C++ Summary notes and exercises

June 4, 2012

These notes provide a guideline for the practical sessions based on the book "C++ Primer, Fourth Edition By Stanley B. Lippman, Josée Lajoie, Barbara E. Moo Addison Wesley Professional". Each chapter should be read in parallel with the practical session. Some technical parts which are less likely to be relevant for the numerical applications we are usually interested in physics will be omitted (or only briefly commented). In each section we will provide a summary of the main points to keep together with exercises. Each exercise will be provided in a zip file with a README file explaining what you have to do and the source code solution that I produce.

In addition to this book, I will also extract some examples and exercises from "C++ by Dissection" by Ira Pohl.

Contents

1 Getting started

This chapter of the book is supposed to: i) Explain the basic files structure of a basic C++ program and explain how to compile simple programs; ii) Give you a flavour of some of the basic language "words" so that you can write very simple programs, and an idea about the data structures to be learned later on. It is only to get you started and everything will be discussed in more detail later.

1.1 Steps to create and compile a simple C++ program

To create an executable program you need to follow the steps:

1. **Create a source code file** (that's just a text file), for example named `program1.cpp` (.cpp means C++ source file). In that file you must have **at least one function which is called** `main` and it is the function that the system calls when you execute the program.

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

   Everything else your program does must be called from within this function directly or indirectly. The body of the function is inside the block delimited by the braces {}. The dots denote your source code. This function returns an integer as indicated in its prototype by the `int` (the first line) and it has no arguments (for now). The last line is the only explicit statement in this schematic example. All statements must end with ";". The value 0 is usually a message of success for the end of the run of the program. In fact, in C++, the last statement is assumed if omitted, so strictly speaking, it does not have to be included.

2. **From the terminal**, go to the directory where your file has been saved and **compile it** by invoking

   ```
   $ g++ program1.cpp -o program1.exe
   ```

   Here, g++ is the compiler which uses the source file `program1.cpp` to create the executable object file (the -o, is a flag which means object file) `program1.exe`

3. Finally you can run yout program by invoking it directly from the command line like this (./ means current directory – this may be system dependent)

   ```
   $ ./program1.exe
   ```

Comment If you need to interrupt your program (because it gets stuck or otherwise...) usually you have to hit Ctrl+c. However this might be system dependent.

1.2 Input and Output

The C++ language does not include any facilities for input and output directly. However, in addition to the language, there is the standard library which extends the language. Basically:

> **The Standard Library** can be thought of as a set of functions and data types that are coded in C++ somewhere in library files in your system. You can use functions or data types in the library by including the relevant files in your source code.

To use the Input/Output facilities in the library you need to **include a line in your source file**:

```
#include<iostream>
int main(){
   ...
```

when you compile your code, the relevant `iostream` files will then be looked for in some default directories.

The basic statements for I/O (input output) are:

- **Input**: This is done through the `cin` object which usually appears in the code as `std::cin`. The :: is the scope operator which says that `cin` is in the namespace `std`. This is a space of names

for the objects in the standard library, to avoid clashes with variables defined by the user. A schematic input statement in a program would be

```
#include<iostream>
int main(){
   std::cin >> var1 >> var2 >> ...>> varN;
   ...
```

where >> is the input operator which reads from the standard input (usually what the user types in the terminal, or a text file if we redirect the input – we will see how to do so) into the variables `var1, var2,..., varN`. We can chain as many of these "reads" from the input, streamed in this way into `std::cin`.
Comment: Some times you may want to send an "end-of-file" signal when entering input, to let the program know you are done with providing input. This may be system dependent, but usually for UNIX it is Ctrl+d and for Windows Ctrl+z.

- **Output:** Similarly to the input stream there are 3 different output streams, which ordinarily are all streamed into the terminal, but can be streamed to data files. These are `std::cout`, `std::clog` and `std::cerr`. You can think of each of these streams' purpose to output data, log information about the run of the program for the user and error information for the user respectively. A schematic input statement can be illustrated with `std::cout`:

  ```
  #include<iostream>
  int main(){
     std::cout << espression1 << expression2 << ...<< expresionN << std::endl;
     ...
  ```

  where each expression will be printed in sequence. Similarly to the input operator, now we have the output operator << which feeds into the stream the expressions to print out. In addition, there is `std::endl` which is the end line manipulator. It's effect is to print out the stream imediately for the user to see (it flushed the stream). If not present it may be delayed and printed later! It is important to use this when debugging code to keep track exactly where at the point of the run of the program we are.

1.2.1 Redirection

The input stream can be read directly from an input file (inputfile.txt) by executing your program as

```
$ ./program1.exe < inputfile.txt
```

where < means redirection of the input to be read by the file in front, instead of the terminal. If the program prompts the user to input any data, it will then be automatically read from the text file.

Similarly, one can redirect the output streams to files. For `std::cout` we use the > sign. For `std::cerr` and `std::clog` one uses 2> and both are redirected to the same file, so for example

```
$ ./program1.exe > outfile.txt 2> infofile.txt
```

redirects the output of `std::cout` to outfile.txt and the output of `std::cerr` and `std::clog` to infofile.txt.

Do the Exercise ExampleRedirectIO.zip: Hint: You need to include the header for the math facilities of the standard library `#include<cmath>` so that the $\log(x)$ function is known.

1.3 Comments

Lines of code are commented by placing a double slash at the begining of the line

```
// This is a commented line
```

For blocks of lines one can use

```
/* This is a commented block of lines
...
...
...*/
```

1.4 Control structures

The most basic control structures we are introducing for now allow you to: i) test conditions and ii) perform recurssive tasks.

1.4.1 The `if` statement

This allows you to test a condition or a set of conditions sequentially. The simplest usage is (schematically)

```
if(condition)
    statement;
```

where if the condition is true then the statement is executed. If the code to be executed has several statements then we must use a block in braces:

```
if(condition){
    statement1;
    statement2;
}
```

There is also a more general form where we can include an arbitrary number of elseif conditions to be tested in case of failure and an optional else statement. The most general form is (schematically)

```
if(condition1){
    statement1;
    statement2;
}
else if(condition2){
    statement3;
    ...
}
else if(condition3){
    statement5;
    statement6;
}
else{
    statement7;
    statement8;
}
```

1.4.2 `while` and `for`

The while statement allows for iterative execution of a block of statements while a condition is true. The general form is

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

The for statement is similar except that there is an integer iteration variable of type `int` which keeps being increased. The general form is

```
for(int i=0; condition;i++){
    statements
}
```

The first statement in the argument initializes the iterator, the second is a condition that is tested each time the block statements are repeated and the last one is an incrementation of the iterator. Here we use the ++ operator, which increments by one the integer i.

1.5 Comment on class types

In the next chapter we will go through the built in data types and keywords of the language. C++ provides an extra feature which allows to define classes. A class is a data type that can de defined by the user and it can contain any structure of data. In addition, the operations on the data and how the class type interacts with other class objects is defined in the class. For example it is usual to provide an interface for a class type with `std::cin, std::cout` etc...

Introducing classes at this stage can become confusing. Instead we will skip to the next chapter to learn first the basics of the language.

2 Variables and basic data types

In this chapter the basic data types of the language and the operators they support are presented.

2.1 Primitive built-in types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

Table 2.1. C++: Arithmetic Types

In addititon to these types there is the `void type` which is used for example as the type returned by a function which acts on some variables but does not return any value. Note that the size (in bits) may depend on your machine achitecture. For example, in 64 bits machines usually a `double` holds more digits.

The types `int, bool, char` (and the corresponding long or short versions) are known as integral types and they hold integers, boolean (`true or false`) and characters (each character variable holds a number corresponding to a character).

Types can be signed or unsigned. This is achieved by adding `unsigned` before the corresponding type.

Note: You should be aware of the limits of the representation of the types above. If you enter a value out of range (for example an integer that has too many digits), you will get a value that is completely different. When coding you should be careful to make sure this does not happen otherwise you will most definitely get wrong results and the code may still compile and run!

Regarding floating point number, the type `float` is usually not precise enough so it is standard to use `double` for most numerical applications (though the longer version may be necessary in some situations).

2.1.1 Arithmetic and logical operators

Though in the book this is only introduced later on. It makes more sense to introduce them right now so that we can do some examples.

Operator	Function	Use
+	unary plus	+ expr
-	unary minus	- expr
*	multiplication	expr * expr
/	division	expr / expr
%	remainder	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

Table 5.1. Arithmetic Operators

Each of these operators yields bool		
Operator	Function	Use
!	logical NOT	!expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equality	expr == expr
!=	inequality	expr != expr
&&	logical AND	expr && expr
||	logical OR	expr || expr

Table 5.2. Relational and Logical Operators

The action of each of the arithmetic operators depends on the type to which it is applied. For example division of integers yields the integer part of the result.

2.2 Literal constants

Literal integers and floating point numbers are entered in the usual way without or with a dot ".". In addition there are ways of specifying if integers are signed, unsigned, float or long:

```
128u     /* unsigned   */          1024UL    /* unsigned long   */
1L       /* long    */             8Lu        /* unsigned long   */
```

Other examples including scientific notation are

```
3.14159F            .001f          12.345L            0.
3.14159E0f          1E-3F          1.2345E1L          0e0
```

Printable characters literal expressions are written in several forms:

1. In single quotes with the character inside

   ```
   'a'          '2'          ','          ' ' // blank
   ```

2. Through an escape sequence which starts with a slash \. One can have special escape sequences for special characters such as:

newline	\n	horizontal tab	\t
vertical tab	\v	backspace	\b
carriage return	\r	formfeed	\f
alert (bell)	\a	backslash	\\
question mark	\?	single quote	\'
double quote	\"		

Or a generalised escape sequence for any character in the character set, which consistes of a slash with the number of the corresponding character

```
\7 (bell)      \12 (newline)     \40 (blank)
\0 (null)      \062 ('2')        \115 ('M')
```

For example for output (or later on when we introduce strings), it is useful to note how to write a literal expression for an array of characters, i.e. a string. String literals are represented in double quotes "". Some examples are

```
"Hello World!"                // simple string literal
""                            // empty string literal
"\nCC\toptions\tfile.[cC]\n"  // string literal using newlines and tabs
```

All string literals have in addition a null character at the end of the string.

Long forms, both for characters and strings, are specified by adding L before the corresponding literal. To concatenate string literals one just writes them adjacent to each other with white space in between.

2.3 Variables

We have used already variables in some examples before. Variables are objects whose values can change through the execution. They are declared in the program by specifying the data type (which determines the amount of storage needed) and the name of the variable.

2.3.1 Naming conventions

Variable names are usually lower case, but this is not a strict rule. Some keywords are reserved for the language so they cannot be used as variable names:

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

Table 2.2. C++ Keywords

Another rule is that an identifier cannot start with a digit. Underscores are also allowed. The following tables show common conventions for identifiers, together with bad practices and illegal names

Table 2.3 Valid Identifiers	
n	Typically an integer variable
count	Meaningful as documentation
buff_size	C++ style—underscore separates words
bufferSize	Java and Pascal style—capital separates words
q2345	Obscure
cout	Used in the standard library iostream
_Sysfoo	Underscore capital is for system use
too__bad	Double underscore is for system use

Table 2.4 Illegal as Identifiers	
for	Keyword
3q	Cannot start with digit
-count	Do not mistake - for _

2.3.2 Declaration and Initialisation

Some examples on how to declare variables are as follows:

```
double salary, wage;    // defines two variables of type double
int month,
    day, year;          // defines three variables of type int
std::string address;    // defines one variable of type std::string
```

where several variables of the same type are declared on the same line, separated by commas.

Objects can be initialised in the following ways:

```
int ival(1024);     // direct-initialization
int ival = 1024;    // copy-initialization
```

For more complicated class types, we will see later that there may be more forms of initialising the corresponding class objects. That is defined in the class itself by something called the constructor.

Initialisation may be done for several objects in one line, or with an arbitrarily complicated expression, including functions. For example

```
double price = 109.99, discount = 0.16;
double sale_price = apply_discount(price, discount);
```

where in the second line a there is a function `apply_discount` with two arguments, which will return the value to initialise `sale_price`.

There is a way of declaring a variable without defining it (i.e. without reserving space in memory and initialising it). This just declares the presence of the variable somewhere in the program which will be defined there. This is done by using the `extern` keyword.

```
extern int i;   // declares but does not define i
int i;          //  declares and defines i
extern double pi = 3.1416; // definition
```

where we note in the last line that if we asign it a value then if becomes a definition as well. This is useful to declare variables which are common to different files.

2.3.3 Scope of a name

A name of a variable, function or even data type may be restricted to certain files, or certain blocks of code and may not be known to other blocks of code. The part of the program where a certain name applies is called its **scope.** Scopes in C++ are usually delimited by curly braces. We have seen already this for some of the control structures such as `if, while, for`. Variables are usually defined from their point of declaration and are not known outside the scope defined by the curly braces where they are.

In C++ it is usually good practice to define variables where they are used. Also you should avoid using the same variable names in nested scopes, which may be confusing and create bugs.

Exercise Write a program which:

1. Writes a formatted header paragraph of text with several lines informing the user about: the author of the program, the year and version of the program, A short abstract of the program describing what it does (look at the most suitable characters from the table in the previous sections, to format the text).

2. Prompts the user for an input integer and then computes the factorial of that number, using a for loop. The program should report an error if the input number is negative and print an error message (try to use the bell character to see what happens).

3. Test the program for increasing values of the input. Can you explain what's happening. Repeat the calculation using double and try again.

2.4 `const` qualifier

In many situations it is convenient to define constants, for example if there is a fixed dimension (which you may want to change later and would be a pain to change all occurences in the program), or a fundamental constant in your problem an you want an error to ocurr if there is some bug which changes its value.

This is done by adding the keyword `const` to the data type and initialising the object. For example:

```
const int bufSize = 512;     // input buffer size
```

A constant is local to the scope where it is defined. To make a constant visible to several files, it should be declared as external in all the files where it is used and defined in one of them. For example:

```
// file_1.cc
// defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_2.cc
extern const int bufSize; // uses bufSize from file_1
// uses bufSize defined in file_1
for (int index = 0; index != bufSize; ++index)
      // ...
```

Exercise Write a program which computes the area under a parabola in a interval using a trapezium rule. The program should:

1. Use a starting step of integration and compute a first estimate, and then repeat with a `while` loop with a smaller step while the error estimate (obtained by comparing to the previous value) is below a threshold.

2. You should define `constants` in your code which hold: the maximum number of subdivisions of the integration domain, minimum number of subdivision (to start off with), and the goal for the relative error.

