

Introduction to Scientific Programming in C++/Fortran2003

Victor Eijkhout

2017

Contents

I	Introduction	9
1	Introduction	11
1.1	<i>Programming and computational thinking</i>	11
1.1.1	History	11
1.1.2	Computational thinking	13
1.1.3	Hardware	14
1.1.4	Algorithms	14
1.2	<i>About the choice of language</i>	15
1.3	<i>Further reading</i>	15
2	Warming up	17
2.1	<i>Programming environment</i>	17
2.2	<i>Compiling</i>	17
II	C++	19
3	Basic elements of C++	21
3.1	<i>Statements</i>	21
3.2	<i>Variables</i>	21
3.3	<i>Input/Output, or I/O as we say</i>	23
3.4	<i>Expressions</i>	24
3.5	<i>Conditionals</i>	25
3.5.1	Further practice	26
4	Looping	29
4.1	<i>Basic ‘for’ statement</i>	29
4.2	<i>Looping until</i>	29
5	Scope, functions, classes	33
5.1	<i>Scope</i>	33
5.2	<i>Functions</i>	34
5.2.1	Parameter passing	36
5.2.2	Recursive functions	38
5.2.3	Polymorphic functions	38
5.2.4	Default arguments	39
5.2.5	Static variables	39
6	Structures	41
6.1	<i>Why structures?</i>	41

6.2	<i>The basics of structures</i>	41
7	Classes and objects	43
7.1	<i>What is an object?</i>	43
7.1.1	<i>Accessors</i>	45
7.1.2	<i>Methods</i>	46
7.2	<i>Relations between classes</i>	46
7.3	<i>Inclusion relations between classes</i>	46
7.4	<i>Inheritance</i>	47
7.4.1	<i>Methods of base and derived classes</i>	48
8	Other	51
8.1	<i>Output your own classes</i>	51
9	Arrays	53
9.1	<i>Traditional arrays</i>	53
9.2	<i>Multi-dimensional arrays</i>	54
9.3	<i>Other allocation mechanisms</i>	54
9.4	<i>Vector class for arrays</i>	56
9.4.1	<i>Vector methods</i>	57
9.4.2	<i>Vectors are dynamic</i>	57
9.4.3	<i>Vector assignment</i>	57
9.4.4	<i>Vectors and functions</i>	58
9.4.5	<i>Dynamic size of vector</i>	58
9.4.6	<i>Timing</i>	59
9.5	<i>Wrapping a vector in an object</i>	60
9.6	<i>Multi-dimensional cases</i>	60
9.6.1	<i>Matrix as vector of vectors</i>	60
9.6.2	<i>Matrix class based on vector</i>	60
9.7	<i>Exercises</i>	61
10	Strings	63
10.1	<i>Basic string stuff</i>	63
10.2	<i>Conversion</i>	64
11	Input/output	65
11.1	<i>Formatted output</i>	65
11.2	<i>Floating point output</i>	67
11.3	<i>Saving and restoring settings</i>	69
11.4	<i>File output</i>	70
11.4.1	<i>Output your own classes</i>	70
11.5	<i>Input streams</i>	71
12	References and addresses	73
12.1	<i>Reference</i>	73
13	Polymorphism	75
13.1	<i>The basic idea</i>	75
14	Memory	77
14.1	<i>Memory and scope</i>	77
15	Pointers	79
15.1	<i>What is a pointer</i>	79
15.2	<i>Pointers and addresses, C style</i>	79

15.2.1	Parameter passing	80
15.2.2	Allocation	81
15.2.3	Memory leaks	82
15.3	<i>Safer pointers in C++</i>	83
16	Prototypes	85
16.1	<i>Prototypes for functions</i>	85
16.1.1	Header files	86
16.1.2	C and C++ headers	87
16.2	<i>Global variables</i>	87
16.3	<i>Prototypes for class methods</i>	88
16.4	<i>Header files and templates</i>	88
16.5	<i>Namespaces and header files</i>	88
17	Preprocessor	89
17.1	<i>Textual substitution</i>	89
17.2	<i>Parametrized macros</i>	90
17.3	<i>Conditionals</i>	90
17.3.1	Check on a value	91
17.3.2	Check for macros	91
18	Templates	93
18.1	<i>Templating over non-types</i>	94
19	Error handling	95
19.1	<i>General discussion</i>	95
19.2	<i>Exception handling</i>	95
20	Standard Template Library	99
20.1	<i>Containers</i>	99
20.1.1	Iterators	99
20.2	<i>Complex numbers</i>	100
20.3	<i>About the 'using' keyword</i>	100
21	Obscure stuff	101
21.1	<i>Casts</i>	101
21.1.1	Static cast	101
21.1.2	Dynamic cast	101
21.2	<i>lvalue vs rvalue</i>	101
21.2.1	Conversion	103
21.2.2	References	103
21.2.3	Rvalue references	103
22	More exercises	105
22.1	<i>cplusplus</i>	105
22.2	<i>world best learning center</i>	105
III	Fortran	107
23	Basics of Fortran	109
23.1	<i>Main program</i>	109
23.1.1	Program structure	109
23.1.2	Statements	109

23.1.3	Comments	110
23.2	Variables	110
23.2.1	Data types	110
23.2.2	Constants	111
23.2.3	Initialization	111
24	Loop constructs	113
24.1	Loop types	113
24.2	Interruptions of the control flow	113
24.3	Implied do-loops	114
25	Scope, subprograms, modules	115
25.1	Scope	115
25.2	Procedures	115
25.2.1	Subroutines and functions	115
25.2.2	Return results	116
25.2.3	Types of procedures	116
25.2.4	Optional arguments	117
25.3	Modules	117
26	Structures, eh, types	119
27	Classes and objects	121
27.1	Classes	121
28	Arrays	127
28.1	Static arrays	127
28.1.1	Multi-dimensional	127
28.1.2	Query the size of an array	128
28.1.3	Allocatable arrays	128
28.2	Arrays to subroutines	128
28.3	Operating on a whole array	129
28.3.1	Arithmetic operations	129
28.3.2	Restricting with <code>where</code>	129
28.4	Array slicing	129
29	Pointers	131
29.1	Basic pointer operations	131
29.2	Example: linked lists	133
30	Input/output	137
30.1	Print to terminal	137
30.1.1	Printing arrays	137
30.1.2	Formats	137
30.2	File I/O	138
31	Array operations	139
31.1	Loops without looping	139
31.1.1	Loops without dependencies	139
31.1.2	Loops with dependencies	140
IV	Exercises and projects	143
32	Simple exercises	145

