int f() const { /* ... */ };
    int g() const { /* ... */ };
  }
  ...
  ```

```
Things t;
int x,y,z;
x = t.f();
y = t.g();
z = t.f();
```

Since the methods did not alter the object, the compiler can conclude that `x,z` are the same, and skip the calculation for `z`.

23.2 Casts

In C++, constants and variables have clear types. For cases where you want to force the type to be something else, there is the *cast* mechanism. With a cast you tell the compiler: treat this thing as such-and-such a type, no matter how it was defined.

23.2.1 Casting constants

One use of casting is to convert to constants to a 'larger' type. For instance, allocation does not use integers but `size_t`.

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
/* ... */
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
```

C++ pointers are really memory addresses, with no type information to it. With a *cast* it becomes possible change your mind about what a pointer is.

The statis cast is also useful for dealing with a *void pointer*.

The `static_cast` has the safety feature that the compiler will complain if the conversion is not possible.

23.2.2 Dynamic cast

If we have a pointer to a derived object, stored in a pointer to a base class object, it's possible to turn it safely into a derived pointer again:

```
derived_object *derived_pointer;
basic_class *basic_pointer;
derived_pointer = dynamic_cast<derived_object*>(basic_pointer);
if (derived_pointer==nullptr)
  // cast failed
```

Using a `static_cast` here would lead to a compiler error.

23.2.3 Legacy mechanism

The syntax 'open parenthesis, type, closing parenthesis' means:

- take whatever you have here,
- and interpret it as the specified type.

Example:

```
int i[2];
double *point_at_real = (double*)i;
cout << "Print two integers as double: " << *point_at_real << endl;
```

This is very dangerous. It is also impossible to search for such a thing in your editor. Please use the mechanisms above.

23.3 lvalue vs rvalue

The terms 'lvalue' and 'rvalue' sometimes appear in compiler error messages.

```
int foo() {return 2;}

int main()
{
    foo() = 2;

    return 0;
}

# gives:
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment
```

See the 'lvalue' and 'left operand'? To first order of approximation you're forgiven for thinking that an *lvalue* is something on the left side of an assignment. The name actually means 'locator value': something that's associated with a specific location in memory. Thus is lvalue is, also loosely, something that can be modified.

An *rvalue* is then something that appears on the right side of an assignemnt, but is really defined as everything that's not an lvalue. Typically, rvalues can not be modified.

The assignment `x=1` is legal because a variable `x` is at some specific location in memory, so it can be assigned to. On the other hand, `x+1=1` is not legal, since `x+1` is at best a temporary, therefore not at a specific memory location, and thus not an lvalue.

Less trivial examples:

```
int foo() { x = 1; return x; }
int main() {
  foo() = 2;
```

```
}
```

is not legal because `foo` does not return an lvalue. However,

```
class foo {
private:
  int x;
public:
  int &xfoo() { return x; };
};
int main() {
  foo x;
  x.xfoo() = 2;
```

is legal because the function `xfoo` returns a reference to the non-temporary variable `x` of the `foo` object.

Not every lvalue can be assigned to: in

```
const int a = 2;
```

the variable `a` is an lvalue, but can not appear on the left hand side of an assignment.

23.3.1 Conversion

Most lvalues can quickly be converted to rvalues:

```
int a = 1;
int b = a+1;
```

Here `a` first functions as lvalue, but becomes an rvalue in the second line.

The ampersand operator takes an lvalue and gives an rvalue:

```
int i;
int *a = &i;
&i = 5; // wrong
```

23.3.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s = std::string(); // wrong
```

is illegal. The type of `s` is an 'lvalue reference' and it can not be assigned from an rvalue.

On the other hand

```
const std::string &s = std::string();
```

works, since `s` can not be modified any further.

23.3.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through *move semantics*.

Consider a copy assignment operator

```
BigThing& operator=( const BigThing &other ) {
  BigThing tmp(other); // standard copy
  std::swap( /* tmp data into my data */ );
  return *this;
};
```

This calls a copy constructor and a destructor on `tmp`. (The use of a temporary makes this safe under exceptions. The `swap` method never throws an exception, so there is no danger of half-copied memory.)

However, if you assign

```
thing = BigThing(stuff);
```

Now a constructor and destructor is called for the temporary rvalue object on the right-hand side.

Using a syntax that is new in *C++*, we create an *rvalue reference*:

```
BigThing& operator=( BigThing &&other ) {
  swap( /* other into me */ );
  return *this;
}
```

PART III

FORTRAN

Chapter 24

Basics of Fortran

Fortran is an old programming language, dating back to the 1950s, and the first 'high level programming language' that was widely used. In a way, the fields of programming language design and compiler writing started with Fortran, rather than this language being based on established fields. Thus, the design of Fortran has some idiosyncracies that later designed languages have not adopted. Many of these are now 'deprecated' or simply inadvisable. Fortunately, it is possible to write Fortran in a way that is every bit as modern and sophisticated as other current languages.

In this part of our book, we will teach you safe practices for writing Fortran. Occasionally we will not mention practices that you will come across in old Fortran codes, but that we would not advise you taking up. While our exposition of Fortran can stand on its own, we will in places point out explicitly differences with C++.

For secondary reading, this is a good course on modern Fortran: `http://www.pcc.qub.ac.uk/tec/courses/f77tof90/stu-notes/f90studentMIF_1.html`

24.1 Compiling Fortran

For Fortran programs, the compiler is `gfortran` for the GNU compiler, and `ifort` for Intel. Fortran programs can have a number of extensions, but some of them have special meaning to the compiler, by convention. In this book we adopt the `F90` extension.

The minimal Fortran program is:

```
Program SomeProgram
  ! stuff goes here
End Program SomeProgram
```

Exercise 24.1. Add the line

```
print *,"Hello world!"
```

to the empty program, and compile and run it.

24.2 Main program

Fortran does not use curly brackets to delineate blocks, instead you will find `end` statements. The very first one appears right when you start writing your program: a Fortran program needs to start with a `Program` line, and end with `End Program`. The program needs to have a name on both lines:

```
Program SomeProgram
  ! stuff goes here
End Program SomeProgram
```

and you can not use that name for any entities in the program.

24.2.1 Program structure

Unlike C++, Fortran can not mix variable declarations and executable statements, so both the main program and any subprograms have roughly a structure:

```
Program foo
  < declarations >
  < statements >
End Program foo
```

(The *emacs* editor will supply the block type and name if you supply the 'end' and hit the TAB or RETURN key; see section 2.1.1.)

24.2.2 Statements

Let's say a word about layout. Fortran has a 'one line, one statement' principle.

- As long as a statement fits on one line, you don't have to terminate it explicitly with something like a semicolon:
  ```
  x = 1
  y = 2
  ```
- If you want to put two statements on one line, you have to terminate the first one:
  ```
  x = 1; y = 2
  ```
- If a statement spans more than one line, all but the first line need to have an explicit *continuation character*, the ampersand:
  ```
  x = very &
    long &
    expression
  ```

  (This is different between *free format* and *fixed format*, where it's the lines after the first that are marked a continuation, but we don't teach that here.)

24.2.3 Comments

Fortran knows only single-line *comments*, indicated by an exclamation point:

```
x = 1 ! set x to one
```

Everything from the exclamation point onwards is ignored.

Maybe not entirely obvious: you can have a comment after a continuation character:

```
x = f(a) & ! term1
  + g(b)   ! term2
```

24.3 Variables

Unlike in C++, where you can declare a variable right before you need it, Fortran wants its variables declared near the top of the program or subprogram:

```
Program YourProgram
  ! variable declaration
  ! executable code
End Program YourProgram
```

A variable declaration looks like:

```
type, attributes :: name1, name2, ....
```

where

- *type* is most commonly `integer`, `real(4)`, `real(8)`, `logical`. See below; section 24.3.1.
- *attributes* can be `dimension`, `allocatable`, `intent`, `parameters` et cetera.
- *name* is something you come up with. This has to start with a letter.

24.3.1 Data types

Fortran has a somewhat unusual treatment of data types: if you don't specify what data type a variable is, Fortran will deduce it from some default or user rules. This is a very dangerous practice, so we advocate putting a line

```
implicit none
```

immediately after any program or subprogram header.

You can query how many bytes a data type takes with `kind`.

You can set this in the declaration:

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
```

```
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

Numerical precision

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error $\approx 10^{-6}$
- Computations in 8-byte have relative error $\approx 10^{-15}$

Also different exponent range: max 10^{50} and 10^{300} respectively.

Storage size

F08: `storage_size` reports number of bits.

F95: `bit_size` works on integers only.

`c_sizeof` reports number of bytes, requires `iso_c_binding` module.

You can set the precision of floating point numbers with `selected_real_kind`, where the argument is the number of significant digits.

24.3.2 Constants

Since there are 4-byte and 8-byte reals, how is that for real constants? Writing `3.14` will usually be a single precision real. The question is then, if you write

Single precision constants

```
real(8) :: x
x = 3.14
y = 6.022e-23
```

how is that converted to double? This can introduce random junk bits.

Force a constant to be `real(8)`:

Double precision constants

- Use a compiler flag such as `-r8` to force all reals to be 8-byte.
- Write `3.14d0`
- `x = real(3.14, kind=8)`

24.3.3 Initialization

Variables can be initialized in their declaration:

```
integer :: i=2
real(4) :: x = 1.5
```

That this is done at compile time, leading to a common error:

```
subroutine foo()
  integer :: i=2
  print *,i
  i = 3
end subroutine foo
```

On the first subroutine call `i` is printed with its initialized value, but on the second call this initialization is not repeated, and the previous value of `3` is remembered.

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

I/O routines

- Input:

  ```
  READ *,n
  ```

- Output:

  ```
  PRINT *,n
  ```

There is also `WRITE`.

Other syntax for read/write with files and formats.

24.5 Expressions

Arithmetic expressions

- Pretty much as in C++
- Exception: `r**2` for power.
- Modulus is a function: `MOD(7,3)`.

Boolean expressions

- Long form `.and. .not. .or. .lt. .eq. .ge. .true. .false.`
- Short form: `< <= == /= > >=`

Conversion and casting

Conversion is done through functions.

- `INT`: truncation; `NINT` rounding
- `REAL`, `FLOAT`, `SNGL`, `DBLE`
- `CMPLX`, `CONJG`, `AIMIG`

http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm

Complex

Complex numbers exist

24.6 Review questions

Exercise 24.2. In declarations

```
real(4) :: x
real(8) :: y
```

what do the `4` and `8` stand for?
What is the practical implication of using the one or the other?

Exercise 24.3. In the following code, if `value` is nonzero, what do expect about the output?