3. The program prints the result with error estimate and error compared to the exact result.

4. Always comment your program with a lot of detail so that others reading it will quickly understand it.

2.5 References

A reference is a compound type (i.e. defined in terms of another type) which works as an alias, by referring to a object of the corresponding type. Operations on the reference act on the object it refers to, so a reference is just another name for an object. Because it always has to refer to a specific object, a reference must be initialised when it is defined.

A reference declaration and initialisation requires adding the character & attached to the name of the alias:

```
int ival = 1024;
int &refVal = ival; // ok: refVal refers to ival
int &refVal2;       // error: a reference must be initialized
int &refVal3 = 10;  // error: initializer must be an object
```

For example if we add to `refVal` or assign it to a variable `ival`, it will be modified or used respectively.

The rule for declaration is to attach the & character to the reference name. For example in multiple references

```
int i = 1024, i2 = 2048;
int &r = i, r2 = i2;      // r is a reference, r2 is an int
int i3 = 1024, &ri = i3;  // defines one object, and one reference
int &r3 = i3, &r4 = i2;   // defines two references
```

`const` type references are references to constants, i.e.

```
const int ival = 1024;
const int &refVal = ival;      // ok: both reference and object are const
int &ref2 = ival;              // error: non const reference to a const object
```

2.5.1 Quick introduction to functions

An essential feature of the language is to allow the definition of functions. Functions are essential to produce structured code, because it allows to break a large project into smaller problems. A C++ program is essentially a set of functions which are called by the `main` function.

The basic structure of a function can be divided into a **header** and the **body** block of the function. The header specifies the type of data to be returned, the name of the function, and list of arguments. The body contains a list of statements and return statement at the end which returns the value of the function.

```
returntype functionname(arg1, arg2, ...){
   statements...;
   return returntypeVariable;
}
```

The header of the function can appear as a declaration of the function before it is defined, with a ; at the end

```
returntype functionname(arg1, arg2, ...);
```

This is called a **prototype** and it is used by the compiler when it checks all the types declared in the program. For a simple program with just one file it must appear before `main`. For example, the following program

```
#include<iostream>
int main(){
   printme();
}
void printme(){
    std::cout« "Print"« std::endl;
}
```

will not compile because the function `printme` has not been declared before being used in `main`. The correct form is to include its definition before, or the prototype:

```
#include<iostream>
void printme();
int main(){
   printme();
}
void printme(){
    std::cout« "Print"« std::endl;
}
```

This simple example also illustrates the point that a function may have no arguments and it may not return anything (void).

The connection with references, appears when we consider functions with arguments. If the arguments of a function is a data type which is not a reference, then the code wil create a copy of that data type to be used in the function. This is called pass-by-value. However, at times one may want to write functions which act on a variable and store the value of the computation directly in it without having to create a copy (this is much more efficient for more complex data types which take up a lot of memory space). This is known as call-by-reference.

An example of call-by-value:

In file compute_sum.cpp

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

int compute_sum(int n)  // sum from 1 to n
{
   int   sum = 0;
   for ( ; n > 0; --n)  // value of n is changed
      sum += n;
   return sum;
}

int main()
{
   int   n = 3, sum;

   cout << n << endl;     // 3 is printed
   sum = compute_sum(n);
   cout << n << endl;     // 3 is printed
   cout << sum << endl;
}
```

An example of a function which calls by reference is as follows

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

2.6 Typedef

Another type of alias is `typedef`. This allows to define a synonym for a certain data type which may be convenient for readibility, or to simplify long types to make them easier to type. Examples of typedef definitions are

```
typedef double wages;       //  wages is a synonym for double
typedef int exam_score;     //  exam_score is a synonym for int
typedef wages salary;       //  indirect synonym for double
```

and their usage later in the code

```
wages hourly, weekly;     // double hourly, weekly;
exam_score test_result;   // int test_result;
```

2.7 Header files

Header files are a way of having declarations which are common to different source code files. For example if your program is large you may want to break it down into several files where sets of related functions are all grouped in a file. A header file allows to include the declarations of functions and data types in your program which are common to several source files, and can be include in all of them through an #include preprocessor directive. Then each source code file can be compiled separately.

Headers usually have an extension `.h` and are included in the following way

```
#include "headername.h"
```

This form in quotes searches for the header file in the current directory. The form we have used for example for `#include<iostream>` searches in the system pre-defined paths. In the form with quotes, you can also specify a full path.

The compilation of several source code files can be done as follows

```
$ g++ -o executable source1.cpp source2.cpp source3.cpp ...
```

Exercise

1. Adapt the program `compute_sum.cpp` so that `n` is called by reference. Check what happens now to the value of `n`.
2. Re-write one (or both) of the programs which compute the factorial or integral of a parabola, with a function for each of the operations (taking as arguments either the integer or integration limits respectively). Place the function in a separate source file and write a header to be included in the two source files. Try to compile.

3 Library Types

The C++ standard library defines convenient library types for strings and vectors, (the corresponding built in types arrays and pointers will be seen later). These come with iterator types for an easy access to elements. These library types are higher level in the sense that we only need to know the operations they support without worrying about the way they are stored in memory.

3.1 `using` declarations

To avoid having to use the scope operator repeatedly, one can write a `using` declaration at the top of the source code. The general form is

```
using namespace::name;
```

For example

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1
         << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

where a `using` declaration was written for each name. In some cases however, it may be convenient to declare not only some names in the namespace, but the entire namespace. For the standard library this is particularly useful because you end up using a lot of names which are defined in the library and it may become cumbersome to write `std::` before each name. This is done by declaring the entire namespace

```
using namespace std;
```

Using declarations are not used in header files. In header files, the full qualified name of the library type must be used.

3.2 Library `string` type

This type supports variable length strings of characters, manages the memory and provides operations. To use it one must include

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

or alternatively to the last using declaration, the entire standard namespace.

`string s1;`	Default constructor; s1 is the empty string
`string s2(s1);`	Initialize s2 as a copy of s1
`string s3("value");`	Initialize s3 as a copy of the string literal
`string s4(n, 'c');`	Initialize s4 with n copies of the character `'c'`

Table 3.1. Ways to Initialize a `string`

The interface with `cin, cout` is such that each string read or write corresponds to a chain of characters with no white space. A white space signals the end of the string. For example the following code reads the first string which is typed before a white space and returns it

```
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;          // empty string
    cin >> s;          // read whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

One can read an unkown number of strings as follows

```
int main()
{
    string word;
    // read until end-of-file, writing each word to a new line
    while (cin >> word)
        cout << word << endl;
    return 0;
}
```

To read an entire line we can use the `getline` string operation which reads a line until the newline character is introduced (which is not stored in the string)

```
int main()
{
    string line;
    // read line at time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

The most commonly used string operations are

`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`
`s.size()`	Returns number of characters in `s`
`s[n]`	Returns the character at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` equal to the concatenation of `s1` and `s2`
`s1 = s2`	Replaces characters in `s1` by a copy of `s2`
`v1 == v2`	Returns `true` if `v1` and `v2` are equal; `false` otherwise
`!=, <, <=, >, and >=`	Have their normal meanings

Table 3.2. `string` Operations

Notes

- The type returned by .size() is actually not an integer but a companion type of `string`. This is necessary because you can easily read an entire file into a string whose size cannot be stored in a

regular `unsigned int`. That special type is `string::size_type` and it should be used instead of integers to avoid runtime errors.

- The + operator must have at least one string as argument (if all literals it does not work)

```
string s1 = "hello";   // no punctuation
string s2 = "world";
string s3 = s1 + ", ";           // ok: adding a string and a literal
string s4 = "hello" + ", ";      // error: no string operand
string s5 = s1 + ", " + "world"; // ok: each + has string operand
string s6 = "hello" + ", " + s2; // error: can't add string literals
```

- The subscript of a string (when fetching an element) starts at 0. One can change the character in position `n` by the character `x` for example through `str1[n-1]='x'`

There's a set of functions defined in the `cctype` header to process individual characters of a string:

`isalnum(c)`	TRue if c is a letter or a digit.
`isalpha(c)`	true if c is a letter.
`iscntrl(c)`	true if c is a control character.
`isdigit(c)`	true if c is a digit.
`isgraph(c)`	true if c is not a space but is printable.
`islower(c)`	true if c is a lowercase letter.
`isprint(c)`	TRue if c is a printable character.
`ispunct(c)`	TRue if c is a punctuation character.
`isspace(c)`	true if c is whitespace.
`isupper(c)`	TRue if c is an uppercase letter.
`isxdigit(c)`	true if c is a hexadecimal digit.
`tolower(c)`	If c is an uppercase letter, returns its lowercase equivalent; otherwise returns c unchanged.
`toupper(c)`	If c is a lowercase letter, returns its uppercase equivalent; otherwise returns c unchanged.

Table 3.3. `cctype` Functions

Exercises:

1. Test the string input and output examples in this section

2. Do the following exercises from the C++ primer:

Exercise 3.7: Write a program to read two `strings` and report whether the `strings` are equal. If not, report which of the two is the larger. Now, change the program to report whether the `strings` have the same length and if not report which is longer.

Exercise 3.8: Write a program to read `strings` from the standard input, concatenating what is read into one large `string`. Print the concatenated `string`. Next, change the program to separate adjacent input `strings` by a space.

Exercise 3.9: What does the following program do? Is it valid? If not, why not?

```
string s;
cout << s[0] << endl;
```

Exercise 3.10: Write a program to strip the punctuation from a `string`. The input to the program should be a string of characters including punctuation; the output should be a `string` in which the punctuation is removed.

3.3 Library type `vector`

This is a class template which allows to define variable size vectors of any type (including any user defined type!), so it is a container (because it contains other objects). Without knowing anything about the class we can just specify the type of the vector. We need to include `#include<vector>` and a `using` statement either for the name `vector` or the whole standard namespace as before.

The generic ways of declaring are

`vector<T> v1;`	`vector` that holds objects of type T;
	Default constructor v1 is empty
`vector<T> v2(v1);`	v2 is a copy of v1
`vector<T> v3(n, i);`	v3 has n elements with value i
`vector<T> v4(n);`	v4 has n copies of a value-initialized object

Table 3.4. Ways to Initialize a `vector`

Examples

```
vector<int> ivec1;           // ivec1 holds objects of type int
vector<int> ivec2(ivec1);    // ok: copy elements of ivec1 into ivec2
vector<string> svec(ivec1);  // error: svec holds strings, not ints

vector<int> ivec4(10, -1);       // 10 elements, each initialized to -1
vector<string> svec(10, "hi!");  // 10 strings, each initialized to "hi!"
```

If we do not specify a value, the vector is initialised for us.

Operations

`v.empty()`	Returns `true` if v is empty; otherwise returns `false`
`v.size()`	Returns number of elements in v
`v.push_back(t)`	Adds element with value `t` to end of v
`v[n]`	Returns element at position n in v
`v1 = v2`	Replaces elements in v1 by a copy of elements in v2
`v1 == v2`	Returns TRue if v1 and v2 are equal
`!=, <, <=, >, and >=`	Have their normal meanings

Table 3.5. `vector` Operations

Just as for strings for each vector of a given type there is a size type `vector<T>::size_type`.

One can add elements with the `push_back()` operation:

```
// read words from the standard input and store them as elements in a vector
string word;
vector<string> text;    // empty vector
while (cin >> word) {
    text.push_back(word);     // append word to text
}
```

and access an element (which must already exist!!) by specifying a subscript:

```
// reset the elements in the vector to zero
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

Note: Using the `size()` function is good programming practice rather than remembering the size of the vector (since they can grow).

Exercises

1. Write a program to declare several vectors of diferent built in types and the string type, in all the ways on the table at the beginning of the section and print their values.
2. Do the following exercises from, the C++ primer

Exercise 3.13: Read a set of integers into a `vector`. Calculate and print the sum of each pair of adjacent elements in the `vector`. If there is an odd number, tell the user and print the value of the last element without summing it. Now change your program so that it prints the sum of the first and last elements, followed by the sum of the second and second-to-last and so on.

Exercise 3.14: Read some text into a `vector`, storing each word in the input as an element in the `vector`. transform each word into uppercase letters. Print the transformed elements from the `vector`, printing eight words to a line.

Exercise 3.15: Is the following program legal? If not, how might you fix it?

```
vector<int> ivec;
ivec[0] = 42;
```

Exercise 3.16: List three ways to define a `vector` and give it 10 elements, each with the value 42. Indicate whether there is a preferred way to do so and why.

3.4 Iterators

They are built in library types which allow to navigate through the elements of a vector. These are in general safer than using integers and subscripting, for example, because they prevent accessing elements outside the bounds of the vector (these can be serious bugs which are hard to find because the code may compile and run apparently normally).

There is an iterator for each container type, for example for a vector of integers:

```
vector<int>::iterator iter;
```

Each vector container defines two functions which return a value with the type of the corresponding iterator, they are the `begin()` function

```
vector<int>::iterator iter = ivec.begin();
```

which returns a value referring to the first element of the vector (same as `ivec[0]`); and the `end()` function which returns a value beyond the last element, which is a sentinel indicating it refers to a non-existing element, and cannot be increased!

The operations on iterators are as follows:

- Dereferencing: it allows to access the element an iterator refers to and it is done through the *dereference operator* *:

  ```
  *iter = 0;
  ```

  This statement, for example, sets the current element that `iter` refers to, to zero.

- Increment: This moves the iterator to refer to the next element `++iter;`

- Positive or negative shift by an integer: `iter+n` or `iter-n` returns an iterator of the same type referring to a position ahead or behind the current element.

- Comparison: Through the operations ==, !=

There is also a `difference_type` similar to `size_type` which holds the distance between two iterators (and it is guarateed to be large enough for the largest distance between any two iterators) so

```
iter1-iter2;
```

returns a `difference_type`.

