CONCENTRATED C++

An Introduction to the Language
(2nd Edition)

Adrian P. Robson
The University of Northumbria at Newcastle

June 1994

Contents

Chapter 1

INTRODUCTION

This is a basic introduction to Object Oriented Programming with C++. An ability to design and write programs in a modern procedural language like Ada, Modula-2 or Pascal is assumed. However, C experience is not required.

1.1 Object Oriented Programming

- Object oriented programming approaches the design of software by building a program from a number of objects that communicate by exchanging messages.

  This promotes high cohesion and low coupling.

- Objects are defined by classes. Any number of objects of a class can be created.

- A class has a data part, data members in C++, and a set of messages, member functions in C++.

- A Class defines a public interface for a object in the form of a set of messages to which it will respond.

  - The implementation of the class is hidden. An object's user does not have know about its internal working.
  - This separation of interface and implementation is called abstraction. Classes support abstract data types.

- New classes can be defined by inheriting the attributes of existing classes.

  Inheritance makes it easier for a programmer to express commonality among classes. It promotes code reuse.

- Object oriented programming supports a form of polymorphism.

  This allows different objects to understand the same message, although they may respond to it in different ways. We can send such a message to an object without knowing its exact type.

- Classes that are defined with parameterised attributes are call generic classes.

  These are used as general purpose or utility classes. C++ has template classes for this purpose.

- The features that identify an object oriented language are:
  - Objects
  - Classes
  - Abstraction
  - Inheritance
  - Polymorphism and Generic Classes

- Stroustrup defines the object oriented programming paradigm as:

  > *"Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance."*

- C++ is a hybrid OOL, smalltalk is a pure OOL

1.2 History of C++

- C++ was developed by Bjarne Stroustrup, who works for AT&T Bell Telephone Laboratories.
- Originally implemented in 1982 as "C with Classes".
- Used by AT&T researchers in 1984.
- Distributed to Universities and throughout AT&T in early 1985.
- AT&T C++ translator 1.0 released to the public in late 1985.
- Commercial compilers began to appear in 1988, Zortec and GNU
- C++ moved into the mainstream of programming languages with Release 2.0 in June 1989

  Major manufactures provide C++ products for their platforms. Borland release a Turbo C++ Compiler.

- ANSI C++ Committee formed in late 1989.

  They approved templates and exception handling.

- In late 1991 template implementations were provided by Borland and by AT&T/USL Cfront Release 3.0. Others soon followed.
- At the moment (1993) no implementations supporting exception handling are available.

1.3 Influences

The design of C++ was effected by many other computer languages:

- C provided the basic syntax and semantics for C++.
- Classes came from Simula67.

  Simula's lack of performance inspired Stroustrup to develop C++.
- Operator overloading and embedded declarations are features of Algol68.
- Ada and Clu influenced C++ templates.
- Ada, Clu and ML influenced the design of C++ exception handling.
- One explanation for the name C++ is that ++ is the C incremental operator.

1.4 C and C++

C++ supports ANSI C (with minor exceptions).

- It is link compatible with the standard C libraries.
- Every program in Kernighan and Richie "The C Programming Language, 2 Edition" will compile under C++.

1.5 Overview

This document is divided into a number of sections:

1. The basics:

   This introduces the C++ language without classes. It purpose is to show programmers with experience of a good procedural language, like Pascal, how to write equivalent programs in C++.

2. Classes in C++:

   This introduces the concept of classes. It shows how class objects can be used to build programs.

3. Inheritance:

   This extends the previous section by showing how inheritance can be used to build rich class hierarchies. Single and multiple inheritance and some aspects of C++ design style are considered.

4. Virtual Functions:

   This looks at some of the more advanced aspects of C++ programming. The concept of Polymorphism is introduced. Virtual functions and pointers to objects are presented as a method of supporting polymorphism in C++.

5. Templates:

   Generic classes and container classes are discussed. C++ templates are introduced to support these concepts.

6. Input and Output:

   The use of the C++ iostream has already been introduced. This section provides much more detail and explains how named files can be used.

7. Object Oriented Design:

   This reviews the object oriented ideas that have been introduced throughout the course. It looks at how software can be designed in an object oriented way, so that it can be easily implemented in a language like C++.

8. Additional material:

   A number of addition sections are provided for reference.

The examples in this document were compiled using a Borland C++ compiler.

1.6 Reading List

Graham, N., *Learning C++,* McGraw-Hill, 1991.

A very good, easy to read introductory book. It does not cover templates or exceptions.

Winder, S., *Developing C++ Software, 2nd Edition,* Wiley, 1993.

A good introduction to C++ and object oriented programming. However, its coverage of inheritance is complicated.

Stroustrup, B., *The C++ Programming Language, 2nd Edition,* Addison-Wesley, 1991.

A rich, in depth introduction to C++. It contains the C++ reference manual.

Lippman, S., *A C++ Primer,* 2nd Edition, Addison-Wesley, 1992.

An alternative introductory book. Stan Lippman works for AT&T and wrote parts of their C++ compiler.

Ellis, M.A. and B. Stroustrup, *The annotated C++ Reference Manual,* Addison-Wesley.

Very heavy reading but very comprehensive. Not an introductory text.

Booch, G., *Object-Oriented Design with Applications,* Benjamin-Cummings, 1991.

An excellent book on OOD. Good background reading.

Coad, P., and E. Yourdon, *Object-Oriented Analysis 2nd Edition,* Prentice-Hall, 1991.

This book describes one version of OOA. Good background reading.

Chapter 2

THE BASICS

This section begins an introduction of the C++ language. Classes, which are fundamental to C++ programs, are covered in the next section.

2.1 Example: Hello World

```
// File: hello.cpp
// Hello word program

#include <iostream.h>

main()
{
   cout << "Hello World\n";
}
```

- The program is stored in a file called hello.cpp. The file type cpp is the default for the Borland C++ compiler.
- The lines beginning with // are comments.
- The line

  `#include <iostream.h>`

  allows us to use the standard C++ terminal I/O library. `#include` is a preprocessor directive.
- `main()` is a function.
  - The system executes a C++ program by calling this function. All C++ programs must have a function called main.

    The empty parentheses `( )` indicate that main takes no arguments.
  - The braces `{` and `}` enclose a block.

    A block is the basic unit for grouping declarations and statements.

    The statements defining a function are always enclosed in a block.

- The line

  `cout << "Hello World\n";`

  is a statement. It writes

  `Hello World`

  on the terminal screen.

- `"Hello World\n"` is a string literal.

  `\n` is an escape sequence. It causes the output to start on a new line.

- All statements must end with a semi-colon.

2.2 Comments

- Any text following `//`, until the end of the line is a comment.
- Comments can also be enclosed in `/*  */` like this:

```
// this is a comment line
   a = b;  // this is a comment at the end of a line

/* this is also
   a
   comment */

this is an error
   a = b;     so is this
```

2.3 Names

- Names (identifiers) in C++ consist of letters, digits and the underscore character `_`.
- Names must begin with a letter or an underscore.
- C++ is case sensitive.
- All names must be declared before they can be used. A declaration associates a name with a type:

```
int count = 0;
float range = 3.6;
char ans;
int max( int a, int b);
```

2.4 The fundamental types

2.4.1 Integer numbers

- Integers types, in increasing size, are:

  ```
  char
  short int or short
  int
  long int or long
  ```

- Signed and unsigned integers:
  - All the integer types can also be signed or unsigned. For example, `unsigned short int`.
  - Unsigned integers cannot represent negative numbers but they can represent double the number of positive values.
  - The signed `int` types are synonyms for their plain versions, so `signed int` is the same as `int`.
- The normal integer type for arithmetic is `int`.
- The `char` type is used for storing single characters.

  The character types `char`, `unsigned char` and `signed char` are distinct.
- In general, the size of an integer type is implementation dependent.

  The compiler's file `limits.h` contains information about the minimum and maximum values that can be stored.

2.4.2 Real numbers

- The varieties of real numbers, in order of increasing precision, are:

  ```
  float
  double
  long double
  ```

2.5 Constants

2.5.1 Integers

- Integer constants are written like this:

  `1249234  99  0  345U  16777215L  45UL  0XFF  010`

  Commas are not allowed.

  Decimal, hexadecimal or octal notation can be used. The number twelve can be written as `12`, `0XC` or `014`.

  A decimal number must not begin with a zero.

- The type of an integer constant depends on its form, value, and suffix. The most compact internal representation is used. Thus, the type of a constant is chosen in the following order.

  A decimal with no suffix: `int`, `long int` then `unsigned long int`.

  An octal or hexadecimal with no suffix: `int`, `unsigned int`, `long int` then `unsigned long int`.

  Suffixed by U: `unsigned int` then `unsigned long int`.

  Suffixed by L: `long int` then `unsigned long int`.

  Suffixed by UL or LU: It is `unsigned long int`.

2.5.2 Characters

- The character types are:

```
char
unsigned char
```

- Character constants are written like this:

  `'a'  'D'  ''  '\n'  '\xaa'  '"'`

- Escape sequences can be used to specify single characters:

new line	`\n`
horizontal tab	`\t`
backspace	`\b`
carriage return	`\r`
alert (bell)	`\a`
form feed	`\f`
back slash	`\\`
single quote	`\'`
double quote	`\"`
hex number	`\xhh`

2.5.3 Real numbers

- Real or floating point constants are written as follows:

  `3.127  .001  5.6E3  2e5L  4e-10F`

  An `F` suffix indicates a `float` and `L` a `long double`. The type is `double` if no suffix is given.

2.5.4 Strings

- String literals are sequences of characters in double quotes:

  `"This is a string literal. It will sound the bell \a"`

2.5.5 Enumerations

- Enumerations are integral types with named constants:

  `enum Grade { GOOD, AVERAGE, POOR };`

  The name `GOOD` represents the value 0, `AVERAGE` represents 1 and `POOR` represents 2.

  Conventional C++ style is to use capitals for the names of constants.

- They can be given specific values:

  `enum Grade { GOOD = 100, AVERAGE = 50, POOR = 0 };`

- Enumerated type values are converted to type int before any operations are carried out on them.

2.6 The iostream

2.6.1 Output

- Use the `<<` operator with `cout`:

  `cout << xxx;`

- Useful manipulators:

`oct`	convert to octal
`dec`	convert to decimal
`hex`	convert to hexadecimal
`endl`	add `\n` and flush
`ends`	add `\0` and flush
`flush`	flush stream
`setw( int w )`	set output field width

  Remember to include `<iomanip.h>` if `setw` is used.

2.6.2 Input

- Use the `>>` operator with `cin`:

  `cin >> xxx;`

- Useful manipulators:

`eatwhite`	eat white space
`setw( int w )`	limit input field width

  White space is a blank, tab, new line, form feed or carriage return.

  Remember to include `<iomanip.h>` if —setw— is used.

2.6.3 Example: Print some constants

```
// File: contest.cpp
// Experiment with C++ constants

#include <iostream.h>
```

```
enum Grade { GOOD, AVERAGE, POOR };

main()
{
   cout << "I said \"Lets look at some constants\".\n";
   cout << "This is a two line string\nSecond line\n";
   cout << "But this is "     "all on the same line\n";
   cout << "Some integers are " << 0xff << ", "
        << -1 << ", " << -1U << " and "
        << hex << -1U << '\n';
   cout << "Some characters are " << '\x61'
        <<  " = " << 'a' << '\n';
   cout << "Now some floating point values\n"
        << 2.5 << ' ' << 0.25E10 << ' ' << 9E20 << ' '
        << 123456789.0 << endl;
   cout << "Grades are " << GOOD << ' '
        << AVERAGE << ' '<< POOR << endl;

   char answer;
   cout << "Give yes(y) or no(n): ";
   cin >> answer;
   cout << "You said " << answer;
}
```

This program produces the following output:

```
I said "Lets look at some constants".
This is a two line string
Second line
But this is all on the same line
Some integers are 255, -1, 65535 and ffff
Some characters are a = a
Now some floating point values
2.5 2.5e+09 9e+20 1.234567e+08
Grades are 0 1 2
Give yes(y) or no(n): X
You said X
```

2.7 Data Objects

- A data object is an area in memory where information is stored. It is a variable.
  - A variable has an address which is its location in memory. Addresses can be stored and manipulated in pointers.
  - A variable can have zero, one or more names.
  - A variable has a type which says how big it is and how its internal bit pattern is interpreted.
  - A variable has a value.

2.7.1 Using data objects

- An variable must be declared before it can be used:

  ```
  int count;
  float total_value;
  int x, y, z;
  ```

- They can be initialised when they are declared:

  ```
  int count = 0;
  float init_value = 3.4;
  ```

- Variables can be assigned values at run time:

  `count = 6;`

- variables are modified via an lvalue.

  An lvalue is an expression that references an object's location (rather than just its value).

  The term lvalue was originally coined to mean something that can be on the left-hand side of an assignment.

- Variables can also be used on the right side of an assignment; where their values are used to compute the value to be assigned:

  `count = a + b;`

2.7.2 Scope and duration

- An object can only be used after it has been declared.
- Objects declared outside any block are global. They have file scope.
- An object declared inside a block is local. Its not available outside the block.
- Blocks and thus scopes can be nested.
- Names can be overridden inside blocks.

  The `::` operator can be used to reference a global object with the same name as a local object. (See the example below.)

- The duration of an object is normally the same as its scope.

  It is destroyed when it can no longer be accessed:

  - Unless it is declared as static.

    It is still only accessible from within the block, but it is not destroyed when the block is finished.

    When the block is re-entered it is still available.

    It is initialised once, when it is created.

  - Unless it is declared using new (more about this later).

    By default objects are destroyed when they go out of scope. This kind of object is called automatic.

- A global or local static object that is not explicitly initialised is implicitly initialised to zero.

  Automatic objects must be explicitly initialised.

2.7.3 Example: Scope experiment

```
// File : scopetst.cpp
// Experiment with C++ scope and duration rules

#include <iostream.h>

int x = 1;      // Global x - file scope

int give_x()    // A function
{
   return x;
}

main()
{
   cout << "A -> x is " << x << endl;
   cout << "B -> x from give_x is " << give_x() << endl;
   int x = 2;
   cout << "C -> x is " << x << endl;
   cout << "D -> x from give_x is " << give_x()
        << endl;
   {
      int x = 3;
      cout << "E -> x is " << x << endl;
      cout << "F -> ::x is " << ::x << endl;
   }
   cout << "G -> x is " << x << endl;
   for ( int i = 0; i < 3; i++ )       // loop 3 times
   {
      int x = 0;
      cout << "H -> automatic x in loop is "
           << x++ << endl;
      static int y;
      cout << "I -> static y in loop is    "
           << y++ << endl;
   }
}
```

This program has the following output:

```
A -> x is 1
```

```
B -> x from give_x is 1
C -> x is 2
D -> x from give_x is 1
E -> x is 3
F -> ::x is 1
G -> x is 2
H -> automatic x in loop is 0
I -> static y in loop is    0
H -> automatic x in loop is 0
I -> static y in loop is    1
H -> automatic x in loop is 0
I -> static y in loop is    2
```

2.8 Operators and expressions

An expression is a sequence of operators and operands that specifies a computation.