```
real(8) :: value,should_be_value
real(4) :: value4
/* ... */
print *,"Input value:",value
value4 = value
should_be_value = value4
print *,"Difference:",value-should_be_value
```

Chapter 25

Conditionals

25.1 Boolean variables

The fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`

25.1.1 Operators and such

Operators: `.and. .or. .not.`

Equivalence between logical expressions:

- `.eqv.` : $(x \wedge y) \vee (\neg x \wedge \neg y)$, with negation
- `.neqv.` : $(x \wedge \neg y) \vee (\neg x \wedge y)$.

25.2 Conditionals

The Fortran conditional statement uses the `if` keyword:

Conditionals

Single line conditional:

```
if ( test ) statement
```

The full if-statement is:

```
if ( something ) then
  do something
else
  do otherwise
end if
```

The 'else' part is optional; you can nest conditionals.

You can label conditionals, which is good for readability but adds no functionality:

```
checkx: if ( ... some test on x ... ) then
checky:   if ( ... some test on y ... ) then
               ... code ...
          end if checky
        else checkx
             ... code ...
        end if checkx
```

25.2.1 Operators

Comparison and logical operators

Operator	meaning	example
`==`	equals	`x==y-1`
`/=`	not equals	`x*x*!=5`
`>`	greater	`y>x-1`
`>=`	greater or equal	`sqrt(y)>=7`
<,<=	less, less equal	
`.and. .or.`	and, or	`x<1 .and. x>0`
`!`	not	`.not.( x>1 .and. x<2 )`

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

25.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and characters.

Select statement

Test single values or ranges, integers or characters:

```
  Select Case (i)
Case (:-1)
   print *,"Negative"
Case (5)
   print *,"Five!"
Case (0)
   print *,"Zero."
Case (1:4,6:) ! can not have (1:)
   print *,"Positive"
end Select
```

Compiler does checking on overlapping cases!

25.4 Review questions

Exercise 25.1. What is a conceptual difference between the C++ `switch` and the Fortran `Select` statement?

Chapter 26

Loop constructs

26.1 Loop types

Fortran has the usual indexed and 'while' loops. There are variants of the basic loop, and both use the `do` keyword. The simplest loop has

- a loop variable, which needs to be declared;
- a lower bound and upper bound.

Do loops

```
integer :: i

do i=1,10
  ! code with i
end do
```

You can include a step size (which can be negative) as a third parameter:

```
do i=1,10,3
  ! code with i
end do
```

While loop

The while loop has a pre-test:

```
do while (i<1000)
  print *,i
  i = i*2
end do
```

You can label loops, which improves readability, but so also below.

```
outer: do i=1,10
    inner: do j=1,10
    end do inner
end do outer
```

The label needs to be on the same line as the `do`, and if you use a label, you need to mention it on the `end do` line.

F77 note: Do not use label-terminated loops. Do not use non-integer loop variables.

26.2 Interruptions of the control flow

For interminate looping, you can use the `while` test, or leave out the loop parameter altogether. In that case you need the `exit` statement to stop the iteration.

Exit and cycle

```
do
  x = randomvalue()
  if (x>.9) exit
  print *,"Nine out of ten exes agree"
end do
```

Skip rest of iteration:

```
do i=1,100
  if (isprime(i)) cycle
  ! do something with non-prime
end do
```

Cycle and exit can apply to multiple levels, if the do-statements are labeled.

```
outer : do i = 1,10
inner : do j = 1,10
    if (i+j>15) exit outer
    if (i==j) cycle inner
end do inner
end do outer
```

26.3 Implied do-loops

There are do loops that you can write in a single line. This is useful for I/O. For instance, iterate a simple expression:

Implied do loops

```
print *,(2*i,i=1,20)
```

You can iterate multiple expressions:

```
print *,(2*i,2*i+1,i=1,20)
```

These loops can be nested:

```
print *,( (i*j,i=1,20), j=1,20 )
```

This construct is especially useful for printing arrays.

26.4 Review questions

Exercise 26.1. What is the output of:

```
do i=1,11,3
  print *,i
end do
```

What is the output of:

```
do i=1,3,11
  print *,i
end do
```

Chapter 27

Scope

27.1 Scope

Fortran ‘has no curly brackets’: you not easily create nested scopes with local variables as in C++. For instance, the range between `do` and `end do` is not a scope. This means that all variables have to be declared at the top of a program or subprogram.

27.1.1 Variables local to a program unit

Variables declared in a subprogram have similar scope rules as in C++:

- Their visibility is controlled by their textual scope:

```
Subroutine Foo()
  integer :: i
  ! `i' can now be used
  call Bar()
  ! `i' still exists
End Subroutine Foo
Subroutine Bar() ! no parameters
  ! The `i' of Foo is unknown here
End Subroutine Bar
```

- Their dynamic scope is the lifetime of the program unit in which they are declared:

```
Subroutine Foo()
  call Bar()
  call Bar()
End Subroutine Foo
Subroutine Bar()
  Integer :: i
  ! `i' is created every time Bar is called
End Subroutine Bar
```

27.1.1.1 Variables in a module

Variables in a module (section 28.3) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

27.1.1.2 Other mechanisms for making static variables

Before Fortran gained the facility for recursive functions, the data of each function was placed in a statically determined location. This meant that the second time you call a function, all variables still have the value that they had last time. To force this behaviour in modern Fortran, you can add the `Save` specification to a variable declaration.

Another mechanism for creating static data was the `Common` block. This should not be used, since a `Module` is a more elegant solution to the same problem.

27.1.2 Variables in an internal procedure

An *internal procedure* (that is, one placed in the `Contains` part of a program unit) can receive arguments from the containing program unit. It can also access directly any variable declared in the containing program unit, through a process called *host association*.

The rules for this are messy, especially when considering implicit declaration of variables, so we advise against relying on it.

Chapter 28

Subprograms and modules

28.1 Procedures

Programs can have subprograms: parts of code that for some reason you want to separate from the main program. If you structure your code in a single file, this is the recommended structure:

Subprograms in contains clause

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < subprogram definitions >
End Program foo
```

That is, subprograms are placed after the main program statements, separated by a `contains` clause.

In general, these are the placements of subprograms:

- Internal: after the `Contains` clause of a program
- In a `Module`; see section 28.3.
- Externally: the subprogram is not internal to a `Program` or `Module`. In this case it's safest to declare it through an `Interface` specification; section 28.2.

28.1.1 Subroutines and functions

Fortran has two types of subprograms:

- Subroutines, which are somewhat like `void` functions in C++: they can be used to structure the code, and they can only return information to the calling environment through their parameters.
- Functions, which are like C++ functions with a return type.

Both types have the same structure, which is roughly the same as of the main program:

```
subroutine foo( <parameters> )
<variable declarations>
<executable statements>
end subroutine foo
```

Exit from a procedure can happen two ways:

1. the flow of control reaches the end of the procedure body, or
2. execution is finished by an explicit `return` statement.

```
subroutine foo()
  print *,"foo"
  if (something) return
  print *,"bar"
end subroutine foo
```

The `return` statement is optional in the first case.

A subroutine is invoked with a `call` statement:

```
call foo()
```

Recursion

Declare function as `Recursive Function`

```
Recursive Function factorial(n) Result(fact)
  if (n==0) then
    fact = 1
  else
    fact = n * factorial(n-1)
  end if
End Function Factorial
```

28.1.2 Return results

While a `subroutine` can only return information through its parameters, a *function* procedure returns an explicit result:

```
logical function test(x)
  implicit none
  real :: x

  test = some_test_on(x)
  return ! optional, see above
end function test
```

You see that the result is not returned in the `return` statement, but rather through assignment to the function name. The `return` statement, as before, is optional and only indicates where the flow of control ends.

A *function* in Fortran is a subprogram that return a result to its calling program, much like a non-void function in C++

Function definition and usage

- Return type, keyword `function`, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

Function example

Code:

```
program plussing
  implicit none
  integer :: i
  i = plusone(5)
  print *,i
contains
  integer function plusone(invalue)
    implicit none
    integer,intent(in) :: invalue
    plusone = invalue+1
  end function plusone
end program plussing
```

Output:

```
           6
```

A function is not invoked with `call`, but rather through being used in an expression:

```
if (test(3.0) .and. something_else) ...
```

You now have the following cases to make the function known in the main program:

- If the function is in a `contains` section, its type is known in the main program.
- If the function is in a module (see section 28.3 below), it becomes known through a `use` statement.

F77 note: Without modules and `contains` sections, you need to declare the function type explitly in the calling program. The safe way is through using an `interface` specification.

Exercise 28.1. Write a program that asks the user for a positive number; negative input should be rejected. Fill in the missing lines in this code fragment:

Code:

```
program readpos
  implicit none
  real(4) :: userinput
  print *,"Type a positive number:"
  userinput = read_positive()
  print *,"Thank you for",userinput
contains
  real(4) function read_positive()
    implicit none
  /* ... */
  end function read_positive
end program readpos
```

Output:

```
 Type a positive number:
 No, not  -5.00000000
 No, not   0.00000000
 No, not  -3.14000010
 Thank you for   2.48000002
```

Why a 'contains' clause?

```
Program ContainsScope
  implicit none
  call DoWhat()
end Program ContainsScope

subroutine DoWhat(i)
  implicit none
  integer :: i
  i = 5
end subroutine DoWhat
```

```
Program ContainsScope
  implicit none
  call DoWhat()
contains
  subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
  end subroutine DoWhat
end Program ContainsScope
```

Warning only, crashes.