Here is an **example** of a comparison between subscripting

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
        ivec[ix] = 0;
```

and using iterators

```
// equivalent loop using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin();
                           iter != ivec.end(); ++iter)
    *iter = 0;  // set element to which iter refers to 0
```

`const_iterator` **type :** This is another type defined by each container which is a "read only" iterator, in the sense that it refers to the elements of a vector (and thus can equally be used to navigate through the vector), but cannot be used to change the value of the elements of the vector. The difference is that when we dereference a `const_iterator` using *, we get a reference to a constant type. The use of this type of iterator may be safer in some "read-only" situations.

Examples:

- Standard usage:

```
// use const_iterator because we won't change the elements
for (vector<string>::const_iterator iter = text.begin();
                                    iter != text.end(); ++iter)
    cout << *iter << endl; // print each element in text
```

- An error:

```
for (vector<string>::const_iterator iter = text.begin();
                                   iter != text.end(); ++ iter)
    *iter = " ";     // error: *iter is const
```

- Possible confusion:

```
vector<int> nums(10);  // nums is nonconst
const vector<int>::iterator cit = nums.begin();
*cit = 1;               // ok: cit can change its underlying element
++cit;                  // error: can't change the value of cit
```

- More examples:

```
const vector<int> nines(10, 9);  // cannot change elements in nines
// error: cit2 could change the element it refers to and nines is const
const vector<int>::iterator cit2 = nines.begin();
// ok: it can't change an element value, so it can be used with a const vector<int>
vector<int>::const_iterator it = nines.begin();
*it = 10; // error: *it is const
++it;     // ok: it isn't const so we can change its value
```

Exercises

Exercise 3.17: Redo the exercises from Section 3.3.2 (p. 96), using iterators rather than subscripts to access the elements in the `vector`.

Exercise 3.18: Write a program to create a `vector` with 10 elements. Using an iterator, assign each element a value that is twice its current value.

Exercise 3.19: Test your previous program by printing the `vector`.

Exercise 3.20: Explain which iterator you used in the previous programs, and why.

Exercise 3.21: When would you use an iterator that is `const`? When would you use a `const_iterator`. Explain the difference between them.

Consider the following way of locating the middle element of a vector
`vector<int>::iterator mid = vi.begin() + vi.size() / 2;`

Exercise 3.22: What happens if we compute `mid` as follows:

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

4 Arrays and Pointers

These are lower level equivalents to vectors and iterators respectively, which are built into the language. The main *disadvantages* are that arrays are fixed size and these types do not provide simple operations to: add elements; to make sure we don't exceed bounds; to check sizes, etc...

However they are still used in the implementations of class types, because they are more efficient. For example, the implementation of vectors and strings in the standard library will most definitly have arrays of the type of the corresponding types involved. As a rule of thumb, arrays and pointers should be used mostly in class definitions, otherwise vectors and iterators should be used because they are safer and will prevent bugs (or help debuging).

4.1 Arrays

Just like vectors they are compound types consisting of a type specifier and a dimension. The type can be any built in data type or class type. With the exception of references (there are no arrays of references), the element type can be compound as well.

To initialise the array we need to specify the dimention in brackets [], either with a literal constant or an expression which yields a constant which must be known at compile time:

```
// both buf_size and max_files are const
const unsigned buf_size = 512, max_files = 20;
int staff_size = 27;            // nonconst
const unsigned sz = get_size();  // const value not known until run time
char input_buffer[buf_size];     // ok: const variable
string fileTable[max_files + 1]; // ok: constant expression
double salaries[staff_size];     // error: non const variable
int test_scores[get_size()];     // error: non const expression
int vals[sz];                    // error: size not known until run time
```

Elements can be initialised explicitly by providing a comma-separated list in braces, otherwise the elemenst are initialised as the corresponding types, or undefined. When initialised explicitly the compiler may infer the dimension of the array if not specified.

```
int ia[] = {0, 1, 2}; // an array of dimension 3
```

If the dimension is specified and the list is smaller, then the remaining elements are initialised to zero or the default constructor if a class type

```
const unsigned array_size = 5;
// Equivalent to ia = {0, 1, 2, 0, 0}
// ia[3] and ia[4] default initialized to 0
int ia[array_size] = {0, 1, 2};
// Equivalent to str_arr = {"hi", "bye", "", "", ""}
// str_arr[2] through str_arr[4] default initialized to the empty string
string str_arr[array_size] = {"hi", "bye"};
```

For arrays of characters, since string literals contain the null character at the end, one has to be careful to take that into account in the dimension of the array (if specified). Examples:

```
char ca1[] = {'C', '+', '+'};                // no null
char ca2[] = {'C', '+', '+', '\0'};         // explicit null
char ca3[] = "C++";     // null terminator added automatically

const char ch3[6] = "Daniel"; // error: Daniel is 7 elements
```

Warning: There is no copy or asignment for arrays (it has to be done through a loop element by element).

Operations on arrays We access elements by subscripting in the same way as for vectors. There is an integer like type `size_t` which is the correct type to be used for the index. Similarly to vectors the index runs from 0 to size-1. It is the responsibility of the programmer to make sure that the index is within the bounds.

In the following example, a `for` loop steps through the 10 elements of an array, assigning to each the value of its index:

```
int main()
{
    const size_t array_size = 10;
    int ia[array_size]; // 10 ints, elements are uninitialized

    // loop through array, assigning value of its index to each element
    for (size_t ix = 0; ix != array_size; ++ix)
          ia[ix] = ix;
    return 0;
}
```

Using a similar loop, we can copy one array into another:

```
int main()
{
    const size_t array_size = 7;
    int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };
    int ia2[array_size]; // local array, elements uninitialized

    // copy elements from ia1 into ia2
    for (size_t ix = 0; ix != array_size; ++ix)
          ia2[ix] = ia1[ix];
    return 0;
}
```

Do the following exercises from the C++ primer:

Exercise 4.6: This code fragment intends to assign the value of its index to each array element. It contains a number of indexing errors. Identify them.

```
const size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
      ia[ix] = ix;
```

Exercise 4.7: Write the code necessary to assign one array to another. Now, change the code to use `vectors`. How might you assign one `vector` to another?

Exercise 4.8: Write a program to compare two arrays for equality. Write a similar program to compare two `vectors`.

Exercise 4.9: Write a program to define an array of 10 `ints`. Give each element the same value as its position in the array.

4.2 Pointers

These are the lower level equivalent of iterators, for arrays. They are compound objects. A pointer is simply an object that points to the address (location) in memory, which it holds. A big difference is that, unlike iterators, pointers can point at a single object, not just the elements of a cointainer like iterators do.

Initialisation

Pointers are declared by adding * together with the identifier we want to declare as a pointer variable (the ** must come* together *with every pointer variable declared, not with the type specified for the container*!).

```
vector<int>   *pvec;      // pvec can point to a vector<int>
int           *ip1, *ip2; // ip1 and ip2 can point to an int
string        *pstring;   // pstring can point to a string
double        *dp;        // dp can point to a double
```

Note: Pointer declarations make more sense when read from left to right!

The values of a pointer can be: i) a memory address, ii) one past the end of an object, iii) or zero (means no value and should be used instead of uninitialised pointers which will produce VERY dangerous bugs),

```
int ival = 1024;
int *pi = 0;       // pi initialized to address no object
int *pi2 = & ival; // pi2 initialized to address of ival
int *pi3;          // ok, but dangerous, pi3 is uninitialized
pi = pi2;          // pi and pi2 address the same object, e.g. ival
pi2 = 0;           // pi2 now addresses no object
```

There are only four kinds of values that may be used to initialize or assign to a pointer:

1. A constant expression with value 0 (e.g., a const integral object whose value is zero at compile time or a literal constant 0)
2. An address of an object of an appropriate type
3. The address one past the end of another object
4. Another valid pointer of the same type

Some examples of these rules are as follows

```
int ival;
int zero = 0;
const int c_ival = 0;
int *pi = ival; // error: pi initialized from int value of ival
pi = zero;      // error: pi assigned int value of zero
pi = c_ival;    // ok: c_ival is a const with compile-time value of 0
pi = 0;         // ok: directly initialize to literal constant 0


double dval;
double *pd = &dval;   // ok: initializer is address of a double
double *pd2 = pd;     // ok: initializer is a pointer to double

int *pi = pd;   // error: types of pi and pd differ
pi = &dval;     // error: attempt to assign address of a double to int *
```

The reason why types must match is because pointers provide indirect access to an object, and the operations which are supported are determined by such type (hence the types must match).

void pointers

This is a special case when the pointer does not have a type specification and it simply holds an address in memory. Thus this is used only to pass addresses around through functions or comparison to other pointers.

```
double obj = 3.14;
double *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;       // obj can be an object of any type
pv = pd;               // pd can be a pointer to any type
```

Exercises

Exercise 4.10: Explain the rationale for preferring the first form of pointer declaration:

```
int *ip; // good practice
int* ip; // legal but misleading
```

Exercise 4.11: Explain each of the following definitions. Indicate whether any are illegal and if so why.

```
(a) int* ip;
(b) string s, *sp = 0;
(c) int i; double* dp = &i;
(d) int* ip, ip2;
(e) const int i = 0, *p = i;
(f) string *p = NULL;
```

Exercise 4.12: Given a pointer, p, can you determine whether p points to a valid object? If so, how? If not, why not?

Exercise 4.13: Why is the first pointer initialization legal and the second illegal?

```
int i = 42;
void *p = &i;
long *lp = &i;
```

Operations on pointers

The basic operations are: Dereferencing (similar to iterators),

```
string s("hello world");
string *sp = &s; // sp holds the address of s
cout  <<*sp;     // prints hello world
```

change value of object

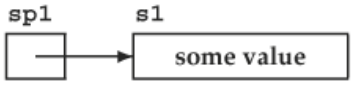
```
*sp = "goodbye"; // contents of s now changed
```

and asign new address

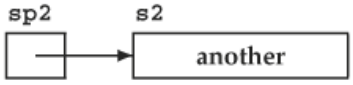
```
string s2 = "some value";
sp = &s2;  // sp now points to s2
```

Some examples with schematics

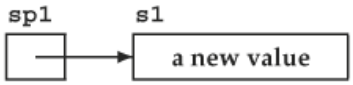
```
string s1("some value");
string *sp1 = &s1;
```



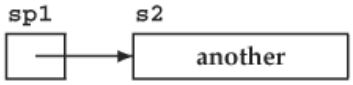

```
string s2("another");
string *sp2 = &s2;
```




```
// assign through sp1
// value in s1 changed
*sp1 = "a new value";
```




```
// assign to sp1
// sp1 points to a different object
sp1 = sp2;
```



Note: Do not confuse pointers with references. Unlike the last line of the last example, references cannot be re-bound, i.e. we cannot change a reference to refer to a different object by asigning it after it is declared (and thus initialised), so for example

```
int &ri = ival, &ri2 = ival2;
ri = ri2;    // assigns ival2 to ival
```

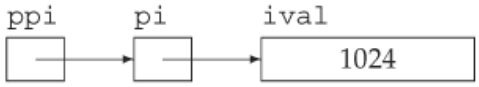
has changed the value of `ival` to be the same as `ival2` and the references stayed the same.

Pointers to pointers

Pointers are themselves objects in memory. They, therefore, have addresses that we can store in a pointer:

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to int
```

which yields a pointer to a pointer. We designate a pointer to a pointer by using **. We might represent these objects as



As usual, dereferencing `ppi` yields the object to which `ppi` points. In this case, that object is a pointer to an `int`:

```
int *pi2 = *ppi; // ppi points to a pointer
```

To actually access `ival`, we need to dereference `ppi` twice:

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

4.2.1 Pointers and Arrays

An array element can be accessed through a pointer. The name of an array is actually a pointer to the first element of the array:

```
int ia[] = {0,2,4,6,8};
int *ip = ia; // ip points to ia[0]
```

or if we want to point to another element

```
ip = &ia[4];    // ip points to last element in ia
```

Pointer arithmetic and dereferencing

Similarly to iterators one can advance or go back positions in memory or find distances between pointers by using pointer arithmetic. To advance to a new element (or subtract to go backwards)

```
ip = ia;            // ok: ip points to ia[0]
int *ip2 = ip + 4;  // ok: ip2 points to ia[4], the last element in ia
```

If a pointer points to an array, it is equivalent to use this arithmetic or to subscript as for the corresponding array, i.e.

```
int *p = &ia[2];     // ok: p points to the element indexed by 2
int j = p[1];        // ok: p[1] equivalent to *(p + 1),
                     //    p[1] is the same element as ia[3]
int k = p[-2];       // ok: p[-2] is the same element as ia[0]
```

Important: Note that dereferrencing and pointer arithmetic do NOT commute (parenthesis are essential!) so

```
int last = *(ia + 4); // ok: initializes last to 8, the value of ia[4]
```

is not the same as

```
last = *ia + 4;     // ok: last = 4, equivalent to ia[0]+4
```

The difference between pointers yields a type `ptrdiff_t`

```
ptrdiff_t n = ip2 - ip; // ok: distance between the pointers
```

Note: It is up to the user to be careful enough not to go beyond bounds to invalid addresses.

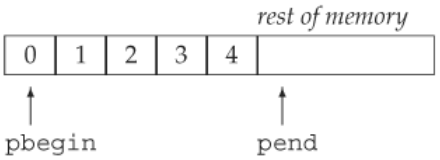
The computation of the pointer to "one position past end" is also possible (as for iterators) however it has to be controled by the user!

```
const size_t arr_size = 5;
int arr[arr_size] = {1,2,3,4,5};
int *p = arr;           // ok: p points to arr[0]
int *p2 = p + arr_size; // ok: p2 points one past the end of arr
                        //    use caution -- do not dereference!
```

Example

Print an array using pointers

```
const size_t arr_sz = 5;
int int_arr[arr_sz] = { 0, 1, 2, 3, 4 };
// pbegin points to first element, pend points just after the last
for (int *pbegin = int_arr, *pend = int_arr + arr_sz;
          pbegin != pend; ++pbegin)
    cout << *pbegin << ' '; // print the current element
```



Pointers to `const`, `const` pointers and `const` pointers to `const`

These are pointers which cannot change the value of the object they point to. They can be asigned the address of an object which is not a `const` but they cannot change their value. They are usually used in argument definition of functions if we want to ensure they are not changed

```
double dval = 3.14; // dval is a double; its value can be changed
cptr = &dval;       // ok: but can't change dval through cptr

dval = 3.14159;       // dval is not const
*cptr = 3.14159;      // error: cptr is a pointer to const
double *ptr = &dval;  // ok: ptr points at non-const double
*ptr = 2.72;          // ok: ptr is plain pointer
cout << *cptr;        // ok: prints 2.72
```

One can also have constant pointers (i.e. the address they point to, cannot be changed and must be initialised when declared)

```
int errNumb = 0;
int *const curErr = &errNumb; // curErr is a constant pointer
```

which however can be used to change the value of the object they point to

```
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset value of the object to which curErr is bound
}
```

Finally, one can have `const` pointers to `const` where neither the pointer nor the object pointed at can be changed

```
const double pi = 3.14159;
// pi_ptr is const and points to a const object
const double *const pi_ptr = &pi;
```

Exercises

1. Write code to change the value of a pointer (test various types). Write code to change the value to which the pointer points (test various types).
2. Write a program that uses pointers to set the elements in an array of ints to zero.

