Creating an object oriented computer program using C++

Level 3

Notes

for

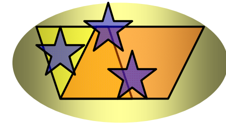
City & Guilds

7540 Unit 034

Written for Microsoft Visual C++ or Borland C++ compilers

©Tench Computing Ltd

Tench Computing Ltd
Pines
Glendale Road
Burgess Hill
West Sussex
RH15 0EJ



Web address: www.tenchcomputing.co.uk
Email address: jtench@globalnet.co.uk

About the author: *Jackie Tench MSc, ACIB, Cert Ed*

Jackie started her working career in branch banking with the Midland Bank (now HSBC) and was transferred to their Computing Department after achieving 100% in their ability test for programmers. She then worked for more than a decade in this department and was one of the first women to achieve a junior management grade at the age of 21. She attended a significant number of IBM programming training courses during her time there.

Jackie was the first woman to pass the ACIB (Associate Chartered Institute of Bankers) examinations in the Midland Bank (HSBC) and the youngest person at 21 years of age.

Jackie then left to raise a family but still found time to teach part-time at a college in Sheffield and to obtain a MSc in Computing and a Cert Ed in teaching.

When her children were old enough Jackie returned to work full-time and was a Senior Lecturer in Software Engineering and Computer Studies at a college in Brighton for nearly 10 years teaching all levels up to and including HND.

Therefore, Jackie has both considerable business knowledge and qualifications plus wide experience in practical computing and training – covering areas such as structured design, analysis, coding, testing and implementing software applications plus training students to fulfil an important role in the computer industry.

Jackie has worked as a consultant for several blue chip companies and examination boards using her software engineering and educational training skills and is now one of the foremost experts in computing with an extensive knowledge of programming languages and applications.

If you wish to see all of Jackie's current training manuals please visit her web site.

Copyright ©1999 Tench Computing Ltd

Microsoft, Windows, Windows NT or other Microsoft products referenced herein are either the trademarks or registered trademarks of the Microsoft Corporation. Other trademarks for products referenced herein are also acknowledged.

All rights are reserved and no part of this training manual may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the purchase of a licence.

This training manual is sold subject to licence and on condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated without the prior consent of Tench Computing Ltd in any form of binding or cover other than that in which it is issued and without a similar condition being imposed on the subsequent purchaser. Any program listings within this training manual may be entered, stored and executed in a computer but they may not be reproduced for publication.

Contents Page

Introduction
- Library files ... 1
- Reserved words/keywords ... 1
- Preprocessor
- Comments ... 2
- Indentation ... 2
- Algorithms ... 2

Basic data types
- Variables ... 3
- Consistency ... 3
- Variable names ... 3
- Data types
  - Integer ... 3
  - Floating point ... 4
  - Character ... 4
  - Character strings ... 5
  - Constants ... 5
- Local variables
  - Automatic (auto) ... 6
  - Static ... 6
  - Static class variable ... 7
- Global variables ... 7
- Enumeration ... 8
- Arithmetic operators ... 9
- Assignment operators ... 10
- Relational operators ... 11
- Logical operators ... 11

Validation
- Types of validation
  - Date checks ... 12
  - Range checks ... 12
  - Presence check ... 13
  - Type checks
    - Numeric ... 13
    - Alphabetic ... 13

Predefined functions
- String functions
  - strlen() function ... 14
  - strcat() function ... 14
  - strcpy() function ... 15
  - gets() function ... 15
  - puts() function ... 16
- Conversion functions
  - atoi() ... 16
  - atof() ... 16
  - toupper() function ... 17
  - tolower() function ... 17

C++ streams libraries
    Standard I/O 18
Questions 1 19
Functions
    Passing arguments to functions 21
    Call by value 21
    Return statement 22
    Call by reference 23
    Default parameter values 25
switch statement 26
continue statement 28
Overloading functions 29
Macros 31
Arrays
    One-dimensional array 32
    Two-dimensional array 33
    Apply a one-dimensional function on each subarray 33
    Apply a one-dimensional function to the whole array 34
    Apply a two-dimensional function 35
Pointers
    Assignment 36
    Value finding 36
    Use a pointer address 36
    Increment or decrement a pointer 36
    Pointers to arrays 37
    Array name argument to function 37
    Casting 38
    Pointer notation 39
Structures
    Access structure members 40
    Array of structures 40
    Identify members of a structure array 41
    Function arguments
        Pass a structure address 42
        Pass an array of structures 42
Questions 2 44
Class
    Attributes/properties 48
    Behaviour 48
    Structure of a class 49
    Instance variables 49
    Class variables 49
    Methods (functions) 49
    Accessing instance variables 53
    Calling methods 53
        Scope resolution operator:: 53
        this hidden pointer 53
Constructors and destructors
    Constructor
        Default constructor 54