32.1	<i>Looping exercises</i>	145
32.1.1	<i>Further practice</i>	146
32.2	<i>Object oriented exercises</i>	147
33	Not-so simple exercises	149
33.1	<i>List access</i>	149
34	Prime numbers	151
34.1	<i>Preliminaries</i>	151
34.2	<i>Arithmetic</i>	151
34.3	<i>Conditionals</i>	151
34.4	<i>Looping</i>	152
34.5	<i>Functions</i>	152
34.6	<i>While loops</i>	152
34.7	<i>Global variables: optional</i>	152
34.8	<i>Structures</i>	153
34.9	<i>Classes and objects</i>	153
34.10	<i>Arrays</i>	154
35	Geometry	155
35.1	<i>Point class</i>	155
35.2	<i>Using one class in another</i>	155
35.3	<i>Has-a relation</i>	156
35.4	<i>Is-a relationship</i>	157
36	PageRank	159
36.1	<i>Basic ideas</i>	159
V	Advanced topics	161
37	Tiniest of introductions to algorithms and data structures	163
37.1	<i>Data structures</i>	163
37.1.1	<i>Linked lists</i>	163
37.1.2	<i>Trees</i>	164
37.2	<i>Algorithms</i>	166
37.2.1	<i>Sorting</i>	166
37.3	<i>Programming techniques</i>	167
37.3.1	<i>Memoization</i>	167
38	Programming strategies	169
38.1	<i>Programming: top-down versus bottom up</i>	169
38.1.1	<i>Worked out example</i>	170
38.2	<i>Coding style</i>	171
38.3	<i>Documentation</i>	171
38.4	<i>Testing</i>	171
39	Complexity	173
39.1	<i>Order of complexity</i>	173
39.1.1	<i>Time complexity</i>	173
39.1.2	<i>Space complexity</i>	173

VI	Index and such	175
40	Index	177

PART I

INTRODUCTION

Chapter 1

Introduction

1.1 Programming and computational thinking

1.1.1 History

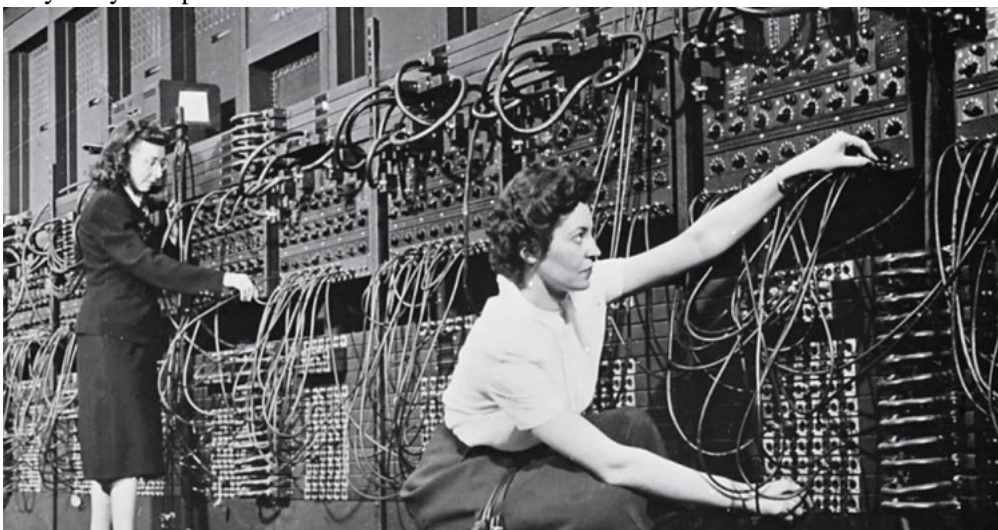
1.1. Earliest computers

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



1.2. Hands-on programming

Very early computers were hardwired



1. Introduction

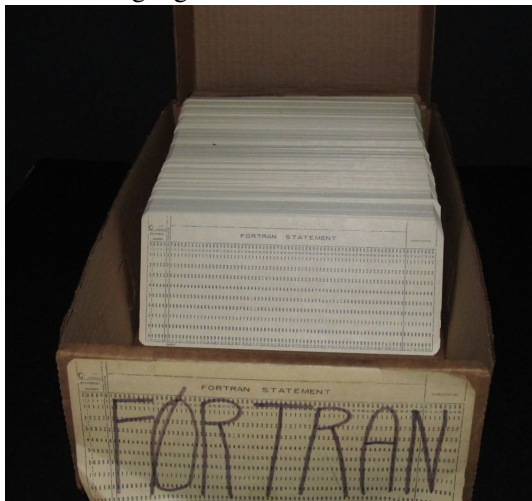
1.3. Program entry

Later programs were written on punchcards



1.4. The first programming language

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



1.5. Programming is everywhere

Programming is used in many different ways these days.

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

This course is aimed at people in the last category.

1.1.2 Computational thinking

1.6. Programming is not simple

Programs can get pretty big



It's not just translating formulas anymore.

Translating ideas to computer code: computational thinking.

1.7. Examples of computational thinking

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

1.8. Abstraction

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

1.9. Data abstraction

What is the structure of the data in your program?

Stack: you can only get at the top item



Queue:

items get added in the back, processed at the front



1.1.3 Hardware

1.10. Do you have to know much about hardware?

Yes, it's there, but we don't think too much about it in this course.

<https://youtu.be/JEpsKnWZrJ8>

1.1.4 Algorithms

1.11. What is an algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time

[A. Levitin, Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2003]

The instructions are in some language:

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

1.12. Program steps

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

1.13. Program data

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

1.2 About the choice of language

1.14. Comparing two languages

Python vs C++ on bubblesort:

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

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

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

1.15. The right language is not all

Python with quicksort algorithm:

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

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

1.3 Further reading

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

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

Chapter 2

Warming up

2.1 Programming environment

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

- If you have a *Linux* computer, you are all set.
- If you have an *Apple* computer, it is easy to get you going. Install *XQuartz* and a *package manager* such as *homebrew* or *macports*.
- *Microsoft Windows* users can use *putty* but it is probably a better solution to install a virtual environment such as *VMware* (<http://www.vmware.com/>) or *Virtualbox* (<https://www.virtualbox.org/>).

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

2.2 Compiling

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

Let’s say that

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

To compile your program, you then type

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

On TACC machines, use the Intel compiler:

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

2. Warming up

which you can verbalize as ‘invoke g++, with output myprogram, on myprogram.cxx’.

So let’s do an example.

This is a minimal program:

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

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

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

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

Here is a statement that at least produces some output:

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

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

PART II

C++

Chapter 3

Basic elements of C++

3.1 Statements

3.1. Program statements

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

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

and arbitrary:

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

Exercise 3.1. Take the ‘hello world’ program you wrote earlier, and duplicate the hello-line. Compile and run.

Does it make a difference whether you have the two hellos on the same line or on different lines?

Experiment with other changes to the layout of your source. Find at least one change that leads to a compiler error.

3.2. Fixed elements

You see that certain parts of your program are inviolable:

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

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

3.2 Variables

3.3. Variable declarations

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

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

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

3.4. Variable names

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

3.5. Declaration

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

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

Declarations can go pretty much anywhere in your program.

3.6. Datatypes

Variables come in different types;

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

3.7. Floating point constants

- Default: `double`
- Float: `3.14f` or `3.14F`
- Long double: `1.41l` or `1.41L`.

This prevents numerical accidents:

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

converts float to double, maybe introducing random bits.

3.8. Assignment

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

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

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

3.9. Assignments

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

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

3.10. Special forms

Update:

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

Integer add/subtract one:

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

Pre/post increment:

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

3.11. Initialization

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

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

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

3.12. Truth values

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

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

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

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

3.13. Terminal output

You have already seen `cout`:

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

3.14. Terminal input

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

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

However, this function is somewhat tricky.