4.3 Dynamic memory allocation & multi-dimensional arrays

Arrays (as we have seen so far) have the limitation of having a fixed size and to be available only to the block where they are declared. Even though their size is fixed, it can be determined at run time by using some dynamical memory allocation facilities in the language. The main thing to be aware of is that when we dynamically allocate an array, it continues to exist unless explicitely freed. When a C++ program runs it has a finite amout of memory availlable for allocation (called the free store or heap) so it is important to free dynamically allocated arrays, after they are used. In C++, to dynamically allocate memory one used the `new` and `delete` expressions.

Defining a Dynamic Array

This is done by declaring a pointer to the first element of the array of a certain type, and then allocating space with `new` some examples are as follows

```
int *pia = new int [10]; // The array is uninitialised for 10 integers
```

```
string *psa = new string [12]; // The array is value initialise for class type objects
by the default constructor

int *pia2 = new int [10](); // The array is value-initialised with 10 integers to zero.
This is done through the parenthesis for built-in types
```

where we have noted the difference between uninitialised and value-initialised arrays. This distinction is important, for example for dynamically allocated arrays of constants, which must be value initialised using the parethesis for built in types, otherwise we get an error.

The examples above are for arrays of fixed size determined when the code is written. However, the big advantage is to be able to determine the array size at run time:

```
size_t n = get_size(); //Let's assume there is a function which determines the size of
the array while running...

int *p = new int [n]; // The array will be uninitialised with size n as determine while
running the code

//Then the rest of the program follows and uses the array...
```

We should note that if the size determined at run time is 0, the code still works.

The final step is to free the memory after we are done with using the array. A good rule of thumb is to think about the point in the code where the array will be freed, at the same time as we write its declaration to allocate it, and write the `delete` statement imediatelly to avoid forgetting. For the last example the statement is

```
delete [] pia;
```

For each new statement we must have a delete statement like this one to free the memory.

Exercise 4.27: Given the following new expression, how would you `delete pa`?

```
int *pa = new int[10];
```

Exercise 4.28: Write a program to read the standard input and build a `vector` of `int`s from values that are read. Allocate an array of the same size as the `vector` and copy the elements from the `vector` into the array.

Initialising a vector from an array

There is a constructor for vectors which allows to do so, by specifying a first argument which is a pointer to the first element of the array we want to pass and another pointer which points to one past the last element of the array we want to pass (i.e. we can also pass a portion of an array as well):

```
const size_t arr_size = 6;
int int_arr[arr_size] = {0, 1, 2, 3, 4, 5};
// ivec has 6 elements: each a copy of the corresponding element in int_arr
vector<int> ivec(int_arr, int_arr + arr_size);


// copies 3 elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> ivec(int_arr + 1, int_arr + 4);
```

Exercise 4.32: Write a program to initialize a `vector` from an array of `int`s.

Exercise 4.33: Write a program to copy a `vector` of `int`s into an array of `int`s.

Exercise 4.34: Write a program to read `string`s into a `vector`. Now, copy that `vector` into an array of character pointers. For each element in the `vector`, allocate a new character array and copy the data from the `vector` element into that character array. Then insert a pointer to the character array into the array of character pointers.

Exercise 4.35: Print the contents of the `vector` and the array created in the previous exercise. After printing the array, remember to delete the character arrays.

Multidimensional arrays

These are arrays of arrays and are initialised by adding another bracket [] with the dimension. Let's look at some examples.

```
// array of size 3, each element is an array of ints of size 4
int ia[3][4];

 int ia[3][4] = {     /*  3 elements, each element is an array of size 4 */
     {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
     {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
     {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
 };

// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

// explicitly initialize only element 0 in each row
int ia[3][4] = {{ 0 } , { 4 } , { 8 } };

// explicitly initialize row 0
int ia[3][4] = {0, 3, 6, 9};
```

Note, in particular, the very different behaviour of the last two examples! Subscripting is done as usual, with one pair of brackets for each dimension.

One can have multidimensional arrays with any number of dimensions. We should keep in mind that multidimensional arrays are actually arrays of pointers.

Actually it is useful to think about multi-dimensional arrays in terms of pointers. This allows for an easy dynamic allocation of multidimensional arrays.

For example, if we need to allocate a `n` by `n` matrix of `double`swhere `n` is determined at run time, one would write

```
double ** matrix = new double * [n]; //Declare a pointer to pointer of type double, and
allocate n pointers to double one for each row of the array
for(size_t i = 0; i!=n;i++){
  matrix[i]= new double [n]; //Looop over the dynamically allocated rows and allocate
space for the columns
  for(size_t j = 0; j!=n;j++) //Looop over the dynamically allocated rows and columns
to initialise to the sum of their positions
    matrix[i][j]=i+j; //Looop over the dynamically allocated rows and columns to initialise
to the sum of their positions
 }
```

In the previous example one can have more nested "indices" i.e. pointers to pointers to pointers... It is important to *free the memory after it is used*. The easiest way is to copy the declarations with the

new statement, invert the order and replace the new statements by delete.

```
for(size_t i = 0; i!=n;i++)
  delete [] matrix[i];
double delete [] matrix;
```

The only difference is that we do not specify the type of the pointer for the delete statement.

Exercises

1. Write a program which reads the elements of two matrices of doubles from the standard input, determining the size of each matrix from the first line, and then returns the elements of the product and difference of the two matrices.
2. Write a program to read n square matrices (number of matrices and dimension specified by the user), compute all possible pairs of products and print the result for the user.

5 Classes as data structures

Classes allow us to define our own data types and the operations they support. We have already seen some examples of class types such as the library type `string`. We were able to learn how to use this data type and its operations without having to go into the details of how it is implemented.

The definition of a give class involves:

- An ***interface***: Which consists on the operations/functions that the user is able to execute on class types and it may also contain variables (data) that the user may manipulate directly.
- An ***implementation***: Which are typically variables (data) and functions which are hidden when the class type is used but are essential to make it work.

This paradigm of separating the interface from the implementation is useful, because we may change the inner workings of the class (its implementation) without having to change the code that uses it.

There are two ways of defining a class using two different keywords. The most common one in C++ is the `class` keyword, but one can also use the `struct` keyword which is inherited from C. The only difference between the two is the default behaviour. Let's present them and contrast.

A `class` is defined usually in a separate file. The general structure is

```
class ClassName{
  public:
  //Declarations which define the operations, the functions which are public for the user
to use, and the data types which the user might access directly
 private:
  //Hidden declarations of data types which hold the class data, and functions which are
used to make its operations and functions work.
};
```

The set of operations and data types defining the class are called the members of the class. The keywords, `public` and `private` are access lables, which indicate whether the members can be accessed directly in the code or not. The rule is that whenever an access lablel is specified, it is valid until the next line with an access label (which changes the access again). The default behaviour for classes is that when the "first" access label is omitted (at the top of the definition), is that whathever is declares there is set to be `private`. The `struct` keyword is exactly the same as `class` except for this default behaviour which is opposite, i.e. if we don't specify the "first" access label it is implicitly `public`.

One can see in this structure the separation from the interface and implementation, so the abstract operations and functions that characterize the data type we want to create, are separate from how those operations are implemented in the code which defines the class

The declaration of the data members are done exactly in the same way as in normal code, so in a first approach one can use a class simple as a way of encapsulating various data types to make a convenient data sructure. In that case the use of the `struct` keyword may be more descriptive (since there is only public data and no acces labels are necessary).

Let's look at a first example which defines the coordinates of a point in 2D:

```
struct point {
   double x, y;
};
```

Now if we want to declare several 2D points in the main code we just declare it as a usual variable

```
point p1, p2, p3;
```

According to the default rules all member in this case are public and they can be accessed through the dot "." operator which is the member access operator. So for example, to asign values to the coordinates and calculate the distance squared from the origin of point `p1` we would write

```
p1.x=2.3;
p1.y=-4.2;
double distance = (p1.x)*(p1.x)+(p1.y)*(p1.y);
```

An equivalent definition using the keyword `class` would be (note how the keyword `public` is now mandatory)

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

Similarly to the built in data types, we can have pointers to structures. Then to access members we have to dereference first (with parethesis) and use the dot operator after. However there is an operator (arrow operator ->) to avoid having to do so each time:

pointer_to_structure -> member_name

An equivalent construct is given by

*(*pointer_to_structure).member_name*

Some examples are as follows:

Table 4.1 Declarations and Initialization

```
point w, *p = &w;   point v[5];
w.x = 1;  w.y = 4;  v[0] = w;
```

Expression	Equivalent Expression	Value
w.x	p -> x	1
w.y	p -> y	4
v[0].x	v -> x	1
(*p).y	p -> y	4

A more complete example:

In file struct_point1.cpp

```
// Compute an average point

struct point { double x, y; };

point average(const point* d, int size)
{
   point sum = {0, 0};

   for (int i = 0; i < size; i++) {
      sum.x += d->x;
      sum.y += d->y;
      d++;      // d is iterator accessing each point
   }
   sum.x = sum.x / size;
   sum.y = sum.y / size;
   return sum;
}


int main()
{
   point data[5] = { {1.0, 2.0}, {1.0, 3.3},
                     {5.1, 0.5}, {2.0, 2.0}, {0, 0} };
   point average_point;

   average_point = average(data, 5);
   cout << "average point = (" << average_point.x
        << ", " << average_point.y << ") " << endl;
}
```

In addition to having data members, in C++ one can also have member functions. It is more usual to use the `class` keyword in C++. So we will do so from now on and start by illustrating a class with public members functions. Let's re-write the previous example:

In file point4.cpp

```
class point {
public:                 // place public members first
   void  print() const { cout << "(" << x << "," << y << ")"; }
   void  print(const string& name) const;
   void  set(double u, double v) { x = u; y = v; }
   void  plus(point c);
private:
   double x, y;
};

int main()
{
   point w1, w2;
   w1.set(0, 0.5);
   w2.set(-0.5, 1.5);
   cout << "\npoint w1 = ";
   w1.print();
   cout << "\npoint w2 = ";
   w2.print();
   cout << endl;
}
```

Exercises:

1. Design a C++ structure to store a dairy product name, portion weight, calories, protein, fat, and carbohydrates. Twenty-five grams of American cheese have 375 calories, 5 grams of protein, 8 grams of fat, and 0 carbohydrates. Show how to assign these values to the member variables of your structure. Write a function that, given a variable of type struct dairy and a weight in grams (portion size), returns the number of calories for that weight.
2. Write a class `point` that has three coordinates x, y, and z, where the coordinates are private. How can you access the individual members? Create a function which is external to the class and computes the dot product between two `point` variables (pass the points by reference as we have seen some weeks ago).

6 More on expressions and statements

6.1 Some types of behaviour to be aware of

Logical statements We have used logical operators repeatedly in the examples of the previous sections. A point that we haven't mentioned which one has to be careful with is that relational operators are left associative. Thus, if we try to chain expessions, such as

```
if(1 < i < 3){
  ...
};
```

the result will be first to check if `1<i` and then the boolean result of that comparison will be compare to 3.

Regarding `bool` variables, another point to keep in mind is that it is usually bad practice to test equality of a `bool` variable to a bool literal (such as `if(boolvar==true)`), because it is an extra operation that is not needed since we can simply test the value of the `bool` variable itself (`if(boolvar)`).

The ++ and − operators One point to keep in mind about the increment (++) and decrement (−) operators is that they come in two forms:

- Prefix form : `++i` or `--i`: This will first increment the value of the variable and then it will return the incremented value.
- Postfix form : `i++` or `i--`: This will return the value of the variable and then increment it.

If this behaviour does not make a difference, then the first form is preferred. This is because for the second form, the original value of the variable has to be stored in addition to increment it, so that such value can be returned.

```
int i = 0, j;
j = ++i; // j = 1, i = 1: prefix yields incremented value
j = i++; // j = 1, i = 2: postfix yields unincremented value
```

6.2 Other types of expressions

The `sizeof` operator This returns a value of type `size_t` which is the size in bytes of the object. This value is a compile time constant. The possible forms are

```
sizeof (type name);
sizeof (expr);
sizeof expr;
```

and examples

```
Sales_item item, *p;
// three ways to obtain size required to hold an object of type Sales_item
sizeof(Sales_item); // size required to hold an object of type Sales_item
sizeof item; // size of item's type, e.g., sizeof(Sales_item)
sizeof *p;   // size of type to which p points, e.g., sizeof(Sales_item)

// sizeof(ia)/sizeof(*ia) returns the number of elements in ia
int sz = sizeof(ia)/sizeof(*ia);
```

Comma expressions These are series of statement which are executed sequentially from left o right. The value returned by the expression is the one of the rightmost expression. Example:

```
int cnt = ivec.size();
// add elements from size... 1 to ivec
for(vector<int>::size_type ix = 0;
                ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

Precedence It is very important to keep in mind the rules of precedence and how operators associate with each other in a compound expression. The default precedence rules can be overriden by using parenthesis, and such should be done whenever in doubt. For the usual arithmetic operators, the rules are the usual ones. However, for logical, assignment operators, etc... what you think the behaviour is, may be very different from what it actually is! A table of precedence rules is in section 5.10 of the C++ primer which you can consult.

Another point to keep in mind is that the order in which operands of an operator are evaluated is not always necessarily defined, for example

```
// oops! language does not define order of evaluation
if (ia[index++] < ia[index])
```

The `new` and `delete` statements We have already seen how these statements can be used to dynamically allocate memory for arrays. These should be important statements, for class design so let's dissect in more detail their properties:

1. The `new` statement, returns a pointer to a newly allocated object of the type that is specified. Thus this can be used for any data type (not just arrays), including single objects.

   ```
   int i;             // named, uninitialized int variable
   int *pi = new int; // pi points to dynamically allocated,
                      // unnamed, uninitialized int
   ```

   One can initialise the newly allocated object using direct initialisation

   ```
   int i(1024);             // value of i is 1024
   int *pi = new int(1024); // object to which pi points is 1024
   string s(10, '9');                  // value of s is "9999999999"
   string *ps = new string(10, '9');   // *ps is "9999999999"
   ```

   otherwise, the default is used. However it is usually a bad idea to rely on this behaviour! If we want to use explicit value initialisation we add ()

```
string *ps = new string();  // initialized to empty string
int *pi = new int();  // pi points to an int value-initialized to 0
cls *pc = new cls();  // pc points to a value-initialized object of type cls
```

2. As we have seen before, as a general guideline, for every `new` statement we should have a delete statement, to free the memory as soon as we do not need it anymore. This must always be applied only to an object that has been dynamically allocated. Examples:

```
int i;
int *pi = &i;
string str = "dwarves";
double *pd = new double(33);
delete str; // error: str is not a dynamic object
delete pi;  // error: pi refers to a local
delete pd;  // ok
```

   After deletion, the value of the object becomes undefined, though a valid address is still held in the pointer!

3. One can dynamically allocate constant objects and delete them. The rule is that the object must be initialised when created, which means that for built in types we must initialise explicitly, and for class types we may omit initialisation because the default constructor is applied:

```
// allocate and initialize a const object
const int *pci = new const int(1024);

// allocate default initialized const empty string
const string *pcs = new const string;
```

Type conversion When a certain operation involves different types, one of the types may be converted to the other one before the operation can be carried out. The main arithmetic conversions can be understood from looking at some examples:

```
bool      flag;         char           cval;
short     sval;         unsigned short usval;
int       ival;         unsigned int   uival;
long      lval;         unsigned long  ulval;
float     fval;         double         dval;
3.14159L + 'a'; // promote 'a' to int, then convert to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, that int converted to float
sval + cval;    // sval and cval promoted to int
cval + lval;    // cval converted to long
ival + ulval;   // ival converted to unsigned long
usval + ival;   // promotion depends on size of unsigned short and int
uival + lval;   // conversion depends on size of unsigned int and long
```

There is little point on memorising all the conversion rules in mind, so I refer to section 5.12 of the C++ primer for other conversion rules involving other built in data types and class types.

Explicit conversion may be achieved which is called a cast. The general form is

```
cast-name<type>(expression);
```

where `cast-name` is one of the following cast operators:

- `static_cast`: This performs a conversion explicitly at compile time.