User-defined constructor 54
Destructor 54
Inline functions 55
Class variables 58
Inheritance
Define derived classes 60
Friend functions 64
Files
Text files 65
Binary files 65
Sequential files 65
Open a file 65
close() function 67
get() and out() functions 67
getline() function 67
Printer output 69
Questions 3 70
Testing
Purpose of testing 73
Compilation error 73
Logical error 73
Run-time error 73
Target environment 73
Test plan 74
Test data 74
Test log 77
Change control 77
Test plan and test log forms 77
Check digits 79
Dry run 81
Member example 84
User instructions
Member user instructions 96
Error messages 97
Large-scale projects
Change control 98
Interface 99
Integration 99
Technical documentation 100
Concepts of Object Orientation
Objects 102
Class 102
Instantiation 102
Abstraction 102
Data abstraction 103
Encapsulation 103
Attributes 103
Methods 103
Object classification 104
Subclassing 104

Inheritance 104
Messaging 104
Types 104
Polymorphism 105
Classification of objects 106
Object Oriented design
Object model 110
Encapsulation 110
Class interface 111
Reusability 111
Questions 4 112
Sample assignment 115
Sample questions 117

Introduction

Library files

C++ contains pre-written functions that are stored in library files.
These library files can be included in a program so that the program can use the pre-written functions. The library files are provided to save a programmer writing commonly used code from scratch. All the standard functions e.g. for input and output which are needed in a program are contained in the library files.

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

at the start of a program indicates that the library file **iostream.h** for input and output stream functions is to be included in the program.

Reserved words/keywords

Reserved words are words that are part of the programming language. In C++ words like **while** and **switch** are reserved words that have a special meaning.

Preprocessor

C++ uses a preprocessor which scans the source code and looks for any preprocessor directives. A preprocessor directive starts with a # in column 1 of a line.
The preprocessor scans the code and looks for any # directives and processes them before the compiler compiles the code. #include directives tell the preprocessor to include the code from the specified file into the source code to be compiled.

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

This directive tells the preprocessor to include the file iostream.h and the brackets <> tell it that it is a standard include file.

```
#include "myfile.h"
```

This directive tells the preprocessor to include the file myfile.h and the quotes "" tell it to look first in the current directory and then in the standard include directories. A pathname can be specified if the file is not in the current directory. This allows you to create your own header file and include it in the same way as the pre-written library files.

The # symbol can also be used for other directives. #define can be used to define constants. By convention the name of the constant is in upper case to distinguish it from a variable. #define is the C method of defining constants.

```
#define PI 3.142
// N.B. there is no = sign and also no ; at the end of the line
#define MESSAGE1 "This is a string\n"
```

When the preprocessor finds a #define it replaces all occurrences of the constant name with the constant value. In the above examples where it finds PI in the code it will replace it with 3.142 and where it finds MESSAGE1 it will replace it with "This is a string\n"

Comments

Comments are used to make programs more readable for the programmer. They are ignored by the compiler.
// This is used for a single line comment
/* This is used for multiple lines
  of comments */

> Do not forget to put the end comment symbols */ when you use the /*   */ format for multiple comment lines. If the */ end comment is omitted all lines of code up to the next */ or until the end of the code if no further comments are used will be ignored because the compiler will think they are comment lines.

Indentation

Code is normally indented for each block of code delimited by braces {}. Indentation is used to make program code more readable and is ignored by the compiler.

Algorithms

Algorithms are written to provide software solutions. Algorithms could be written in pseudocode, structured English or a programming language. All algorithms for software require the use of program constructs. The three main program constructs are sequence, selection and iteration.

Sequence is where the instructions are executed one after the other.

Selection is where different paths can be taken dependent on a condition or choice. Selection uses the **if** and **switch** statements.

Iteration (repetition) is where a block of instructions is repeated. The block of instructions can be repeated a set number of times or repeated dependent on a condition. Iteration is achieved using the **while**, **do…while** or **for** constructs.