Error, does not compile

Why a 'contains' clause, take 2

Code:

```
Program ContainsScope
  implicit none
  integer :: i=5
  call DoWhat(i)
end Program ContainsScope

subroutine DoWhat(x)
  implicit none
  real :: x
  print *,x
end subroutine DoWhat
```

Output:

```
   7.00649232E-45
```

At best compiler warning if all in the same file
For future reference: if you see very small floating point numbers, maybe you have made this error.

28.1.3 Arguments

Subprogram arguments

Arguments are defined in subprogram body:

```
subroutine f(x,y,i)
  implicit none
  integer,intent(in) :: i
  real(4),intent(out) :: x
  real(8),intent(inout) :: y
  x = 5; y = y+6
end subroutine f
! and in the main program
call f(x,y,5)
```

Parameter passing

- Everything is passed by reference.
- Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.
- Terminology: Fortran talks about 'dummy' and 'actual' arguments. Dummy: in the definition; actual: in the calling program.

Intent checking

Compiler checks your intent against your implementation. This code is not legal:

```
subroutine ArgIn(x)
  implicit none
  real,intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

Why intent checking?

Allow compiler optimizations:

```
x = f()
call ArgOut(x)
print *,x
```

Call to `f` removed

```
do i=1,1000
  x = ! something
  y1 = .... x ....
  call ArgIn(x)
  y2 = ! same expression as y1
```

`y2` is same as `y1` because `x` not changed

28.1.4 Types of procedures

Procedures that are in the main program (or another type of program unit), separated by a `contains` clause, are known as *internal procedures*. This is as opposed to *module procedures*.

There are also *statement functions*, which are single-statement functions, usually to identify commonly used complicated expressions in a program unit. Presumably the compiler will *inline* them for efficiency.

The `entry` statement is so bizarre that I refuse to discuss it.

28.1.5 More about arguments

Keyword arguments

- Use the name of the *formal parameter* as keyword.
- Keyword arguments have to come last.

Code:

```
  call say_xy(1,2)
  call say_xy(x=1,y=2)
  call say_xy(y=2,x=1)
  call say_xy(1,y=2)
  ! call say_xy(y=2,1) ! ILLEGAL
contains
  subroutine say_xy(x,y)
    implicit none
    integer,intent(in) :: x,y
    print *,"x=",x,", y=",y
  end subroutine say_xy
```

Output:

```
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
 x=           1 , y=           2
```

Optional arguments

- Extra specifier: `Optional`
- Presence of argument can be tested with `Present`

28.2 Interfaces

If you want to use a subprogram in your main program, the compiler needs to know the signature of the subprogram: how many arguments, of what type, and with what intent. You have seen how the `contains` clause can be used for this purpose if the subprogram resides in the same file as the main program.

If the subprogram is in a separate file, the compiler does not see definition and usage in one go. To allowed the compiler to do checking on proper usage, we can use an `interface` block. This is placed at the calling site, declaring the signature of the subprogram.

Main program:

```
interface
   function f(x,y)
     real*8 :: f
     real*8,intent(in) :: x,y
   end function f
end interface

real*8 :: in1=1.5, in2=2.6, result

result = f(in1,in2)
```

Subprogram:

```
function f(x,y)
  implicit none
  real*8 :: f
  real*8,intent(in) :: x,y
```

The `interface` block is not required (an older `external` mechanism exists for functions), but is recommended. It is required if the function takes function arguments.

28.2.1 Polymorphism

The `interface` block can be used to define a generic function:

```
interface f
function f1( ..... )
function f2( ..... )
end interface f
```

where `f1`,`f2` are functions that can be distinguished by their argument types. The generic function `f` then becomes either `f1` or `f2` depending on what type of argument it is called with.

28.3 Modules

A module is a container for definitions of subprograms and types, and for data such as constants and variables. A module is not a structure or object: there is only one instance.

What do you use a module for?

- Type definitions: it is legal to have the same type definition in multiple program units, but this is not a good idea. Write the definition just once in a module and make it available that way.
- Function definitions: this makes the functions available in multiple sources files of the same program, or in multiple programs.
- Define constants: for physics simulations, put all constants in one module and use that, rather than spelling out the constants each time.
- Global variables: put variables in a module if they do not fit an obvious scope.

F77 note: Modules are much cleaner than common blocks. Do not use those.

Module definition

```
Module FunctionsAndValues
  implicit none

  real(8),parameter :: pi = 3.14

contains
  subroutine SayHi()
    print *,"Hi!"
  end subroutine SayHi

End Module FunctionsAndValues
```

Any routines come after the `contains`

A module is made available with the `use` keyword, which needs to go before the `implicit none`.

Module use

```
Program ModProgram
  use FunctionsAndValues
  implicit none

  print *,"Pi is:",pi
  call SayHi()

End Program ModProgram
```

Also possible:

```
Use mymodule, Only: func1,func2
Use mymodule, func1 => new_name1
```

By default, all the contents of a module is usable by a subprogram that uses it. However, a keyword `private` make module contents available only inside the module. You can make the default behaviour explicit by using the `public` keyword. Both `public,private` can be used as attributes on definitions in the module. There is a keyword `protected` for data members that are public, but can not be altered by code outside the module.

If you compile a module, you will find a `.mod` file in your directory. (This is little like a `.h` file in C++.) If this file is not present, you can not `use` the module in another program unit, so you need to compile the file containing the module first.

Exercise 28.2. Write a module `PointMod` that defines a type `Point` and a function `distance` to make this code work:

```
use pointmod
implicit none
type(Point) :: p1,p2
real(8) :: p1x,p1y,p2x,p2y
read *,p1x,p1y,p2x,p2y
p1 = point(p1x,p1y)
p2 = point(p2x,p2y)
print *,"Distance:",distance(p1,p2)
```

Put the program and module in two separate files and compile thusly:

```
ifort -g -c pointmod.F90
ifort -g -c pointmain.F90
ifort -g  -o pointmain pointmod.o pointmain.o
```

28.3.1 Polymorphism

```
module somemodule
```

```
INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

  contains
  subroutine swapreal
  ...
  end subroutine swapreal
  subroutine swapint
  ...
  end subroutine swapint
```

28.3.2 Operator overloading

```
MODULE operator_overloading
IMPLICIT NONE
...
INTERFACE OPERATOR (+)
MODULE PROCEDURE concat
END INTERFACE
```

including the assignment operator:

```
INTERFACE ASSIGNMENT (=)
subroutine_interface_body
END INTERFACE
```

This mechanism can also be used for dot-operators:

```
INTERFACE OPERATOR (.DIST.)
MODULE PROCEDURE calcdist
END INTERFACE
```

Chapter 29

String handling

29.1 String denotations

Code:

```
print *,'This string was in single quotes'
print *,'This string in single quotes contains a single '' quote'
print *,"This string was in double quotes"
print *,"This string in double quotes contains a double "" quote"
```

Output:

29.2 Characters

29.3 Strings

Intrinsic functions: `LEN(string)`, `INDEX(substring,string)`, `CHAR(int)`, `ICHAR(char)`, `TRIM(string)`

Code:

```
character(len=10) :: firstname,lastname
character(len=15) :: shortname,fullname
firstname = "Victor"; lastname = "Eijkhout"
shortname = firstname // lastname
print *,"without trimming: ",shortname
fullname = trim(firstname) // " " // trim(lastname)
print *,"with trimming: ",fullname
```

Output:

```
without trimming: Victor    Eijkh
with trimming: Victor Eijkhout
```

Code:

```
character(len=12) :: strvar
strvar = "word"
print *,len(strvar),len(trim(strvar))
```

Output:

```
         12           4
```

29.4 Strings versus character arrays

Chapter 30

Structures, eh, types

Structures: `type`

The Fortran name for structures is `type` or *derived type*.

Type definition

`Type name` / `End Type` block. Variable declarations inside the block

```
type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

Creating a type structure

Declare a typed object in the main program:

```
Type(mytype) :: typed_object,object2
```

Initialize with type name:

```
typed_object = mytype( 1, 'my_name', 3.7 )
object2 = typed_object
```

Member access

Access structure members with `%`

```
Type(mytype) :: typed_object
type_object%member = ....
```

Example

```
type point
   real :: x,y
end type point
```

```
type(point) :: p1,p2
p1 = point(2.5, 3.7)

p2 = p1
print *,p2%x,p2%y

type(my_struct) :: data
type(my_struct),dimension(1) :: data_array
```

Types as subprogram argument

```
real(4) function length(p)
  implicit none
  type(point),intent(in) :: p
  length = sqrt( p%x**2 + p%y )
end function length
```

Chapter 31

Classes and objects

31.1 Classes

Classes and objects

Fortran classes are based on `type` objects, a little like the analogy between C++ `struct` and `class` constructs.

New syntax for specifying methods.

Object is type with methods

You define a type as before, with its data members, but now the type has a `contains` for the methods:

```
Module multmod

  type Scalar
     real(4) :: value
   contains
     procedure,public :: print
     procedure,public :: scaled
  end type Scalar

contains ! methods
end Module multmod

Program Multiply
  use multmod
  implicit none

  type(Scalar) :: x
  real(4) :: y
  x = Scalar(-3.14)
  call x%print()
  y = x%scaled(2.)
  print '(f7.3)',y

end Program Multiply
```

Method definition

```
subroutine print(me)
  implicit none
  class(Scalar) :: me
  print '("The value is",f7.3)',me%value
end subroutine print
function scaled(me,factor)
  implicit none
  class(Scalar) :: me
  real(4) :: scaled,factor
  scaled = me%value * factor
```

```
end function scaled
```

Methods have object as argument

You define functions that accept the type as first argument, but instead of declaring the argument as `type`, you define it as `class`.