```
double d = 97.0;
// cast specified to indicate that the conversion is intentional
char ch = static_cast<char>(d);
```

- `const_cast`: It removes the "constancy" property of the value returned by the variable. An example is if we have a function which contains an argument which is not of `const` type which it doesn't change, but we want to pass the value of a variable which is of `const` type, so we need to remove the constancy property. Then form example

```
const char *pc_str;
char *pc = string_copy(const_cast<char*>(pc_str));
```

- `dynamic_cast` and `reinterpret_cast`: These will be covered later. The first is the equivalent of the `static_cats` operator for dynamically allocated objects.

As a general rule, casting should be avoided unless strictly necessary.

Exercises

1. Write a program that writes the size of each of the built-in types. Also test other types we have been using such as class types from the standard library.
2. Write a program that dynamically allocates a two dimensional array and try to obtain its size information using sizeof.

6.3 Further control structures

The `switch` statement This provides a way to avoid nested `if` statements, by running through a list of cases. This statement compares the value of an arbitrary expression (which must yield an integral result) to the value of each of the cases (which must be an integral result). Let's look at an example:

```
char ch;
// initialize counters for each vowel
int aCnt = 0, eCnt = 0, iCnt = 0,
    oCnt = 0, uCnt = 0;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout  << "Number of vowel a: \t" << aCnt << '\n'
      << "Number of vowel e: \t" << eCnt << '\n'
      << "Number of vowel i: \t" << iCnt << '\n'
      << "Number of vowel o: \t" << oCnt << '\n'
      << "Number of vowel u: \t" << uCnt << endl;
```

An important behaviour to keep in mind is that execution continues to test all the cases unless the break statements are present. This should always be present unless you really mean to do so, because it may create unexpected errors.

An example of intentional absence of break is:

```
int vowelCnt = 0;
// ...
switch (ch)
{
    // any occurrence of a,e,i,o,u increments vowelCnt
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

or equivalently

```
switch (ch)
{
    // alternative legal syntax
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

In addition, just as for the `else` statement, we may include a default line to be executed:

```
// if ch is a vowel, increment the appropriate counter
switch (ch) {
    case 'a':
        ++aCnt;
        break;
    // remaining vowel cases as before
    default:
        ++otherCnt;
        break;
}
```

Some comments on `for` loops We have already seen `for` loops in some detail. Some further rules to keep in mind are:

- Any part of the header of a `for` loop can be omitted by using a null statement. Null statements should always be commented!

```
vector<string>::iterator iter = svec.begin();
for( /* null */ ; iter != svec.end(); ++iter) {
    cout << *iter; // print current element
    // if not the last element, print a space to separate from the next one
    if (iter+1 != svec.end())
        cout << " ";
}
```

- Multiple definitions (of the same type though) and multiple expressions may be included through Comma statements. For example:

```
const int size = 42;
int val = 0, ia[size];
// declare 3 variables local to the for loop:
// ival is an int, pi a pointer to int, and ri a reference to int
for (int ival = 0, *pi = ia, &ri = val;
      ival != size;
      ++ival, ++pi, ++ri)
              // ...
```

The `do while` statement This is similar to the for loop except that the test statement is only performed after each cycle in the loop so it is executed at least once. Because of such, loop variables (or any variable that is tested in the statement) have to be defined before the loop. Also note that the block always ends with a semi-colon `;`

Example:

```
// repeatedly ask user for pair of numbers to sum
string rsp; // used in the condition; can't be defined inside the do
do {
   cout << "please enter two values: ";
   int val1, val2;
   cin  >> val1 >> val2;
   cout << "The sum of " << val1 << " and " << val2
        << " = " << val1 + val2 << "\n\n"
        << "More? [yes][no] ";
   cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

The `break` statement This stops the nearest enclosing `while, for, do while` or `switch` loop. A break statement is only legal when it is inside a loop. It can be inside an `if` statement, only when the

`if` itself is inside a loop.

Example:

```
string inBuf;
while (cin >> inBuf && !inBuf.empty()) {
    switch(inBuf[0]) {
    case '-':
        // process up to the first blank
        for (string::size_type ix = 1;
                    ix != inBuf.size(); ++ix) {
              if (inBuf[ix] == ' ')
                   break; // #1, leaves the for loop
              // ...
        }
        // remaining '-' processing: break #1 transfers control here
        break; // #2, leaves the switch statement
    case '+':
        // ...
    } // end switch
    // end of switch: break #2 transfers control here
}  // end while
```

The `continue` statement This terminates the current loop iteration skiping directly to the next run around the loop. Example:

```
string inBuf;
while (cin >> inBuf && !inBuf.empty()) {
        if (inBuf[0] != '_')
             continue; // get another input
        // still here? process string ...
}
```

Exercises .

Exercise 6.7: There is one problem with our vowel-counting program as we've implemented it: It doesn't count capital letters as vowels. Write a program that counts both lower- and uppercase letters as the appropriate vowelthat is, your program should count both `'a'` and `'A'` as part of `aCnt`, and so forth.

Exercise 6.8: Modify our vowel-count program so that it also counts the number of blank spaces, tabs, and newlines read.

Exercise 6.9: Modify our vowel-count program so that it counts the number of occurrences of the following two-character sequences: `ff`, `fl`, and `fi`.

Exercise 6.12: Write a small program to read a sequence of `strings` from standard input looking for duplicated words. The program should find places in the input where one word is followed immediately by itself. Keep track of the largest number of times a single repetition occurs and which word is repeated. Print the maximum number of duplicates, or else print a message saying that no word was repeated. For example, if the input is

```
how, now now now brown cow cow
```

the output should indicate that the word "now" occurred three times.

Exercise 6.18: Write a small program that requests two `strings` from the user and reports which `string` is lexicographically less than the other (that is, comes before the other alphabetically). Continue to solicit the user until the user requests to quit. Use the `string` type, the `string` less-than operator, and a `do while` loop.

7 Functions

Functions are essential to any elaborate project. They can be thought of as a way of extending the capabilities of the built in operators, such that they can take an arbitrary number of operands.

7.1 Function parameter list & argument passing

We have already seen the basic structure of a function definition in previous chapters. We have also seen that any built in data-type can be an argument (including an empty list) or a return type of a function. However there are some restrictions to this rule in the way such types are passed or returned.

Parameter list Similarly to the local variables, the parameters of a function provide local storage. The difference is that they are initialised from the arguments that are passed when the function is called. The number and type of arguments passed to a function must match the function definition or must be given by an expression that can be implicitely converted to the correct type (we will see an exception below). The types are checked at compile time, so if they do not match an error message will be issued.

Example of errors:

```
int myreturn(int i1){ // Definition of a simple function
  return i1;//which returns its integer argument
}

int main(){
  myreturn("abcd"); //error...  no rule to convert char* to int
  int a=1;
  double f=1.2;
  myreturn(a,f); //error...  wrong number of arguments
  myreturn(); //error...  arguments expected
  myreturn(a); //ok
  myreturn(a*2-1); //ok
  myreturn(f); //ok
  return 0;
}
```

We will see how to pass functions themselves as arguments, later on.

Argument passing As mentioned above, when the parameter of a function is not of reference type, then the corresponding argument is copied to the function when it is called (so the original argument cannot be accessed/changed by the function). Otherwise, if a reference, then the parameter is just a name for the argument which is used directly without copy.

Non-reference parameters:

- *Pointer parameters* - What is passed to the function is a copy of the pointer variable which points to a certain object. In this case the function can change the value of the object that the argument pointer points to, but not the value of the pointer, since what is used in the function is a local copy of the pointer:

```
void reset(int *ip)
{
    *ip = 0; // changes the value of the object to which ip points
    ip = 0;   // changes only the local value of ip; the argument is unchanged
}
```

so when we call ...

```
int i = 42;
int *p = &i;
cout << "i: " << *p << '\n';   // prints i: 42
reset(p);                      // changes *p but not p
cout << "i: " << *p << endl;   // ok: prints i: 0
```

For example to avoid the value of the object the pointer points to, to be changed, we should declare it instead as a pointer to `const`.

- `const` *parameters* - Because of the rule that a local copy is created, we can pass either a `const` or non-`const` argument because a local copy will be created anyway (the reverse is also true). Thus in the case of non-reference types, `const` can be seen just as a way of ensuring the value of this argument is not changed as the function is executed.

Passing arguments as a copy is not enough if we want to: i) change the value of the argument, ii) pass a large object, which may be inefficient, iii) pass an object for which copy is not possible, etc... In such case, pointers or non-reference paramaters will be necessary.

Reference Parameters We have already seen an example of parameters passed by reference. The swap function is a typical example for which a copy would not work, because the local copies of the object would be swaped instead of the original objects.

Another usage of reference parameters is to return additional information from the function. Let's look at an example where a function is supposed to find whether a certain integer occurs in an vector, return an iterator which refers to the first occurrence, and also we want to return the number of occurrences of such integer:

```
// returns an iterator that refers to the first occurrence of value
// the reference parameter occurs contains a second return value
vector<int>::const_iterator find_val(
    vector<int>::const_iterator beg,             // first element
    vector<int>::const_iterator end,             // one past last element
    int value,                                    // the value we want
    vector<int>::size_type &occurs)              // number of times it occurs
{
    // res_iter will hold first occurrence, if any
    vector<int>::const_iterator res_iter = end;
    occurs = 0; // set occurrence count parameter
    for ( ; beg != end; ++beg)
        if (*beg == value) {
            // remember first occurrence of value
            if (res_iter == end)
               res_iter = beg;
            ++occurs; // increment occurrence count
        }
    return res_iter;  // count returned implicitly in occurs
}
```

which we would call for example as
`iterfirst = find_val(ivec.begin(),ivec.end(),42,counter);`

We may also want to pass a large object as a reference, just because it is inneficient to copy it, but we may at the same time want to prevent it is changed. In that case we can use a `const` reference to

avoid copy but have the same behaviour. For example:

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

In general, a rule of good practice is to declare references which are not supposed to be changed as `const`.

Another important rule for reference arguments, is that we cannot pass an expression or an argument that can in principle be implicitely converted to the data type of the reference. Let's look at some examples:

```
// function takes a non-const reference parameter
int incr(int &val)
{
    return ++val;
}
int main()
{
    short v1 = 0;
    const int v2 = 42;
    int v3 = incr(v1);   // error: v1 is not an int
    v3 = incr(v2);       // error: v2 is const
    v3 = incr(0);        // error: literals are not lvalues
    v3 = incr(v1 + v2);  // error: addition doesn't yield an lvalue
    int v4 = incr(v3);   // ok: v3 is a non const object type int
}
```

Passing a reference to a pointer In this case, we are able to change the value of the pointer, and the obect it points to. For example, the `swap` function would become:

```
// swap values of two pointers to int
void ptrswap(int *&v1, int *&v2)
{
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}


int main()
{
    int i = 10;
    int j = 20;
    int *pi = &i;  // pi points to i
    int *pj = &j; // pj points to j
    cout << "Before ptrswap():\t*pi: "
         << *pi << "\t*pj: " << *pj << endl;
    ptrswap(pi, pj); // now pi points to j; pj points to i
    cout << "After ptrswap():\t*pi: "
         << *pi << "\t*pj: " << *pj << endl;
    return 0;
}
```

Exercises:

Exercise 7.3: Write a program to take two `int` parameters and generate the result of raising the first parameter to the power of the second. Write a program to call your function passing it two `int`s. Verify the result.

Exercise 7.4: Write a program to return the absolute value of its parameter.

Exercise 7.5: Write a function that takes an `int` and a pointer to an `int` and returns the larger of the `int` value of the value to which the pointer points. What type should you use for the pointer?

Exercise 7.6: Write a function to swap the values pointed to by two pointers to `int`. Test the function by calling it and printing the swapped values.

Vector and other container parameters It is usually good practice to pass an iterator to the vector (or another container) to be processed, as an argument to a function, instead of the container itseld. If the container must really be passed, then it should be done by reference. An example is:

```
// pass iterators to the first and one past the last element to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end)
{
    while (beg != end) {
        cout << *beg++;
        if (beg != end) cout << " "; // no space after last element
    }
    cout << endl;
}
```

Array parameters Because arrays cannot be copied, they are passed using pointers. Be aware that whenever the parameter of a function is an array, a pointer is being passed, for example:

```
// three equivalent definitions of printValues
void printValues(int*) { /* ... */ }
void printValues(int[]) { /* ... */ }
void printValues(int[10]) { /* ... */ }
```

the two second syntaxes are usually not good practice because they may be misleading, in particular the dimension in the third example is ignored by the compiler! The first form is the one that is good practice, because it corresponds explicitely to what is being passed.

Similarly to other data types, arrays are passed as reference or non-reference types, and `const` or non-`const`.

Non-reference array parameters These are always converted to a pointer to the first element of the array which is then copied. As for other types, if we do not want to change the value of the array elements such parameter should be defined as a pointer to `const`

```
// f won't change the elements in the array
void f(const int*) { /* ... */ }
```

Passing an array by reference In this special case, the size of the array is not ignored! This is because a reference to the array is passed, so the array is not converted to a pointer. The compiler will check the size of the parameter and the argument being passed at compile time (which must match). For example:

```
// ok: parameter is a reference to an array; size of array is fixed
void printValues(int (&arr)[10]) { /* ... */ }
int main()
{
    int i = 0, j[2] = {0, 1};
    int k[10] = {0,1,2,3,4,5,6,7,8,9};
    printValues(&i); // error: argument is not an array of 10 ints
    printValues(j);  // error: argument is not an array of 10 ints
    printValues(k);  // ok: argument is an array of 10 ints
    return 0;
}
```

Note that the parentheses are necessary in `(& arr)[10]` because the subscript operator has higher precedence.