Basic data types

Variables

Variables are used for data items whose value will be changed while a program is running and must be declared before being used. To do this a type must be specified and a variable name.

Consistency

Consistency should be maintained within software when naming constants, variables, classes, functions etc. This makes the code more understandable. Generally constants are in uppercase and variables are in lowercase or possibly an initial capital with lowercase. C++ is case sensitive so that the variable names num1, NUM1 and Num1 would be seen as different variables by the compiler.

Variable names

Variable names must not be the same as any of the keywords in the programming language. Use meaningful names for variables. The number of characters is normally up to 32. The name can be made up of lowercase letters, uppercase letters, the digits and the underscore _ which is counted as a letter. Variable names must not be the same as the keywords (reserved words), which are part of the C++ language, e.g. float, short.

The first character must be a letter or an underscore. Variable names are case sensitive indx, Indx, INDX are all seen as different variable names.

Data types

The following keywords can be used to declare variable types:

int long short unsigned char float double

Integer

Integers are whole numbers.
The **int** type is a signed integer (allows positive and negative whole numbers) and uses 1 machine word for storage. Different computers have different size machine words. This means that the range of numbers that can be stored is machine dependent.

The following are examples of variable declarations for signed integers.

```
int num1;          // num1 is a signed integer type
// num2 and num3 are declared on same line as
// signed integers
int num2,num3;
// num4 is a signed integer type with an initial value of 10
int num4=10;
// num5 and num6 are declared on the same line and initialised
int num5=32,num6=15;
```

The **unsigned int** type allows a larger range of positive numbers but no negative numbers.

```
// num1 is declared as an unsigned integer
unsigned int num1;
```

The **short int** or **short** type may use less storage than an **int** type

```
      short int num1;   // num1 is declared as a short int
// num2 is declared as a short int. The word int may be omitted.
      short num2;
```

The **long int** or **long** type may use more storage than an **int** type.

```
// num2 is declared as a long int and initialised to 10
      long int num2=10L; // L must be used to indicated a long
// num3 is declared as a long int. The int may be omitted
      long num3;
```

Floating point

The **float** type is used for numbers that contain decimal places.
The following are examples of declarations for floating point numbers.

```
      float num1;          // num1 is declared as float
// num2 and num3 are declared as float on the same line
      float num2,num3;
// num5 is declared as float and initialised as 55.62
      float num5=55.62;
```

The **double** type is used to double the storage space allowed for a floating point number. This allows larger numbers to be stored.

```
      double num4;      // num4 is declared as a double size float
```

Character

The char type is used to store one character and takes up 1 byte of storage.
The following are examples of declarations for a single character.

```
      char response;    // response is declared as a char
// grade and code are declared on the same line as char
      char grade,code;
// yes is declared as a char and initialised as the character y.
// Note the use of single quotes round a character
      char yes='y';
```

> If you miss out the quotes around a character the compiler will see it as a variable name. For example char yes=y; will be interpreted by the compiler as assign the variable y to the variable yes.

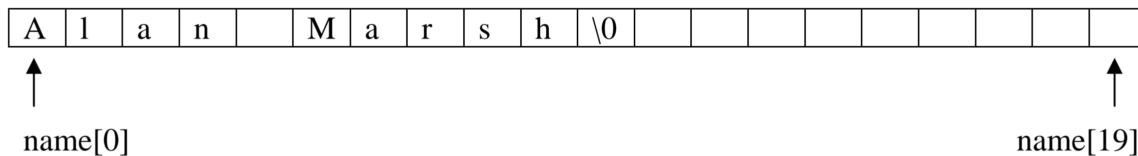
The computer holds characters internally in a binary format. Each character is held as 1 byte. One byte is 8 bits (binary digits). The PC uses the ASCII character set to hold characters internally. For instance the character A is held internally as the decimal number 65 which is 01000001 in binary, B is 66 (01000010) and C is 67 (01000011) and so on to Z. The character a is held internally as the decimal number 97 which is 01100001, b is 98 (01100010), c is 99 (01100011) and so on to z.

Character strings

Character strings contain a series of one or more characters.
The char type is used to declare character strings but the maximum size the character string can contain is also specified.

```
//name is declared as a character string with a maximum size of 20
      char name[20];
```

Strings have the null character ('\0') stored at the end so that a character string declared with a maximum size of 20 can contain a maximum of 19 characters plus the null character.
If the variable name contained the string "Alan Marsh" it would be stored as follows:



The character string "Alan Marsh" is 10 characters long but it needs 11 storage locations. The storage locations for the variable **name** are numbered 0 to 19. Strings are delimited by double quotes.