2.8.1 Precedence and associativity

- `3 * 4 + 3`

  The operator * has a higher precedence than +, so the grouping is

  `( 3 * 4 ) + 4`

- `3 - 4 + 5`

  The operators `+` and `-` have the same precedence and have left-to-right associativity, so this groups as follows:

  `( 3 - 4 ) + 5`

- `a = b = c`

  The `=` operator has right-to-left associativity, so this groups like this:

  `a = ( b = c )`

- Precedence and associativity can normally be modified with parentheses, so:

  `( 3 + 4 ) * 5` has the value 35, but `( a = b ) = c` is an error because `( a = b )` is not an lvalue.

  Use parenthesis to make expressions easy to understand.

- Precedence and associativity determine the grouping of operator with operands. The order of calculation is not defined.

  The expression `a * b + c * d` groups `(a * b) + (c * d)` but we cannot predict if `a * b` or `c * d` will be calculated first.

2.8.2 Arithmetic operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	integer division, quotient floating point division
`%`	integer division, remainder (modulus)

The operation of `+`, `-` and `*` are obvious, but

`7 / 2`	gives 3
`7 % 2`	gives 1
`7.0 / 2.0`	gives 3.5
`7.0 / 2`	gives 3.5 (promotion)
`7.0 % 2`	is an error

2.8.3 Assignment operators

- x = 6 is an expression. It has a value and can be used like this:

  `cout << ( x = 6 );`

  assigns the value 6 to `x` and prints 6.

  Without the parentheses the statement would group `(cout << x) = 6` which is invalid.

- Expressions of the form `x = x + 6` are common. C++ provides an abbreviated form for this type of statement:

  `a += b` is the same as `a = a + b`
  `a -= b` is the same as `a = a - b`
  `a *= b` is the same as `a = a * b`
  `a /= b` is the same as `a = a / b`
  `a %= b` is the same as `a = a % b`

  These expressions also have values, so `n = m *= 3` id valid.

  There are also bit-wise assignment operators, which have a similar form.

2.8.4 Increment and decrement operators

- `n = n - 1` and `n = n + 1` are very common expressions, so C++ provides the special operators `++` and `--`.

- `++n` increments `n` and gives the incremented value. `n++` gives the value of `n` and has the side effect of incrementing `n`.

  `a = n++` is the same as `a = n; n = n + 1;`
  `a = ++n` is the same as `n = n + 1; a = n;`
  `a = n--` is the same as `a = n; n = n - 1;`
  `a = --n` is the same as `n = n - 1; a = n;`

2.8.5 Relationship and Logic operators

- C++ does not have a Boolean type.

  The integer value 0 represents false. Any non-zero integer value represents true.

  The expression `n - n` evaluates to false, while `3 * 6` is always true.

- The equivalence operator is `==`

  `7 == 7` yield true (i.e. 1)
  `3 == 7` yields false (i.e. 0)

  Take care. A very common mistake, which the C++ compiler may not catch, is using `=` instead of `==`.

- The other relation operators are:

`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

- Relational expressions can be combined with the logical operators:

`!`	NOT
`||`	OR
`&&`	AND

  For example, `n == 1 || m > 6`

  Parentheses are not required because `||` has a lower precedence than `==` or `>`. However, use them for clarity, like this `(n == 1) || (m > 6)`.

  The left operand of `||` or `&&` is always evaluated first. The right hand operand is not evaluated unless is has to be.

  The expression `1 || n` is always true and `n` is not evaluated. The expression `0 && n` is always false and `n` is not evaluated.

  Take care. The symbols `|` and `&` are valid C++ operators. They are bitwise `OR` and bitwise `AND` respectively.

2.8.6 The conditional expression

- This has the form

  `expression-c ? expression-t : expression-f`

  `Expression-c` is always evaluated first. If the condition represented by this is true then the whole expression has the value of `expression-t`, otherwise it has the value of `expression-f`.

2.8.7 The comma operator

- This is an obscure operator (!).
- A pair of expressions separated by a comma are evaluated left to right and the value of the left expression is discarded.

  The type and value of the result are those of the right operand.

  All side effects of the left expression are applied before the right operand is evaluated.
- In the context where comma has a special meaning, such as lists of actual arguments to a function, the comma operator can only appear in parentheses.

  In `f(a,(t=3,t+2),c)` the second argument is 5. This also assigns the value 3 to `t`.
- A more useful example is

  ```
  for ( i = 0, j = 4; i < 10; i++, j++)
     ....;
  ```

  This varies `i` from 0 to 9 while `j` goes from 4 to 13.

  Parentheses are not required because comma has the lowest precedence.

2.8.8 Type conversions

- C++ allows mixed mode expressions such as `2 + 3.5`

  Both operands must be the same type for computation, so one is promoted.
- The full rules are complex, but generally `char`, `short int` and `enum` are converted to `int`.

  Then one operand is promoted through `int` to `unsigned int` to `long` to `unsigned long` to `float` to `double` to `long double` until it matches the other operand.
- Conversions are performed if needed during initialisation.

2.8.9 Casts

- Type conversions can be explicit

  `(double)7/3` or `double(7)/3` gives 2.5

  Take care. Casts can produce garbage, for example `char(1000)`. They can be used thoughtlessly to by-pass C++ type restrictions. Do not use casts if they can be avoided.

2.8.10 Side effects

- C++ expressions can have useful side effects,

  `i = n++` has the side effect that `n` is incremented.

  The `=` operator has a side effect. Its left hand operand is modified.

- The vague ordering of operations in C++ expressions can cause problems:

  We cannot predict the value of `i` after `++i + i` or `i = 6 + i++`

  We do not know the argument values in `f(--i,i++)`

  There is some help. All side effects are guarantied to happen:

  - At the end of statements.
  - At comma, `||`, `&&` and `?:` operators. The left most expression is evaluated first.
  - Before arguments are passed to functions. However, the order of evaluation of arguments is not defined.

2.9 Control structures

2.9.1 While statement

```
while ( expression ) statement;
```

```
  int i = 1;
  while ( i <= 5 ) {
     cout << i << endl;
     i++;
  }
```

2.9.2 Do statement

```
do statement while ( expression );
```

```
  int i = 1;
  do {
     cout << i << endl;
     i++
  } while ( i <= 5 );
```

2.9.3 For statement

```
for ( statement-i expression-c; expression-s )
     statement;
```

This is equivalent to:

```
     statement-i;
     while ( expression-c ) {
        statement;
        expression-s;
     }

  for ( int i = 1; i <= 5; i++ )
     cout << i << endl;
```

Parts of the for statement cam be omitted. For example, `for (;;);` loops forever.

2.9.4 If statement

```
if ( expression ) statement;
   if ( expression ) statement-1 else statement-2;

  if ( amount > 10 ) {
     cout << "Too many items for one delivery\n";
     amount = 10;
  }
  else
     cout << amount << " items will be delivered\n";
```

In nested `if` statements, an `else` is connected with the last encountered else-less `if`. Use `{ }` to override this relationship.

2.9.5 Switch statement

```
switch ( expression ) statement;
case constant-expression : statement;
break;
default : statement;

  switch ( ans ) {
  case 'y':
  case 'Y':
     cout << "Resetting totals\n";
     stock_total = 0;
     break;
  case 'n':
  case 'N':
     cout << "Totals not modified\n";
     break;
  default:
     cout << "Invalid reply. Please try again\n";
  }
```

2.10 Example: Multiplication Table

```
//file contr.cpp
// demonstrate some C++ control structures

#include <iostream.h>
#include <iomanip.h>

enum Boolean { FALSE, TRUE };
const int TOP = 12;

main()
{
   cout << "\nTables Print Program\n";
   cout <<   "--------------------\n\n";

   Boolean more = TRUE;
   while ( more ) {
      int table;
      Boolean good_ans;
      do {
         cout << "Give required table (2 to 12): ";
         cin >> table;
         if (table >= 2 && table <= 12)
            good_ans = TRUE;
         else {
            cout << "Table must be in range 2 to 12, "
                    "please try again\n";
            good_ans = FALSE;
         }
      } while ( !good_ans );
      cout << endl;
      for ( int i = 1; i <= TOP; i++ ) {
         cout << table << " x " << setw(2) << i
              << " = " << table * i << endl;
      }
      char ans;
      cout << "\nAnother table ( y or n ) ";
      cin >> ans;
      switch ( ans ) {
      case 'Y':
      case 'y':
         more = TRUE;
         break;
      case 'N':
      case 'n':
         more = FALSE;
      }
   }
}
```

2.11 Pointers

- A pointer is the address of an object of a specific type:

  `int* p;    // p is a pointer to int`

  Take care. In the declaration `int* p, r` `r` is an integer, NOT a pointer to `int`.

- The address-of operator is `&`. When `&` is applied to an lvalue it yields a pointer:

```
float a;
int i;
int *p;
p =&i;
p = &a;    // error p is not a pointer to float
```

- The object addressed by a pointer is accessed by dereferencing the pointer:

```
int m = 0;
int *p = &m;
*p += 6;          // add 6 to m
```

- Arithmetic can be performed on pointers (See the section on arrays for an example.)

2.12 Arrays

2.12.1 Declaration

- An array in C++ is declared like this:

  `int a[5];`

  This is an array of five integers.

2.12.2 Initialisers

- Array can have initialisers:

  `int total[5] = { 3, 5, 2, 5, 8 };`

- If the initialiser has too few items to fill the array, zeros are added.

  `int total[5] = { 23, 65 };`

  is the same as

  `int total[5] = { 23, 65, 0, 0, 0 };`

- The size of the array can be omitted if an initialiser is used.

  `int total[] = { 3, 5, 2, 5, 8 };`

  declares an array of five integers

2.12.3 Access

- The elements of an array are accessed using subscripts.

  ```
  total[3] += 7;     // add 7 to the 4th
                     // element of total
  ```

- The first element is `total[0]`.
- The last element is `total[4]` if the array has 5 elements.
- **Take care.** You can read and write off the end of an array. This can damage data or even crash the program, including the development tool. (The author has been hit by this one.)

2.12.4 Pointers

- An array variable is actually a pointer.

  The name of an array can be dereferenced as a pointer variable:

  ```
  main()
  {
     int array[] = { 1, 2, 3, 4, 5 };
     int i;

     for ( i = 0; i < 5; i++ )
        cout << array[i] << ' ';
     cout << endl;

     int* array_point = array;   // <- address of array

     for ( i = 0; i < 5; i++ ) {
        cout << *array_point << ' ';
        array_point++;
     }
     cout << endl;

     array++;      // ERROR - An array variable is CONST

  }
  ```

2.13 Strings

- A string is an array of characters. Its end is indicated by a null character which is `'\0'`.

  `char dog[5] = "Cleo";`

  is the same as

  `char dog[5] = { 'C', 'l', 'e', 'o', '\0' };`

- The array must be at least one character longer than the longest string it will have to store.
- There is a string library that offers a number of string functions. To use it include `<string.h>`.

2.13.1 Example: Strings

```
// file string1.cpp
// demonstrate strings

#include <iostream.h>
#include <string.h>

main()
{
   char dog1[10] = "Cleo";
   char dog2[10] = "Penny";
   char dog4[] = "Brollie";
   char dog3[10];

// dog3 = dog2; ERROR - can't assign a string to dog3

// Copy string
   int i = 0;
   while ( dog2[i] != '\0' ) {
      dog3[i] = dog2[i];
      i++;
   }
   dog3[i] = '\0';

   cout << "dog 3 is " << dog3 << endl;


// A more compact method
   char* s1 = dog3;
   const char* s2 = dog1;
   while ( *s1++ = *s2++ );

   cout << "dog 3 is " << dog3 << endl;

// Use string library function
   strcpy(dog3,dog4);

   cout << "dog 3 is " << dog3 << endl;

// OUCH!!!  The following prints "should Xe goodbye"
//          with a Borland C++ compiler !!!!!

   char* se1 = "goodbye";
```

```
    se1[14] = 'X';
    cout << "should be goodbye " << endl;

}
```

2.14 Structures

- Many languages offer a record type. In C++ this is called a structure.
- In an array all the elements are the same type. In a structure the element (or members) can be of different types:

  ```
  struct dog {
     char name[20];
     char sex;
     char breed[20];
  };
  ```

- This structure can be declared and accessed as follows:

  ```
  dog my_dog;
  my_dog.sex = 'F';
  my_dog.name = "Cleo";
  ```

- They can be intialised like arrays:

  `dog my_dog = { "Cleo", 'F', "Alsatian" };`
- The members of a structure can be accessed through pointers:

  ```
  dog* p = &my_dog;
  cout << "my dogs name is " << p->name << endl;
  ```

- structures are not very important in C++. Classes are much more useful. A structure is just a public access class (more about this later).
- C++ also provides unions which are similar to variant records in Pascal. However, these are not described in this report.

2.15 Named constants

- The keyword const can be added to the declaration of an object to make it a constant rather than a variable.

  ```
  const int LIMIT = 10;
  LIMIT = 20;             // error
  LIMIT++;                // error
  int i = LIMIT;          // ok
  ```

- It can be used with pointers:

```
const char* cs = "abc";   // pointer to constant
cs[2] = 'x';              // error
cs = "xyz";               // ok
cs++;                     // ok
char a = cs[0];           // ok
```

and ...

```
char *const cs = "abc";   // constant pointer
cs[2] = 'x';              // ok
cs = "xyz";               // error
cs++;                     // error
char a = cs[0];           // ok
```

and ...

```
const char *const cs = "abc";   // both constant
cs[2] = 'x';                    // error
cs = "xyz";                     // error
cs++;                           // error
char a = cs[0];                 // ok
```

- Access thorough pointers to constants is restricted:

```
int a = 1;
const int b = 2;
const int* p1 = &a;    // ok
const int* p2 = &b;    // ok
int* p3 = &a;          // ok
int* p4 = &b;          // error
*p4 = 6;               // can't be allowed
```

2.16 Reference Types

- A reference is an alternative name for an object.
- A reference type is specified by putting `&` after a type, `X&` means reference to `X`.

```
int i = 2;
int& r = i;   // r and i refer to same variable
int j = r;    // j= 2
r = 6;        // i = 6
```

- A reference must be initialised to an lvalue but a constant reference can be initialised to a constant:

```
int& i = 2;           // error
const int& j = 2;     // ok
```

- Reference types are most useful for function values and arguments.

2.17 Typedef Names

- The keyword typedef can be used to define alternative names for types:

  ```
  typedef char* string;

  string a;
  // is the same as
  char* a;

  string a, b, c;
  // is the same as
  char *a, *b, *c;
  ```

- Typedef names are synonyms for types.
- Useful for introducing meaningful names.
- Used in libraries to improve portability.

2.18 Functions

2.18.1 Declaring functions

- A function must be declared before it can be called.

  A function declaration gives the function's name, its return type and the number and type of its arguments.

  `double max( double a, double b );`

- The argument names in a declaration are ignored by the compiler. They can be omitted but it is best not to. They are useful for documentation.

2.18.2 Defining functions

- A function is defined by giving it a body

  ```
  double max( double a, double b )
  {
     if ( a > b )
        return a;
     else
        return b;
  }
  ```

- The function max might be called like this

  `cout << max(v,val);`

2.18.3 Return statements

- A function must finish (exit) with a return statement which supplies a value of the correct type.
- A function can be declared as void:

  `void print_amount( int amount );`

  In which case it should not contain a return statement.

2.18.4 Arguments

- Arguments can be specified for input or output:

  ```
  void silly( int argin, int& argout )
  {
     argin++;
     argout++;
  }
  ```

  When `silly` is called `argin` will not be modified but `argout` will be incremented.