Multi-dimensional arrays These are actually arrays of pointers, so it is clear to pass them as such:

`// first parameter is an array whose elements are arrays of 10 ints`
`void printValues(int (* matrix)[10], int rowSize);` though it is possible to do it in the following way (where the first dimension has been omitted on purpose because it is not used)

```
// first parameter is an array whose elements are arrays of 10 ints
void printValues(int matrix[][10], int rowSize);
```

Note: It is up to the programmer to manage arrays passed to functions. The most common way of preventing errors such as going beyond bounds are:

- *Put a marker* in the last element of the array itself which denotes the end of the array (an example are C-style character strings which contain the termination character).
- *Use strategies similar to the standard library*: Pass pointers to the begining of the array, and one past the endo of the array

```
void printValues(const int *beg, const int *end)
{
    while (beg != end) {
        cout << *beg++ << endl;
     }
}
int main()
{
    int j[2] = {0, 1};
    // ok: j is converted to pointer to 0th element in j
    //     j + 2 refers one past the end of j
    printValues(j, j + 2);
    return 0;
}
```

- *Explicitely pass a size parameter*

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void printValues(const int ia[], size_t size)
{
     for (size_t i = 0; i != size; ++i) {
         cout << ia[i] << endl;
     }
}
int main()
{
    int j[] = { 0, 1 }; // int array of size 2
    printValues(j, sizeof(j)/sizeof(*j));
    return 0;
}
```

Exercises:

Exercise 7.13: Write a program to calculate the sum of the elements in an array. Write the function three times, each one using a different approach to managing the array bounds.

Exercise 7.14: Write a program to sum the elements in a `vector<double>`.

- Go back to the integrate parabola program. Create a function which takes as argument iterator variables to run through a vector which contains several intervals of integration, and which uses such intervals to repeatedly use IntegrateParabola and average out the result of all integrations.
- Re-write the previous program passing an array by reference.

7.2 The `return` statement

There are two forms of this statement:

No value returned: `return;`
This is used for `void` functions which do not return anything:

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
     // if values already the same, no need to swap, just return
     if (v1 == v2)
         return;
     // ok, have work to do
     int tmp = v2;
     v2 = v1;
     v1 = tmp;
     // no explicit return necessary
}
```

A function of this type can also return a function which also has a `void` return type:

```
void do_swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // ok: void function doesn't need an explicit return
}
void swap(int &v1, int &v2)
{
    if (v1 == v2)
        return false; // error: void function cannot return a value
    return do_swap(v1, v2); // ok: returns call to a void function

}
```

Return a value of the function return type: .
`return statement;`
This must be an expression which is either of the correct type, or can be implicitely converted. The only exception is for the `main()` function, for which a `return 0;` statement may be omitted since the compiler implicitely inserts that.

One can also return a reference, with the WARNING that this cannot be a local object!

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

The same rule applies for a function which returns a pointer, i.e. it should NEVER return a local pointer.
An interesting property of returning references is that they are Lvalues:

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix];
}
int main()
{
    string s("a value");
    cout << s << endl;   // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A

    cout << s << endl;   // prints A value
    return 0;
}
```

Note that functions can recursively call themselves (except for `main()`):

```
// recursive version greatest common divisor program
int rgcd(int v1, int v2)
{
    if (v2 != 0)               // we're done once v2 gets to zero
        return rgcd(v2, v1%v2); // recurse, reducing v2 on each call
    return v1;
}
```

Exercises .

Exercise 7.20: Rewrite `factorial` as an iterative function.

Exercise 7.21: What would happen if the stopping condition in `factorial` were:

```
if (val != 0)
```

7.3 Declaration & default arguments

As we have seen in previous chapters, a function must be declared before it is used. Usually the prototype goes into a header file that is included at the top of the source code file.

Another property of functions is that there may be a list of arguments (they must always be the last ones in the prototype) which have default values. As such, they may be omitted when the function is called. If the first default is omitted, all have to be omitted. For example:

```
string screenInit(string::size_type height = 24,
                  string::size_type width = 80,
                  char background = ' ' );
```

which can be called as:

```
string screen;
screen = screenInit();       // equivalent to screenInit (24,80,' ')
screen = screenInit(66);     // equivalent to screenInit (66,80,' ')
screen = screenInit(66, 256);       // screenInit(66,256,' ')
screen = screenInit(66, 256, '#');
```

Default arguments may be an expression which can be evaluated at run time, and may be specified in the prototype or function definition. It is however best practice to place it in the prototype in headers, because the default can be specified only once.

An example of some of the rules above:

```
string::size_type screenHeight();
string::size_type screenWidth(string::size_type);
char screenDefault(char = ' ');
string screenInit(
    string::size_type height = screenHeight(),
    string::size_type width = screenWidth(screenHeight()),
    char background = screenDefault());
```

7.4 Inline functions

Functions have several advantages as seen above. However, there is the drawback that calling a function is slower than evaluating the corresponding expression. If one wants to avoid this, one can define an inline function (by adding the label `inline` before the data type), which is expanded at each point of the program where it is called. The compiler will usually do so, however it should be kept in mind that this is only a request. Inline function definitions should always appear in headers, and whenever they are changed, the source files where they are included must be recompiled.

Let's look at some examples. Assume that you have the following inline function defined in a header:

```
// inline version: find longer of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
        return s1.size() < s2.size() ? s1 : s2;
}
```

then

```
        cout << shorterString(s1, s2) << endl;
```

would be expanded during compilation into something like

```
        cout << (s1.size() < s2.size() ? s1 : s2)
             << endl;
```

Exercise: An important habit for any programmer is to be able to come up with test programs for the features (s)he is trying to implement.

1. Come up with a test program that uses a function to perform a mathematical operation, contains 2 arguments which must always be suplied and 2 others that are optional and are related to controlling the precision of your calculation. Write the main function such that it tests the case when none, 1 or 2 default arguments are supplied.
2. Think about a repetitive short function that may be used a lot in a specific calculation and write and test and inline function.

7.5 Overloaded functions

These are functions (within a common scope) which have the same name, but different parameter list. Typical examples are the arithmetic operations which act on diferent types. Similarly, one may define our own overloaded functions. Example:

```
Record lookup(const Account&);  // find by Account
Record lookup(const Phone&);    // find by Phone
Record lookup(const Name&);     // find by Name
Record r1, r2;
r1 = lookup(acct);                  // call version that takes an Account
r2 = lookup(phone);                 // call version that takes a Phone
```

The main restrictions are:

- If only the return type differs, then, that's an error
- If both return type and parameter list match then it is just a redeclaration.
- If only non-reference parameters differ by being of `const` type, then it is just a redeclaration.
- The previous point does not apply to reference parameters

One should be careful not to abuse overloading, because the extra information by using different function names may be useful and make the program less obscure.

Let's look at some examples and the way the matching is done (this can become tricky):

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);  // calls void f(double, double)
```

```
void ff(int);
void ff(short);
ff('a');    // char promotes to int, so matches f(int)
```

7.6 Pointers to functions

This is an extremely useful construct that allows for example to define functionals acting on functions. For example, if we want to build an integrator based on a numerical integration rule, we would like to be able to pass a function as an argument to be integrated.

A function pointer, points to a particular type in the same way as a pointer to a data type. It points to a function with certain return type and paramater list types regardless of the function name:

```
// pf points to function returning bool that takes two const string references
bool (*pf)(const string &, const string &);
```

the **parenthesis are essential**, otherwise the star * would refer to the return type which would be a pointer. Because repeating this type of declarations may become cumbersome, creating a typedef by adding the keword to the previous example is useful. The name of the defined type is the one assigned to the pointer to function declaration in that case.

Notes:

- the name of a function is a pointer to that function, so if we use the name without calling the value is a pointer to the function which returns the address of the function.
- a pointer to a function can only be asigned a pointer of the same type or an expression which has a value 0, so there is no conversion between different type pointers to functions.

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(char*, char*);
// pointer to function returning bool taking two const string&
cmpFcn pf;
pf = sumLength;      // error: return type differs
pf = cstringCompare; // error: parameter types differ
pf = lengthCompare;  // ok: function and pointer types match exactly
```

- a function can be called through a pointer to function.

```
typedef bool (*cmpFcn)(const string &, const string &);

// compares lengths of two strings
bool lengthCompare(const string &, const string &);

cmpFcn pf = lengthCompare;
lengthCompare("hi", "bye"); // direct call
pf("hi", "bye");            // equivalent call: pf1 implicitly dereferenced
(*pf)("hi", "bye");         // equivalent call: pf1 explicitly dereferenced
```

- A function parameter can be a pointer to function!

```
    /* useBigger function's third parameter is a pointer to function
     * that function returns a bool and takes two const string references
     * two ways to specify that parameter:
     */
    // third parameter is a function type and is automatically treated as a pointer to
➥ function
    void useBigger(const string &, const string &,
                   bool(const string &, const string &));
    // equivalent declaration: explicitly define the parameter as a pointer to function
    void useBigger(const string &, const string &,
                   bool (*)(const string &, const string &));
```

- One can also return a pointer to function. The explicit declaration is very confusing, so the best strategy is to write a `typedef`

```
// PF is a pointer to a function returning an int, taking an int* and an int
typedef int (*PF)(int*, int);
PF ff(int);  // ff returns a pointer to function
```

- One can asign an overloaded function to a pointer to function, as long as there is a definition with matching argument type and return type.

Exercises

1. Write a function which performs the matrix multiplication of any two matrices for which the product is defined and overload it for all possible built in types that can represent the matrix entries. In your main file use both, arrays which are entered directly, and arrays that are entered by the user, using dynamic memory allocation.

2. Re-Write the Integrate Parabola exercise, so that one of its arguments is a pointer to function, that is the function to be integrated. Write several functions to be integrated using the simple integration rule and write a main program which keeps calling the new `Integrate` routine and returns the integral of the function over the interval.

8 The I/O library

In this section we will see some further features of input/output streams, many of which are common with the `istream` type object `cin` and the `ostream` type object `cout`. We will introduce filestreams and stringstreams to read/write to files and strings. An important property of these new streams is that they are derived from the a base class defining `istream` and `ostream`. This is related to the concept of inheritance which we will not cover. The main idea is that all the operations from the base class are supported by the derived class, so any code we have written for the base class already works for the derived class.

The streams we will now address are summarised in the following table, and inheritance hierarchy

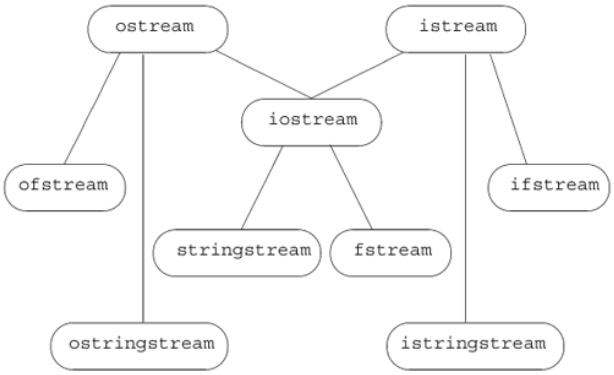
Header	Type
`iostream`	`istream` reads from a stream `ostream` writes to a stream `iostream` reads and writes a stream; derived from `istream` and `ostream`,

`fstream`	`ifstream`, reads from a file; derived from `istream`
	`ofstream` writes to a file; derived from `ostream`
	`fstream`, reads and writes a file; derived from `iostream`

`sstream`	`istringstream` reads from a `string`; derived from `istream`
	`ostringstream` writes to a `string`; derived from `ostream`
	`stringstream` reads and writes a `string`; derived from `iostream`

Table 8.1. IO Library Types and Headers

Figure 8.1. Simplified `iostream` Inheritance Hierarchy



8.1 Condition states

All streams have members (data types or functions) which hold/retrieve information about the state of the stream. We have already seen such an example when testing `cin` for `true` or `false`. However, it is possible to obtain more fine grained information:

strm::iostate	Name of the machine-dependent integral type, defined by each iostream class that is used to define the condition states.
strm::badbit	strm::iostate value used to indicate that a stream is corrupted.
strm::failbit	strm::iostate value used to indicate that an IO operation failed.
strm::eofbit	strm::iostate value used to indicate the a stream hit end-of-file.
s.eof()	true if eofbit in the stream s is set.
s.fail()	true if failbit in the stream s is set.
s.bad()	TRue if badbit in the stream s is set.
s.good()	true if the stream s is in a valid state.
s.clear()	Reset all condition values in the stream s to valid state.
s.clear(flag)	Set specified condition state(s) in s to valid. Type of flag is strm::iostate.
s.setstate(flag)	Add specified condition to s. Type of flag is strm::iostate.
s.rdstate()	Returns current condition of s as an strm::iostate value.

Here is an example of interrogating a stream and managing errors:

```
int ival;
// read cin and test only for EOF; loop is executed even if there are other IO failures
while (cin >> ival, !cin.eof()) {
    if (cin.bad())         // input stream is corrupted; bail out
        throw runtime_error("IO stream corrupted");
    if (cin.fail()) {                       // bad input
        cerr<< "bad data, try again";       // warn the user
        cin.clear(istream::failbit);        // reset the stream
        continue;                           // get next input
    }
    // ok to process ival
}
```

An example of how to access the condition state:

```
// remember current state of cin
istream::iostate old_state = cin.rdstate();
cin.clear();
process_input();  // use cin
cin.clear(old_state); // now reset cin to old state
```

8.2 Output buffer

In the begining of the course we have mentioned that `endl` forces the output buffer holding the output that should be written to the terminal through `cout` to be flushed. The same applies for other output streams, they are held in a buffer which get's flushed when:

1. `endl` is used; `flush` is used; the `unitbuf` manipulator is set, so that after each output line, it is flushed.
2. The program ends normally
3. If the buffer gets full and then it happens automatically

4. If we tie the output stream to an input stream in which case whenever the input stream is used, the output stream is flushed.

Examples:

```
cout << "hi!" << flush;      // flushes the buffer; adds no data
cout << "hi!" << ends;       // inserts a null, then flushes the buffer
cout << "hi!" << endl;       // inserts a newline, then flushes the buffer


cout << unitbuf << "first" << " second" << nounitbuf;
```

is equivalent to writing

```
cout << "first" << flush << " second" << flush;
```

The nounitbuf manipulator restores the stream to use normal, system-managed buffer flushing.

One can tie an output stream to an input stream

```
cin.tie(&cout);   // illustration only: the library ties cin and cout for us
ostream *old_tie = cin.tie();
cin.tie(0); // break tie to cout, cout no longer flushed when cin is read
cin.tie(&cerr);   // ties cin and cerr, not necessarily a good idea!
// ...
cin.tie(0);       // break tie between cin and cerr
cin.tie(old_tie); // restablish normal tie between cin and cout
```

8.3 File streams

The `fstream` header defines three types to support file IO:

1. `ifstream`, derived from `istream`, reads from a file.
2. `ofstream`, derived from `ostream`, writes to a file.
3. `fstream`, derived from `iostream`, reads and writes the same file.

These support all the operations in `iostream` with the addition of `open` and `close` functions, and their own constructors.

To use a file stream, we need to define/declare input objects (just like `cin`) and output objects (like `cout, cerr` or `clog`).

One can either declare first the stream objects we want to use as:

```
ifstream infile;    // unbound input file stream
ofstream outfile;   // unbound output file stream
```

and then bind them to files with a given name

```
infile.open("in");   // open file named "in" in the current directory
outfile.open("out"); // open file named "out" in the current directory
```

or use a constructor that does both (we are *assuming* `ifile` *and* `ofile` *are strings*, note that they msut be converted to C-style strings!):

```
// construct an ifstream and bind it to the file named ifile
ifstream infile(ifile.c_str());
// ofstream output file object to write file named ofile
ofstream outfile(ofile.c_str());
```

Let's look at some common operations:

Checking if open was successful.