The string "X" is not the same as the character 'X'. 'X' is a basic type (char) and takes up 1 byte of storage whereas "X" is a derived type, an array of char and takes up 2 bytes of storage (the X and the null character). The null character ('\0') is used at the end of a string as a delimiter. A string does not need to use the whole of the storage area reserved for it so a delimiter is needed to indicate where the end of the string is.

When you create and manipulate strings in a program by building up a string from single characters you must make sure that you add a null character ('\0') at the end of the string. If a null character is not added as a delimiter at the end of the string the end of the string will not be known. For instance a string without a null character as a delimiter that is output using a cout statement will output characters until the end of memory or until if finds a null character whichever comes first.

Constants

Constants are used for data items whose values will not be changed while a program is running. The keyword **const** is used at the start of the declaration to specify a constant. By convention constant names are specified in uppercase to distinguish them from variables, which are lowercase. Values should not be put directly into code but declared as constants.

```
const float PI=3.142;
const int MAX=30;
```

Constants can be declared outside any class or function or within a function. A constant cannot be declared as a data member of a class.

You must declare variables with the correct data type. If you use the wrong data type you will get a type mismatch when you assign a value to the variable.

Local variables

Automatic (auto)

By default variables are automatic (auto). An automatic variable has local scope. It comes into existence when it is declared and space is reserved for it. It disappears when it goes out of scope. The enclosing braces define its scope.

If a variable is declared within a **for** loop, its scope is only within the for loop.

```
for (int k = 0; k < 2; k++) // k is declared and is visible
{
      cout << k;
}
// k is no longer visible and goes out of scope.
```

An automatic variable declared at the start of a function is visible throughout that function while the function is executing and disappears when the function finishes executing. Any value held in the variable at the end of the function execution is lost. The next time the function is executed the automatic variables will be reset. So if a variable int k = 4; were declared at the start of a function, k would have the value 4 at the start of each execution of the function.

A data member (variable) declared in a class definition is visible and in scope for all the class's methods (functions) and persistent (holds its value) for each instance of the class.

The reason that local variables are used is so that there is no clash of variable names within software. Variable names must be unique within their scope but cannot be seen outside that scope. This means that several functions can declare a variable with the same name but there is no clash, as the variables cannot be seen outside each function. This means that when a large software project is being designed although it is important for the interfaces (number of arguments and data types) for software functions to be known the local variable names do not need to be agreed.

Static

Static variables have local scope but hold their value throughout the program. The computer remembers the values. They are created and memory allocated when the program is compiled.

```
#include <iostream.h>
void teststatic(void);
void main(void)
{
int loopcount;
      for (loopcount=1; loopcount<=4;loopcount++)
      {
            cout << "This is loop " << loopcount << endl;
            teststatic();
      }
}
void teststatic(void)
{
int localscopeauto = 1;
```

```
static int localscopestatic = 1;
      cout << "Local scope auto variable = " << localscopeauto++ << endl;
      cout << "Local scope static variable = " << localscopestatic++;
      cout << endl;
}
```

The output from this program is:

This is loop 1
Local scope auto variable = 1
Local scope static variable = 1
This is loop 2
Local scope auto variable = 1
Local scope static variable = 2
This is loop 3
Local scope auto variable = 1
Local scope static variable = 3
This is loop 4
Local scope auto variable = 1
Local scope static variable = 4

It can be seen that the automatic variable is reset each time the teststatic function is executed whereas the static variable holds its value between executions.

Static class variable

When static is used for a data member (variable) for a class, there is only one variable, which is shared by all instances of the class. This type of variable is used to keep totals or counts where only one is required for the class and not one for each instance of the class. It is a form of global variable but with scope limited to the class.

Global variables

Global constants and variables are constants and variables, which are declared outside of any class or function. Global variables hold their value and global constants and variables have the whole source file as their scope.

Enumeration

This is a data type that allows provision of a given set of possible values for a variable, which relates to integer values but each value can be given a name. Enumerated types make the code more readable because instead of using numbers names are used.

```
enum ans {NO,YES};
```

By default, the states defined in the braces correspond to numbers starting at 0 and incrementing by 1 each time, so that in this case NO = 0 and YES = 1. Wherever the names NO and YES are used in the program the values 0 and 1 will be used respectively.