<http://www.cplusplus.com/forum/articles/6046/>.

3.15. Better terminal input

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

```
#include <iostream>  
#include <sstream>  
using namespace std;  
/* ... */  
std::string saymany;  
int howmany;  
  
cout << "How many times? ";  
getline( cin, saymany );  
stringstream saidmany( saymany );  
saidmany >> howmany;
```

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

Exercise 3.5. Write a program that

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

For more I/O, see chapter 11.

3.4 Expressions

3.16. Arithmetic expressions

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

- Modulus: `%`

3.17. Boolean expressions

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


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

3.18. Conversion and casting

Real to integer: round down:

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

Dangerous:

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

The fraction is executed as integer division. Do:

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

Exercise 3.6. Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

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

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

Exercise 3.7. Write a program that ask for two integer numbers n_1, n_2 .

- Assign the integer ratio n_1/n_2 to a variable.
- Can you use this variable to compute the modulus

$$n_1 \bmod n_2$$

(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 [20.2](#).

3.5 Conditionals

3.19. If-then-else

A *conditional* is a test: ‘if something is true, then do this, otherwise maybe do something else’. The C++ syntax is

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

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

3.20. Complicated conditionals

Chain:

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

Nest:

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

3.21. Switch

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

3.22. Local variables in conditionals

The curly brackets in a conditional allow you to define local variables:

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

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

3.5.1 Further practice

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

1. Write a C program to find maximum between two numbers.
2. Write a C program to find maximum between three numbers.
3. Write a C program to check whether a number is even or odd.
4. Write a C program to check whether a year is leap year or not.

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

Chapter 4

Looping

4.1 Basic ‘for’ statement

4.1. Repeat statement

Sometimes you need to repeat a statement a number of times. That’s where the *loop* comes in. A loop has a counter, called a *loop variable*, which (usually) ranges from a lower bound to an upper bound.

Here is the syntax in the simplest case:

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

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

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

4.2. Loop syntax

- The loop variable can be defined outside the loop:

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

```
for(;;) ...
```

4.3. Nested loops

Traversing a matrix:

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

4.2 Looping until

4.4. Indefinite looping

Sometimes you want to iterate some statements not a predetermined number of times, but until a certain condition is met. There are two ways to do this.

First of all, you can use a 'for' loop and leave the upperbound unspecified:

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

4.5. Break out of a loop

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

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

4.6. Skip iteration

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

Alternative:

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

4.7. While loop

The other possibility is a `while` loop, which repeats until a condition is met.

Syntax:

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

or

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

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

4.8. While syntax 1

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

Problem: code duplication.

4.9. While syntax 2

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

More elegant.

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

Chapter 5

Scope, functions, classes

5.1 Scope

Global variables, local variables.

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

Here is a program that uses a global variable:

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

5.2 Functions

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

Any program you can write with functions you can also write without. It will be just as fast, but maybe harder to read, it may be longer, and harder to debug.

5.1. Turn blocks of code into functions

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

5.2. Function definition and call

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

void report_evenness(int n) {
    cout << i;
    if (i%2==0)
        cout << " is even";
    else
        cout << " is odd";
    cout << endl;
}

...
int main() {
    ...
    for (int i=0; i<N; i++)
        report_evenness(i);
}
```

5.3. Why functions?

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

5.4. Prime function

Using a function, primality testing would look like:

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

5.5. Anatomy of a function definition

- Result type: what's computed. `void` if no result
- Name: make it descriptive.
- Arguments: zero or more.
`int i, double x, double y`
- Body: any length.
- Return statement: usually at the end, but can be anywhere; the computed result.

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

- *function result type*: you need to indicate the type of the result;
- *name*: you get to make this up;
- *zero or more function parameters or function arguments*: the data that the function operates on; and the
- *function body*: the statements that make up the function; and
- a *return* statement. Which doesn't have to be last, by the way.

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

5.6. Program with function

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

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

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

5.7. Function call

The function call

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

5.8. Functions without input, without return result

```
void print_header() {
    cout << "*****" << endl;
    cout << "* Output      *" << endl;
    cout << "*****" << endl;
}

int main() {
```

```
print_header();
cout << "The results for day 25:" << endl;
// code that prints results ....
return 0;
}
```

5.9. Functions with input

```
void print_header(int day) {
    cout << "*****" << endl;
    cout << "* Ouput      *" << endl;
    cout << "*****" << endl;
    cout << "The results for day " << day << ":" << endl;
}

int main() {
    print_header(25);
    // code that prints results ....
    return 0;
}
```

5.10. Functions with return result

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

5.11. Scope

Function body is a *scope*: local variables.

No local functions.

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

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

5.2.1 Parameter passing

5.12. Mathematical type function

Pretty good design:

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

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

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

This is equivalent to finding the zero of

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

Newton's method does this by evaluating

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

until the guess is accurate enough.

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

5.13. Results other than through return

Also good design:

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

5.14. Pass by reference example

```
bool can_read_value( int &value ) {
    int file_status = try_open_file();
    if (file_status==0)
        value = read_value_from_file();
    return file_status!=0;
}

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

Exercise 5.2. Write a function `swap` of two parameters that exchanges the input values:

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

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

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

5.2.2 Recursive functions

5.15. Recursion

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

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

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

Exercise 5.4. The sum of squares:

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

can be defined recursively as

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

Write a recursive function that implements this second definition. Test it on numbers that are input by the user.

Then write a program that prints the first 100 sums of squares.

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

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

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 [37.3.1](#).

5.2.3 Polymorphic functions

5.16. Polymorphic functions

You can have multiple functions with the same name:

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

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

5.2.4 Default arguments

5.17. Default arguments

Functions can have *default argument(s)*:

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

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

5.2.5 Static variables

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

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

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

For example

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

gives a stream of integers.

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

Chapter 6

Structures

6.1 Why structures?

6.1. Bundling information

Sometimes a number of variables belong logically together. For instance two doubles can be the x, y components of a vector.

This can be captured in the `struct` construct.

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

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

Initialize:

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

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

6.2 The basics of structures

6.2. Using structures

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

```
struct vector p1,p2;

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

p2 = p1;
```

6.3. Functions on structures

You can pass a structure to a function:

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

6.4. Functions on structures'

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 );  
}
```

6.5. Returning structures

You can return a structure from a function:

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

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

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

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

Chapter 7

Classes and objects

7.1 What is an object?

7.1. *Classes look a bit like objects*

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

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

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

7.2. *Class initialization and use*

Use a *constructor*:

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

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

7.3. *Member initialization*

Other syntax for initialization:

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

7.4. Private data

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

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

7.5. Functions on objects

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

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

We call such internal functions ‘methods’

7.6. Methods that alter the object

```
class Vector {
    /* ... */
    void scaleby( double a ) {
        vx *= a; vy *= a; };
    /* ... */
};

/* ... */
Vector p1(1.,2.);
cout << "p1 has length " << p1.length() << endl;
p1.scaleby(2.);
cout << "p1 has length " << p1.length() << endl;
```

7.7. Methods that create a new object

```

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

```

7.8. Constructor

```

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

```

gives:

```

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

```

So:

```

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

```

7.1.1 Accessors

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`.

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.

7.1.2 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 7.4.1).

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*.

7.2 Relations between classes

7.3 Inclusion relations between classes

7.9. Has-a relationship

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

```

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

```

This is called the *has-a relation*.

At this time, do exercises in section 35.2.

7.10. Literal and figurative has-a

Compare:

```

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

```

Versus:

```

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

```

Implementation vs API.

7.11. Polymorphism in constructors

You have to decide what to store and what to derive, but you can construct two ways:

```

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

```

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

At this time, do the exercises in section 35.3

7.4 Inheritance

7.12. General case, special case

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

You declare that as

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

7.13. Inheritance: derived classes

Derived class *Special* *inherits* methods and data from *base class*

General:

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

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

7.14. Constructors

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

By default the ‘default constructor’, but:

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

7.15. More

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

7.4.1 Methods of base and derived classes

7.16. 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 { ... };  
};
```

7.17. Abstract classes

Special syntax for *abstract method*:

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

- The base class declares that every derived class has to define `f`
- The base class itself does not define this method: *abstract class*
- You can not make objects of the base class.