```
// check that the open succeeded
if (!infile) {
    cerr << "error: unable to open input file: "
         << ifile << endl;
    return -1;
}
```

Rebinding (re-using) a file stream.

```
ifstream infile("in");      // opens file named "in" for reading
infile.close();             // closes "in"
infile.open("next");        // opens file named "next" for reading
```

Clearing a stream before re-using.

```
ifstream input;
vector<string>::const_iterator it = files.begin();
//   for each file in the vector
while (it != files.end()) {
    input.open(it->c_str());  // open the file
    // if the file is ok, read and "process" the input
    if (!input)
        break;                    // error: bail out!
    while(input >> s) // do the work on this file
        process(s);
    input.close();        // close file when we're done with it
    input.clear();        // reset state to ok
    ++it;                 // increment iterator to get next file
}
```

Neglecting the `clear` would cause only the first file to be read, since the stream would be in an error state or end-of-file. If we re-use a stream we must always clear it!!!!

8.3.1 File modes

The file stream constructor has a default argument to set the files in one of the following modes:

`in`	open for input
`out`	open output
`app`	seek to the end before every write
`ate`	seek to the end immediately after the open
`trunc`	truncate an existing stream when opening it
`binary`	do IO operations in binary mode

Table 8.3. File Modes

where `out, app, trunc` are exclusive of `ofstream, fstream`, and `in` is exclusive of `ifstream, fstream`.

Example:

```
//  output mode by default; truncates file named "file1"
ofstream outfile("file1");
// equivalent effect: "file1" is explicitly truncated
ofstream outfile2("file1", ofstream::out | ofstream::trunc);
//  append mode; adds new data at end of existing file named "file2"
ofstream appfile("file2", ofstream::app);
```

Note that a mode is an aribute of a file not a stream, i.e.

```
ofstream outfile;
// output mode set to out, "scratchpad" truncated
outfile.open("scratchpad", ofstream::out);
outfile.close();    // close outfile so we can rebind it
// appends to file named "precious"
outfile.open("precious", ofstream::app);
outfile.close();
// output mode set by default, "out" truncated
outfile.open("out");
```

Here are some valid combinations of modes:

`out`	open for output; deletes existing data in the file
`out | app`	open for output; all writes at end of file
`out | trunc`	same as out
`in`	open for input
`in | out`	open for both input and output; positioned to read the beginning of the file
`in | out | trunc`	open for both input and output, deletes existing data in the file

Table 8.4. File Mode Combinations

Finally an example of a standard function for opening a file, bind it to an input stream and checking the state.

```
// opens in binding it to the given file
ifstream& open_file(ifstream &in, const string &file)
{
    in.close();     // close in case it was already open
    in.clear();     // clear any existing errors
    // if the open fails, the stream will be in an invalid state
    in.open(file.c_str()); // open the file we were given
    return in; // condition state is good if open succeeded
}
```

8.4 String streams

The `sstream` header defines:

- `istringstream`, derived from `istream`, reads from a `string`.
- `ostringstream`, derived from `ostream`, writes to a `string`.
- `stringstream`, derived from `iostream`, reads and writes a `string`.

Note that only the functions which are common with `iostream` are allowed, not the ones that `fstream` supports! The extra operations supported are:

`stringstream strm;`	Creates an unbound `stringstream`.
`stringstream strm(s);`	Creates a `stringstream` that holds a copy of the `string s`.
`strm.str()`	Returns a copy of the `string` that `strm` holds.
`strm.str(s)`	Copies the `string s` into `strm`. Returns `void`.

Table 8.5. `stringstream`-Specific Operations

Using stringstream.

```
string line, word;      // will hold a line and word from input, respectively
while (getline(cin, line))   {            // read a line from the input into line
   // do per-line processing
   istringstream stream(line);            // bind to stream to the line we read
   while (stream >> word){          // read a word from line
       // do per-word processing
   }
}
```

Using stringstream for conversion/Formatting We can use string streams to convert numeric values to a string:

```
int val1 = 512, val2 = 1024;
ostringstream format_message;
// ok: converts values to a string representation
format_message << "val1: " << val1 << "\n"
               << "val2: " << val2 << "\n";
```

And to extract back:

```
// str member obtains the string associated with a stringstream
istringstream input_istring(format_message.str());
string dump; // place to dump the labels from the formatted message
// extracts the stored ascii values, converting back to arithmetic types
input_istring >> dump >> val1 >> dump >> val2;
cout << val1 << " " << val2 << endl;  // prints 512 1024
```

EXERCISES:

Exercise 8.3: Write a function that takes and returns an `istream&`. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid and return the stream.

Exercise 8.4: Test your function by calling it passing `cin` as an argument.

Exercise 8.6: Because `ifstream` inherits from `istream`, we can pass an `ifstream` object to a function that takes a reference to an `istream`. Use the function you wrote for the first exercise to read a named file.

Exercise 8.9: Write a function to open a file for input and read its contents into a `vector` of `strings`, storing each line as a separate element in the `vector`.

Exercise 8.10: Rewrite the previous program to store each word in a separate element.

Exercise 8.13: Write a program similar to `open_file` that opens a file for output.

Exercise 8.16: Write a program to store each line from a file in a `vector<string>`. Now use an `istringstream` to read each line from the `vector` a word at a time.

9 Classes

We have already seen that classes allow us to define our own data types, and member functions associated with that abstract type we wish to define. They also allow for a way to encapsulate the details of the implementation, so that once we have defined a class, it behaves as an independent unit which is easier to debug and optimise, so that it can be later used by any program. Another advantage is that the implementation and the interface are done separately, so that if we have to change the implementation due to bugs, performance, etc... the client code stays the same. In this section we will look at some further features of class definition, and how to control further the data types created and their behaviour.

9.1 Recap and some further features

Let us look at the `Sales_item` class that was mentioned in the begining of the course.

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

There are several things in this class that we have already talked about, such as the keywords `public` and `private` and the definition of member data and functions. Anoter thing to recall is that if a function is `const`, it means that it will not change the values of the data members of the class.

The constructor One novelty of the example above is that there is a member function with the same name as the class. This is called the constructor and it is a function which can take no arguments or a number of arguments. Besides the special property of having the same name as the class, it contains a

colon and an initializer list before the function arguments, which defines the initial values of the data of the class. Finally a constructor has no return type.

Constructor functions can be overloaded, and the definition that it used is determined by the arguments provided to the constructor when an object of that class type is declared.

Example of overloading:

```
class Sales_item;
// other members as before
public:
    // added constructors to initialize from a string or an istream
    Sales_item(const std::string&);
    Sales_item(std::istream&);
    Sales_item();
};
```

and its use

```
// uses the default constructor:
// isbn is the empty string; units_soldand revenue are 0
Sales_item empty;
// specifies an explicit isbn; units_soldand revenue are 0
Sales_item Primer_3rd_Ed("0-201-82470-1");
// reads values from the standard input into isbn, units_sold, and revenue
Sales_item Primer_4th_ed(cin);
```

Note: Similarly to the constructor, other member functions can also be overloaded, provided a suitable definition is provided. Another point to keep in mind is that member functions which are defined inside the class definition are implicitely inline. It is also possible to declare them explicitly inline as usual.

Besides being called in the forms above, we can also call the constructor to dynamically allocate memory by using the class name

```
// constructor that takes a string used to create and initialize variable
Sales_item Primer_2nd_ed("0-201-54848-8");
// default constructor used to initialize unnamed object on the heap
Sales_item *p = new Sales_item();
```

Typedefs Using typedefs is useful for two reasons:

- Firstly it provides a better way to name types which have to be resolved for several scopes.
- Secondly, the user of the class may just use those internal types defined in the classs without the worry of how they are implemented, and also the implementation itself may change later, without the code using the class. For example:

```
class Screen {
public:
    // interface member functions
    typedef std::string::size_type index;
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

Forward declarations It is possible to declare a class without defininig it. This is called a forward declaration

```
class ClassName;
```

This can be used only in a limited number of ways, since their members are not yet defined. Forward declarations *can only be used to define pointers or references to that class type*.

Exercises .

Exercise 12.1: Write a class named `Person` that represents the name and address of a person. Use a `string` to hold each of these elements.

Exercise 12.2: Provide a constructor for `Person` that takes two `strings`.

Exercise 12.3: Provide operations to return the name and address. Should these functions be `const`? Explain your choice.

Exercise 12.4: Indicate which members of `Person` you would declare as `public` and which you would declare as `private`. Explain your choice.

Exercise 12.11: Define a pair of classes X and Y, in which X has a pointer to Y, and Y has an object of type X.

Exercise 12.19: Provide one or more constructors that allows the user of this class to specify initial values for none or all of the data elements of this class:

```
class NoName {
public:
    // constructor(s) go here ...
private:
    std::string *pstring;
    int         ival;
    double      dval;
};
```

Explain how you decided how many constructors were needed and what parameters they should take.

Exercise 12.20: Choose one of the following abstractions (or an abstraction of your own choosing). Determine what data is needed in the class. Provide an appropriate set of constructors. Explain your decisions.

```
(a) Book        (b) Date      (c) Employee
(d) Vehicle    (e) Object    (f) Tree
```

9.2 The implicit `this` pointer

Member functions have an extra implicit parameter which is a pointer to the class type object to which they are bound. It is called the `this` pointer, and it may be thought of as a pointer to the class type object which calls the member function.

Let's look at an example of a Screen class with the following data:

```
class Screen {
public:
    // interface member functions
private:
    std::string contents;
    std::string::size_type cursor;
    std::string::size_type height, width;
};
```

and the following member functions:

```
  class Screen {
  public:
      typedef std::string::size_type index;
      // implicitly inline when defined inside the class declaration
      char get() const { return contents[cursor]; }
      // explicitly declared as inline;
will be defined outside the class declaration
      inline char get(index ht, index wd) const;
      // inline not specified in class declaration, but can be defined inline later
      index get_cursor() const;
      // ...
   };
  // inline declared in the class declaration; no need to repeat on the definition
  char Screen::get(index r, index c) const
  {
      index row = r * width;    // compute the row location
      return contents[row + c]; // offset by c to fetch specified character
  }
  // not declared as inline in the class declaration, but ok to make inline
in definition
  inline Screen::index Screen::get_cursor() const
  {
      return cursor;
  }
```

Let's assume we add the following to the class definition

```
class Screen {
public:
     // interface member functions
     Screen& move(index r, index c);
     Screen& set(char);
     Screen& set(index, index, char);
     // other members as before
};
```

```
Screen& Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}
Screen& Screen::move(index r, index c)
{
    index row = r * width; // row location
    cursor = row + c;
    return *this;
}
```

The `this` pointer is `const`, so the object address to which `this` is bound cannot be changed, though the object itself can be changed. If the member function is itself `const`, then neither can be changed.

Using the class definition above, now we can chain a serie of actions on the same screen:

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

and an example of when we have to be careful with `const` functions, if we define the function `display`

```
Screen myScreen;
// this code fails if display is a const member function
// display return a const reference; we cannot call set on a const
myScreen.display().set('*');
```

as `const`
This can be solved by overloading the function:

```
class Screen {
public:
    // interface member functions
    // display overloaded on whether the object is const or not
    Screen& display(std::ostream &os)
                  { do_display(os); return *this; }
    const Screen& display(std::ostream &os) const
                  { do_display(os); return *this; }
private:
     // single function to do the work of displaying a Screen,
     // will be called by the display operations
     void do_display(std::ostream &os) const
                       { os << contents; }
     // as before
 };
```

Now, when we embed `display` in a larger expression, the nonconst version will be called. When we `display` a `const` object, then the `const` version is called:

```
Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); // calls nonconst version
blank.display(cout);             // calls const version
```

9.3 Some scope rules

Since class member functions are often defined outside the class definition, it is important to note some scope rules. We have already seen that members are accessed through the dot operator.

When defining a member function one needs to specify the scope of the function in its name, and then everything following is in the same scope. For example:

```
char Screen::get(index r, index c) const
{
    index row = r * width;      // compute the row location

    return contents[row + c];   // offset by c to fetch specified character
}
```

Note that the return type is not after such scope resolution, so it may not be in class scope! For example:

```
class Screen {
public:
    typedef std::string::size_type index;
    index get_cursor() const;
};
inline Screen::index Screen::get_cursor() const
{
    return cursor;
}
```

Further rules for name lookup are as follow:

- Declarations in the class containing other defined type members (other class types for example):

1. First definitions within the current class scope which are before the use of the name are looked up.
2. If the previous step fails, types which are defined in the current class scope after, and types which are defined in global scope, before the class definition are looked up.
   Example:

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

- Name lookup in Class member function definitions:
  1. First check the local scope of the function.
  2. Then check declarations for all members of the class to which the function belongs.
  3. Finally if not found, declarations that appear before the member function are checked.

Exercise .