This could also be achieved by using constants:

```
const int NO = 0;
const int YES = 1;
```

but the enumerated type **ans** allows only the two values YES and NO whereas the constants are not related.

```
enum trafficlight {RED=1, AMBER=2, GREEN=3};
```

Values can be supplied as above so that RED = 1, AMBER = 2 and GREEN = 3. This limits the value of the **trafficlight** variable to the values RED, AMBER and GREEN.

Arithmetic operators

`+` Addition
`-` Subtraction
`*` Multiplication
`/` Division
`%` Modulus (the result is the remainder after division)

++ Adds 1 to the variable to its right (prefix mode) or of the variable to its left (postfix mode)

```
++num2;  // prefix mode
```

will add 1 to the value in num2. Same as num2 = num2 + 1;

```
num4++;  // postfix mode
```

will add 1 to the value in num4. Same as num4 = num4 + 1;
When operators are combined it makes a difference when you use prefix and postfix.
Example prefix

```
q = 2 * ++b;
```

Because b has a prefix ++ this operation is applied first. This means that b is incremented by 1 before being multiplied by 2 and then the result placed in q.
Example postfix

```
q = 2 * b++;
```

Because b has a postfix ++ this operation is applied last. This means that b is multiplied by 2 and the result placed in q before b is incremented by 1.

If b contained the value 2 at the start then the result placed in q for the example for prefix would be 6 and for the example for postfix would be 4. In both cases b would contain 3 at the end of the operation.

-- Subtracts 1 from the variable to its right (prefix mode) or from the variable to its left (postfix mode)

```
--num1;  // prefix mode
```

will subtract 1 from the value in num1. Same as num1 = num1 – 1;

```
num2--;  // postfix mode
```

will subtract 1 from the value in num2. Same as num2 = num2 - 1;
The same rules apply for prefix and postfix as for ++.

Assignment operators

`=` Assigns the value at its right to the variable at its left

```
num1 = 4;
```

will assign the value 4 to num1.

```
num1 = num2;
```

will assign the value in num2 into num1.

`+=` Adds the right hand quantity to the left hand variable

```
num1 += 4;
```

will add 4 to the value in num1. This is the same as num1 = num1+ 4;

`-=` Subtracts the right hand quantity from the left hand variable

```
num2 -= 3;
```

will subtract 3 from the value in num2. This is the same as num2 = num2 - 3;

`*=` Multiplies the left hand variable by the right hand quantity

```
num3 *= 2;
```

will multiply the value in num3 by 2. This is the same as num3 = num3 * 2;

`/=` Divides the left hand variable by the right hand quantity

```
num4 /= 5;
```

will divide the value in num4 by 5. This is the same as num4 = num4 / 5;

`%=` Gives the remainder from dividing the left hand variable by the right hand quantity (Integers only)

```
num5 %= 10;
```

will divide the value in num5 by 10 and put the remainder in num5. This is the same as num5 = num5 % 10;
If num5 contained the value 96 then the result in num5 would be 6 (the remainder).

Relational operators

Relational operators are used to do comparison operations

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`!=`	not equal to
`==`	equal to (N.B. the double == this is not the same as the assignment =)

```
#include <iostream.h>
void main(void)
{
int num1 = 3, num2 = 5;
      if (num1 < num2)
            cout << "num1 is less than num2" << endl;
      else
            cout << "num2 is less than num1" << endl;
}
```

The result from this program is:
num1 is less than num2

Logical operators

Logical operators can be used to join relational expressions.

`&&`	and All expressions linked with an *and* must be true for the result to be true
`||`	or Either expression linked with an *or* can be true or all expressions for the result to be true
`!`	not *not* reverses the expression so true becomes false and false becomes true

Logical expressions are evaluated from left to right; evaluation stops as soon as the result is false

```
#include <iostream.h>
void main(void)
{
int num1 = 3, num2 = 5;
int num3 = 6;
// num1 < num2 and num3 == 5 must both be true for the result to be true
      if ((num1 < num2) && (num3 == 5))
            cout << "num1 is less than num2 and num3 is 5" << endl;
      else
            cout << "num3 is not equal to 5" << endl;
}
```

The result from this program is:
num3 is not equal to 5

Validation

It is important that software does not allow invalid data to be entered. If invalid data is accepted by software then it may crash with a run-time error. If numeric data was required by software because it was to be used in a calculation and non-numeric data was entered and accepted, when the software tried to do the calculation it would crash because the computer cannot do calculations on non-numeric data.