The members of the class object have to be accessed through the `%` operator.

```
subroutine set(p,xu,yu)
  implicit none
  class(point) :: p
  real(8),intent(in) :: xu,yu
  p%x = xu; p%y = yu
end subroutine set
```

Class organization

- You're pretty much forced to use `Module`
- A class is a `Type` with a `contains` clause followed by `procedure` declaration
- Actual methods go in the `contains` part of the module
- First argument of method is the object itself.

Point program

```
Module PointClass
  Type,public :: Point
     real(8) :: x,y
   contains
     procedure, public :: distance
  End type Point
contains
  ! ....
End Module PointClass
```

```
Program PointTest
  use PointClass
  implicit none
  type(Point) :: p1,p2

  p1 = point(1.d0,1.d0)
  p2 = point(4.d0,5.d0)

  print *,"Distance:",p1%distance(p2)

End Program PointTest
```

Exercise 31.1. Take the point example program and add a distance function:

```
Type(Point) :: p1,p2

! initialize
dist = p1%distance(p2)
```

Exercise 31.2. Write a method `add` for the `Point` type:

```
Type(Point) :: p1,p2,sum
! initialize
sum = p1%add(p2)
```

What is the return type of the function `add`?

Chapter 32

Arrays

32.1 Static arrays

The preferred way for specifying an array size is:

Fortran dimension

```
real(8), dimension(100) :: x,y
integer :: i(10,20)
```

Static, obey scope.

Such an array, with size explicitly indicated, is called a *static array* or *automatic array*. (See section 32.4 for dynamic arrays.)

Array indexing in Fortran is 1-based:

1-based Indexing

```
integer,parameter :: N=8
real(4),dimension(N) :: x
do i=1,N
  ... x(i) ...
```

Unlike C++, Fortran can specify the lower bound explicitly:

Lower bound

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Such arrays, as in C++, obey the scope: they disappear at the end of the program or subprogram.

32.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of implicit do-loops:

Array initialization

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
/* ... */
real5 = [ (1.01*i,i=1,size(real5,1)) ]
/* ... */
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

32.1.2 Sections

Array sections

Use the colon notation to indicate ranges:

```
real(4),dimension(5) :: x
x(2:5) = x(1:4)
```

Use of sections

Code:

```
real(8),dimension(5) :: x = &
     [.1d0, .2d0, .3d0, .4d0, .5d0]
x(2:5) = x(1:4)
print '(f5.3)',x
```

Output:

```
0.100
0.100
0.200
0.300
0.400
```

You can even use a stride:

Strided sections

Code:

```
integer,dimension(5) :: &
     y = [0,0,0,0,0]
integer,dimension(3) :: &
     z = [3,3,3]
y(1:5:2) = z(1:3)
print '(i3)',y
```

Output:

```
  3
  0
  3
  0
  3
```

32.1.3 Integer arrays as indices

Index arrays

```
integer,dimension(4) :: i = [2,4,6,8]
real(4),dimension(10) :: x
print *,x(i)
```

32.2 Multi-dimensional

Arrays above had '*rank* one'. The rank is defined as the number of indices you need to address the elements. Calling this the 'dimension' of the array can be confusing, but we will talk about the first and second dimension of the array.

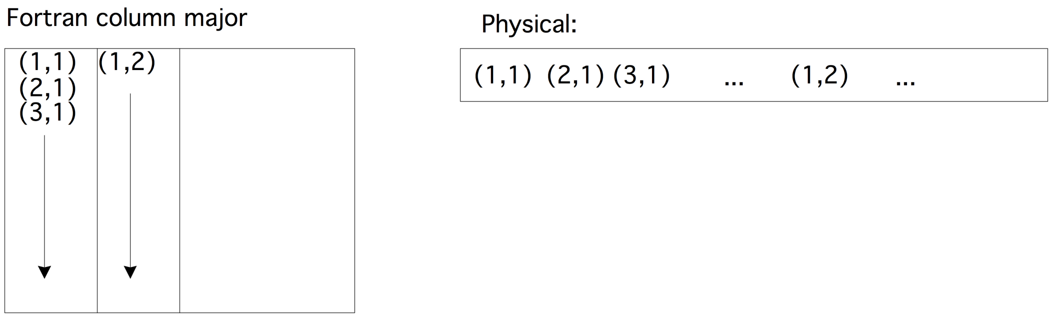
A rank-two array, or matrix, is defined like this:

Multi-dimension arrays

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

Array layout

Sometimes you have to take into account how a higher rank array is laid out in (linear) memory:



32.2.1 Querying an array

We have the following properties of an array:

- The bounds are the lower and upper bound in each dimension. For instance, after

  ```
  integer,dimension(-1:1,-2:2) :: symm
  ```

  the array `symm` has a lower bound of `-1` in the first dimension and `-2` in the second. The functions `Lbound` and `Ubound` give these bounds as array or scalar:

  ```
  array_of_lower = Lbound(symm)
  upper_in_dim_2 = Ubound(symm,2)
  ```

  Code:

  ```
  real(8),dimension(2:N+1) :: Afrom2 = &
       [1,2,3,4,5]
  lo = lbound(Afrom1,1)
  hi = ubound(Afrom1,1)
  print *,lo,hi
  print '(i3,":",f5.3)', &
       (i,Afrom1(i),i=lo,hi)
  ```

  Output:

  ```
             1           5
    1:1.000
    2:2.000
    3:3.000
    4:4.000
    5:5.000
  ```

- The *extent* is the number of elements in a certain dimension, and the *shape* is the array of extents.

- The `size` is the number of elements, either for the whole array, or for a specified dimension.

```
integer :: x(8), y(5,4)
size(x)
size(y,2)
```

32.2.2 Reshaping

RESHAPE

```
array = RESHAPE( list,shape )
```

Example:

```
square = reshape( (/(i,i=1,16)/),(/4,4/) )
```

SPREAD

```
array = SPREAD( old,dim,copies )
```

32.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

Pass array to subroutine

```
    real(8) function arraysum(x)
      implicit none
      real(8),intent(in),dimension(:) :: x
    /* ... */
      do i=1,size(x)
         tmp = tmp+x(i)
      end do
    /* ... */
Program ArrayComputations1D
      use ArrayFunction
      implicit none

      real(8),dimension(:) :: x(N)
    /* ... */
      print *,"Sum of one-based array:",arraysum(x)
```

The array inside the subroutine is known as a *assumed-shape array* or *automatic array*.

32.4 Allocatable arrays

Static arrays are fine at small sizes. However, there are two main arguments against using them at large sizes.

- Since the size is explicitly stated, it makes your program inflexible, requiring recompilation to run it with a different problem size.
- Since they are allocated on the so-called *stack*, making them too large can lead to *stack overflow*.

A better strategy is to indicate the shape of the array, and use `allocate` to specify the size later, presumably in terms of run-time program parameters.

Array allocation

```
real(8), dimension(:), allocatable :: x,y

n = 100
allocate(x(n), y(n))
```

You can `deallocate` the array when you don't need the space anymore.

If you are in danger of running out of memory, it can be a good idea to add a `stat=ierror` clause to the `allocate` statement:

```
integer :: ierr
allocate( x(n), stat=ierr )
if ( ierr/=0 ) ! report error
```

Has an array been allocated:

```
Allocated( x ) ! returns logical
```

Allocatable arrays are automatically deallocated when they go out of scope. This prevents the *memory leak* problems of C++.

Explicit deallocate:

```
deallocate(x)
```

32.5 Array slicing

Fortran is more sophisticated than C++ in how it can handle arrays as a whole. For starters, you can assign one array to another:

```
real(4),dimension(25,26) :: a,b
a = b
```

You can assign subarrays, or *array slices*, as long as they have the same shape. You use colon-syntax to indicate ranges:

Array slicing

- `:` to get all indices,
- `:n` to get indices up to `n`,
- `n:` to get indices `n` and up.

For multi-dimensional arrays, you need to indicate a range in all dimensions.

Array slicing in multi-D

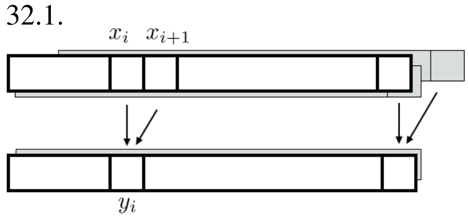
```
real(8),dimension(10) :: a,b
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a
logical,dimension(25)   :: b
a(:,2) = b
```

You can also use strides.

Exercise 32.1.



Code $\forall_i \colon y_i = (x_i + x_{i+1})/2$:

- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array `x` with values that allow you to check the correctness of your code.

32.6 Array output

Use implicit do-loops; section 26.3.

32.7 Operating on an array

32.7.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D*E
```

(where the multiplication is by element).

32.7.2 Intrinsic functions

The following intrinsic functions are avaible for arrays:

Array intrinsics

- `MaxVal` finds the maximum value in an array.
- `MinVal` finds the minimum value in an array.
- `Sum` returns the sum of all elements.
- `Product` return the product of all elements.
- `MaxLoc` returns the index of the maximum element.