Chapter 8

Other

8.1 Output your own classes

See section [11.4.1](#).

Chapter 9

Arrays

9.1 Traditional arrays

Static allocation, initialization.

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

Disadvantage: no test on bounds.

Exercise 9.1. Check whether an array is sorted.

Exercise 9.2. Find the maximum element in an array.

Also report at what index the maximum occurs.

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

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

Exercise 9.3. Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

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

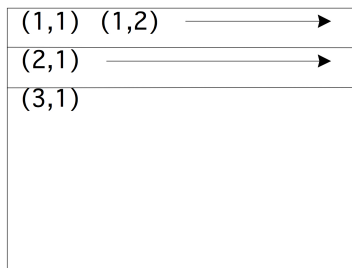
9.2 Multi-dimensional arrays

Declaration, pass to subprogram.

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

int array[5][6];
array[1][2] = 3;
print12(array);
```

C/C++ row major



Physical:

(1,1) (1,2) ... (2,1) ... (3,1)

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

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

int array[5][6];
array[1][0] = 35;
print06(array);
```

9.3 Other allocation mechanisms

A declaration

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

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

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

```

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

```

- Can not be returned from subprogram:

```

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

```

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

```

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

```

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

```

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

```

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

Deprecated use of `malloc`.

9.4 Vector class for arrays

Syntax:

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

vector<type> name(size)
```

where

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

A vector behaves like an array:

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

Initialization:

```
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.)

There is a second way of accessing elements:

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

Safer, but also slower; see below.

Method `size` gives the size of the vector:

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

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

9.4.1 Vector methods

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

9.4.2 Vectors are dynamic

Use `push_back` to add elements at end.

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

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

9.4.3 Vector assignment

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

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

The following mechanism works:

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

You could read this as

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

However, technically, it actually does the following:

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

9.4.4 Vectors and functions

9.4.4.1 Vector as function return

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

```
vector<int> make_array(int n) {
    vector<int> x(n);
    x[0] = n;
    return x;
}

/* ... */
x1 = make_array(10);
```

9.4.4.2 Pass vector to function

You can pass a vector to a function:

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

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

```
void set0( vector<float> &v, float x ) {
    v[0] = x;
}

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

This means you have to pass by reference:

Exercise 9.4. Write functions `random_vector` and `sort` to make the following main program work:

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

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

9.4.5 Dynamic size of vector

Writing

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

creates a vector of size zero. You can then

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

to add elements to the vector, dynamically resizing it.

9.4.6 Timing

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

You can push elements into a vector:

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

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

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

or with subscript:

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

You can also use `new` to allocate:

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

Timings are partly predictable, partly surprising:

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

The increased time for `new` is a mystery.

9.5 Wrapping a vector in an object

You may want to 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);
};
```

9.6 Multi-dimensional cases

9.6.1 Matrix as vector of vectors

Multi-dimensional is harder with vectors:

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

This is not contiguous.

9.6.2 Matrix class based on vector

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

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

```
// array/matrixclass.cxx
class matrix {
private:
    std::vector<double> the_matrix;
    int m,n;
```

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

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

Exercise 9.6. Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

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

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$
$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} & \end{cases}$$

- 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 $n \times n$. When you have that code working, optimize your code to use precisely enough space for the coefficients.

Exercise 9.8. 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 9.9. Put eight queens on a chessboard so that none threatens any other.

Exercise 9.10. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

Chapter 10

Strings

10.1 Basic string stuff

10.1. String declaration

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

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

10.2. String creation

A *string* variable contains a string of characters.

```
string txt;
```

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

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

10.3. Concatenation

Strings can be *concatenated*:

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

10.4. String is like vector

You can query the *size*:

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

or use subscripts:

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

10.5. 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 10.1. Write a function to print out the digits of a number: 156 should print `one five six`. Use a vector of strings.

Hint: it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

Exercise 10.2. Write a function to convert an integer to a string: the input 205 should give `two hundred fifteen, et cetera`.

Exercise 10.3. 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.

10.2 Conversion

`to_string`

Chapter 11

Input/output

11.1 Formatted output

11.1.1. Default output

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

```
cout << "Unformatted:" << endl;
for (int i=1; i<2000000000; i*=10)
    cout << "Number: " << i << endl;
cout << endl;
```

11.2. Output

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

11.3. Reserve space

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

```
cout << "Width is 6:" << endl;
for (int i=1; i<2000000000; i*=10)
    cout << "Number: " << setw(6) << i << endl;
cout << endl;
```

(Only applies to immediately following number)

11.4. Output

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

```
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

11.5. Padding character

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

```
cout << "Padding:" << endl;
for (int i=1; i<2000000000; i*=10)
    cout << "Number: "
        << left << setfill('.') << setw(6) << i << endl;
```

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

Note: many of these output modifiers need

```
#include <iomanip>
```

11.6. Output

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

11.7. Left alignment

Instead of right alignment you can do left:

```
cout << "Padding:" << endl;
for (int i=1; i<2000000000; i*=10)
    cout << "Number: "
        << left << setfill('.') << setw(6) << i << endl;
cout << endl;
```

11.8. Output

```
Padding:
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
```

```

Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

```

11.9. Number base

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

```

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

```

11.10. Output

```

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

```

Exercise 11.1. Make the above output more nicely formatted:

```

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

```

Exercise 11.2. 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.

11.2 Floating point output

11.11. Floating point precision

Use `setprecision` to set the number of digits before and after decimal point:

```

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

```

```
        x *= 10;
    }
```

11.12. Output

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

(Notice the rounding)

11.13. Fixed point precision

Fixed precision applies to fractional part:

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

11.14. Output

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

11.15. Aligned fixed point output

Combine width and precision:

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

```
}
```

11.16. Output

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

11.17. Scientific notation

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

11.18. Output

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

11.3 Saving and restoring settings

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

cout.flags(old_settings);
```

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

cout.precision(old_precision);
```

11.4 File output

11.19. 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();
```

11.20. Binary output

```
ofstream file_out;
file_out.open("fio_binary.out", ios::binary);
```

11.4.1 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 {
        /* ... */
    };
    /* ... */
    std::ostream &operator<<(std::ostream &os, const container &i) {
        os << "Container: " << i.value();
        return os;
    }
};
```

```
};  
/* ... */  
container eye(5);  
cout << eye << endl;
```

11.5 Input streams

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

Chapter 12

References and addresses

12.1 Reference

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

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

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

You can indicate that this is unintended:

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

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

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

There is no change to the calling program. (Some people find this bad, since you can not see from the use of a function whether it passes *by reference* or *by value*.)

Arrays are always pass by reference:

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

The old-style way of doing things:

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


Chapter 13

Polymorphism

13.1 The basic idea

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

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

Chapter 14

Memory

14.1 Memory and scope

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

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

gives:

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


Chapter 15

Pointers

15.1 What is a pointer

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

15.2 Pointers and addresses, C style

15.1. 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:

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

15.2. 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;
```

15.3. 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.