  We say that `argin` is passed by value and that `argout` is passed by reference.
- It can be more efficient to pass a large object by reference rather than by value, even if it will not be changed:

  ```
  void silly( const big_thing& arg1 )
  {
     // arg1 cannot be modified
  }
  ```
- Arrays can be passed as arguments:

  `void copy_string( const char sin[], char sout[] );`

  The size of an array argument is not available in the called function (strings are zero terminated).

  Take care. The compiler will only complain if the actual argument for `sout` is not a pointer to character. A string of any length can be written to `sout`, with probably disastrous results.
- Default arguments can be specified:

  `double percent( double val, double pcent = 1.0 );`

  Can be called with

  ```
  amount = percent(50);     // amount is 0.5
  amount = percent(50,50)   // amount is 25
  ```

  Default arguments must be at the end of the argument list.
- A function does not have to have arguments:

```
int empty();
...
if ( empty() )
   ...;
```

2.18.5 Returning references

- Functions can return references or pointers:

  ```
  int& max( int& a, int& b)
  {
     if (a > b)
        return a;
     else
        return b;
  }
  ```

  This could be used like this:

  ```
  // get the value of the largest
  int c = max(a,b);
  // replace the larger of a or b with 99 !!!
  max(a,b) = 99;
  ```

- More efficient than return by value.
- Function can be on left hand side of assignment.
- Useful for operator overloading (more later).

Take care. Do not return a reference to a local variable. It will not be there after the call.

2.18.6 Recursion

Functions can be recursive:

```
int factorial( int i )
{
   return n == 0 ? 1 : n * factorial(n - 1);
}
```

2.18.7 Inline function

Functions can be declared to be inline:
`inline double max ( double a, double b);`
This is a just a hint to the compiler. You should be warned if the compiler cannot inline the function.

2.18.8 Example: Functions

```
// file funct1.cpp
// Demonstrate functions

#include <iostream.h>

const int MAX_DOGS = 4;
char dogs[MAX_DOGS] [10] = { "Cleo",
                             "Penny",
                             "Brollie" };

int print_all_dogs();
// Print the names of all dogs
//  return the number of dogs

void add_dog( const char* dog, int& done );
// Add a dog
//  done is 0 if no room, +ve if OK

int is_a_dog( const char* dog );
// Check if a dog exists

void copy_string( const char* in_string,
                  char* out_string );
// String copy a to b

int comp_string( const char* a, const char* b );
// String compare
//  return 0 if a = b,
          +ve if a precedes b, and -ve if b precedes a

int factorial( int n );
// Calculate n!


main()
{
   cout << "The dogs are ";
   int dog_count = print_all_dogs();
   cout << endl << " and there are "
        << dog_count << " of them.\n";

   int ok;
   add_dog("Moss",ok);
   if ( ok ) {
      print_all_dogs();
      cout << endl;
   }
   else
```

```
      cout << "no room for dog\n";
   add_dog("Fido",ok);
   if ( ok ) {
      cout << "dont expect this!!!!!\n";
   }
   else
      cout << "no room for dog\n";
   if ( is_a_dog("Cleo") )
      cout << "Cleo is a dog\n";
   else
      cout << "Cleo is not a dog !!!\n";
   if ( is_a_dog("Ann") )
      cout << "Ann is a dog !!!\n";
   else
      cout << "Ann is not a dog \n";

// Test a recursive function
   cout << "factorial 0 is " << factorial(0) << endl
        << "factorial 6 is " << factorial(6) << endl;

}  // end main

int print_all_dogs()
{
   int dcount = 0;
   for ( int i = 0; i < MAX_DOGS; i++ )
      if ( dogs[i][0] != '\0' ) {
         cout << dogs[i] << ' ';
         dcount++;
      }
   return dcount;
}

void add_dog( const char* dog, int& done )
{
   int i = 0;
   while ( dogs[i][0] != '\0' && i < MAX_DOGS )
     i++;

   if ( i < MAX_DOGS ) {
      copy_string(dog,dogs[i]);
      done = 1;
   }
   else
      done = 0;
}

int is_a_dog( const char* dog )
{
   if ( comp_string(dog,"") == 0)
```

```
      return 0;
   else {
      int i = 0;
      while ( comp_string(dog,dogs[i]) != 0 &&
              i < MAX_DOGS )
        i++;
      return ( i < MAX_DOGS );
   }
}

void copy_string( const char * in_string,
                  char* out_string )
{
   while ( (*out_string++ = *in_string++) != 0 );
   // != avoids compiler warning
}

int comp_string( const char* a, const char* b )
{
   while ( *a == *b ) {
      if ( *a == '\0' )
         return 0;
      a++;
      b++;
   }
   return *a - *b;   // subtract char values
}

int factorial( int n )
{
   return n==0 ? 1 : n * factorial(n - 1);
}
```

2.19 Memory Management

2.19.1 Creating objects

- Objects can be dynamically created with new:

  `int* p = new int;`

  This creates an integer object with p as its pointer.

- An object created with new is said to be on the free store.

2.19.2 Accessing objects

- Values can be assigned to the object by dereferencing the pointer:

  ```
  *p = 100;
  cout << *p;
  ```

- If the object is a structure (or a class) the `->` operator can be used to access a member:

  ```
  dog* mydog = new dog;
  mydog->name = "Penny";
  ```

2.19.3 Destroying objects

- An object created with `new` exists until it is destroyed by `delete`. Then the space it occupied can be used by another `new`.

  ```
  delete p;
  delete mydog;
  ```

- The `delete` operator can only be applied to pointers created with `new` or to zero. Applying `delete` to zero has no effect.
- A pointer to constant cannot be deleted.
- To destroy an array use `delete []`

  ```
    int* p = new int[20];
    // use the array like this
    p[2] = 4;
    // and destroy it like this
    delete [] p;
  ```

  - A destructor, if it is defined for class objects in the array, will be invoked for each element.
  - Older C++ compilers do not support this notation:
    The number of elements has to be given explicitly:

    ```
        stringX* table = new stringX[10];
        // bla bla
        delete [10] table;
    ```

  - Do not use `delete []` on non-array objects. Do not use delete on array objects.

- An out of memory condition can be handled using the system variable `_new_handler`.

  Details of how to use this can be found in Graham, N., Learning C++, on page 153 and in listing 3-3 on page 151.

  This function-call mechanism is replace by exception handling in the latest version of C++.

- **Take care.** Do not use a deleted object. The effect of this is undefined.
- **Take care.** There is no *garbage collection.* Destroying the pointers to an object will not destroy the object on free store. This causes memory leakage.

Chapter 3

CLASSES AND OBJECTS

- A class in a program represents the fundamental concepts of the application and in particular the fundamental concepts of the reality being modelled.
- In C++ a class is a type.
- A class consists of:

  A state a collection of data members.

  An interface a collection of member functions.

  A specifier a name.

  Data Hiding levels of program access to data. Also called encapsulation.

  Normally the state is private and the interface is public.

- A stock item might be modelled by:

```
class stock {
   int in_stock;          // private by default
   int reorder_level;     // private by default
   double unit_cost;      // private by default
public:
   void set_stock( int s );
   void set_value( double v );
   void set_reorder( int r )
   double total_value();
   int reorder();
   int out_of_stock();
   int remove( int amount );
   int add( int amount );
   void print();
};
```

- A class object is declared like any other object:

```
stock widget;
stock* wp = &widget;
```

- The public members of a class object can be used like this:

  ```
  widget.set_stock(25);
  if ( widget.reorder() )
     cout << "reorder NOW!";
  cout << "total values is " << widget.total_value();
  delivered = widget.remove(25);
  int newstock = wp->add(5);
  double value = widget.unit_cost;  // error !!
  ```

  - Class members are private by default. They cannot be accessed outside the class.
  - Data members are normally private. Member functions are normally public.
  - Structures (and unions) are just classes with public default access.

- Once a class has been declared an object of that type can defined and referenced.

  However, before the program can run the class's member functions must be defined:

  ```
  double widget::total_value()
  {
     return in_stock * unit_cost;
  }
  ```

  - Function definitions are associated with their class by using the class name with the scope operator.
  - Member functions have full access to data members in their class.
  - Alternatively, they can be defined in the class declaration like this:
    `double total_value() { return in_stock * unit_cost; }`
    This is an inline function, but more about this later.

3.1 Self referencing

- A member function can reference the private members of its object. This is called self referencing.

- A member function has a hidden first argument that points to the object. It is implicitly declared as:

  `X *const this;`

  It is not normally used. Why write `this->unit_cost` when `unit_cost` is all that is needed?

  However, it is used when a reference to the object has to be returned from a function, like this:

  `return *this;`

3.2 Initialisation

- The stock class discussed above is not automatically initialised. The functions `set_stock`, etc have to explicitly called.
- Special functions called constructors can be declared to automatically initialise an object

  These have the same name as their class and do not have a return type:

```
class stock {
   int in_stock;
   int reorder_level;
   double unit_cost;
public:
   stock();
   stock( double unit_cost, int reorder = -1 );
   double total_value();
   int reorder();
   int out_of_stock();
   int remove( int amount );
   int add( int amount );
   void print();
};
```

- Constructors have to be defined like other member functions:

```
stock::stock()
{
   in_stock = 0;
   reorder_level = -1;
   unit_cost = 0;
}
```

  The constructor stock has been overloaded. The version called is determined by the number, type and order of the arguments.

- The relevant constructor is used when a stock object is initialised:

```
// stock() used...
stock widget1;
// stock( double, int ) used...
stock widget2 = stock(1.50,10);
// An abbreviated syntax can be used...
stock widget2(1.50,10);
// This syntax can be used when there is only one
// argument...
stock widget4 = 2.33;
```

3.3 Assignment

- When one object is assigned to another, a memberwise copy is performed.

  This can cause problems when the class contains pointer data members (more later).

- Constructors can be used as normal functions to assign values to objects:

```
stock s1, s2, s3;
s1 = stock(34.54,10)
s3 = stock();
s4 = 5.5;          // stock(5.5,-1) used
s5 = s4;           // memberwise copy
```

3.4 Clean up

- When an object goes out of scope a destructor is called to release storage.
- An explicit destructor function can be declared for a class:

  `~stock();`

  It can have no arguments or a return type. only one destructor per class is allowed.

- Explicit destructors are useful when pointers are used (so stock does not have one).

3.5 Example: Stock Class

```
// File: stock2.cpp
// Using the stock class

#include <iostream.h>

class stock {
   int in_stock;
   int reorder_level;
   double unit_cost;
public:
   stock();
   stock( double unit_cost, int reorder = -1 );
   double total_value();
   int reorder();
   int out_of_stock();
   int remove( int amount );
   int add( int amount );
   void print();
};

stock::stock()
```

```
{
   in_stock = 0;
   reorder_level = -1;   // dont reorder
   unit_cost = 0.0;
}

stock::stock( double unit_cost, int reorder )
{
   in_stock = 0;
   reorder_level = reorder;
   stock::unit_cost = unit_cost;
}

double stock::total_value()
// Calculate stock value
{
   return unit_cost * in_stock;
}

int stock::reorder()
// Check reorder
//  return 0 (false) if reorder not required
//  return 1 (true) if reorder required
{
   if ( in_stock <= reorder_level )
      return 1;
   else
      return 0;
}

int stock::out_of_stock()
// Check if out of stock
//  return 0 (false) if stock left
//  return 1 (true) if out of stock
{
   if ( in_stock == 0 )
      return 1;
   else
      return 0;
}

int stock::remove( int amount )
// Attempt to remove amount of stock
//  return amount actually removed
//  in_stock can never be less than 0
{
   if ( amount > in_stock ) {
     int temp = in_stock;
     in_stock = 0;
     return temp;
   }
   else {
     in_stock -= amount;
     return amount;
   }
}


int stock::add( int amount )
// Add to stock
//  return new stock level
{
   in_stock += amount;
   return in_stock;
}

void stock::print()
// Print stock object
{
   cout << in_stock << '[' << reorder_level << ']'
        << '@' << unit_cost;
}

main()
{
   stock widget1(5.50,30);
   stock widget2 = 3.50;
   int delivery;

   if ( widget1.out_of_stock() )
      cout << "New widget1 stock is "
           << widget1.add(20) << endl;
   if ( widget1.reorder() )
      cout << "Order more number 1 widgets now\n";
   else
      cout << "Stock level number 1 widgets OK\n";
   if ( widget2.reorder() )
      cout << "Order more number 2 widgets now\n";
   else
      cout << "Stock level number 2 widgets OK\n";
   cout << "The value of widgits in stock is "
        << widget1.total_value() +
           widget2.total_value() << endl;
   delivery = widget1.remove(25);
   cout << delivery << " number 1 widgits removed"
                      " from stock for delivery\n";
   cout << "Widget1 status is "; widget1.print();
   cout << endl;
}
```

3.6 Visibility

- Members can be declared as:

  private Name can only be used by member functions and friends of the class in which they are declared.

  protected Name can only be used by member functions and friends plus member functions and friends of derived classes.

  public Name can be used by any function.

  More about friends, derived classes and protected access later.

- As long as the public and protected parts of a class remain unchanged its implementation can be changed without effecting the way the class is used. Its interface is constant. However, the constructors may have to be enhanced.

  > "Designing a class takes more time than just providing a data structure and a set of separate functions. However, this effort should be more than recovered during later design, implementation and testing." Stroustrup

3.7 Function overloading

- Functions declared with the same name, in the same scope, are overloaded.

  Functions with the same name in different classes are not overloaded.

- The actual function called is determined by the type, number and order of the functions' arguments. The compiler matches actual and formal arguments.

  A function's return type does not contribute to its signature.

  ```
  double power( double, int );      // overloaded
  double power( double, double );   // overloaded
  int power( double, int );         // compile time error
  ```

- The detailed rules for matching are very complex but the following simplified rules will do in most cases. They are applied in the given order:

  Exact match: The type of the argument exactly match one of the alternatives

  Standard conversions: The standard conversions are applied to achieve a match.

  User defined conversions: User defined conversions are applied to achieve a match. User defined conversions are constructors taking one argument, and conversion operators (more later).

  When there are more than one argument, an intersection rule is applied:

  A set of "best" matching functions for each argument is found and the intersection of these sets is considered:

  - If the intersection is empty this implies a no match error.
  - If the intersection contains more than one function this implies an ambiguity error.

3.8 Friends

- A friend is a non-member function that is allowed to access the private part of a class.

  A function is made a friend by declaring it as a friend in the class:

  `friend void transfer( stock& s1, stock& s2, int amt );`

  This transfers stock from one item to another. It is definition is:

```
  void transfer( stock& s1, stock& s2, int amt )
  {
     if ( amt > s1.in_stock ) {
       s2.in_stock += s1.in_stock;
       s1.in_stock = 0;
     }
     else {
       s2.in_stock += amount
       s1.in_stock -= amount;
     }
  }
```

- A friend must use qualified names. Friends do not have a this pointer.
- A member function of one class can be the friend of another class:

  `friend void X::f();`

- If all the functions of a class are friend the class is a friend:

  `friend class Y;`

3.9 Operator overloading

- Almost all of the C++ operators can be overloaded, including function call `()` and subscript `[]`.

  `. .* :: ?:` and `sizeof` cannot be overloaded

- New operators cannot be invented.
- An override cannot change the precedence, associativity or arity of an operator.
- Binary operators can be defined as member functions taking one argument or friends taking two arguments.