```
i = MAXLOC( array [, mask ] )
```

- `MinLoc` returns the index of the minimum element.
- `MatMul` returns the matrix product of two matrices.
- `Dot_Product` returns the dot product of two arrays.
- `Transpose` returns the transpose of a matrix.
- `Cshift` rotates elements through an array.

Exercise 32.2. The 1-norm of a matrix is defined as the maximum sum of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Implement these functions using array intrinsics.

Exercise 32.3. Compare implementations of the matrix-matrix product.

1. Write the regular `i,j,k` implementation, and store it as reference.
2. Use the `DOT_PRODUCT` function, which eliminates the `k` index. How does the timing change? Print the maximum absolute distance between this and the reference result.
3. Use the `MATMUL` function. Same questions.
4. Bonus question: investigate the `j,k,i` and `i,k,j` variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

32.7.3 Restricting with `where`

`where`

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
  sequence of array statements
ELSEWHERE
  sequence of array statements
END WHERE
```

32.7.4 Global condition tests

Reduction of a test on all array elements: `all`

```
REAL(8),dimension(N,N) :: A
LOGICAL :: positive,positive_row(N),positive_col(N)
positive = ALL( A>0)
positive_row = ALL( A>0,1 )
positive_col = ALL( A>0,2 )
```

Exercise 32.4. Use array statements (that is, no loops) to fill a two-dimensional array `A` with random numbers between zero and one. Then fill two arrays `Abig` and `Asmall` with the elements of `A` that are great than 0.5, or less than 0.5 respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

Using more array statements, add `Abig` and `Asmall`, and test whether the sum is close enough to `A`.

Similar to `all`, there is a function `any` that tests if any array element satisfies the test.

```
if ( Any( Abs(A-B)>
```

32.8 Array operations

32.8.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

- Slicing: if your loop assigns to an array from another array, you can use slice notation:
  ```
  a(:) = b(:)
  c(1:n) = d(2:n+1)
  ```
- The `forall` keyword also indicates an array assignment:

```
forall (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end forall
```

You can tell that this is for arrays only, because the loop index has to be part of the left-hand side of every assignment.
This mechanism is prone to misunderstanding and therefore now deprecated. It is not a parallel loop! For that, the following mechanism is preferred.

- The *do concurrent* is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end do
```

32.8.2 Loops without dependencies

Here are some illustrations of simple array copying with the above mechanisms.

```
do i=2,n
   counted(i) = 2*counting(i-1)
end do

Original      1   2   3   4   5   6   7   8   9  10
Recursive     0   2   4   6   8  10  12  14  16  18

counted(2:n) = 2*counting(1:n-1)

Original      1   2   3   4   5   6   7   8   9  10
Sliced        0   2   4   6   8  10  12  14  16  18

forall (i=2:n)
   counted(i) = 2*counting(i-1)
end forall

Original      1   2   3   4   5   6   7   8   9  10
Forall        0   2   4   6   8  10  12  14  16  18

do concurrent (i=2:n)
   counted(i) = 2*counting(i-1)
end do

Original      1   2   3   4   5   6   7   8   9  10
```

```
Concurrent   0   2   4   6   8  10  12  14  16  18
```

Exercise 32.5. Create arrays `A,C` of length `2N`, and `B` of length `N`. Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \ldots N$$

and

$$C_i = A_{i/2}, \quad i = 1, \ldots 2N$$

using all four mechanisms. Make sure you get the same result every time.

32.8.3 Loops with dependencies

For parallel execution of a loop, all iterations have to be independent. This is not the case if the loop has a *recurrence*, and in this case, the 'do concurrent' mechanism is not appropriate. Here are the above four constructs, but applied to a loop with a dependence.

```
do i=2,n
   counting(i) = 2*counting(i-1)
end do

Original   1   2   3   4   5   6   7   8   9  10
Recursiv   1   2   4   8  16  32  64 128 256 512
```

The slicing version of this:

```
counting(2:n) = 2*counting(1:n-1)

Original   1   2   3   4   5   6   7   8   9  10
Sliced     1   2   4   6   8  10  12  14  16  18
```

acts as if the right-hand side is saved in a temporary array, and subsequently assigned to the left-hand side.

Using 'forall' is equivalent to slicing:

```
forall (i=2:n)
   counting(i) = 2*counting(i-1)
end forall

Original     1   2   3   4   5   6   7   8   9  10
Forall       1   2   4   6   8  10  12  14  16  18
```

On the other hand, 'do concurrent' does not use temporaries, so it is more like the sequential version:

```
do concurrent (i=2:n)
   counting(i) = 2*counting(i-1)
end do
```

```
Original      1   2   3   4   5   6   7   8   9  10
Concurrent    1   2   4   8  16  32  64 128 256 512
```

Note that the result does not have to be equal to the sequential execution: the compiler is free to rearrange the iterations any way it sees fit.

32.9 Review questions

Exercise 32.6. Let the following declarations be given, and assume that all arrays are properly initialized:

```
real                    :: x
real, dimension(10)     :: a, b
real, dimension(10,10)  :: c, d
```

Comment on the following lines: are they legal, if so what do they do?

1. `a = b`
2. `a = x`
3. `a(1:10) = c(1:10)`

How would you:

1. Set the second row of `c` to `b`?
2. Set the second row of `c` to the elements of `b`, last-to-first?

Chapter 33

Pointers

Pointers in C++ were largely the same as memory addresses (until you got to smart pointers). Fortran pointers on the other hand, are more abstract.

33.1 Basic pointer operations

Pointers are aliases

- Pointer points at an object
- Access object through pointer
- You can change what object the pointer points at.

```
real,pointer :: point_at_real
```

Pointers could also be called 'aliases': they act like an alias for an object of elementary or derived data type. You can access the object through the alias. The difference with actually using the object, is that you can decide what object the pointer points at.

The `pointer` definition

```
real,pointer :: point_at_real
```

defined a pointer that can point at a real variable.

Setting the pointer

- You have to declare that a variable is pointable:
  ```
  real,target :: x
  ```
- Set the pointer with => notation:
  ```
  point_at_real => x
  ```
- Now using `point_at_real` is the same as using `x`.

Pointers can not just point at anything: the thing pointed at needs to be declared as `target`

```
real,target :: x
```

and you use the => operator to let a pointer point at a target:

```
point_at_real => x
```

If you use a pointer, for instance to print it

```
print *,point_at_real
```

it behaves as if you were using the value of what it points at.

Pointer example

```
real,target :: x,y
real,pointer :: that_real

x = 1.2
y = 2.4
that_real => x
print *,that_real
that_real => y
print *,that_real
y = x
print *,that_real
```

1. The pointer points at `x`, so the value of `x` is printed.
2. The pointer is set to point at `y`, so its value is printed.
3. The value of `y` is changed, and since the pointer still points at `y`, this changed value is printed.

Assign pointer from other pointer

```
real,pointer :: point_at_real,also_point
point_at_real => x
also_point => point_at_real
```

Now you have two pointers that point at `x`.

Very important to use the =>, otherwise strange memory errors

If you have two pointers

```
real,pointer :: point_at_real,also_point
```

you can make the target of the one to also be the target of the other:

```
also_point => point_at_real
```

This is not a pointer to a pointer: it assigns the target of the right-hand side to be the target of the left-hand side.

Using ordinary assignment does not work, and will give strange memory errors.

Exercise 33.1. Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

```
call SetPointer(array,biggest_element)
print *,array(1),biggest_element,array(size(array))
```

33.2 Example: linked lists

Linked list

- Linear data structure
- more flexible for insertion / deletion
- ... but slower in access

One of the standard examples of using pointers is the *linked list*. This is a dynamic one-dimensional structure that is more flexible than an array. Dynamically extending an array would require re-allocation, while in a list an element can be inserted.

Exercise 33.2. Using a linked list may be more flexible than using an array. On the other hand, accessing an element in a linked list is more expensive, both absolutely and as order-of-magnitude in the size of the list.
Make this argument precise.

Linked list datatypes

- Node: value field, and pointer to next node.
- List: pointer to head node.

```
type node
   integer :: value
   type(node),pointer :: next
end type node

type list
   type(node),pointer :: head
end type list

type(list) :: the_list
nullify(the_list%head)
```

A list is based on a simple data structure, a node, which contains a value field and a pointer to another node.

By way of example, we create a dynamic list of integers, sorted by size. To maintain the sortedness, we need to append or insert nodes, as required.

Here are the basic definitions of a node, and a list which is basically a repository for the head node:

```
type node
   integer :: value
   type(node),pointer :: next
end type node
```

```
type list
   type(node),pointer :: head
end type list

type(list) :: the_list
nullify(the_list%head)
```

List initialization

First element becomes the list head:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
the_list%head => new_node
```

Initially, the list is empty, meaning that the 'head' pointer is un-associated. The first time we add an element to the list, we create a node and assign it as the head of the list:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
the_list%head => new_node
```

Attaching a node

Keep the list sorted: new largest element attached at the end.

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
current%next => new_node
```

Adding a value to a list can be done two ways. If the new element is larger than all elements in the list, a new node needs to be appended to the last one. Let's assume we have managed to let `current` point at the last node of the list, then here is how to attaching a new node from it:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
current%next => new_node
```

Inserting 1

Find the insertion point:

```
current => the_list%head ; nullify(previous)
do while ( current%value<value &
     .and. associated(current%next) )
   previous => current
```

```
   current => current%next
end do
```

Inserting an element in the list is harder. First of all, you need to find the two nodes, `previous` and `current`, between which to insert the new node:

```
current => the_list%head ; nullify(previous)
do while ( current%value<value &
     .and. associated(current%next) )
   previous => current
   current => current%next
end do
```

Inserting 2

The actual insertion requires rerouting some pointers:

```
allocate(new_node)
new_node%value = value
new_node%next => current
previous%next => new_node
```

Chapter 34

Input/output

34.1 Print to terminal

The simplest command for outputing data is `print`.