15.4. 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:

- `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.

15.5. 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;
```

15.6. 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!

15.2.1 Parameter passing

15.7. 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;
}
```


15.8. 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 15.1. Write another version of the swap function:

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

15.2.2 Allocation

In section 9.1 you learned how to create arrays that are local to a scope:

15.9. 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 9.3) allows you to allocate storage that transcends its scope:

15.10. Declaration and allocation

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

```
}
```

15.11. De-allocation

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

```
delete(array);
```

15.2.2.1 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:

15.12. Allocation in C

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

15.2.2.2 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.

15.13. 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.

15.2.3 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.

15.3 Safer pointers in C++

Section 15.2.3 showed how how memory can become unreachable. The *C++11* standard has mechanisms that can help solve this problem¹.

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:

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

creates an object and overwrites the pointer, causing the object to be destroyed:

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

Illustrating how the object is not destroyed until all references are gone:

```
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;
```

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

gives:

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

Chapter 16

Prototypes

16.1 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:

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

and the main program in another:

```
// 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.

16.1.1 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 definitions file would include this:

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

and so does the main program

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

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

Having a header file is an important safety measure:

- Suppose you change your function definition, 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.

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 ) { ..... }
```

16.1.2 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>
```

16.2 Global variables

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

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

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

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

```
extern int processnumber;
```

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

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

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

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

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

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

This also prevents recursive inclusion of header files.

16.3 Prototypes for class methods

Header file:

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

Implementation file:

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

16.4 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.

16.5 Namespaces and header files

Never put `using namespace` in a header file.

Chapter 17

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 pre-processing stage takes form of literally inserting another file, here a *header file* into your source.

There are more sophisticated uses of the preprocessor.

17.1 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.

17.2 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 17.1. 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=1; j<=n; j++)
        array[ INDEX2D1BASED(i,j,n) ] = ...
```

17.3 Conditionals

There are a couple of *preprocessor conditions*.

17.3.1 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
```

17.3.2 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
#endif
```

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 [16.2](#).

Chapter 18

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 9 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.

18.1. Templated type name

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

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

18.2. Example: function

Definition:

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

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 18.1. 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;
```

18.1 Templating over non-types

THESE EXAMPLES ARE NOT GOOD.

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

18.3. *Templating a value*

Templeting over integral types, not double.

The templated quantity is a value:

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

Exercise 18.2. Write a class that contains an array. The length of the array should be templated.

Chapter 19

Error handling

19.1 General discussion

Sources of errors:

- Array indexing. See section 9.4.
- Null pointers
- Division by zero and other numerical errors.

Guarding against errors.

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

Error reporting:

- Message
- Total abort
- Exception

Assertions:

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

assertions are omitted with optimization

Function return values

19.2 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 exception; to rethrow the same exception use `throw;` without arguments.
- Exceptions delete all stack data, but not new data.

Chapter 20

Standard Template Library

The C++ language has a *Standard Template Library* (STL), which contains functionality that is considered standard, but that is actually 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 9),
- strings (chapter 10),
- streams (chapter 11).

20.1 Containers

Vectors (section 9.4) and strings (chapter 10) are special cases of a STL *container*. Methods such as `push_back` and `insert` apply to all containers.

20.1.1 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 15.2. 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 ) {  
    ...  
}
```

20.2 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.

20.3 About the ‘using’ keyword

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

Chapter 21

Obscure stuff

21.1 Casts

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 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. In C++ there are two safer cast mechanisms.

21.1.1 Static cast

21.1.2 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
```

21.2 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 assignment, 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.

21.2.1 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
```

21.2.2 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.

21.2.3 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;  
}
```


Chapter 22

More exercises

22.1 cplusplus

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

Dungeon crawl.

22.2 world best learning center

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

PART III

FORTRAN

Chapter 23

Basics of Fortran

23.1 Main 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.

23.1.1 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
```

23.1.2 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.)

23.1.3 Comments

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

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

23.2 Variables

23.2.1 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.

23.1. Floating point types

Indicate number of bytes:

```
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
```

23.2. 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.

23.2.2 Constants

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

```
1.d0
```

23.2.3 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  
  cout << i << endl;  
  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.

Chapter 24

Loop constructs

24.1 Loop types

Indexed do loop:

```
do i=1,10,2
  print *,i
end do
```

While:

```
do
  print *,i
  i = i*2
while (i<1000)
```

F77 note: Do not use label-terminated loops.

24.2 Interruptions of the control flow

You can have a do loop without index or while test. In that case you need the `exit` statement to stop the iteration.

```
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.

24.3 Implied do-loops

Iterate an item:

```
print *, (2*i, i=1, 20)
```

Multiple items:

```
print *, (2*i, 2*i+1, i=1, 20)
```

Nested:

```
print *, ( (i*j, i=1, 20), j=1, 20 )
```

Useful in I/O, especially of arrays.

Chapter 25

Scope, subprograms, modules

25.1 Scope

Fortran ‘has no curly brackets’: you not easily create nested scopes. The range between `do` and `end do` is not a scope.

However, see modules below [25.3](#).

25.2 Procedures

Programs can have procedures: 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:

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < procedure definitions >
End Program foo
```

That is, procedures are placed after the main program statements, separated by a `contains` clause.

25.2.1 Subroutines and functions

Fortran has two types of procedures:

- 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()
```

25.2.2 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 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 25.3 below), it becomes known through a `use` statement.

F77 note: Without modules and `contains` sections, you need to declare the function type explicitly in the calling program.

25.2.3 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.

25.2.4 Optional arguments

25.3 Modules

A module is a container for variables and functions, but it is not a class or object: there is only one instance.

```
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`.

```
Program ModProgram
  use FunctionsAndValues
  implicit none

  print *, "Pi is:", pi
  call SayHi()

End Program ModProgram
```

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.

Chapter 26

Structures, eh, types

26.1. Structures: *type*

The Fortran name for structures is `type` or *derived type*.

26.2. Type definition

Type name / End Type block. Variable declarations inside the block

```
type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

26.3. 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
```

26.4. Member access

Access structure members with %

```
Type(mytype) :: typed_object
typed_object%member = ....
```

26.5. 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
```


Chapter 27

Classes and objects

27.1 Classes

27.1. *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.

27.2. *Object is type with methods*

You define a type as before, with its data members, but now the type has a `contains`.