  `aa@bb` can be `aa.operator@(bb)` or `operator@(aa,bb)`

- Unary operators can be a member function with no arguments or a friend with one argument:

  `@aa` can be `aa.operator@()` or `operator@(aa)`

- An operator function taking a basic type as it first argument cannot be a member function:

  `aa+2` can be `aa.operator+(2)` but `2+aa` has to be `operator+(2,aa)`

- Define an operator so that it works in the "same way" as the standard version. Do not surprise the user.

3.10 Guide lines

- Some guide lines for designing operator functions are:
  - Generally if a binary operation does not update the objects make it a friend. Friends are easier to understand.
  - If it updates the object make it a member.
  - If it has to return an lvalue make it a member.
  - If it references only the object make it a member. Make it const if it does not update.
  - Return a reference if its safe, for efficiency with large objects.

  A suitable style is shown in the vector class example below.

- Operators can be declared as private to prevent them from being used e.g. `=`, `&` and comma can normally be used for class objects but not if declared private.

3.11 I/O operator overloads

- The iostream operators `<<` and `>>` can be overridden.
- The operator functions are declared as friends of the class.
- They must designed to be compatible with other iostream `<<` and `>>` operators. See the vector class program below for an example of how to do this.

3.12 Example: A Vector Class

```
// file : vector.cpp
// vector class

#include <iostream.h>
#include <math.h>      // for sqrt
#include <stdlib.h>    // for exit
```

```
const SIZE =3;

class vector {
   double element[SIZE];
public:

// constructors...

   vector();             // create null vector
   vector( double* a );  // create from array

// conversion...

   operator double() const;  // magnitude

// arithmetic operations...

   friend vector operator+ ( const vector& v1,
                             const vector& v2 );
   friend vector operator- ( const vector& v1,
                             const vector& v2 );
   friend vector operator- ( const vector& v1 );
   friend vector operator* ( const vector& v1,
                             double d );
   friend vector operator* ( double d,
                             const vector& v1 );
   friend double operator* ( const vector& v1,
                             const vector& v2 );

// Comparison...

   friend int operator== ( const vector& v1,
                           const vector& v2 );
   friend int operator!= ( const vector& v1,
                           const vector& v2 );

// Assignment...

   vector& operator+= ( const vector& v1 );
   vector& operator-= ( const vector& v1 );
   vector& operator*= ( double d );

// subscript...

   double& operator[] ( int i );

// I/O operators...

   friend istream& operator>> ( istream& c,
                                vector& v1 );
   friend ostream& operator<< ( ostream& c,
                                const vector& v1 );
};

vector::vector()
// Default constructor
{
   for ( int i = 0; i < SIZE; i++ )
      element[i] = 0.0;
}

vector::vector( double* a )
// Construct from array of doubles
{
  for ( int i = 0; i < SIZE; i++ )
     element[i] = a[i];
}

vector::operator double() const
// Type conversion to double = magnitude
{
   double magnitude = 0.0;
   for ( int i = 0; i < SIZE; i++ )
      magnitude += element[i] * element[i];
   return sqrt(magnitude);
}

vector operator+ ( const vector& v1,
                   const vector& v2 )
// vector addition v1 + v1
{
   vector out;
   for ( int i = 0; i < SIZE; i++ )
     out.element[i] = v1.element[i] + v2.element[i];
   return out;
}

vector operator- ( const vector& v1,
                   const vector& v2 )
// vector subtraction v1 - v2
{
   vector out;
   for ( int i = 0; i < SIZE; i++ )
     out.element[i] = v1.element[i] - v2.element[i];
   return out;
}

vector operator- ( const vector& v1 )
// vector negation -v1
{
   vector out;
   for ( int i = 0; i < SIZE; i++ )
      out.element[i] = -v1.element[i];
   return out;
}

vector operator* ( const vector& v1,
                   double d )
// scalar multiplication v1 * d
{
   vector out;
   for ( int i = 0; i < SIZE; i++ )
     out.element[i] = v1.element[i] * d;
   return out;
}

vector operator* ( double d,
                   const vector& v1 )
// scalar multiplication d * v1
{
   return v1 * d;
}

double operator* ( const vector& v1,
                   const vector& v2 )
// Scalar product v1 * v2
{
   double prod = 0.0;

   for ( int i = 0; i < SIZE; i++ )
      prod += v1.element[i] * v2.element[i];
   return prod;
}

int operator== ( const vector& v1, const vector& v2 )
// Compare equal
//  return 0 (false) if not equal
//  return 1 (true) if equal
{
   for ( int i = 0; i < SIZE; i++ )
      if ( v1.element[i] != v2.element[i] )
         return 0;
   return 1;
}

int operator!= ( const vector& v1, const vector& v2 )
// Compare not equal
//  return 0 (false) if equal
//  return 1 (true) if not equal
{
   return !(v1 == v2);
}

vector& vector::operator+= ( const vector& v1 )
// vector add and assign
{
   for ( int i = 0; i < SIZE; i++)
      element[i] += v1.element[i];
   return *this;
}

vector& vector::operator-= ( const vector& v1 )
// vector subtract and assign
{
   for ( int i = 0; i < SIZE; i++)
      element[i] -= v1.element[i];
   return *this;
}

vector& vector::operator*= ( double d )
// scalar multiply and assign
{
   for ( int i = 0; i < SIZE; i++)
      element[i] *= d;
   return *this;
}

double& vector::operator[] ( int i )
// subscript operator
//  subscript must be in range 1 to SIZE -1
{
   if ( i < 1 || i > SIZE ) {
      cerr << "Vector subscript out of range\n";
      exit(1);
   }
   return element[i-1];
}

istream& operator>> ( istream& c, vector& v1 )
{
   for ( int i = 0; i < SIZE; i++ )
      cin >> v1.element[i];
   return c;
}

ostream& operator<< ( ostream& c, const vector& v1 )
{
   c << '(' << v1.element[0];
   for ( int i = 1; i < SIZE; i++ )
      c << ',' << v1.element[i];
   c << ')';
   return c;
}

main()
{
   cout << "demonstrate vector class\n";
   cout << "------------------------\n";

   double a2[] = {1,2,3};
   double a3[] = {4,5,6};

   vector v1;
   vector v2 = a2;
   vector v3 = a3;
   cout << "v1 initialised to " << v1 << endl;
   v1[1] = 1.0;
   v1[2] = 1 + v1[1];
   v1[3] = 1 + v1[2];
// v1[4] = 99;                  // stops program
   cout << " then updated to " << v1 << endl;

   cout << "Enter a vector for v1: ";
   cin >> v1;
   cout << "v1 is " << v1 << endl;
   cout << "v2 is " << v2 << endl;
   cout << "v3 is " << v3 << endl;

   cout << "v1 + v2 = " << v1 + v2 << endl;
   cout << "v1 - v2 = " << v1 - v2 << endl;
   cout << "-v2 is =  " << -v2 << endl;
   cout << "v2 * 2 = " << v2 * 2 << endl;
   cout << "3 * v2 = " << 3 * v2 << endl;
   cout << "v2 * v3 = " << v2 * v3 << endl;

   if ( v1 == v2 )
      cout << "v1 is equal to v2\n";
   else
      cout << "v1 is not equal to v2\n";
   if ( v1 != v2 )
      cout << "v1 is not equal to v2\n";
   else
      cout << "v1 is equal to v2\n";

   v2 += v1;
   cout << "v2 after v2 += v1 is " << v2 << endl;
   v3 -= v1;
   cout << "v3 after v3 -= v1 is " << v3 << endl;
   v1 *= 4;
   cout << "v1 after v1 *= 4 is  " << v1 << endl;
```

```
    cout << "magnitude of v2 is " << double(v2) << endl;
    cout << "and what about +v2 " << +v2 << endl; //!!!

}
```

3.13 Constant Objects and Functions

- An object can be defined as constant. In which case it cannot be modified.

  A class object must be initialised with constructors. Assignment to the object is not allowed.

  The ordinary member functions of a constant class object cannot be called. Warning and error messages will be issued by the compiler.

  The key word const is used to tell the compiler that a function will not change any data members:

  ```
  public:
     int length() const;
  ```

  and

  ```
  int string::length() const
  {
  ......
  }
  ```

  Take care. The `const` key word is just a promise. The function can modify member data.

  Input arguments must be declared as `const` to support constant objects:

  ```
  friend int operator== ( const string& s1,
                          const string& s2 );
  ```

3.14 Inlining

- The compiler can be asked to use inline expansion rather than a subroutine call for the execution of member functions.

  The object oriented style tends to produce lots of function calls. Inlining can reduce the performance overhead.

- Inlining can be requested in the class declaration by providing a body for the function:

```
public:
   stock()
      { in_stock = 0;
        reorder_level = -1;
        unit_cost = 0.0; }
```

- Alternatively the `inline` keyword can be used with the function's definition:

  ```
  inline int stock::add( int amount )
  // Add to stock
  {
     in_stock += amount;
     return in_stock;
  }
  ```

  In this case, the definition must be in the header file.

- The compiler can ignore the inline request if the function is too long or too complicated.
- Inlining should always be considered for constructors and destructors. They can be implicitly called a lot of times and they are normally simple.

3.15 Static members

- A static data member is shared by all instances of a class.

  Every instance of the class uses the same variable:

  ```
  class silly {
     int data1;                // instance data
     int data2;                // instance data
     static int usage;         // class data
  public:
     static int reset_count;   // public class data!!
     static reset();           // static member function
     .....
     .....
  }
  ```

  static member data has to be defined in the source file:

  ```
  int silly::usage = 0;
  int silly::reset_count;    // default int to zero
  ```

- A static function can only manipulate static variables. It cannot access instance variables.

  The static key word is not used when a static member function is defined:

  ```
  silly::reset()
  {
     usage = 0;
     reset_count++;
  }
  ```

  A static member is used via its class name rather than the name of a class object:

  ```
  silly a;
  silly b;
  silly::reset();               // OK
  cout << silly.reset_count;
  cout << silly::usage;         // incorrect
  a.reset();
  ```

  A static member can be used even when there are no instances of the class defined.

3.16 Nested Classes

- Class declarations can be nested.

  A nested class, if it is not public, is hidden within its enclosing class. This reduces the global name space.

- Can be useful for classes that implement dynamic data structures:

  ```
  class list {
     struct listmem {
        int data;
        listmem* next;
        listmem( int d, listmem* n )
           { data = d; next = n; }
     };
     listmem* root;
  public:
     list() { root = 0; }
     insert( int d ) { first = new listmem(d,root); }
     // .... more stuff ....
  };
  ```

- **Take care.** Do not use them unless they are VERY simple. They can be messy.
- A friend class can be used instead:

```
class listmem {
friend class list;
   int data;
   listmem* next;
   listmem( int d, listmem* n ) { data = d; next = n; }
   // .... more stuff ....
};

class list {
   listmem* root;
public:
   list() { root = 0; }
   insert( int d ) { first = new listmem(d,root); }
   // .... more stuff ....
};
```

3.17 Classes with Pointer Data Members

- There can be problems when class objects that contain pointers are created, copied or destroyed.
- When class object is copied a memberwise copy is performed.

  If a data member is a pointer, it is the value of the pointer that is copied rather than the pointed to data.

  So two objects will end up pointing at the same data.
- Consider:

```
class customer {
   char* name;
   int   dnum;
   int*  disc; // variable discount array
public:
   customer( char* n, int d );
   void set_discount( int d, int v )
      { disc[d-1] = v; }
}

customer::customer( char* n, int d )
{
   name = n;
   dnum = d;
   disc = new int[d];
```

```
      for ( int i = 0; i < d; i++ )
         disc[i] = 0;
   }
```

When this is used we have problems:

```
customer c1("Mr Smith",4);
customer c3("Mr Brown",2);

customer c2 = c1;
c1.set_discount(1,99);    // OUCH!! both changed

c3 = c1;
c1.set_discount(1,33);    // OUCH!! both changed
```

3.17.1 Copy constructors

- To avoid this problem we declare a copy constructor that binds with

  `customer c2 = c1;`

  This is done like this:

```
customer::customer( const customer& c )
{
   name = c.name;
   dnum = c.dnum;
   disc = new int[c.dnum];
   for ( int i = 0; i < dnum; i++ )
      disc[i] = c.disc[i];
}
```

Now, after `c2` is defined, `c1` and `c2` are independent objects.

3.17.2 Assignment operators

- However, the assignment c3 = c1 still has the same problem.

  It is important to realise that assignment and initialisation are different operations.

  We will have to introduce an assignment operator that binds with

  `c3 = c1;`

  This is done as follows:

```
customer& customer::operator= ( const customer& c )
{
   delete [dnum] disc;  // destroy existing array
   name = c.name;
   dnum = c.dnum;
```

```
      disc = new int[c.dnum];
      for ( int i = 0; i < dnum; i++ )
         disc[i] =  c.disc[i];
      return *this;
   }
```

Now `c1` and `c3` are independent objects (at last!).

3.17.3 Destructors

- The customer class still has a fault.

  What happens when a customer object is destroyed?

  The pointer to the discount array will be thrown away but the array will still be allocated in free store.

- We need a destructor:

  ```
  customer::~customer()
  {
     delete [] disc;
  }
  ```

3.18 Header files and Linkage

- C++ programs are normally constructed from separate files.

  Files are the units of compilation and provide file scope.

  A component (module) is specified by a header file and a source file:

  - Families of classes, non-member functions, enumerations and typedefs are combined to make a component.
  - A header file defines the interface to the component.
  - A source file provides the implementation of this interface.

  Separate components and header files are a GOOD THING! They encourage:

  - Flexibility and reusability.
  - The development of common components.
  - The separation of interface from implementation.
  - High functional cohesion and low coupling.

- A header file provides declarations of classes and function, constants, enumerations and definitions of inline functions.

  - A header file should never contain ordinary function definitions or data definitions:

```
int twice ( int a )     // not in header !!
{
   return a * 2;
}

int a;                  // not in header !!
```

  - Header files are included in the file that wants to use the component:

    ```
    #include <iostream.h>  // system
    #include "stringc.h"   // user
    ```

    Header file names are conventionally suffixed by .h

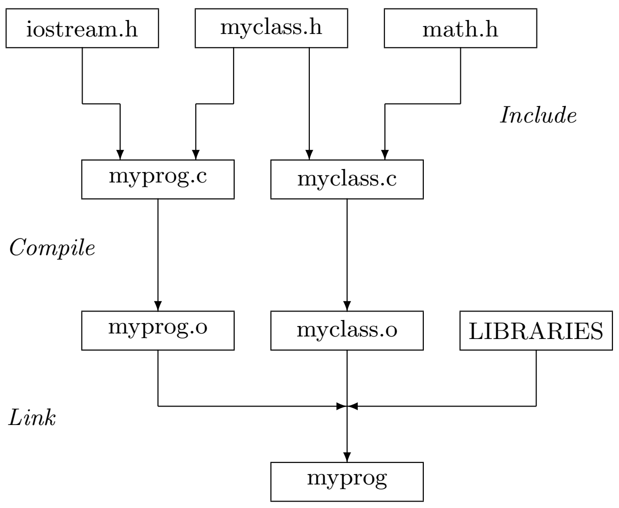
- Source files contain definitions of the functions and objects declared in their associated header files.
  - They are compiled to make object files that are used by the linkage editor to build a complete program.
  - A source file includes its header file.
  - Source file names are often suffixed by .c Other conventions are used. For example, this document uses .cpp because the examples were compiled with Borland C++.
  - The user of a component does not need access to its source file.
- The main program file is a source file containing a function called main. A program can have only one main function.
- Consider this simplified example:

```
// file myclass1.h

// might need some includes

class silly {
   int a
public
   int f1();
   void f2();
   // ... more stuff ...
}

inline void silly::f2 ()
{
   // .....
}
```

```
// file myclass1.cpp
#include <maths.h>
#include "myclass1.h"

int silly:: f1()
{
   // use maths.h
}



// file myprog.cpp

#include <iostream.h>
#include "myclass1.h"

// definitions of local functions for main
....

main()
{
   // use myclass.h and iostream.h
}
```

On some platforms, the following commands are used to build this program:

```
CPP myprog.cpp
CPP myclass.cpp
LINK myprog.obj, myclass.obj
```



- The main program myprog.cpp can be written and compiled as soon as myclass.h is written.
- The source file myclass.cpp is only needed to create the object file myclass.obj for linking.
- The object files for the system include files are automatically provided by the link command.