```
print *,"The result is",result
```

In its easiest form, you use the star to indicate that you don't care about formatting; after the comma you can then have any number of comma-separated strings and variables.

Formatted and unformatted I/O

- Formatted: ascii output. This is good for reporting, but not data analysis.
- Unformatted: binary output. Great for further processing of output data. See section 34.3.
- Beware: binary data is machine-dependent. Use *hdf5* for portable binary.

34.1.1 Print on one line

The statement

```
print *,item1,item2,item3
```

will print the items on one line.

Implicit do loops

Parametrized printing with an *implicit do loop*:

```
print *,( i*i,i=1,n)
```

34.1.2 Printing arrays

If you print a variable that is an array, all its elements will be printed, in *column-major* ordering if the array is multi-dimensional.

You can also control the printing of an array by using an *implicit do loop*:

```
print *,( A(i),i=1,n)
```

34.1.3 Formats

The default formatting uses quite a few positions for what can be small numbers. To indicate explicitly the formatting, for instance limiting the number of positions used for a number, or the whole and fractional part of a real number, you can use a format string.

```
print '(a6,3f5.3)',"Result",x,y,z
```

The format specifier is inside single quotes and parentheses, and consists of comma-separated specifications for a single item:

- '`a`n' specifies a string of n characters. If the actual string is longer, it is truncated in the output.
- '`i`n' specifies an integer of up to n digits. If the actual number takes more digits, it is rendered with asterisks.
- '`f`$m.n$' specifies a fixed point representation of a real number, with m total positions (including the decimal point) and n digits in the fractional part.
- '`e`$m.n$' Exponent representation.
- Strings can go into the format:
  ```
  print '("Result:",3f5.3)',x,y,z
  ```
- '`x`' for a space, '`/`' for newline

Putting a number in front of a single specifier indicates that it is to be repeated.

If you find yourself using the same format a number of times, you can give it a *label*:

```
  print 10,"result:",x,y,z
10 format('(a6,3f5.3)')
```

https://www.obliquity.com/computer/fortran/format.html

Format repetitions

```
print '( 3i4 )', i1,i2,i3
print '( e(i2,":",f7.4) )', i1,r1,i2,r2,i3,r2
```

Repeats and line breaks

- If `abc` is a format string, then `10(abc)` gives 10 repetitions. There is no line break.
- If there is more data than specified in the format, the format is reused in a new print statement. This causes line breaks.
- The / (slash) specifier causes a line break.
- There may be a 80 character limit on output lines.

Exercise 34.1. Use formatted I/O to print the number $0 \cdots 99$ as follows:

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
```

```
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

34.2 File and stream I/O

If you want to send output anywhere else than the terminal screen, you need the `write` statement, which looks like:

```
write (unit,format) data
```

where `format` and `data` are as described above. The new element is the *unit*, which is a numerical indication of an output device, such as a file.

34.2.1 Units

```
Open(11)
```

will result in a file with a name typically `fort11`.

```
Open(11,FILE="filename")
```

Many other options for error handling, new vs old file, etc.

After this:

```
Write (11,fmt) data
```

Again options for errors and such.

34.2.2 Other write options

```
write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

`open close`

34.3 Unformatted output

So far we have looked at ascii output, which is nice to look at for a human , but is not the right medium to communicate data to another program.

- Ascii output requires time-consuming conversion.
- Ascii rendering leads to loss of precision.

Therefore, if you want to output data that is later to be read by a program, it is best to use *binary output* or *unformatted output*, sometimes also called *raw output*.

Unformatted output

Indicated by lack of format specification:

```
write (*) data
```

Note: may not be portable between machines.

Chapter 35

Leftover topics

35.1 Timing

Timing is done with the `system_clock` routine.

- This call gives an integer, counting clock ticks.
- To convert to seconds, it can also tell you how many ticks per second it has: its *timer resolution*.

```
integer :: clockrate,clock_start,clock_end
call system_clock(count_rate=clockrate)
print *,"Ticks per second:",clockrate

call system_clock(clock_start)
! code
call system_clock(clock_end)
print *,"Time:",(clock_end-clock_start)/REAL(clockrate)
```

PART IV

EXERCISES AND PROJECTS

Chapter 36

Simple exercises

36.1 Arithmetic

1. Given

   ```
   int n;
   ```

   write code that uses elementary mathematical operators to compute n-cubed: n^3. Do you get the correct result for all n? Explain.
2. What is the output of:

   ```
   int m=32, n=17;
   cout << n%m << endl;
   ```

36.2 Scope

1. Is this a valid program?

   ```
   void f() { i = 1; }
   int main() {
     int i=2;
     f();
     return 0;
   }
   ```

   If yes, what does it do; if no, why not?
2. What is the output of:

   ```
   #include <iostream>
   using namespace std;
   int main() {
     int i=5;
     if (true) { i = 6; }
     cout << i << endl;
     return 0;
   }
   ```

3. What is the output of:

```
#include <iostream>
using namespace std;
int main() {
  int i=5;
  if (true) { int i = 6; }
  cout << i << endl;
  return 0;
}
```

4. What is the output of:

```
#include <iostream>
using namespace std;
int main() {
  int i=2;
  i += /* 5;
  i += */ 6;
  cout << i << endl;
  return 0;
}
```

36.3 Looping

1. Suppose a function

```
bool f(int);
```

   is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which `f` is true.
2. Suppose a function

```
bool f(int);
```

   is given, which is true for some negative input value. Write a main program that finds the (negative) input with smallest absolute value for which `f` is true.

36.4 Subprograms

Exercise 36.1. Write the missing function `pos_input` that

- reads a number from the user
- returns it
- and returns whether the number is positive

in such a way to make this code work:

Code:

```
program looppos
  implicit none
  real(4) :: userinput
  do while (pos_input(userinput))
     print &
      '("Positive input:",f7.3)',&
      userinput
  end do
  print &
      '("Negative input:",f7.3)',&
      userinput
  /* ... */
end program looppos
```

Output:

```
Running with the following inputs:

5
1
-7.3

Positive input:  5.000
Positive input:  1.000
Negative input: -7.300
```

Hint: is `pos_input` a `SUBROUTINE` or `FUNCTION`? If the latter, what is the type of the function result? How many parameters does it have otherwise? Where does the variable `user_input` get its value? So what is the type of the parameter(s) of the function?

36.5 Object oriented exercises

Exercise 36.2. Why is it a good idea to use an accessor function for the data members of a class, rather than declaring data members `public` and accessing them directly?

Exercise 36.3. You are programming a video game. There are moving elements, and you want to have an object for each. Moving elements need to have a method `move` with an argument that indicates a time duration, and this method updates the position of the element, using the speed of that object and the duration.
Supply the missing bits of code.

```
class position {
  /* ... */
public:
  position() {};
  position(int initial) { /* ... */ };
  void move(int distance) { /* ... */ };
};
class actor {
protected:
  int speed;
  position current;

public:
  actor() { current = position(0); };
  void move(int duration) {
    /* THIS IS THE EXERCISE: */
```

```
    /* write the body of this function */
  };
};
class human : public actor {
public:
  human() // EXERCISE: write the constructor
};
class airplane : public actor {
public:
  airplane() // EXERCISE: write the constructor
};

int main() {
  human Alice;
  airplane Seven47;
  Alice.move( 5 );
  Seven47.move( 5 );
```

Exercise 36.4. Let a `Point` class be given:

```
class Point {
private:
  double x,y;
public:
  Point( double px,double py ) { x = px; y = py; };
  // maybe some more methods
}
```

How would you design a `Set` class so that you could write

```
Point p1,p2,p3;
Set pointset;
pointset.add(p1); pointset.add(p2);
```

Chapter 37

Not-so simple exercises

37.1 List access

Exercise 37.1. Explore the efficiency of using an array versus a linked list.

1. Compare re-allocating the array versus adding elements to the linked list. Start with a simple case: add elements only at the end, and keep a pointer to the final element in the list.
2. Investigate access times: retrieve many (as in: thousands if not more) elements from the array. Do this as follows: allocate an array of indexes, and repeatedly retrieve those list/array elements, for instance adding them together. Does the access time for the array go up if the number of elements gets large?
3. Optimize allocation for the list: create an array of list nodes and use those. Does this make a difference in access times?

Chapter 38

Prime numbers

38.1 Arithmetic

Before doing this section, make sure you study section 4.5.

Exercise 38.1. Read two integers into two variables, and print their sum, product, quotient, modulus.

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

Exercise 38.2. 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.

38.2 Conditionals

Before doing this section, make sure you study section 5.1.

Exercise 38.3. 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.

38.3 Looping

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

Exercise 38.4. 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 38.5. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

38.4 Functions

Before doing this section, make sure you study section 7.

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

Exercise 38.6. Write a function `is_prime` that takes an integer input, and returns a boolean corresponding to whether the input was prime.

```
int main() {
  bool isprime;
  isprime = prime_test_function(13);
```

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

38.5 While loops

Before doing this section, make sure you study section 6.2.

Exercise 38.7. Take your prime number testing function `is_prime`, and use it to write program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

38.6 Structures

Before doing this section, make sure you study section 9.1, 14.1.

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

Exercise 38.8. 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:

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

38.7 Classes and objects

Before doing this section, make sure you study section 10.1, 31.1.

In exercise 38.8 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 38.9. Write a class `primegenerator` that contains the members of the structure, and the functions `nextprime`, `isprime`. The function `nextprime` does not need the object as argument, because the members are in the object, and therefore global to that function.
Your main program should look as follows:

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

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

```
primegenerator sequence;
```

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

Exercise 38.10. 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 e. In each iteration, make a `primegenerator` object to generate p values. For each p test whether $e-p$ is prime.
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.

Exercise 38.11. The *Goldbach conjecture* says that every even number $2n$ (starting at 4), is the sum of two primes $p+q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes. In particular this holds for each prime number:

$$\forall_{p\,\mathrm{prime}} \exists_{q\,\mathrm{prime}} : r \equiv p + (p - q) \text{ is prime.}$$

Write a program that tests this. You need two prime number generators, one for the p-sequence and one for the q-sequence. For each p value, when the program finds the q value, print the q, p, r triple and move on to the next p.
Allocate an array where you record all the $p-q$ distances that you found. Print some elementary statistics, for instance: what is the average, do the distances increase or decrease with p?