```
% -*- latex -*-
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% This TeX file is part of the course
%%% Introduction to Scientific Programming in C++/Fortran2003
%%% copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
%%%
%%% printf.tex : arrays in Fortran
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

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.

```
\Level 0 {Basic pointer operations}

\begin{block}{Pointers are aliases}
  \begin{itemize}
    \item Pointer points at an object
    \item Access object through pointer
    \item You can change what object the pointer points at.
  \end{itemize}
\begin{verbatim}
real,pointer :: point_at_real
```

```
\end{verbatim}  
\end{block}
```

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 `\indexterm{pointer}` definition

```
\begin{verbatim}  
real,pointer :: point_at_real  
\end{verbatim}
```

defined a pointer that can point at a real variable.

```
\begin{block}{Setting the pointer}  
  \begin{itemize}  
    \item You have to declare that a variable is pointable:  
    \begin{verbatim}  
real,target :: x  
\end{verbatim}  
    \item Set the pointer with \verb+=>+ notation:  
    \begin{verbatim}  
point_at_real => x  
\end{verbatim}  
    \item Now using \n{point_at_real} is the same as using \n{x}.  
  \end{itemize}  
\end{block}
```

Pointers can not just point at anything: the thing pointed at needs to be declared as `\indexterm{target}`

```
\begin{verbatim}  
real,target :: x  
\end{verbatim}  
and you use the \verb+=>+ operator to let a pointer point at a target  
\begin{verbatim}  
point_at_real => x  
\end{verbatim}
```

If you use a pointer, for instance to print it

```
\begin{verbatim}  
print *,point_at_real  
\end{verbatim}
```

it behaves as if you were using the value of what it points at.

```
\begin{block}{Pointer example}  
  \verbatimsnippet{pointatreal}
```

```

%
\begin{enumerate}
\item The pointer points at~\n{x}, so the value of \n{x} is printed.
\item The pointer is set to point at~\n{y}, so its value is printed.
\item The value of \n{y} is changed, and since the pointer still
      points at~\n{y}, this changed value is printed.
\end{enumerate}
\end{block}

\begin{block}{Assign pointer from other pointer}
\begin{verbatim}
real,pointer :: point_at_real,also_point
point_at_real => x
also_point => point_at_real
\end{verbatim}
      Now you have two pointers that point at~\n{x}.

      \textbf{Very important to use the \n{=>}, otherwise strange
        memory errors}
\end{block}

If you have two pointers
\begin{verbatim}
real,pointer :: point_at_real,also_point
\end{verbatim}
you can make the target of the one to also be the target of the other.
\begin{verbatim}
also_point => point_at_real
\end{verbatim}
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.

\textbf{Using ordinary assignment does not work, and will give strange
memory errors.}

\begin{exercise}
      Write a routine that accepts an array and a pointer, and on return
      has that pointer pointing at the largest array element.
\end{exercise}
\begin{answer}
      \verbatimsnippet{arraypointf}
\end{answer}

\Level 0 {Example: linked lists}

\begin{block}{Linked list}

```

```
\begin{itemize}
\item Linear data structure
\item more flexible for insertion~/ deletion
\item \ldots~but slower in access
\end{itemize}
\end{block}
```

One of the standard examples of using pointers is the `\indexterm{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.

```
\begin{exercise}
    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 s
    of the list.
```

```
    Make this argument precise.
\end{exercise}
```

```
\begin{block}{Linked list datatypes}
    \begin{itemize}
    \item Node: value field, and pointer to next node.
    \item List: pointer to head node.
    \end{itemize}
    \verbatimsnippet{linklistf}
\end{block}
```

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:

```
%
\verbatimsnippet{linklistf}

\begin{block}{List initialization}
    First element becomes the list head:

\verbatimsnippet{listheadf}
```

```
\end{block}
```

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:

```
%  
\verbatimsnippet{listheadf}
```

```
\begin{block}{Attaching a node}  
  Keep the list sorted: new largest element attached at the end.
```

```
  \verbatimsnippet{listattachf}  
\end{block}
```

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 `\n{current}` point at the last node of the list, then here is how to attaching a new node from it:

```
%  
\verbatimsnippet{listattachf}
```

```
\begin{block}{Inserting 1}  
  Find the insertion point:  
\verbatimsnippet{listfindf}  
\end{block}
```

Inserting an element in the list is harder. First of all, you need to find the two nodes, `\n{previous}` and `\n{current}`, between which to insert the new node:

```
%  
\verbatimsnippet{listfindf}
```

```
\begin{block}{Inserting 2}  
  The actual insertion requires rerouting some pointers:  
  %  
  \verbatimsnippet{listinsertf}  
\end{block}
```

27.3. 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
```

27.4. Objects definition, method invocation

Class objects are defined as `type` objects, just as if there were no class functions on them. The class functions are accessed as

```
object%function(arg1, arg2)
```

where the arguments do not include the `class` argument.

```
use PointClass
implicit none
type(Point) :: p1,p2

call p1%set(1.d0,1.d0)
call p2%set(4.d0,5.d0)
```

27.5. Use modules!

It is of course best to put the type definition and method definitions in a module, so that you can use it.

Mark methods as `private` so that they can only be used as part of the `type`:

```
Module PointClass
  private
contains
  subroutine setzero(p)
    implicit none
    class(point) :: p
    p%x = 0.d0 ; p%y = 0.d0
  end subroutine setzero
End Module PointClass
```

Exercise 27.1. Take the point example program and add a distance function.

Chapter 28

Arrays

28.1 Static arrays

The preferred way for specifying an array size is:

```
real(8), dimension(100) :: x,y
```

Such an array, with size explicitly indicated, is called a *static array* or *automatic array*. (See section [28.1.3](#) for dynamic arrays.)

Array indexing in Fortran is 1-based:

```
real :: x(8)
do i=1,8
... x(i) ...
```

Unlike C++, Fortran can specify the lower bound explicitly:

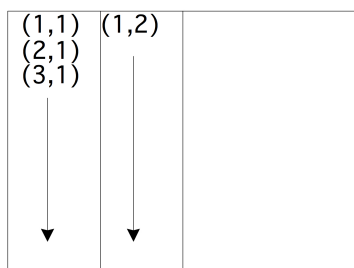
```
real :: x(-1:7)
do i=-1,7
... x(i) ...
```

Such array, as in C++, obey the scope: they disappear at the end of the program or subprogram.

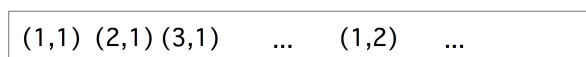
28.1.1 Multi-dimensional

```
integer, dimension(20,30) :: array
```

Fortran column major



Physical:



28.1.2 Query the size of an array

Various query functions:

- Total number of elements (in a specified dimension) can be queried with `size`:

```
integer :: x(8), y(5,4)
size(x)
size(y,2)
```

- Lower and upper bound:

```
Lbound(x)
Ubound(x)
```

28.1.3 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.

```
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=ierr` 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
```

28.2 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:


```
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)
```

28.3 Operating on a whole array

28.3.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D+E
```

(where the multiplication is by element).