3.18.1 Multiple inclusion

- A header file can be included more than once.

  For example, two separately included files can both include the same file.

  Such multiple inclusion will cause duplication errors.

- This is avoided by using the pre-processor like this:

```
// file: mystuff.h

#ifdef MYSTUFF_H
#define MYSTUFF_H

// remander of header file

#endif
```

- System header files use this method.

3.18.2 Example: A String Class

The header file for the class is:

```
// file stringp.h
// super strings - header

#ifndef STRINGP_H
#define STRINGP_H

#include <iostream.h>

class string {
   char* data;  // pointer to data

public:
// constructors etc...

   string();                   // string x
   string( const char* a );    // string x = "abc"
   string( const string& s );  // copy constructor
   ~string();                  // destructor
// assignment...

   string& operator= ( const string& s );

// subscript etc...

   char& operator[] (int i);
   const char& operator[] (int i) const;
   int length() const;

// i/o operators...

   friend ostream& operator<< ( ostream& c, const
string& s );
   friend istream& operator>> ( istream& c, string& s
);

// logical operators...

   friend int operator== ( const string& s1,
                           const string& s2 );
   friend int operator!= ( const string& s1,
                           const string& s2 );

};

#endif
```

Its source file is:

```
// file stringp.cpp

// Super string class - implementation

#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <stdlib.h>

#include "stringp.h"

string::string()
{
   data = new char[1];    // create null string
   data[0] = 0;
}

string::string( const char* a )
{
   data = new char[strlen(a)+1];
   strcpy(data,a);
}

string::string ( const string& s )
{
   data = new char[strlen(s.data)+1];
   strcpy(data,s.data);
}

string::~string()
{
   delete[ strlen(data)+1 ] data; // old version of C++

// delete[] data;                 // ver 2.1 C++
}

string& string::operator= ( const string& s )
{
// destroy exiting data
   delete[ strlen(data)+1 ] data; // old version of C++
// create new data
   data = new char[strlen(s.data) +1];
   strcpy(data,s.data);
   return *this;
}

void error ( const char* mess )
{
   cerr << mess << " identified by " << __FILE__
        << "\nProgram abnormaly terminated.";
   exit(1);
}

char& string::operator[] (int i)
{
   if ( i < 0 || i >= strlen(data) )
      error("Index out of range");
   if ( strlen(data) == 0 )
      error("Index access to empty string");
   return data[i];
}

const char& string::operator[] (int i) const
{
   if ( i < 0 || i >= strlen(data) )
      error("Index out of range");
   if ( strlen(data) == 0 )
      error("Index access to empty string");
   return data[i];
```

```
}

int string::length() const
{
   return strlen(data);
}

ostream& operator<< ( ostream& c, const string& s )
{
   c << s.data;
   return c;
}

istream& operator>> ( istream& c, string& s )
{
   const BUFSIZE = 256;
   char buff[BUFSIZE];

   cin >> setw(BUFSIZE) >> buff;
   s = buff;
   return c;
}

int operator== ( const string& s1, const string& s2 )
{
   if ( strcmp(s1.data,s2.data) == 0 )
      return 1;
   else
      return 0;
}

int operator!= ( const string& s1, const string& s2 )
{
   return !( s1 == s2);
}
```

The following program exercise the string class:

```
// file stringpe.cpp
// exercise super strings

#include <iostream.h>
#include "stringp.h"

#include "stringp.h"      // woops!!!

main()
{

   cout << "--------------------------------\n";
   cout << "This exercises the stringp class\n";
   cout << "--------------------------------\n";
   string a;
   string b = "";
   string c = "hello";
   string d = c;
   cout << '"' << a << "\" " << '"' << b << "\" "
        << '"' << c << "\" " << '"' << d << "\"\n";
   a = "1234";
   cout << "should be 1234 " << a << endl;
   a = c;     // cant do this with ordinary strings !!
   cout << "should be hello " << a << endl;
   cout << "give a string: ";
   cin  >> d;
   cout << "You gave       " << d << endl;
   int i;
   cout << "hello string is ";
   for ( i = 0; i < c.length(); i++ )
      cout << c[i];
   cout << endl;
   cout << "blank string is ";
   for ( i = 0; i < b.length(); i++ )
      cout << b[i];
   cout << endl;
   c[4] = 'X';
   cout << "hellX ?? " << c << endl;
//   c[5] = 'Z'; // error!!
//   b[0] = 'a'; // error!!

   string cs = "abc";
   if ( cs == "abc" )
      cout << "strings equal\n";
   if (  "abc" == cs )
      cout << "strings equal\n";

   const string constS = "qwerty";
   cout << constS.length() << endl;
   cout << constS[1] << endl;
   cout << constS << endl;
//   constS[1] = 'X';     error - cant mod const!!
}
```

While this program demonstrates a more sensible use of the class:

```
// file stringpd.cpp
// Use the super string class.

#include <iostream.h>
#include "stringp.h"
```

```
int my_friend( const string& name );

main()
{
   cout << "----------------------------------\n";
   cout << "Demonstration of the stringp class\n";
   cout << "----------------------------------\n";
   string query;
   cout << "Give a name: ";
   cin >> query;
   if ( my_friend(query) )
      cout << query << " is a friend\n";
   else
      cout << query << " is not a friend\n";
} // end main

int my_friend( const string& name )
{
   string friends[5] = {"Adrian", "John", "Ann",
"Jane", "Mary"};

   int found = 0;
   int i = 0;
   while ( !found && i < 5 )
      if ( name == friends[i] )
         found = 1;
      else
         i++;
   return found;
}
```

Chapter 4

INHERITANCE

4.1 Classes and Inheritance

- Classes can be used to build other classes.
- A class can have an "is a" relationship with another class. For example a dog is an animal or a bus is a vehicle.

  We make this relationship explicit by using inheritance.
- We declare a derived class (e.g. dog) by inheriting one or more base classes (e.g. animal)
- A derived class has all the attributes, data and functions, of its base classes although access may be restricted.
- To avoid confusion: A programmer using a class to define an object is called a user. A programmer using a class as a base class is called a client.

4.2 Single Inheritance

4.2.1 Access to base class members

- Consider the following declaration of a time class that is available for use as a base class:

```
class timec {
   long int seconds;
public:
   timec()
      { seconds = 0; };
   timec( long int sec )
      { seconds = sec; };
   timec( int hr, int min, int sec )
      { put_seconds( hr, min, sec ); };
   void forward();
   void reset();
   int is_pm() const;
   friend ostream& operator<< ( ostream& c,
                                const timec& t );
protected:
   void show( ostream& c ) const;
private:
   void put_seconds( int h, int m, int s );
   void get_hms( int& h, int& m, int& s ) const;
};
```

- The functions `put_seconds` and `get_hms` are private.

  They are for "internal" use by other member functions.

  They are not available for users or clients.

- The function `show` is protected.

  It can be used by clients but not by a users of the class.

  It is provided to allow a derived class to have a print function that displays timec in a suitable format such as h:m:s, without giving it full access to seconds.

  This approach gives a constant interface to both users and clients.

  The operator `<<` can be used but this involves awkward casts.

- The data members of a base class can be declared as protected but then the implementation of the base class cannot be altered without disrupting its derived classes.

  This can be acceptable if the member data structure is very simple and the class is not going to used in a publicly available class library.

4.2.2 Declaring a derived class

- Now lets declare a new time class called `tagged_time` that is a time with an identifying `tag` value:

```
class tagged_time : public timec {
   int tag;
public:
   tagged_time( int t,
                int h = 0, int m = 0, int s = 0);
   tagged_time( int tag, const timec& t );
   tagged_time& operator= ( const tagged_time& t );
   friend ostream& operator<<
                 ( ostream& c, const tagged_time& t );
protected:
   void show( ostream& c ) const;
   void show_tag( ostream& c) const;
private:
   void reset();
};
```

- A derived class has access to the public and protected member data and functions of its base classes. It cannot access their private members.

  This class is derived from timec. Thus:

  - The public members of timec are public members in `tagged_time`.
  - It has full access to the protected members of `timec`.
  - The private members of `timec` exist but cannot accessed directly. The member functions of `timec` must be used if necessary.

- The constructors of `tagged_time` cannot access `seconds` in `timec`, so an initialiser list must be used:

```
tagged_time::tagged_time( int t, int h, int m, int s )
            : timec(h,m,s)
{
   tag = t;
}

tagged_time::tagged_time( int tag,
                          const timec& t ) : timec(t)
{
   tagged_time::tag = tag;
}
```

  The Initialiser lists invoke `timec` constructors. When this is done the body of `tagged_time` is executed to initialise the `tag` variable.

- A memberwise copy is performed when a `timec` object is assigned:

```
timec time1,
timec time2(12,0,0);
...
time1 = time2;
```

  If this is done for a `tagged_time` object the `tag` identifier will be modified. This is not satisfactory, so an assignment operator is provided:

```
tagged_time& tagged_time::operator=
                          ( const tagged_time& t )
{
   *(timec*)this = (timec)t;
   return *this;
}
```

  - This does not modify the `tag` variable.
  - It has no direct access to the `timec` part of `tagged_time`, so casts are used to invoke a `timec` memberwise copy.

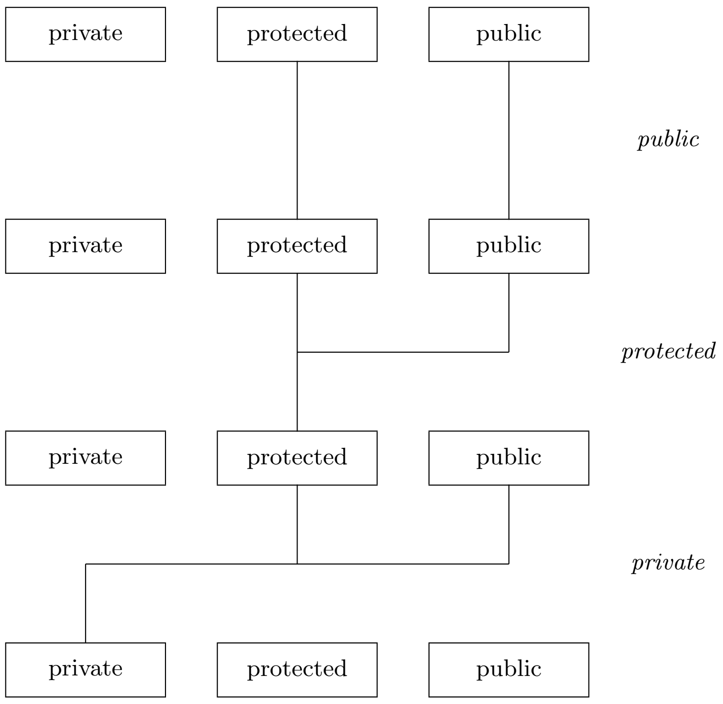
    The `this` pointer is cast to `timec*`, a pointer to `timec`, and the input argument is cast to `timec`.

- We have decided that `tagged_time` objects should not be reset, so the `void reset()` is declared (overloaded) as private so that it cannot be accessed.
- The `tagged_time` class is used just like the `timec` class:

```
timec time1(12,0,0);
time1.forward();
time1.reset();
tagged_time time2(1,12,0,0);
time2.forward();
time2.reset();       // error
```

4.3 Access to Base Classes

- A base class can be declared as public, private or protected when it is inherited. This effects the visibility of the base class members.



- Individual members can have their access adjusted:

```
class B {          class C : private B {
   int a;             int c;
public:            public:
   int b;             B::b; // adjust access to b
};                 };
```

The access level cannot be made any more or less restrictive than the level in the base class.

4.4 Overloading inherited functions

- Inherited functions can be redefined or overridden:

```
class day_time : public timec {
   int day;
public:
   date_time( int da, int hr, int mi, int se );
   void reset();
   void forward();
};

void day_time::reset()
{
   day = 0;
   timec::reset();
}

void day_time::forward()
{
   day++;
   timec::reset();
}
```

The reset and forward functions in timec are unsuitable, so they are overloaded with a new versions.

- Note the scope operator applied to `reset()` in the function `forward()`. Without it, `day` would be reset. Its omission in the body of `reset()` would cause a recursive loop.

4.5 Multiple Inheritance

- A class can have more than one base class. This is called multiple inheritance.

- Lets declare two classes:

```
class timem {
   int mins;
   int hours;
public:
   timem() { mins = 0; hours = 0; };
   int set( int hr, int min );
   void get( int& h, int& m ) const;
};
```

```
class date {
   int day;
   int month;
   int year;
public:
   date() { day = 0; month = 0; year = 0; }
   int set( int d, int m, int y );
   void get( int& d, int& m, int& y ) const;
};
```

and uses them as base classes for :

```
class entry : public timem, public date {
   char* comment;
public:
   entry()
      { comment = "no entry"; };
   void set( char* c )
      { comment = c; };
   void display();
};

void entry::display()
{
   int da, mo, yr, hr, mi;
   date::get(da,mo,yr);
   timem::get(hr,mi);
   cout << da << ',' << mo << ',' << yr;
   cout << '@' << hr << ':' << mi;
   cout << '|' << comment;
}
```

- The scope operator `::` is used in display to select the right version of `get`.
- There is no initialiser list for `entry()`, which means the default constructors `timem()` and `date()` are used.

• The `entry` class is used like this:

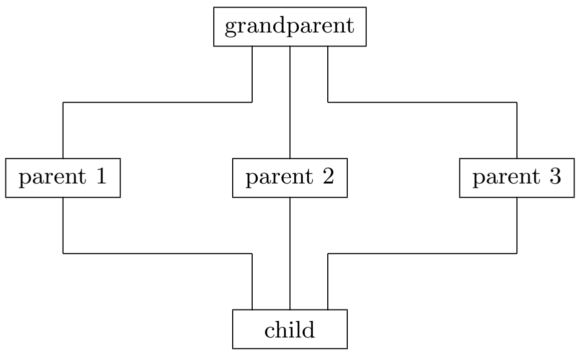
```
entry e1;
e1.timem::set(1,30);
e1.date::set(1,2,1992);
e1.set("go to shops");
e1.display();
cout << endl;
```

4.6 Initialisation

- The constructors for a class object are called in a specific order:
  1. Base classes in declaration order.
  2. Class object data members in declaration order.
  3. The body of the specified constructor is executed.
- The initialisation list specifies which constructors should be used:
  - The default constructor for a class is used if one is not given in the list. Thus a constructor that takes no arguments need not appear in the list.
  - The list can contain constructors for member class objects.
  - The order of constructors in the list is irrelevant.
- Destructors are called in reverse order to the constructors.