Extend your version of the Screen class to include the move, set, and display operations. Test your class by executing the expression:

Exercise 12.13: [View full width]

```
     //
move cursor to given position, set that character and
➥ display the screen
     myScreen.move(4,0).set('#').display(cout);
```

Exercise 12.14: It is legal but redundant to refer to members through the this pointer. Discuss the pros and cons of explicitly using the this pointer to access members.

9.4 Friends

This is a mechanism which allows a class type, or function, which are not members of the class to have access to the private members of the class. Such "friends" are declared anywhere in the class by writing the keword `friend` followed by the class or the function name which are granted access to the private members of the class.

Example: Consider the `Screen` class introduced before. We might want to have a window manager, that manages several Screens on a display, which will have to have access to private members of the Screen class

```
class Screen {
    // Window_Mgr members can access private parts of class Screen
    friend class Window_Mgr;
    // ...restofthe Screen class
};
```

so that the window manager may use the private members as follows

```
Window_Mgr&
Window_Mgr::relocate(Screen::index r, Screen::index c,
                     Screen& s)
{
     // ok to refer to height and width
     s.height += r;
     s.width += c;

     return *this;
}
```

A friend can be a non-member function, an entire class, or a function of a class that was defined before. It is usually a good idea to restrict friend access, so the latter is sometimes useful. An example of making a function which is member of another class, a friend would be in the previous example, to give access only to a function of the window manager:

```
class Screen {
    // Window_Mgrmust be defined before class Screen
    friend Window_Mgr&
        Window_Mgr::relocate(Window_Mgr::index,
                             Window_Mgr::index,
                             Screen&);
    // ...restofthe Screen class
};
```

An important point to keep in mind in this example, is that we must be careful about the order in which these definitions are provided. This is because a friend function, must know about the class members with which it is friend. So in this example, we would need:

- first `Window_Mgr` would have to be defined, together with a declaration of their members, so that the `Screen` class can declare `relocate` as a friend
- Then `Screen` has to be defined, before `relocate` is defined, because the members of `Screen` will have to be known for the definition of `relocate`
- Finally `relocate` can be defined.

Another important note regarding overloaded functions which are friends of a class. The rule is that for every overloaded version, a `friend` declaration must be present. Example:

```
// overloaded storeOn functions
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);
class Screen {
    // ostream version of storeOn may access private parts of Screen objects
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};
```

Exercises .

Exercise 12.34: Define a nonmember function that adds two `Sales_item` objects.

Exercise 12.35: Define a nonmember function that reads an `istream` and stores what it reads into a `Sales_item`.

9.5 Static Class members

Static Class members, are members which are shared among all class objects that are defined. This might be useful for example to keep a count of objects of a certain class type that have been defined.

In some sense, these are members which are global to all class objects, while not being accessible generically in the user code. Static objects can be data or member functions. `static` member functions do not have a `this` pointer because they are not bound to any particular object.

The main advantages of using static members rather than globals are as follows:

1. The name of a static member is in the scope of the class, thereby avoiding name collisions with members of other classes or global objects.
2. Encapsulation can be enforced. A static member can be a private member; a global object cannot.
3. It is easy to see by reading the program that a static member is associated with a particular class. This visibility clarifies the programmer's intentions.

Let us look at an example of static member declaration with public and private members, where the class is bank account with an owner:

```
class Account {
public:
    // interface functions here
    void applyint() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double); // sets a new rate
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};
```

and its usage:

```
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
double rate;
rate = ac1.rate();        // through an Account object or reference
rate = ac2->rate();       // through a pointer to an Account object
rate = Account::rate();   // directly from the class using the scope operator
```

Note:

- Just like other members, static members can be accessed inside the class definitions without the scope resolution operator.
- The `static` keyword appears only inside the definition in the class, where the member is declared, and if the member is defined outside, the keyword is NOT repeated. In the example above:

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```

- Static members must be defined exactly once, because unlike other members they are not initialised by a constructor.
- Static data members cannot be initialised in the class definition like other data members, with the exception of integral `static const`, if a constant expression

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);  // sets a new rate
private:
    static const int period = 30; // interest posted every 30 days
    double daily_tbl[period]; // ok: period is constant expression
};
```

- Another interesting property is that, since static members are not part of the object, but instead are shared, they can be of the class type itself

```
class Bar {
public:
    // ...
private:
    static Bar mem1; // ok
    Bar *mem2;       // ok
    Bar mem3;        // error
};
```

Similarly, a static data member can be used as a default argument:

```
class Screen {
public:
    // bkground refers to the static member
    // declared later in the class definition
    Screen& clear(char = bkground);
private:
    static const char bkground = '#';
};
```

Exercise 12.38: Define a class named Foo that has a single data member of type int. Give the class a constructor that takes an int value and initializes the data member from that value. Give it a function that returns the value of its data member.

Exercise 12.39: Given the class Foo defined in the previous exercise, define another class Bar with two static data elements: one of type int and another of type Foo.

Exercise 12.40: Using the classes from the previous two exercises, add a pair of static member functions to class Bar. The first static, named FooVal, should return the value of class Bar's static member of type Foo. The second member, named callsFooVal, should keep a count of how many times xval is called.

Exercise 12.41: Given the classes Foo and Bar that you wrote for the exercises to Section 12.6.1 (p. 470), initialize the static members of Foo. Initialize the int member to 20 and the Foo member to 0.

9.6 Constructors, copy control and destructors

9.6.1 More on constructors

We have seen that constructors are member functions which have no return type and are executed automatically when a class type object is declared. A special feature is the initialiser list which is run to initialise the class objects. Some properties are as follows:

- If the initialiser list is not provided, the class will initialise its data members using the default constructor for each member.
- If there is no default constructor for a member, then an initialiser is required.
- The initialiser expression can be any valid expression
- The order in which members are initialised follows the order of declaration in the class. This is relevant when a member is initialised in terms of other members.

```
class X {
    int i;
    int j;
public:
    // run-time error: i is initialized before j
    X(int val): j(val), i(j) { }
};
```

- As for any other function, the constructor can have default arguments. This can be particularly useful to create a class with default constructor where one of the members has no default constructor

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
              isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
};
```

The default constructor If there is no default constructor being defined, the compiler tries to define a synthesized default constructor. It is usually good practice to define the constructor instead of relying on this default behaviour whenever other constructors are defined. The default constructor is not called with parethesis!!!, only the object name is used:

```
Sales_item myobj();   // ok: but defines a function, not an object
if (myobj.same_isbn(Primer_3rd_ed))   // error: myobj is a function
```

Implicit conversions Another useful behaviour to be aware of, is that a constructor with a single parameter, defines an implicit conversion from the parameter type to the class type.

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
              isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
 };
```

Each of these constructors defines an implicit conversion. Accordingly, we can use a `string` or an `istream` where an object of type `Sales_item` is expected:

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_itemwith 0 units_soldand revenue from
// and isbn equal to null_book
item.same_isbn(null_book);
```

This behaviour can be avoided by declaring the corresponding constructor `explicit`, the only restriction being that this keyword can appear only inside the class definition:

```
class Sales_item {
public:
    // default argument for book is the empty string
    explicit Sales_item(const std::string &book = ""):
              isbn(book), units_sold(0), revenue(0.0) { }
    explicit Sales_item(std::istream &is);
    // as before
};
```

so that when used...

```
item.same_isbn(null_book); // error: string constructor is explicit
item.same_isbn(cin);       // error: istream constructor is explicit
```

The explicit keyword means that we can use the constructor in an initialisation, as long as it is done explicitely, i.e. by using the class name to call the constructor:

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_itemwith 0 units_soldand revenue from
// and isbn equal to null_book
item.same_isbn(Sales_item(null_book));
```

9.6.2 Copy control

The copy constructor This is a constructor with a single parameter which is usually a `const` reference to an object of the same type as the class.

```
class Foo {
public:
   Foo();           // default constructor
   Foo(const Foo&); // copy constructor
   // ...
};
```

It can be used to:

- Explicitly or implicitly initialise an object from another of the same type. Some examples contrasting copy-initialisation with direct initialisation

  ```
  string null_book = "9-999-99999-9"; // copy-initialization
  string dots(10, '.');               // direct-initialization

  string empty_copy = string();       // copy-initialization
  string empty_direct;                // direct-initialization
  ```

  Note that copy initialisation is only ok when the class type suports copy. For example `istream` variables do not allow copy:

  ```
  ifstream file1("filename"); // ok: direct initialization
  ifstream file2 = "filename"; // error: copy constructor is private
  // This initialization is okay only if
  // the Sales_item(const string&) constructor is not explicit
  Sales_item item = string("9-999-99999-9");
  ```

- Copy a non-reference object to be returned by a function, or similarly, a non-reference parameter of a function

  ```
  // copy constructor used to copy the return value;
  // parameters are references, so they aren't copied
  string make_plural(size_t, const string&, const string&);
  ```

- Initialise elements of a container or a list of elements in an array: An example for a vector:

```
// default string constructor and five string copy constructors invoked
vector<string> svec(5);
```

and with an array, where copy initialisation is used

```
Sales_item primer_eds[] = { string("0-201-16487-6"),
                            string("0-201-54848-8"),
                            string("0-201-82470-1"),
                            Sales_item()
                          };
```

If we do not provide a copy constructor, the compiler will synthesize one for us, which simply copies member by member. For example:

```
class Sales_item {
// other members and constructors as before
private:
    std::string isbn;
    int units_sold;
    double revenue;
};
```

the synthesized Sales_item copy constructor would look something like:

```
Sales_item::Sales_item(const Sales_item &orig):
    isbn(orig.isbn),              // uses string copy constructor
    units_sold(orig.units_sold),  // copies orig.units_sold
    revenue(orig.revenue)         // copy orig.revenue
    {    }                        // empty body
```

The assignment operator This is an overloaded operator (=) which is responsible for assigning an object of class type to another. We will look at overloaded operators next, an example is:

```
class Sales_item {
public:
    // other members as before
    // equivalent to the synthesized assignment operator
    Sales_item& operator=(const Sales_item &);
};
```

Similarly to the copy construct, there is a synthesized assigment operator that assigns member by member. For `Sales_item` an assignment operator could be

```
// equivalent to the synthesized assignment operator
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    isbn = rhs.isbn;              // calls string::operator=
    units_sold = rhs.units_sold;  // uses built-in int assignment
    revenue = rhs.revenue;        // uses built-in double assignment
    return *this;
}
```

The Destructor This is another special member function which is responsible for deallocating an object. This is called automatically whenever an object of the class type is destroyed

```
// p points to default constructed object
Sales_item *p = new Sales_item;
{
                          // new scope
    Sales_item item(*p);  // copy constructor copies *p into item
    delete p;             // destructor called on object pointed to by p
}                         // exit local scope; destructor called on item
```

Note that the destructor is not run on a reference or pointer to a dynamically allocated object that goes out of scope (a delete must be done explicitly). Similarly, for containers:

```
{
    Sales_item *p = new Sales_item[10]; // dynamically allocated
    vector<Sales_item> vec(p, p + 10);  // local object
    // ...
    delete [] p; // array is freed; destructor run on each element
 }   // vec goes out of scope; destructor run on each element
```

As a general rule, usually an explicit definition of a destructor is needed whenever copy or assignment are needed. If not defined, the compiler always defines one for us which deletes each element in reverse order. The object to which a pointer member points to is not destroyed, only the pointer itself.

The form of a destructor is similar to a constructor except that there is no return type or parameters and a tilde ~ is added:

```
class Sales_item {
public:
   // empty; no work to do other than destroying the members,
   // which happens automatically
    ~Sales_item() { }
   // other members as before
};
```

Exercises .

Exercise 12.23: Assume we have a class named `NoDefault` that has a constructor that takes an `int` but no default constructor. Define a class `C` that has a member of type `NoDefault`. Define the default constructor for `C`.

Exercise 12.30: Compile the following code:

```
f(const vector<int>&);
int main() {
    vector<int> v2;
    f(v2);  // should be ok
    f(42);  // should be an error
    return 0;
}
```

What can we infer about the `vector` constructors based on the error on the second call to `f`? If the call succeeded, then what would you conclude?

Exercise 13.4: Given the following sketch of a class, write a copy constructor that copies all the elements. Copy the object to which `pstring` points, not the pointer.

```
struct NoName {
    NoName(): pstring(new std::string), i(0), d(0) { }
private:
    std::string *pstring;
    int    i;
    double d;
};
```

Exercise 13.10: Define an `Employee` class that contains the employee's name and a unique employee identifier. Give the class a default constructor and a constructor that takes a `string` representing the employee's name. If the class needs a copy constructor or assignment operator, implement those functions as well.

9.7 Overloading operators

This allows for a behaviour of our class types, which are similar to the built in types. This should be used with care when there is an obvious an natural reason to overload an operator. For example, overloading the + operator to represent division o members would clearly be a bad idea. The most common overloaded operators are the shift operators. As an example of usefulness of overloading, compare

```
cout << "The sum of " << v1 << " and " << v2
          << " is " << v1 + v2 << endl;
```

to the more verbose code that would be necessary if IO used named functions:

```
// hypothetical expression if IO used named functions
cout.print("The sum of ").print(v1).
       print(" and ").print(v2).print(" is ").
       print(v1 + v2).print("\n").flush();
```

Overloaded operators are functions which special names with the keyword `operator` and a parameter list corresponding to the operands. For example, the declaration of an overloaded + operator in the class `Sales_item` would be

```
class Sales_item;
// other members as before
public:
    // added constructors to initialize from a string or an istream
    Sales_item(const std::string&);
    Sales_item(std::istream&);
    Sales_item();
};
```

The operators that are allowed to be overloaded are:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Table 14.1. Overloadable Operators

The main rules for overloading operators are:

- At least one of the operands of the overloaded operator must be of class type, otherwise we would be overloading a built in operator! Similarly we cannot defined new operators for built in types.
- Precedence and associativity cannot be changed (it is the same as for built in types)
- Short circuit evaluation is not preserved
- Class member operator seem to have one less parameter, which however corresponds to the `this` pointer

```
// member binary operator: left-hand operand bound to implicit this pointer
Sales_item& Sales_item::operator+=(const Sales_item&);
// nonmember binary operator: must declare a parameter for each operand
Sales_item operator+(const Sales_item&, const Sales_item&);
```

- Often, overloaded operators which are non-member functions are made friends to classes. An example of such is:

```
class Sales_item {
    friend std::istream& operator>>
                    (std::istream&, Sales_item&);
    friend std::ostream& operator<<
                    (std::ostream&, const Sales_item&);
public:
    Sales_item& operator+=(const Sales_item&);
};
    Sales_item operator+(const Sales_item&, const Sales_item&);
```

Exercise: Write a class which has a dynamically allocated pointer to hold a position in N-dimensions. Include a constructor and default destructor and overloaded input and output operators.