28.3.2 Restricting with where

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
  sequence of array statements
ELSEWHERE
  sequence of array statements
END WHERE
```

28.4 Array slicing

Chapter 29

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.

29.1 Basic pointer operations

29.1. 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.

29.2. 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.

29.3. 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.

29.4. 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 29.1. Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element.

29.2 Example: linked lists

29.5. 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 29.2. 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.

29.6. 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)

```

29.7. 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
```

29.8. 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
```

29.9. 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
```

29.10. 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 30

Input/output

30.1 Print to terminal

The simplest command for outputting 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.

30.1.1 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), i=1,n)
```

30.1.2 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:

- *an* specifies a string of *n* characters. If the actual string is longer, it is truncated in the output.
- *in* 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 floating point number, with *m* total positions (including the decimal point) and *n* digits in the fractional part.
- *e_m.n* Exponent representation.

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>

30.2 File I/O

Units. open close

Chapter 31

Array operations

31.1 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
```

31.1.1 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
```

31. Array operations

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 31.1. Create arrays A , C of length $2N$, and B of length N . Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \dots, N$$

and

$$C_i = A_{i/2}, \quad i = 1, \dots, 2N$$

using all four mechanisms. Make sure you get the same result every time.

31.1.2 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
Recursive	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.

PART IV

EXERCISES AND PROJECTS

Chapter 32

Simple exercises

32.1 Looping exercises

Exercise 32.1. 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 32.2. The integer sequence

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

leads to the Collatz conjecture: no matter the starting guess u_1 , the sequence $n \mapsto u_n$ will always terminate.

For $u_1 < 1000$ find the values that lead to the longest sequence: every time you find a sequence that is longer than the previous maximum, print out the starting number.

Exercise 32.3. 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 32.4. 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

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

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 $|x_- - x_+| < 10^{-10}$.

32.1.1 Further practice

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

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

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

32.2 Object oriented exercises

Exercise 32.5. Use the `Point` class from exercise ... and make a `Set` class, where each `Set` object contains multiple points.

```
class Point {
private:
    float x,y;
public:
    Point(float ux,float uy) { x = ux; y = uy; };
    // other methods omitted
};

int main() {
    Set set;

    set.add( Point(1.0,1.0) );
    set.add( Point(2.0,2.0) );
    cout << "Number of points: " << set.cardinality() << endl;

    Point p3(4.0,5.0);
    set.add( p3 );
    cout << "Number of points: " << set.cardinality() << endl;

    return 0;
}
```

This program prints out first 2, then 3, as the number of points in the set.

Chapter 33

Not-so simple exercises

33.1 List access

Exercise 33.1. 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 34

Prime numbers

34.1 Preliminaries

Assuming you have learned about

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

34.2 Arithmetic

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

Exercise 34.1. Read two integers into two variables, and print their sum, product, quotient, modulus.

A less common operator is the modulo operator %.

Exercise 34.2. 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.

34.3 Conditionals

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

Exercise 34.3. 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.

34.4 Looping

Control structures such as loops; section 4.1.

Exercise 34.4. 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 34.5. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

34.5 Functions

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

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

Exercise 34.6. Write a function that takes an integer input, and return a boolean corresponding to whether the input was prime.

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

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

34.6 While loops

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

Exercise 34.7. Take the prime number testing program, and modify it to read in how many prime numbers you want to print. Print that many successive primes. Keep a variable `number_of_primes_found` that is increased whenever a new prime is found.

34.7 Global variables: optional

Before doing this section, make sure you study section 5.1.

Exercise 34.8. Use global variables to rewrite exercise 34.7. Your main program should exactly be this:

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



```

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

    return 0;
}

```

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

34.8 Structures

Before doing this section, make sure you study section 6.1, 12.1.

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

Exercise 34.9. Rewrite the exercise that found a predetermined number of primes, putting the `number_of_primes_found` and `last_number_tested` variables in a structure. Your main program should now look like:

```

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

```

34.9 Classes and objects

Before doing this section, make sure you study section 7.1, 27.1.

In exercise 34.9 you made a structure that contains the data for a primesequence, and you have separate functions that operate on that structure or on its members.

Exercise 34.10. Write a class `primesequence` that contains the members of the structure, and the functions `nextprime`, `isprime`. The function `nextprime` does not need the structure as argument, because the structure members are in the class, and therefore global to that function.

Your main program should look as follows:

```

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

```

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

```
primesequences sequence;
```

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

Exercise 34.11. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p+q$. Write a program to test this for the even numbers up to 20 million. Make an outer loop over the even numbers. In each iteration, make a `primesequences` object to generate p values. Then, for each p , make a second `primesequences` object to generate q values, and test with these. For each even number, print out how it is the sum of two primes. If multiple possibilities exist, only print the first one you find.

Exercise 34.12. 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:

$$\forall_{p \text{ prime}} \exists_{q \text{ prime}} : r \equiv p + (p - q) \text{ is prime.}$$

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 ?

34.10 Arrays

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

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

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

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

Exercise 34.13. Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers.

Chapter 35

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 7.

35.1 Point class

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

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 35.1. Make class `Point` with a constructor

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

Write a method `distance_to_origin` that returns a `float`.

Write a method `printout` that uses `cout` to display the point.

Write a function `distance` so that if `p, q` are `Point` objects,

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

computes the distance.

Exercise 35.2. Make a default constructor for the point class:

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

which you can use as:

```
Point p;
```

Exercise 35.3. Advanced. Can you make a `Point` class that can accomodate any number of space dimensions? Hint: use a `vector`; section 9.4. Can you make a constructor where you do not specify the space dimension explicitly?

35.2 Using one class in another

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

Exercise 35.4. Make a class `LinearFunction` with a constructors:

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

and a function

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

which you can use as:

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

Exercise 35.5. Make a class `LinearFunction` with two constructors:

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

where the first stands for a line through the origin.

Implement again the `evaluate` function so that

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

35.3 Has-a relation

Before doing this section, make sure you study section 7.3.

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.

35.1. Axi-parallel rectangle class

Intended API:

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

It would be convenient to store width and height; for

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

it would be convenient to store bottomleft/topright points.

Exercise 35.6. Make a class `Rectangle` (sides parallel to axes) with two constructors:

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

and functions

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

Let the `Rectangle` object store two `Point` objects.

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

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

35.4 Is-a relationship

Before doing this section, make sure you study section 7.4.

Exercise 35.7. Take your code where a `Rectangle` was defined from one point, width, and height.

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

First ask yourself: what should the constructor of a `Square` look like?

Exercise 35.8. Revisit the `LinearFunction` class. Add methods `slope` and `intercept`. Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

Chapter 36

PageRank

36.1 Basic ideas

Assuming you have learned about arrays [9](#), 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 36.1. Store the sparse matrix representing the web as a

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

structure.

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

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

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

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

PART V

ADVANCED TOPICS

Chapter 37

Tiniest of introductions to algorithms and data structures

37.1 Data structures

This really goes beyond this book.

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

37.1.1 Linked lists

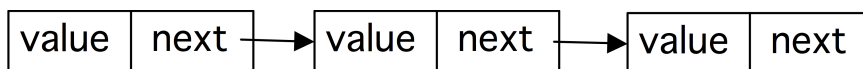
Before doing this section, make sure you study section 15.

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 9.4.2.)