4.7 Virtual Base Classes

- If we have class inheritance of this form:



  There can be a problem.
- The class child will have three copies of the grandparent class's public and private members because each parent inherits a separate copy of these from grandparent.
- This is avoided by specifying grandparent as a virtual class in the parent classes as in the following:

```
// file virtual.cpp
// play with virtual classes

#include <iostream.h>

class reg {
protected:
    int value;
public:
    reg( int n = 0 ) { value = n; }
    int clock()
        { return value; }
    void latch( int v )
        { value = v; };
};

class shiftreg : public virtual reg {
public:
  void shift_left()
      { value *= 2; }
};

class bitreg : public virtual reg {
public:
    void set_bit( int mask )
        { value = value | mask; }
};

class shift_bitreg : public shiftreg,
                     public bitreg {
public:
    shift_bitreg( int n = 0 ): reg(n) {};
};

main()
{
    cout << "reg test\n";
    cout << "--------\n";
    shift_bitreg R1;
    R1.latch(16);
    R1.shift_left();
    R1.set_bit(2);
    cout << R1.clock() << endl;
}
```

- If a virtual base class has any constructors, there must be one that requires no arguments or has defaults for all its argument.

  The default constructor, with no arguments, will be called if a constructor is not called from an initialisation list.

- A constructor for a virtual base class must be called from the derived class that is actually creating an object. Calls from intermediate base classes are ignored.

4.8 Design Style

- We use inheritance to model an ”is a” relationship between classes.

- A ”has a” relationship can also be modelled with inheritance using private base classes, which results in complete encapsulation of the inherited classes.

  However, this is not an ideal approach...

- A ”has a” relationship is best modelled by containment.

  For example, an ALU has three registers:

  ```
  class ALU {
     A1 : shift_bitreg;
     A2 : shift_bitreg;
     A3 : shift_bitreg;
  public:
     void latch_A1 ( int value );
     void latch_A2 ( int value );
     int clock_A3();
     void operation( int op_code );
  };
  ```

- Private and protected inheritance should be used for ”is a” relationships. The qualification should be used simply to manage access to the inherited attributes.

  For example, if we have a double ended integer queue:

  ```
  class dqueue {
  // implementation stuff
  public:
     dqueue();
     void head_in( int data );
     int head_out();
     void tail_in( int data )
     int tail_out();
     int empty();
     int full();
  };
  ```

  We might want to declare an integer stack, which is a restricted double queue:

  ```
  class stack : private dqueue {
  public:
     stack();
     void push ( int data )
        { head_in(data); };
     int pop()
        { return head_out(); }
     dqueue::empty;
     dqueue::full;
  };
  ```

  Note the use of `::` to give access to some `dqueue` functions.

Chapter 5

VIRTUAL FUNCTIONS

5.1 Polymorphism

- Polymorphism refers to the situation where objects belonging to different classes can respond to the same message.

  Thus, we can send a message without knowing the type of the recipient.

- The ”classic” example is a graphics interface:

  Different shape objects such as circle and square respond to the same messages such as show or rotate.

  A list of objects can be displayed by sending a show message to every member of the list with out knowing the exact type of each member.

  This type of list is called heterogeneous.

  A *heterogeneous* list contains objects from different classes. All the members of a *homogeneous* list are the same type.

- In C++ polymorphism is implemented using virtual functions.

- To support polymorphism we must be able to refer to objects without regard to their classes:

  Ordinary non-reference, non-pointer variables cannot be used for this purpose. Pointers or references must be used:

  – An ordinary variable cannot refer to objects from different classes because the space occupied by the classes will not be the same.

  – A pointer or a reference to a class is an address. This is the same size regardless of the size of the referenced object.

5.2 Assignment and Truncation

- Consider:

```
class parent {
   int p1, p2;
public:
   ...
};

class child : public parent {
   int c1, c2;
public:
   ...
};
```

Objects of class parent have two instance variable p1 and p2. Objects of class child have four instance variables p1, p2, c1 and c2.

- If we define

  ```
  parent parent1, parent2;
  child child1, child2;
  ```

  The assignment

  `parent1 = child1;`

  is allowed but will result in truncation.

  The value of `child1` is converted to type `parent` by discarding the instance variables `c1` and `c2`. This assignment is equivalent to

  ```
  parent1.p1 = child1.p1;
  parent1.p2 = child1.p2;
  ```

  The assignment

  `child1 = parent1;`

  is an error because there is no default cast from `parent` to `child`.

- We can use pointers and references without truncation:

  ```
  parent* ptr = &child1;
  parent& ref = child1;
  ```

  Pointers are generally more useful for manipulating objects. The following are both valid:

  ```
  ptr = &parent2;
  ptr = &child2;
  ```

- We can use pointers to define an heterogeneous list:

  ```
  parent* list[4];

  list[0] = &parent1;
  list[1] = &child1;
  list[2] = &child2;
  list[3] = &parent2;
  ```

5.3 Virtual Functions

- Consider the following program:

```
// file poly1.cpp
// play with virtual functions

#include <iostream.h>

class parent {
public:
   virtual void Vfunct()
      { cout << "   virtual function in parent\n"; };
   void Nfunct()
      { cout << "   normal  function in parent\n"; };
};

class child : public parent {
public:
   void Vfunct()
      { cout << "   virtual function in child\n"; };
   void Nfunct()
      { cout << "   normal  function in child\n"; };
};

main()
{
   parent p1;
   child c1;
   parent* p2;

   cout << "parent direct\n";
   p1.Vfunct();
   p1.Nfunct();
   cout << "child direct\n";
   c1.Vfunct();
   c1.Nfunct();

   cout << "parent indirect\n";
   p2 = &p1;
   p2->Vfunct();
   p2->Nfunct();

   cout << "child indirect\n";
   p2 = &c1;
   p2->Vfunct();
   p2->Nfunct();

}
```

This produces the following output:

```
parent direct
   virtual function in parent
   normal  function in parent
child direct
   virtual function in child
   normal  function in child
parent indirect
   virtual function in parent
   normal  function in parent
child indirect
   virtual function in child
   normal  function in parent
```

- For a non-virtual function the declaration of the pointer variable determines which function definition will be used.

  For virtual functions the class of the object pointed to determines which function is used.

- Nonvirtual functions are bound, associated with definitions, at compile time (early binding).

  Virtual functions are bound at run time (late binding).

- A function that overrides a virtual function is itself a virtual function.

- A functions name and arguments determine an override.

  It is illegal to override a virtual function in a base class with a function returning a different type. (not so for non-virtual functions).

- Constructors cannot be virtual.

  The class of an object is always known when it is created, so virtual constructors do not make sense.

- Destructors can be virtual.

  If a destructor in a base class is declared as virtual, then destructors in derived classes override the base class destructor.

- Friends cannot be virtual.

  A friend is not a member function, so it cannot be virtual.

5.4 Abstract Base Classes

- Sometimes a class is only used for deriving other classes. This is an abstract base class.

- It is often impossible to give useful definitions of the functions declared in a base class. The declaration is just a dummy that will be overridden in a derived class.

  A dummy could be provided:

```
  virtual void display()
     { cout << "display() called for a class in "
            << "which it is not overridden\n"; }
```

A better alternative is available. The pure virtual function:

`virtual void display() = 0;`

- A class with a pure virtual function is always an abstract base class.
- An abstract base class can only be used to derive other classes.

  It cannot be used to define objects.

  If a class has a pure virtual function these rules will be enforced.

Chapter 6

TEMPLATES

Templates support generic classes and functions in C++. They are most useful for defining container classes such as lists and queues.

6.1 A Simple Class Template

- We define a template class like this:

  ```
  template<class T>
  class stack {
     T* v;
     T* p;
     int sz;
  public:
     stack( int s ) { v = p = new T[sz=s]; }
     ~stack() { delete[] v; }
     void push( T a ) { *p++ = a; }
     T pop() { return *--p; }
     int size() const { return p-v; }
  };
  ```

  The argument T is the name of a type. It is replaced when the template is used:

  ```
  stack<int> sc(100);   // a stack of integers
  stack<accounts> Acc(13);  // a stack of account objects
  ```

- It is usually best to debug an actual class, such as an integer stack, before converting it to a template.
- Template functions do not have to be inline:

  ```
  template<class T> void stack<T>::push( T a )
  {
     *p++ = a;
  }
  ```

6.2 A Simple Function Template

- Functions can be defined in a similar way:

```
template<class T> void swap( T& a, T& b )
{
   T temp = a;
   a = b;
   b = temp;
}
```

- The compiler does all the dirty work and the function can be used without effort:

```
int i,j;
double x,y

swap(i,j);
swap(x,y);
swap(i,x);   // error does not match template
```

6.3 Template Arguments

- Template arguments do not have to be type names.

  Character strings, function names and constant expressions can be used:

```
template<class T, int size> class buffer {
   T item[size];
...
};
...
buffer<char,200> cbuff;
buffer<int,10> ibuff;
```

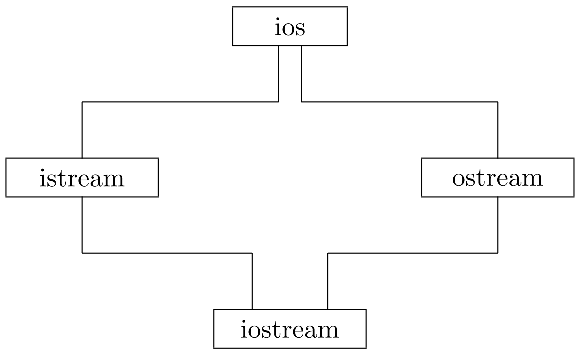
- Making size a template argument means that:

  1. It must be known at compile time.
  2. The buffer can be allocated without using free store.

Chapter 7

MORE INPUT AND OUTPUT

- By default every C++ program can use three streams standard output (`cout`), standard input (`cin`) and error output (`cerr`)
- These are supported by the header file `<iostream.h>` which declares four classes:



7.1 Input and output operators

- Standard `<<` and `>>` operators for the built in types are provided:

```
cout << "give value";
int i;
cin >> i;
if ( i < 0 ) cerr << "error on input";
```

- The streams `cin` and `cout` are tied, so the output buffer will be flushed before input begins.

7.1.1 Get and Put Functions

- Lower level manipulation is possible with the `get()` functions:

```
istream& get( char& c );
// Get a single character

istream& getline( char* p, int n, char t = '\n' );
// Get line of at most n-1 characters and place in p,
// \n\0 placed at end of p.
```

- The `>>` operators and `get()` return non-zero if successful.
- There is an equivalent `put()` function for output, so we can write:

```
char c;
while (cin.get(c)) cout.put(c);
```

7.1.2 Other useful Functions

- The standard header file `<ctype.h>` contains some functions that are useful for processing input.

  For example, `int isprint( char c )` returns true if `c` is a printable character.

- There is a function for returning a character to the input stream:

```
// do nothing !!
cin.get(c);
cin.putback(c);
```

7.2 Format State Flags

ios::skipws	skip white space on input
ios::adjustfield	field adjustment bit field
ios::left	pad after value
ios::right	pad before value
ios::internal	pad between sign and value
ios::basefield	integer base bit field
ios::dec	decimal
ios::oct	octal
ios::hex	hexadecimal
ios::showbase	show integer base
ios::showpos	explicit + for positine integers
ios::uppercase	E and X rtaher than e or x
ios::showpoint	print trailing zeros
ios::floatfield	floating point notation bitfield
ios::scientific	.ddd Edd
ios::fixed	ddd.dd
ios::unitbuf	flush output after each output operation
ios::stdio	flush output after each character

- In each of the named bit fields the flags are mutually exclusive. No more than one flag should be set.
- Default is `ios::skipws` and `ios::dec` set with all the other flags clear.

7.3 Parameters

- Width

  For output this sets width of the print field.

  For the input of strings this sets the maximum number of characters read (width - 1).
- Fill

  Specifies the pad character.
- Precision

  If the ios::scientific or ios::fixed flags are set this specifies the number of digits to the right of the point.

  Otherwise, it specifies the total number of digits to be printed.

7.4 Setting Flags and Parameters

- The iostream class provides `fill()`, `flags()`, `precision()` and `width()` for setting up format states and parameters.
- They return the relevant state component before it is changed. The following get the state without modification.

  ```
  char c = cout.fill();
  long f = cout.flag();
  int  p = cout.precision();
  int  w = cout.width();
  ```

  The stored values can be used to restore the state later by specifing them as arguments in the relevant function.
- The `width`, `fill` and `precision` parameters are simple to modify:

  ```
  cout.fill('*');
  cout.width(10);
  cout.precision(6);
  cout << value1 << value2;
  ```

  A call of `width()` affects only the immediately following output operation. For example, the `value2` above will be printed with the default width.

  ¿ Setting the ios flags with `flag()` is a little more complicated.

  The flag argument is built by ORing `ios` flags

  ```
  cout.flags(ios::hex);
  cout.flags(ios::hex | ios::left);
  const long my_options = ios::left | ios::fixed;
  long old_options = cout.flags(myoptions);
  ```

  The `flags()` function sets the specified flags and clears all the others.
- The functions `setf()` and `unsetf()` can be used instead of `flags()`. They only modify the flags specified in their argument
  - The `setf()` function sets flags and `unsetf()` clears them.
  - Like `flags()` they return the current flag settings.
- There is a two argument version of `setf()` for setting flags in named bit fields:

  `cout.setf(ios::hex, ios::basefield);`

  This sets the `ios::hex` flag and clears all the other flags in `ios::basefield`. All the flags not in `ios::basefield` are unchanged.

  If the first argument is zero, all the flags in the field are cleared:

  `cout.setf(0,ios::basefield);`

7.4.1 Manipulators

- Manipulators an be used with the `<<` and `>>` operators to modify the state and parameters:

manipulator	**equivalent**
`setiosflags()`	single argument `setf()`
`resetiosflags()`	`unsetf()`
`setfill()`	`fill()`
`setw()`	`width()`
`setprecision()`	`precision()`

  - They do not return the current state.
  - The header file `<iomanip.h>` must be included.

- Other useful manipulators that do not need arguments are:

manipulator	**use**
`dec`	use octal notation
`hex`	use hexadecimal notation
`oct`	use decimal notation
`endl`	add `\n` and flush
`ends`	add `\0` and flush
`flush`	flush stream
`ws`	eat white space

  - These do not need `<iomanip.h>`.

- They are used like this:

```
cout << setwidth(10) << 13 << endl;
cout << hex << setw(10) << 13 << endl;
cout << dec << setiosflags(ios::left) << 13 << endl;
cout << setw(10) << hex << 13 << endl;
```

7.5 Detecting Errors and End of File

- The error state of a stream can be examined with

function	**returns nonzero (true) if**
`int eof();`	end of file seen
`int fail();`	next operation will fail
`int good();`	next operation might succeed
`int bad();`	stream corrupted

- When `good()` or `eof()` are true the previous operation succeeded.
- If `good()` is false all further operations are ignored.