38.8 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 38.12. 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.

38.9 Inheritance and virtual methods

If you turn the Eratosthenes sieve into a generator class, you see that it has a `nextprime` function, just like the other generator class you wrote above.

Exercise 38.13. Use inheritance to express the commonality between the classes:

- Make a `generator` base class; it probably remembers the last prime found;
- Make `bruteforce_generator` and `sieve_generator` classes that derive from `generator`, and that each have their own `nextprime` method.

Since the derived classes have different `nextprime` methods, you can not really put that method in the base class. However:

Exercise 38.14. Add `nextprime` as a virtual method to the base class.

Chapter 39

Infectuous disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

39.1 Model design

It is possible to model disease propagation statistically, but here we will build an explicit simulation: we will maintain an explicit description of all the people in the population, and track for each of them their status.

We will use a simple model where a person can be:

- sick: when they are sick, they can infect other people;
- susceptible: they are healthy, but can be infected;
- recovered: they have been sick, but no longer carry the disease, and can not be infected for a second time;
- inocculated: they are healthy, do not carry the disease, and can not be infected.

In more complicated models a person could be infectious during only part of their illness, or there could be secondary infections with other diseases, et cetera. We keep it simple here: any sick person is infectious.

In the exercises below we will gradually develop a somewhat realistic model of how the disease spreads from an infectious person. We always start with just one person infected. The program will then track the population from day to day, running indefinitely until none of the population is sick. Since there is no re-infection, the run will always end.

39.1.1 Other ways of modeling

Instead of capturing every single person in code, a 'contact network' model, it is possible to use an Ordinary Diffential Equation (ODE) approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [**?**].

http://mathworld.wolfram.com/Kermack-McKendrickModel.html

This is known as a 'compartmental model'. Both the contact network and the compartmental model capture part of the truth. In fact, they can be combined. We can consider a country as a set of cities, where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

39.2 Coding up the basics

Before doing this section, make sure you study section 10.

We start by writing code that models a single person. The main methods serve to infect a person, and to track their state. We need to have some methods for inspecting that state.

The intended output looks something like:

```
On day 10, Joe is susceptible
On day 11, Joe is susceptible
On day 12, Joe is susceptible
On day 13, Joe is susceptible
On day 14, Joe is sick (5 to go)
On day 15, Joe is sick (4 to go)
On day 16, Joe is sick (3 to go)
On day 17, Joe is sick (2 to go)
On day 18, Joe is sick (1 to go)
On day 19, Joe is recovered
```

Exercise 39.1. Write a `Person` class with methods:

- `status_string()` : returns a description of the person's state;
- `update()` : update the person's status to the next day;
- `infect(n)` : infect a person, with the disease to run for n days;
- `is_stable()` : report whether the person has been sick and is recovered.

The main program should execute these statements in a loop:

```
Person joe;
/* ... */
  joe.update();
  float bad_luck = (float) rand()/(float)RAND_MAX;
  if (bad_luck>.95)
    joe.infect(5);

  cout << "On day " << step << ", Joe is "
       << joe.status_string() << endl;
  if (joe.is_stable())
    break;
```

Your main program could for instance look like:

```
Person joe;
/* ... */
  joe.update();
  float bad_luck = (float) rand()/(float)RAND_MAX;
  if (bad_luck>.95)
    joe.infect(5);

  cout << "On day " << step << ", Joe is "
       << joe.status_string() << endl;
```

```
if (joe.is_stable())
  break;
```

Here is a suggestion how you can model disease status. Use a single integer with the following interpretation:

- healthy but not inoculated, value 0,
- recovered, value -1,
- inocculated, value -2,
- and sick, with n days to go before recovery, modeled by value n.

The `Person::update` method then updates this integer.

39.3 Population

Before doing this section, make sure you study section 11.

Next we need a `Population` class. Implement a population as a `vector` consisting of `Person` objects. Initially we only infect one person, and there is no transmission of the disease.

The trace output should look something like:

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

Exercise 39.2. Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:
  ```
  Population population(npeople);
  ```
- Write a method that infects a random person:
  ```
  population.random_infection();
  ```
- Write a method `count_infected` that counts how many people are infected.
- Write an `update` method that updates all persons in the population.
- Loop the `update` method until no people are infected: the `Population::update` method should apply `Person::update` to all person in the population.

Write a routine that displays the state of the popular, using for instance: `?` for susceptible, `+` for infected, `-` for recovered.

39.4 Contagion

This past exercise was too simplistic: the original patient zero was the only one who ever got sick. Now we incorporate contagion, seeing how far the disease can spread from a single infected person.

We start with a very simple model of infection.

Exercise 39.3. Read in a number $0 \leq p \leq 1$ representing the probability of disease transmission upon contact. Incorporate this into the program: in each step the direct neighbours of an infected person get sick themselves.

```
population.set_probability_of_transfer(probability);
```

Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

Exercise 39.4. Incorporate inoculation: read another number representing the percentage of people that has been vaccinated. Choose those members of the population randomly. Describe the effect of vaccinated people on the spread of the disease. Why is this model unrealistic?

39.5 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model*; https://en.wikipedia.org/wiki/Epidemic_model.

Set the number of people that a person comes into contact with, per day, to 6 or so. You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

Exercise 39.5. Code the random interactions. Now run a number of simulations varying

- The percentage of people inoculated, and
- the chance the disease is transmitted on contact.

Record how long the disease runs through the population. With a fixed number of contacts and probability of transmission, how is this number of function of the percentage that is vaccinated?
Investigate the matter of 'herd immunity': if enough people are vaccinated, then some people who are not vaccinated will still never get sick. Let's say you want to have this probability over 95 percent. Investigate the percentage of inoculation that is needed for this as a function of the contagiousness of the disease.

39.6 Project writeup and submission

39.6.1 Program files

In the course of this project you have written more than one main program, but some code is shared between the multiple programs. Organize your code with one file for each main program, and a single 'library' file with the class methods. This requires you to use *separate compilation* for building the program, and you need a *header* file; section 17.1.1.

Submit all source files with instructions on how to build all the main programs. You can put these instructions in a file with a descriptive name such as `README` or `INSTALL`, or you can use a *makefile*.

39.6.2 Writeup

In the writeup, describe the 'experiments' you have performed and the conclusions you draw from them. The exercises above give you a number of questions to address.

For each main program, include some sample output, but note that this is no substitute for writing out your conclusions in full sentences.

The last exercise asks you to explore the program behaviour as a function of one or more parameters. Include a table to report on the behaviour you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

Chapter 40

Geometry

In this set of exercises you will write a small 'geometry' package: code that manipulates points, lines, shapes. These exercises mostly use the material of section 10.

40.1 Basic functions

Exercise 40.1. Write a function with (float or double) inputs x, y that returns the distance of point (x, y) to the origin.
Test the following pairs: $1, 0, 0, 1, 1, 1, 3, 4$.

Exercise 40.2. Write a function with inputs x, y, α that alters x and y corresponding to rotating the point (x, y) over an angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

40.2 Point class

Before doing this section, make sure you study section 10.1.

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 40.3. Make class `Point` with a constructor

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

Write the following methods:

- `distance_to_origin` returns a `float`.
- `printout` uses `cout` to display the point.
- `distance` computes the distance between this point and another: if `p,q` are `Point` objects,

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

  computes the distance.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 40.4. Make a default constructor for the point class:

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

which you can use as:

```
Point p;
```

Exercise 40.5. Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 11.2. Can you make a constructor where you do not specify the space dimension explicitly?

40.3 Using one class in another

Before doing this section, make sure you study section 10.2.

Exercise 40.6. Make a class `LinearFunction` with a constructor:

```
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 40.7. 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;
```

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.

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 40.8. 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.

40.4 Is-a relationship

Before doing this section, make sure you study section 10.3.

Exercise 40.9. 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 40.10. 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 41

PageRank

41.1 Basic ideas

Assuming you have learned about arrays 11, 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 41.1. Store the sparse matrix representing the web as a

```
vector< vector<bool> > web;
```

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.

Exercise 41.2. Track the probability that a user is on a certain page with a *probability vector*:

```
vector<float> state;
```

The sum of the entries needs to be 1. While the algorithms will mathematically guarantee this, it may be a good idea to test for it every once in a while.

Now model the user clicking on a link by computing a new probability vector:

- if for some page p `state[p]` is positive,
- then for all links q, where `web[p][q]` is true,
- the new state gets a contribution of `state[p]` divided by the number of q-links.

Do you recognize a linear algebra algorithm?

Together this gives an approximation of Google's *PageRank* algorithm.

PART V

ADVANCED TOPICS

Chapter 42

Programming strategies

42.1 A philosophy of programming

Code for the reader, not the writer

Yes, your code will be executed by the computer, but:

- You need to be able to understand your code a month or year from now.
- Someone else may need to understand your code.
- ⇒ make your code readable, not just efficient

High level and low level

- Don't waste time on making your code efficient, until you know that that time will actually pay off.
- Knuth: 'premature optimization is the root of all evil'.
- ⇒ first make your code correct, then worry about efficiency

Abstraction

- Variables, functions, objects, form a new 'language':
  code in the language of the application.
- ⇒ your code should look like it talks about the application, not about memory.
- Levels of abstraction: implementation of a language should not be visible on the use level of that language.

42.2 Programming: top-down versus bottom up

The exercises in chapter 38 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 38.9.

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...