If you need to do lots of insertions, make a *linked list*.



```
class Node {
private:
    int data{0}, count{0};
    Node *next{nullptr};
public:
    Node() {}
    Node(int value) { data = value; count++; };
    bool hasNext() {
        return next!=nullptr; };
};
```

```
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;
    }
};
```

37.1.2 Trees

Before doing this section, make sure you study section 15.

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};
    std::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 = std::shared_ptr<Node>( new Node(value) );
    hasleft = true;
};
void addright( int value ) {
    right = std::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
};
```

37.2 Algorithms

This *really really* goes beyond this book.

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

37.2.1 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.

37.2.1.1 Bubble sort

An array a of length n is sorted if

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

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 swap( std::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 39.1.1) 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.

37.2.1.2 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 37.1. 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?

37.3 Programming techniques

37.3.1 Memoization

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

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

Here is an outline.

```
int fibonacci(int n) {
    std::vector<int> fibo_values(n);
    for (int i=0; i<n; i++)
        fibo_values[i] = 0;
    fibonacci_memoized(fibo_values, n-1);
    return fibo_values[n-1];
}

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

//codesnipet end
```

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

    return 0;
}
```


Chapter 38

Programming strategies

38.1 Programming: top-down versus bottom up

The exercises in chapter 34 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 34.10.

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

38.1.1 Worked out example

Take a look at exercise 32.2. We will solve this in steps.

1. State the problem:

```
// find the longest sequence
```

2. Refine by introducing a loop

```
// find the longest sequence:
```

```
// Try all starting points
```

```
// If it gives a longer sequence report
```

3. Introduce the actual loop:

```
// Try all starting points
```

```
for (int starting=2; starting<1000; starting++)
```

```
// If it gives a longer sequence report
```

4. Record the length:

```
// Try all starting points
```

```
int maximum_length=-1;
```

```
for (int starting=2; starting<1000; starting++) {
```

```
    // If the sequence gives a longer sequence report:
```

```
    int length=0;
```

```
    // compute the sequence
```

```
    if (length>maximum_length) {
```

```
        // Report this sequence as the longest
```

```
    }
```

```
}
```

5. // Try all starting points

```
int maximum_length=-1;
```

```
for (int starting=2; starting<1000; starting++) {
```

```
    // If the sequence gives a longer sequence report:
```

```
    int length=0;
```

```
    // compute the sequence
```

```
    int current=starting;
```

```
    while (current!=1) {
```

```
        // update current value
```

```
        length++;
```

```
    }
```

```
    if (length>maximum_length) {
```

```
        // Report this sequence as the longest
```

```
    }
```

```
}
```

38.2 Coding style

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

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

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

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

38.3 Documentation

Take a look at Doxygen.

38.4 Testing

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

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

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

Chapter 39

Complexity

39.1 Order of complexity

39.1.1 Time complexity

Exercise 39.1. 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 $1 \dots n$. You can have a solution that keeps a running sum, and a solution with an inner loop.

Exercise 39.2. How many operations, as a function of n , are performed in these two solutions?

39.1.2 Space complexity

Exercise 39.3. 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 39.4. How much space do the two solutions require?

PART VI

INDEX AND SUCH

Chapter 40

Index

allocate, 128
Apple, 17
argument
 default, 39
array
 automatic, 127
 static, 127
assignment, 22

bad_alloc, 97
bad_exception, 97
bit_size, 110
bottom-up, 169
break, 30
bubble sort, 61, 166

C preprocessor, see preprocessor
C++, 104
C++11, 56, 83
 range-based iterator, 100
c_sizeof, 110
call
 function, 34
call, 116
calling environment, 73
case sensitive, 22
cast, 101
cin, 23
class
 abstract, 49
 base, 48
 derived, 48
class, 125
close, 138
code
 maintenance, 171
 reuse, 34
column-major, 137
compiler, 17
 and preprocessor, 89
compiling, 17
conditional, 25
constructor, 43
container, 99
contains
 for class functions, 121
 in modules, 117
contains, 115
continuation character, 109
cout, 23

datatype, 22
deallocate, 128
destructor
 at end of scope, 39
do concurrent, 139
do loop
 implicit, 137

emacs, 17
Eratosthenes sieve, 154
exception
 catch, 96
 throwing, 95
executable, 17
exit, 113
expression, 22

false, 23
forall, 139

- Fortran
 - comments, 110
 - source format
 - fixed, 110
 - free, 110
- forward declaration, 85
- function, 34, 116
 - arguments, 35
 - body, 35
 - defines scope, 36
 - parameters, 35
 - result type, 35
- g++, 17
- getline, 24
- GNU, 17
- Goldbach conjecture, 154
- has-a relation, 47
- header file, 86, 89
 - and global variables, 87
 - treatment by preprocessor, 87
- hexadecimal, 79
- homebrew, 17
- icpc, 17
- inheritance, 48
- inline, 117
- is_eof, 71
- is_open, 71
- iso_c_binding, 110
- iterator, 99
- keywords, 21
- label, 138
- linked list, 133
- Linux, 17
- list
 - linked, 163
- loop, 29
- loop variable, 29
- lvalue, 102
- macports, 17
- makefile, 86
- malloc, 56, 82
- matrix
 - sparse, 159
- members, 41
- memoization, 167
- memory leaking, 82
- method
 - abstract, 49
 - overriding, 48
- Microsoft
 - Windows, 17
 - Word, 12
- move semantics, 103
- new, 55, 81
- Newton's method, 36
- object
 - destructor, 77
- open, 138
- overloading, 46
- package manager, 17
- parameter, see function, parameter
 - passing
 - by reference, 73
 - by value, 73
- pass by reference, 37
- pass by value, 36
- pointer
 - arithmetic, 80
- pointer, 131
- preprocessor
 - and header files, 87
 - conditionals, 90–91
 - macro
 - parametrized, 90
 - macros, 89–90
- print, 137
- procedures
 - internal, 116
 - module, 116
- program
 - statements, 18
- prototype, 85
- putty, 17
- quicksort, 167
- recurrence, 140

- recursion, 38
- return, 35
- return, 116
- return status, 37
- root finding, 145
- rvalue, 102
 - reference, 104
- scope
 - dynamic, 39
 - lexical, 39
 - of function body, 36
 - out of, 77
- size, 128
- sizeof, 82
- smartphone, 12
- source code, 17
- stack, 54, 128
 - overflow, 128
- Standard Template Library, 99
- statement functions, 117
- storage_size, 110
- string, 63
 - concatenation, 63
 - size, 63
- struct, 41
- subprogram, *see* function
- swap, 103
- target, 131
- templates
 - and separate compilation, 88
- test-driven development, 171
- testing, 171
- top-down, 169
- true, 23
- type
 - derived, 119
- type, 119
- unit testing, 171
- Unix, 17
- use, 117
- values
 - boolean, 23
- variable, 22
 - assignment, 22
 - declaration, 22, 22
 - global
 - in header file, 87
 - initialization, 23
 - numerical, 22
 - static, 39
- variables
 - global, 33
- vector, 163
- vi, 17
- Virtualbox, 17
- Visual Studio, 17
- VMware, 17
- void, 35
- while, 30
- Xcode, 17
- XQuartz, 17