  Calling the function `void clear()` with no arguments will resets the error state so that processing can continue. This function can be used to clear individual bits:

```
clear();
clear(ios::badbit);
```

- When end of file is encountered `eof()` and `fail()` are true:

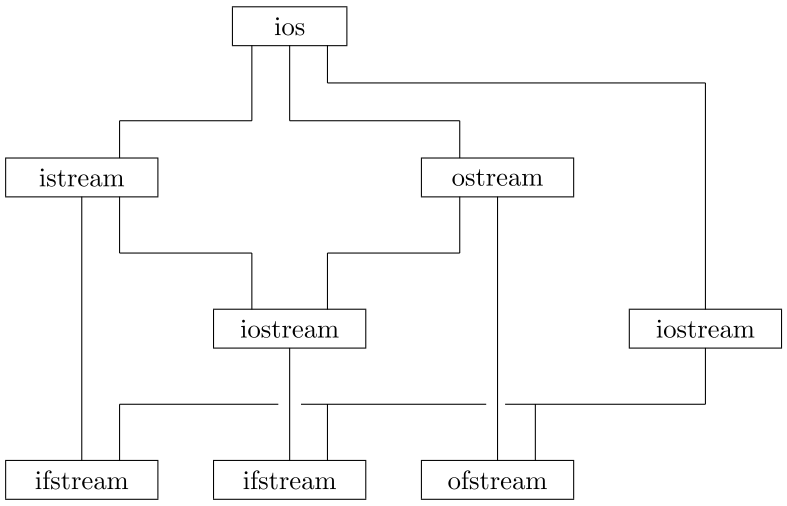
```
while ( !cin.get(c).fail() ) {
   // process c
}
if ( cin.eof() ) {
   // normal termination
}
else if ( cin.bad() ) {
   // Fatal error message
}
else  {
   cin.clear();
   // attempt to recover
}
```

- The stream state can also be accessed with `rdstate()`:

```
switch (cin.rdstate()) {
case ios::goodbit:
   // the last cin operation succeeded
   break;
case ios::eofdbit:
   // at end of file
   break;
case ios::failbit:
   // some kind of formatting error
   // Most likely not too bad
   break;
case ios::badbit:
   // cin characters might be lost
   break;
}
```

7.6 Named Files

The header file `<fstream.h>` declares four classes:



7.6.1 Opening and closing files

- When we open a named file we provide a name and a file mode.
- File modes are:

`ios::in`	open for reading
`ios::out`	open for output
`ios::app`	append
`ios::ate`	open and seek to end of file
`ios::nocreate`	fail if file does not exist
`ios::noreplace`	fail if file exists

  Files opened in modes `ios::out` and ios::app will be created if they do not exist.

  Modes can be combined by ORing them:

  – `ios::out | ios::nocreate` specifies an output file that must exist.

  – `ios::out | ios::in | ios::ate` specifies an input output file that should retain existing data.

- We open a file by defining a file object:

```
ifstream infile("report.doc",ios:in);
ofstream list("result.txt",ios::out);
fstream master("main.dta",ios::in|ios:out);
```

- `ifstream` defaults to input and the `ofstream` defaults to output. So we can write

```
ifstream infile("memo.doc");
ofstream list("result.txt");
```

- We can use the `ios` operator `!` to test if a file opened without error.

```
ofstream list("result.txt");
if ( !list ) cerr << "open failed!!!";
```

- Files can be closed explicitly:

```
ifstream in("memo.doc");
in.close();
```

  However, the stream's destructor will be called when the stream object goes out of scope.

7.6.2 Reading and writing files

Writing text to a file and reading it is easy:

```
// file file1.cpp
// Write to a disk file then read it!

#include <fstream.h>
#include <stdlib.h>

main()
{
   ofstream out("iotest1.txt");
   if ( !out ) {
      cerr << "Output open failed\n";
      exit(1);
   }
   out << "This is some test data.\n";
   out << "This is the last line\n";

   ifstream in("iotest1.txt");
   if ( !in ) {
      cerr << "Input open failed\n";
      exit(1);
   }
   char line[81];
   in.getline(line,81,'\n');
   while ( in ) {
      cout << '*' << line;
      in.getline(line,81,'\n');
   }
}
```

7.7 Command Line Parameters

- Parameters can be given with the command that executes a program.

  A program `xcopy` that copies file might be executed with the command:

  `xcopy file1.dat file2.dat`

- A program gets its parameters by declaring main() with two arguments

  `main( int argc, char* argv[] ) { ... }`

  The first argument gives the number of parameters.

  The second is an array of strings, each of which is one of the parameters.

  The first parameter is always the program name.

  For the command given above the value af `argc` is 3 and the elements of `argv` are:

  ```
  argv[0]      "xcopy"
  argv[1]      "file1.dat"
  argv[2]      "file2.dat"
  ```

- The following program shows how to process command line parameter:

  ```
  // file xcopy.cpp
  // command line arguments

  #include <fstream.h>
  #include <stdlib.h>

  void error( char* s1, char* s2 = "")
  {
     cerr << s1 << ' ' << s2 << endl;
     exit(1);
  }

  main(int argc, char* argv[])
  {
     if ( argc != 3 )
        error("Wrong number of arguments");
     ifstream in(argv[1]);
     if ( !in )
        error("Cannot open",argv[1]);
     ofstream out(argv[2]);
     if ( !out )
        error("Cannot open",argv[2]);

     char c;
     while ( out && in.get(c) )
        out.put(c);
  }
  ```

Appendix A

OBJECT ORIENTED DESIGN

A.1 The design process

- The design process has three stages:
  - Analysis: Defining the scope of the problem to be solved.
  - Design: Creating an overall structure for a system.
  - Implementation: Writing and testing the code.

  This is an iterative process.

- The following activities should permeate the process:
  - Experimentation.
  - Testing.
  - Analysis of the design and the implementation.
  - Documentation.
  - Management.

  Software maintenance is just more iterations though this process.

- It is important that analysis, design and implementation do not become too detached from each other.
  - The implementation platform and language can be ignored during the analysis phase, because only the problem domain is being considered.
  - The implementation language has to be considered during design, but remember that you do not have to use all the language's features all the time.

A.2 Classes in the design

- Classes represent concepts. There are two types of class in a system:
  - The classes that directly reflect concepts in the application domain.

    These are the concepts used by end-users to describe their problems and solutions.
  - The classes that are artifacts of the software implementation.

    These are the concepts used by the designers and programmers to describe their implementation techniques.
- Classes are arranged in hierarchies. They are composed into components.

A.3 Objects in the design

- Objects are instances of classes.
- There might be only one instance of a class in a design or there might be many.
- Objects are arranged in a hierarchy.

  Instance connections represent message paths between objects. This is similar to data modelling with entity relationship diagrams.
- Objects reflect the distinction between the application and design domains embodied in their classes.
- In C++ control or management objects will have to be introduced to initialise and sequence the programs processing. The main() function can be considered as sort of management object.

A.4 Design Steps

- If Object Oriented Analysis has been used, design can begin with a review of the application's classes and objects, and their relationships.

  If structured analysis has been used, then some OOA must be performed to establish the classes and objects in the problem domain before a review can begin.
- The steps are similar for both the analysis and design phases. The difference being the level of abstraction:

  The primary abstraction of the problem domain is analysis.

  The supporting abstractions of the implementation are design.
- The design steps are iterative.

A.4.1 Finding classes

- Talk to users, read any specification documents.

  Nouns are a good indicator for potential classes and objects.
- Talk to experts in the application area and to your colleagues. Software design is difficult on you own.
- Consider the problem domain in order to refine classes.

  Eliminate abstractions outside the problem domain.

  Check if any classes are duplicated.
- Introduce classes to support implementation.

  Additional implementation dependent classes will be uncovered during the design process.
- Use existing classes if possible. Look at other programs.

> "Software design is hard and we need all the help we can get."
> Stoustrup

A.4.2 Specifying attributes

- The attributes of a class define its state. Attributes are specified as data members in C++.
- Look for the *adjectives* used to describe classes and objects. These are a indication of potential class attributes.

  Form an a abstract view. Do not worry about hidden implementation detail too early.
- Look for attributes that are objects and classes:
  - Look for groups of attributes that may form classes in their own right.
  - Some of the attributes might be class objects. Represent these as class dependences.

A.4.3 Specifying dependencies

- Analyse the classes and establish:
  - Any "is a" relationships = inheritance.
  - Any "has a" relationships = containment.
  - Any collaborative relationships = friendship.
- Look for possible generic classes, like container classes such as tables or lists.
- Group classes into components.
- Review the distribution of operations in the class hierarchies:

  - Is an operation common to an entire class hierarchy or just to part of it?
  - Should an operation be available to the entire program or restricted to the class hierarchy?

- Use a graphical notation. Draw diagrams.

A.4.4 Specifying operations

- A good starting point is to look for verbs in discussions with users or in specification documents.
- Define the minimal set of operations required by the concept the represents.
  - Define the public interface by identify the operation required by the user of each class.

    Consider how an object of the class is to be constructed, copied (if at all) and destroyed.

    Consider which operations could be added for notational convenience, but include only a few really important ones.
  - Consider which operations are to be virtual.

    These are operations for which the class can act as an interface for an implementation supplied by a derived class.
  - Consider what commonality of naming and functionality can be achieved across all the classes of a component.
- Define the private and protected interfaces:

  Identify helper functions to support the operation of the public interface.
- It can be useful to classify operations in terms of how they effect the internal state of an object:
  - Foundation operations: Constructors, destructors and copy operations.
  - Selectors: Operations that do not modify the state of an object.
  - Modifiers: Operations that do alter the state of an object.
  - Conversion operations: Operations that produce an object of another type based on the state of the object to which they are applied.
  - Iterators: Operations that somehow allow access to or use a sequence of contained objects.

A.4.5 Specifying object connections

- Define the system's class instances. Its objects.
- Establish object connections.
  - Connections represent message paths.

- At higher levels of abstraction represent the connections in a cardinal form:

  Object connections can be zero-to-one, one-to-one or one-to-many, etc. Use a notation similar to entity relationship diagrams. (Data modeling can be viewed as a subset of OOA.)

- At lower (program design) levels of abstraction replace the x-to-many connections with container objects and one-to-one object connections.

  Suitable containers might be tables or files (or even an object oriented data base).

  In simple cases the container can be implemented as a raw data structure, such as an array of objects.

  Classes have the advantage of providing constructors and destructors, even if the contained objects are declared as public for simplicity.

Appendix B

STROUSTRUP'S RULES OF THUMB

These rules of thumb are taken from Bjarne Stroustrup's book The C++ Programming Language, Second Edition.
He advises the reader not to take them too literally.

> "To write a good program takes intelligence, taste and patience. You are not going to get it right first time; experiment!"

B.1 While Learning C++

1. When you program, you create a concrete representation of the ideas in your solution to some problem. Let the structure of the program reflect those ideas as directly as possible:
   (a) If you can think of "it" as a separate idea, make it a class.
   (b) If you can think of "it" as a separate entity, make it an object of some class.
   (c) If two classes have something significant in common, make that commonality a base class.
   (d) If a class is a container of objects make it a template.
2. When you define a class that does not implement a mathematical entity like a matrix or a complex number or a low-level type such as a linked list:
   (a) Don't use global data.
   (b) Don't use global (nonmember) functions.
   (c) Don't use public data members.
   (d) Don't use friends except to avoid (a), (b) or (c).
   (e) Don't access the data members of other object directly.
   (f) Don't put a type field in a function; use virtual functions.
   (g) Don't use inline functions, except as a significant optimisation.

B.2 Design and Development

B.2.1 Concepts

- The most important aspect of software design is to be clear about what you are trying to build.
- Successful software development is a long term activity.
- The systems we construct tend to be at the limit of the complexity that we and our tools can handle.
- There are no good "cookbook" methods that can replace intelligence, experience and good taste in design and programming.
- Experimentation is essential all non-trivial software development.
- Design and programming are iterative activities. They converge in steps towards, but never reach, a perfect solution.
- The separate phases of a software project, such as design, programming and testing, cannot be strictly separated.
- Programming and design cannot be considered without also considering the management of these activities.

B.2.2 Approach

- Know what you are trying to achieve.
- Have specific and tangible aims.
- Don't try to use technological fixes for sociological problems.
- Consider the long term
  - in design and
  - in the treatment of people.
- use existing systems as models, inspiration and as starting points.
- design for change:
  - flexibility,
  - extensibility,
  - portability and
  - re-use.
- Document, market and support re-usable components.
- Reward and encourage re-use of
  - design,
  - libraries, and

  - classes.

- Focus on component design.
  - Use classes to represent concepts.
  - Define interfaces to reveal the minimal amount of information needed.
  - Keep interfaces strongly typed wherever possible.
  - Use application level types in interfaces wherever possible.
- Repeatedly review and refine both the design and the implementation.
- Use the best tools available for testing and for analysing
  - the design and
  - the implementation.
- Experiment, analyse and test as early as possible and as often as possible.
- Keep it simple; as simple as possible, but no simpler.
- Keep it small; don't add features "just in case".
- Don't forget about efficiency.
- Keep the level of formality appropriate to the scale of the project.
- Don't forget that designers, programmers, and even managers are human.

B.3 Design and C++

- Evolve use towards data abstraction and object-oriented programming.
  - Adopt new technology gradually; don't rush.
  - Use C++ features and techniques as needed (only).
  - Match design and programming style.
- Focus on component design.
- Use classes to represent concepts.
  - Use public inheritance to represent is a relationships.
  - Use membership to represent has a relationships.
  - Make sure uses dependencies are understood, non-cyclic whenever possible, and minimal.
  - Actively search for commonality in the concepts of the application and implementation, and represent the resulting more general concepts as base classes.
- Define interfaces to reveal the minimal amount of information needed:
  - Use private data and member functions wherever possible.

  - Use the public/protected distinction to distinguish between the needs of designers of derived classes and general users.

- Minimise an interface's dependencies on other interfaces.
- Keep interfaces strongly typed.
- Express interfaces in terms of application-level types.

Appendix C

OPERATOR SUMMARY

Unary operators and assignment operators are right associative. All others are left associative. Each box holds operators with the same precedence. The boxes are in desending order of precedence.

Operator	Description	Example
`::`	scope resolution	`class_name :: member`
`::`	global	`:: name`
`.`	member selection	`object . member`
`->`	member selection	`pointer -> member`
`[]`	subscripting	`pointer [ expr ]`
`()`	function call	`expr ( expr_list )`
`()`	value construction	`type ( expr_list )`
`sizeof`	size of object	`sizeof expr`
`sizeof`	size of type	`sizeof ( type )`
`++`	post increment	`lvalue ++`
`++`	pre increment	`++ lvalue`
`--`	post decrement	`lvalue --`
`--`	pre decrement	`-- lvalue`
`~`	complement	`~ expr`
`!`	not	`! expr`
`-`	unary minus	`- expr`
`+`	unary plus	`+ expr`
`&`	address of	`& lvalue`
`*`	dereference	`* expr`
`new`	create	`new type`
`delete`	destroy	`delete pointer`
`delete[]`	destroy array	`delete [] pointer`
`()`	cast	`( type ) expr`

Operator	Description	Example
`*`	multiply	`expr * expr`
`.*`	member selection	`object . pointer-to-member`
`->*`	member selection	`pointer -> pointer_to_member`
`/`	divide	`expr / expr`
`%`	modulo	`expr % expr`
`+`	add	`expr + expr`
`-`	subtract	`expr - expr`
`<<`	shift left	`expr << expr`
`>>`	shift right	`expr >> expr`
`<`	less than	`expr < expr`
`<=`	less than or equal	`expr <= expr`
`>`	greater than	`expr > expr`
`>=`	greater than or equal	`expr >= expr`
`==`	equal	`expr == expr`
`!=`	not equal	`expr != expr`
`&`	bitwise AND	`expr & expr`
`^`	bitwise exclusive OR	`expr ^ expr`
`|`	bitwise inclusive OR	`expr | expr`
`&&`	logical AND	`expr && expr`
`||`	logical OR	`expr || expr`
`? :`	conditional expression	`expr ? expr : expr`
`=`	simple assignment	`lvalue = expr`
`*=`	multiply and assign	`lvalue *= expr`
`/=`	divide and assign	`lvalue /= expr`
`%=`	modulo and assign	`lvalue %= expr`
`+=`	add and assign	`lvalue += expr`
`-=`	subtract and assign	`lvalue -= expr`
`<<=`	shift left and assign	`lvalue <<= expr`
`>>=`	shift right and assign	`lvalue >>= expr`
`&=`	AND and assign	`lvalue &= expr`
`|=`	inclusive OR and assign	`lvalue |= expr`
`^=`	exclusive OR and assign	`lvalue ^= expr`
`,`	comma (sequencing)	`expr , expr`

Appendix D

C++ KEYWORDS

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Appendix E

Arithmetic Conversions

These lists explain, in detail, the conversions performed on operands during the evaluation of expressions. Knowledge at this low level is not normally needed.