42.2.1 Worked out example

Take a look at exercise 6.6. 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 from 'start' gives a longer sequence report:
  int length=0;
  // compute the sequence from 'start'
  if (length>maximum_length) {
    // Report this sequence as the longest
  }
}
```

5. Refine computing the sequence:

```
// compute the sequence from 'start'
int current=starting;
while (current!=1) {
  // update current value
  length++;
}
```

6. Refine the update of the current value:

```
// update current value
if (current%2==0)
  current /= 2;
else
  current = 3*current+1;
```

42.3 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.

42.4 Documentation

Take a look at Doxygen.

42.5 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 43

Tiniest of introductions to algorithms and data structures

43.1 Data structures

The main data structure you have seen so far is the array. In this section we briefly sketch some more complicated data structures.

43.1.1 Stack

A *stack* is a data structure that is a bit like an array, except that you can only see the last element:

- You can inspect the last element;
- You can remove the last element; and
- You can add a new element that then becomes the last element; the previous last element becomes invisible: it becomes visible again as the last elememt if the new last element is removed.

The actions of adding and removing the last element are known as *push* and *pop* respectively.

Exercise 43.1. Write a class that implements a stack of integers. It should have methods

```
void push(int value);
int pop();
```

43.1.2 Linked lists

Before doing this section, make sure you study section 16.

Arrays are not flexible: you can not insert an element in the middle. Instead:

- Allocate larger array,
- copy data over (with insertion),
- delete old array storage

This is very expensive. (It's what happens in a C++ `vector`; section 11.2.2.)

If you need to do lots of insertions, make a *linked list*. The basic data structure is a `Node`, which contains

1. Information, which can be anything; and
2. A pointer (sometimes called 'link') to the next node. If there is no next node, the pointer will be `NULL`. Every language has its own way of denoting a *null pointer*.

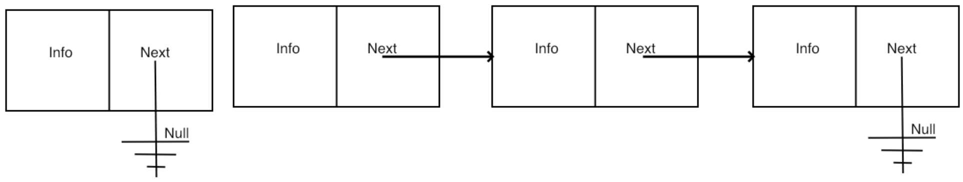


Figure 43.1: Node data structure and linked list of nodes

```
class Node {
private:
  int data{0},count{0};
  Node *next{nullptr};
public:
  Node() {}
  Node(int value) { data = value; count++; };
  bool hasnext() {
    return next!=nullptr; };
```

We illustrate this in figure 43.1.

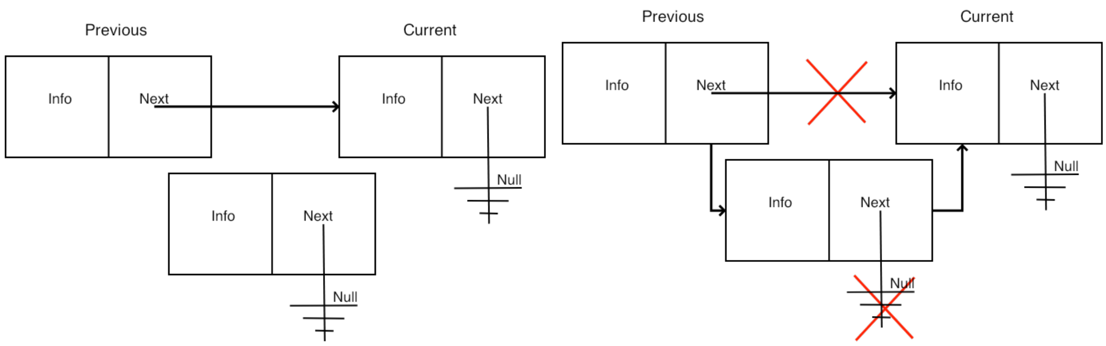


Figure 43.2: Insertion in a linked list

The obvious operations on a linked list are searching for an element, and inserting a new element. See figure 43.2.

```
  Node *insert(int value) {
    if (value==this->data) {
      // we have already seen this value: just count
      count ++;
      return this;
    } else if (value>this->data) {
      // value belong in the tail
      if (!hasnext())
next = new Node(value);
      else
next = next->insert(value);
      return this;
```

```
    } else {
      // insert at the head of the list
      Node *newhead = new Node(value);
      newhead->next = this;
      return newhead;
    }
  };
```

43.1.3 Trees

Before doing this section, make sure you study section 16.

A tree can be defined recursively:

- A tree is empty, or
- a tree is a node with some number of children trees.

Let's design a tree that stores and counts integers: each node has a label, namely an integer, and a count value that records how often we have seen that integer.

Our basic data structure is the node, and we define it recursively to have up to two children. This is a problem: you can not write

```
class Node {
private:
  Node left,right;
}
```

because that would recursively need infinite memory. So instead we use pointers.

```
class Node {
private:
  int key{0},count{0};
  shared_ptr<Node> left,right;
  bool hasleft{false},hasright{false};
public:
  Node() {}
  Node(int i,int init=1 ) { key = i; count = 1; };
  void addleft( int value) {
    left = shared_ptr<Node>( new Node(value) );
    hasleft = true;
  };
  void addright( int value ) {
    right = shared_ptr<Node>( new Node(value) );
    hasright = true;
  };
};
```

and we record that we have seen the integer zero zero times.

Algorithms on a tree are typically recursive. For instance, the total number of nodes is computed from the root. At any given node, the number of nodes of that attached subtree is one plus the number of nodes of the left and right subtrees.

```
int number_of_nodes() {
  int count = 1;
  if (hasleft)
    count += left->number_of_nodes();
  if (hasright)
    count += right->number_of_nodes();
  return count;
};
```

Likewise, the depth of a tree is computed as a recursive max over the left and right subtrees:

```
int depth() {
  int d = 1, dl=0,dr=0;
  if (hasleft)
    dl = left->depth();
  if (hasright)
    dr = right->depth();
  d = max(d+dl,d+dr);
  return d;
};
```

Now we need to consider how actually to insert nodes. We write a function that inserts an item at a node. If the key of that node is the item, we increase the value of the counter. Otherwise we determine whether to add the item in the left or right subtree. If no such subtree exists, we create it; otherwise we descend in the appropriate subtree, and do a recursive insert call.

```
  void insert(int value) {
    if (key==value)
      count ++;
    else if (value<key) {
      if (hasleft)
left->insert(value);
      else
addleft(value);
    } else if (value>key) {
      if (hasright)
right->insert(value);
      else
addright(value);
    } else throw(1); // should not happen
  };
```

43.2 Algorithms

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

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

43.2.1 Sorting

Unlike the tree algorithms above, which used a non-obvious data structure, sorting algorithms are a good example of the combination of very simple data structures (mostly just an array), and sophisticated analysis of the algorithm behaviour. We very briefly discuss two algorithms.

43.2.1.1 Bubble sort

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.

```
void swapij( vector<int> &array,int i ) {
  int t = array[i];
  array[i] = array[i+1];
  array[i+1] = t;
}
```

(Why is the array argument passed by reference?)

If you go through the array once, swapping elements, the result is not sorted, but at least the largest element is at the end. You can now do another pass, putting the next-largest element in place, and so on.

This algorithm is known as *bubble sort*. It is generally not considered a good algorithm, because it has a time complexity (section 44.1.1) of $n^2/2$ swap operations. Sorting can be shown to need $O(n \log n)$ operations, and bubble sort is far above this limit.

43.2.1.2 Quicksort

A popular algorithm that can attain the optimal complexity (but need not; see below) is *quicksort*:

- Find an element, called the pivot, that is approximately equal to the median value.
- Rearrange the array elements to give three sets, consecutively stored: all elements less than, equal, and greater than the pivot respectively.
- Apply the quicksort algorithm to the first and third subarrays.

This algorithm is best programmed recursively, and you can even make a case for its parallel execution: every time you find a pivot you can double the number of active processors.

Exercise 43.2. Suppose that, by bad luck, your pivot turns out to be the smallest array element every time. What is the time complexity of the resulting algorithm?

43.3 Programming techniques

43.3.1 Memoization

In section 7.3 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) {
  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( 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;
}
```

Chapter 44

Complexity

44.1 Order of complexity

44.1.1 Time complexity

Exercise 44.1. 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 44.2. How many operations, as a function of n, are performed in these two solutions?

44.1.2 Space complexity

Exercise 44.3. 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 44.4. How much space do the two solutions require?

Chapter 45

C for C++ programmers

C++ is, to a large extent, a superset of C. But that doesn't mean that you should look at C++ as 'C with some extra mechanisms'. C++ has a number of new mechanisms that offer the same functionality as older C mechanisms, and **that should replace them**.

An entertaining talk that makes this point: `https://www.youtube.com/watch?v=YnWhqhNdYyk`.

Here are some topics.

45.1 I/O

There is little employ for `printf` and `scanf`. Use `cout` (and `cerr`) and `cin` instead.

Chapter 13

45.2 Arrays

Arrays through square bracket notation are unsafe. They are basically a pointer, which means they carry no information beyond the memory location.

It is much better to use `vector`. Use range-based loops, even if you use bracket notation.

Chapter 11

45.3 Strings

No more hassling with *null terminators*. A `string` is an object with operations defined on it.

Chapter 12.

45.4 Pointers

There is very little need for pointers.

- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 11.1.1.
- To pass an argument *by reference*, use a *reference*. Section 7.2.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

Legitimate needs:

- Objects on the heap. Use `shared_ptr` or `unique_ptr`; section 16.3.
- Use `nullptr` as a signal.

45.4.1 Parameter passing

No longer by address: now true references! Section 7.2.

45.5 Objects

Objects are structures with functions attached to them. Chapter 10.

45.5.1 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 18.

45.5.2 Templates

If you find yourself writing the same function for a number of types, you'll love templates. Chapter 20.

45.6 Obscure stuff

45.6.1 Const

Functions and arguments can be declared const. This helps the compiler. Section 23.1.1.

45.6.2 Lvalue and rvalue

Section 23.3

PART VI

INDEX AND SUCH

Chapter 46

Index