E.1 The usual arithmetic conversions

1. If either operand is of type long double, the other is converted to long double.
2. Otherwise, if either operand is double, the other is converted to double.
3. Otherwise, if either operand is float, the other is converted to float.
4. Otherwise, the integral promotions are performed on both operands.
5. Then, if either operand is unsigned long the other is converted to unsigned long.
6. Otherwise, if one operand is a long int and the other unsigned int, then if a long int can represent all the values of unsigned int, the unsigned int is converted to a long int; otherwise both operands are converted to unsigned long int.
7. Otherwise, if either operand is unsigned, the other is converted to unsigned.
8. Otherwise, both operands are int.

E.2 Integral promotions

1. A char, short int, enumerator or an int bit-field, in both their signed and unsigned varieties, may be used wherever an int may be used.
2. If an int can represent all the values of the original type, the value is converted to int; otherwise it is converted to unsigned int.

Appendix F

C Library Functions

For more information about C functions look at the relevent header files on your computer. Alternatively, for more help, consult any good book on ANSI C, such as Banahan, M., D. Brady and M. Doran, *The C Book, 2nd Edition*, Addison-Wesley, 1991.

F.1 Functions by Type

F.1.1 Type and Conversion Functions

`atof`	Convert string to double	`<stdlib.h>`
`atoi`	Convert string to integer	`<stdlib.h>`
`atol`	Convert string to long	`<stdlib.h>`
`isalnum`	Alphanumeric character?	`<ctype.h>`
`isalpha`	Alphabetic character?	`<ctype.h>`
`iscntrl`	Control character?	`<ctype.h>`
`isdigit`	Decimal digit?	`<ctype.h>`
`isgraph`	Printable character but not space?	`<ctype.h>`
`islower`	Lower case alphabetic character?	`<ctype.h>`
`isprint`	Printable character?	`<ctype.h>`
`ispunct`	Not alphanumeric or space?	`<ctype.h>`
`isspace`	White space?	`<ctype.h>`
`isupper`	Upper case alphabetic character?	`<ctype.h>`
`isxdigit`	Hexadecimal digit?	`<ctype.h>`
`strtod`	String to long	`<stdlib.h>`
`strtol`	String to double	`<stdlib.h>`
`strtoul`	String to unsigned long	`<stdlib.h>`
`tolower`	Convert char to lower case	`<ctype.h>`
`toupper`	Convert char to upper case	`<ctype.h>`

F.1.2 Mathematical Functions

`abs`	Absolute value of an int	`<stdlib.h>`
`acos`	Arccosine	`<math.h>`
`asin`	Arcsine	`<math.h>`
`atan`	Arctangent	`<math.h>`
`atan2`	Principle value of arctangent of y/x	`<math.h>`
`atof`	Convert string to double	`<stdlib.h>`
`atoi`	Convert string to integer	`<stdlib.h>`
`atol`	Convert string to long	`<stdlib.h>`
`ceil`	Ceiling	`<math.h>`
`cos`	Cosine	`<math.h>`
`cosh`	Hyperbolic cosine	`<math.h>`
`div`	Quotient and remainder of int divide	`<stdlib.h>`
`exp`	e^x	`<math.h>`
`fabs`	Absolute value of a double	`<math.h>`
`floor`	Floor	`<math.h>`
`fmod`	Floating point remainder of x/y	`<math.h>`
`frexp`	Floating point number to normalized fraction and integer power of two	`<math.h>`
`labs`	Absolute value of a long	`<stdlib.h>`
`ldexp`	$2^y x$	`<math.h>`
`ldiv`	Quotient and remainder of long divide	`<stdlib.h>`
`log`	$\ln x$	`<math.h>`
`log10`	$\log x$	`<math.h>`
`modf`	Floating point to integer fractional parts	`<math.h>`
`pow`	x^y	`<math.h>`
`rand`	Generate a random integer	`<stdlib.h>`
`sin`	Sine	`<math.h>`
`sinh`	Hyperbolic sine	`<math.h>`
`sqrt`	Square root	`<math.h>`
`srand`	Seed the random number generator	`<stdlib.h>`
`strtod`	String to long	`<stdlib.h>`
`strtol`	String to double	`<stdlib.h>`
`strtoul`	String to unsigned long	`<stdlib.h>`
`tan`	Tangent	`<math.h>`
`tanh`	Hyperbolic tangent	`<math.h>`

F.1.3 Miscellaneous Functions

`abort`	Abnormality terminate program	`<stdlib.h>`
`atexit`	Register function for auto call on exit	`<stdlib.h>`
`bsearch`	Binary search sorted array	`<stdlib.h>`
`exit`	Normal exit	`<stdlib.h>`
`getenv`	Obtain environment information	`<stdlib.h>`
`qsort`	Sort array	`<stdlib.h>`
`rand`	Generate a random integer	`<stdlib.h>`
`signal`	Invoke a function to handle a signal	`<signal.h>`
`srand`	Seed the random number generator	`<stdlib.h>`
`system`	Process system command	`<stdlib.h>`
`va_arg`	Variable argument list access	`<stdarg.h>`
`va_end`	Variable argument list termination	`<stdarg.h>`
`va_start`	Variable argument list intitalization	`<stdarg.h>`

F.1.4 Input/Output Functions

`clearerr`	Clear a file error and eof	`<stdio.h>`
`fclose`	Close a file	`<stdio.h>`
`feof`	End of file?	`<stdio.h>`
`ferror`	File error?	`<stdio.h>`
`fflush`	Flush buffers	`<stdio.h>`
`fgetc`	Read a character from a file	`<stdio.h>`
`fgetpos`	Get position within a file	`<stdio.h>`
`fgets`	Read a string from a file	`<stdio.h>`
`fopen`	Open a file	`<stdio.h>`
`fprintf`	Write formatted output to a file	`<stdio.h>`
`fputc`	Write a character to a file	`<stdio.h>`
`fputs`	Write a string to a file	`<stdio.h>`
`fread`	Read several items from a file	`<stdio.h>`
`freopen`	Reopen a file	`<stdio.h>`
`fscanf`	Read formatted input from a file	`<stdio.h>`
`fseek`	Move within a file	`<stdio.h>`
`fsetpos`	Move within a file	`<stdio.h>`
`ftell`	Get position within a file	`<stdio.h>`
`fwrite`	Write a number of items to a file	`<stdio.h>`
`getc`	Read a character from a file	`<stdio.h>`
`getchar`	Read a character from stdin	`<stdio.h>`
`gets`	Read a string from stdin	`<stdio.h>`
`perror`	Print error	`<stdio.h>`
`printf`	Write formatted output to stdout	`<stdio.h>`
`putc`	Write a character to a file	`<stdio.h>`
`putchar`	Write a character to stdout	`<stdio.h>`
`puts`	Write a string to stdout	`<stdio.h>`
`remove`	Delete a file	`<stdio.h>`
`rename`	Rename a file	`<stdio.h>`
`rewind`	Move to the start of a file	`<stdio.h>`
`scanf`	Read formatted output from stdin	`<stdio.h>`
`setbuf`	Change buffering strategy	`<stdio.h>`
`setvbuf`	Change buffering strategy	`<stdio.h>`
`tmpfile`	Create temporary file	`<stdio.h>`
`tmpnam`	Generate a unique file name	`<stdio.h>`
`ungetc`	Unget a character	`<stdio.h>`
`vfprintf`	Write formatted output to a file	`<stdio.h>`
`vprintf`	Write formatted output to stdout	`<stdio.h>`
`vsprintf`	Write formatted output to stdout	`<stdio.h>`

F.1.5 Memory Functions

`calloc`	Allocate storage	`<stdlib.h>`
`free`	Free storage	`<stdlib.h>`
`malloc`	Allocate storage	`<stdlib.h>`
`memchr`	Find first of char in memory block	`<string.h>`
`memcmp`	Compare blocks of memory	`<string.h>`
`memcpy`	Copy block of memory	`<string.h>`
`memmove`	Copy block of memory	`<string.h>`
`memset`	Set value of memory block	`<string.h>`
`realloc`	Change the size of allocated storage	`<stdlib.h>`

F.1.6 String Functions

`asctime`	Convert time to string	`<time.h>`
`ctime`	Convert time to string	`<time.h>`
`sprintf`	Write formatted output to a string	`<stdio.h>`
`sscanf`	Read formatted input from a string	`<stdio.h>`
`strcat`	Concatenate strings	`<string.h>`
`strchr`	Find first of char in string	`<string.h>`
`strcmp`	Compare strings	`<string.h>`
`strcoll`	Compare strings	`<string.h>`
`strcpy`	Copy string	`<string.h>`
`strcspn`	Find first of char set in string	`<string.h>`
`strerror`	String equivalent of errno	`<string.h>`
`strftime`	Convert time to string	`<time.h>`
`strlen`	Length of string	`<string.h>`
`strncat`	Concatenate strings	`<string.h>`
`strncmp`	Compare strings	`<string.h>`
`strncpy`	Copy string	`<string.h>`
`strpbrk`	Find last of char set in string	`<string.h>`
`strrchr`	Find last of char in string	`<string.h>`
`strspn`	Length of first char set block in string	`<string.h>`
`strstr`	Find substring	`<string.h>`
`strtok`	Break string into tokens	`<string.h>`
`strxfrm`	String transform	`<string.h>`

F.1.7 Date and Time Functions

`asctime`	Convert time to string	`<time.h>`
`clock`	Time in 'ticks'	`<time.h>`
`ctime`	Convert time to string	`<time.h>`
`difftime`	Difference between two calendar times	`<time.h>`
`gmttime`	Greenwich mean time	`<time.h>`
`localtime`	Local time	`<time.h>`
`mktime`	Calendar time	`<time.h>`
`strftime`	Convert time to string	`<time.h>`
`time`	Calendar time	`<time.h>`

F.2 Functions by Library

F.2.1 ctype

`isalnum`	Alphanumeric character?
`isalpha`	Alphabetic character?
`iscntrl`	Control character?
`isdigit`	Decimal digit?
`isgraph`	Printable character but not space?
`islower`	Lower case alphabetic character?
`isprint`	Printable character?
`ispunct`	Not alphanumeric or space?
`isspace`	White space?
`isupper`	Upper case alphabetic character?
`isxdigit`	Hexadecimal digit?
`tolower`	Convert char to lower case
`toupper`	Convert char to upper case

F.2.2 math

`acos`	Arccosine
`asin`	Arcsine
`atan`	Arctangent
`atan2`	Principle value of arctangent of y/x
`ceil`	Ceiling
`cos`	Cosine
`cosh`	Hyperbolic cosine
`exp`	e^x
`fabs`	Absolute value of a double
`floor`	Floor
`fmod`	Floating point remainder of x/y
`frexp`	Floating point number to
normalized fraction and integer power of two	
`ldexp`	$2^y x$
`log`	$\ln x$
`log10`	$\log x$
`modf`	Floating point to integer and
fractional parts	
`pow`	x^y
`sin`	Sine
`sinh`	Hyperbolic sine
`sqrt`	Square root
`tan`	Tangent
`tanh`	Hyperbolic tangent

F.2.3 signal

`signal`	Invoke a function to handle a signal

F.2.4 stdarg

`va_arg`	Variable argument list access
`va_end`	Variable argument list termination
`va_start`	Variable argument list intitalization

F.2.5 stdio

`clearerr`	Clear a file error and eof
`fclose`	Close a file
`feof`	End of file?
`ferror`	File error?
`fflush`	Flush buffers
`fgetc`	Read a character from a file
`fgetpos`	Get position within a file
`fgets`	Read a string from a file
`fopen`	Open a file
`fprintf`	Write formatted output to a file
`fputc`	Write a character to a file
`fputs`	Write a string to a file
`fread`	Read several items from a file
`freopen`	Reopen a file
`fscanf`	Read formatted input from a file
`fseek`	Move within a file
`fsetpos`	Move within a file
`ftell`	Get position within a file
`fwrite`	Write a number of items to a file
`getc`	Read a character from a file
`getchar`	Read a character from stdin
`gets`	Read a string from stdin
`perror`	Print error
`printf`	Write formatted output to stdout
`putc`	Write a character to a file
`putchar`	Write a character to stdout
`puts`	Write a string to stdout
`remove`	Delete a file
`rename`	Rename a file
`rewind`	Move to the start of a file
`scanf`	Read formatted output from stdin
`setbuf`	Change buffering strategy
`setvbuf`	Change buffering strategy
`sprintf`	Write formatted output to a string
`sscanf`	Read formatted input from string
`tmpfile`	Create temporary file
`tmpnam`	Generate a unique file name
`ungetc`	Unget a character
`vfprintf`	Write formatted output to a file
`vprintf`	Write formatted output to stdout
`vsprintf`	Write formatted output to stdout

F.2.6 stdlib

`abort`	Abnormality terminate program
`abs`	Absolute value of an int
`atexit`	Register function for auto call on exit
`atof`	Convert string to double
`atoi`	Convert string to integer
`atol`	Convert string to long
`bsearch`	Binary search sorted array
`calloc`	Allocate storage
`div`	Quotient and remainder of int divide
`exit`	Normal exit
`free`	Free storage
`getenv`	Obtain environment information
`labs`	Absolute value of a long
`ldiv`	Quotient and remainder of long divide
`malloc`	Allocate storage
`qsort`	Sort array
`rand`	Generate a random integer
`realloc`	Change the size of allocated storage
`srand`	Seed the random number generator
`strtod`	String to long
`strtol`	String to double
`strtoul`	String to unsigned long
`system`	Process system command

F.2.7 string

`memchr`	Find first of char in memory block
`memcmp`	Compare blocks of memory
`memcpy`	Copy block of memory
`memmove`	Copy block of memory
`memset`	Set value of memory block
`strcat`	Concatenate strings
`strchr`	Find first of char in string
`strcmp`	Compare strings
`strcoll`	Compare strings
`strcpy`	Copy string
`strcspn`	Find first of char set in string
`strerror`	String equivalent of errno
`strlen`	Length of string
`strncat`	Concatenate strings
`strncmp`	Compare strings
`strncpy`	Copy string
`strpbrk`	Find last of char set in string
`strrchr`	Find last of char in string
`strspn`	Length of first char set block in string
`strstr`	Find substring
`strtok`	Break string into tokens
`strxfrm`	String transform

F.2.8 time

`asctime`	Convert time to string
`clock`	Time in ’ticks’
`ctime`	Convert time to string
`difftime`	Difference between two calendar times
`gmttime`	Greenwich mean time
`localtime`	Local time
`mktime`	Calendar time
`strftime`	Convert time to string
`time`	Calendar time