Software developers should write routines in the software that check the entered data to make sure it is valid. A tester must use test data that is invalid to prove that the software rejects invalid data.

Types of validation

Date checks

Whenever a date is used as input, the software should check that it is valid. The type of check made depends on the format of the date entered. If a date is entered in the format dd/mm/yyyy then full checks are required.

The days (dd) must be inside the boundaries for days in a month. For most months this is between 1 and 31 but for the months April, June, September and November the days are between 1 and 30. For February the days are between 1 and 28 except for a leap year when the days are between 1 and 29.

The month (mm) is checked to make sure it is between 1 and 12.

The year (yyyy) cannot be checked properly but could be checked for the first two digits being 19 or 20. It depends on the application what the range of years can be. If the software was for a museum which was holding details about historical artefacts then the range of years could not be checked because there would be a wide range of years allowable.

The format in which the date is entered should be specified in the design specification and it is up to a tester to decide what test data and tests should be done to make sure that the software will not accept invalid dates.

Range checks

Some data input may be specified as restricted to a range. For instance a value entered may be restricted to values between 100-500. A range check must be written in the software to prevent any other values being accepted. The range for data input should be specified in the design specification. A tester must decide what test data to use to prove that invalid values would not be accepted by the software.

For data input restricted to the values 100-500 the test data would be 99, 100, 101, 499, 500 and 501. This data tests the boundaries of the values. The values 99 and 501 should be rejected and the values 100, 101, 499 and 500 should be accepted.

The following example ensures that the program will keep looping until the correct data (100-500) is input.

```
#include <iostream.h>
void main(void)
{
int num;
	do
	{
		cout << "Enter a number between 100 and 500 " << endl;
		cin >> num;
		if (num < 100 || num > 500)
			cout <<  "Invalid number entered" << endl;
	}
	while (num < 100 || num > 500);
}
```

This code does not prevent characters being entered instead of numeric data. The code would not execute correctly if characters were entered.

Presence check

Input data may be specified as required which means it cannot contain spaces. Fields such as customer account number for a sales order or student enrolment number for a student record must be present and cannot be omitted. The software must check that these fields do not contain spaces. A tester must create test data that includes spaces for this data to ensure that the software will reject the spaces.

Type checks

Numeric

Input data that has numeric calculations done on it must be checked to ensure that it is numeric. If non-numeric data is allowed as input the software will crash when the computer attempts to do the calculations. The software must reject any data input that is non-numeric. A tester must create test data that includes alphanumeric data to ensure that the software rejects the non-numeric data.

Alphabetic

Input data that must be alphabetic can be checked to ensure that it is alphabetic. Alphabetic data only includes the characters A-Z and a-z therefore data that includes spaces and special characters e.g. a hyphen (-) cannot be alphabetic.

Predefined functions

Predefined functions are functions that are already written and stored in libraries as part of the programming language and can be used by a software developer. This saves a software developer from having to write basic functions that may be required in software. The problem with C++ is that it is not a standardised language so care must be taken when using different compilers, as the predefined functions do not always execute in the same way even if they have the same name. A program that has been written, compiled and tested using one compiler may compile with no errors using another compiler and execute without crashing but the output or processing may be incorrect. This means that software moved from one type of computer to another should be fully tested again.

The correct library file must be included in the program using a #include when predefined functions are used, otherwise an error will occur when the software is compiled because the function cannot be found.

String functions

The following functions need the **string.h** library included.

strlen() function

This function is used when you need to find out the length of a string. It takes a string argument and returns the length of the string. It does not include the '\0' (null) character in its count.

```
#include <iostream.h>
#include <string.h> //This is library for the strlen function
void main(void)
{
int len;
char str2[25] = {"This is the content"};
	len = strlen(str2);
	cout << "This is the length of str2 - " << len << endl;
}
```

The output from this program is:
This is the length of str2 - 19

strcat() function

This function is a concatenation function. It takes two strings for arguments and a copy of the second string is concatenated (added) to the end of the first string. The second string is not altered. The function does not check that the second string will fit in the first array. It is the developer's responsibility to make sure that the first array size is large enough.

```
#include <iostream.h>
#include <string.h> //This is library for the strcat function
void main(void)
{
char str1[25] = {"James "};
char str2[20] = {"Nightingale"};
	strcat(str1,str2);
	cout << "This is str1 after str2 is added - " << str1 << endl;
```

```
}
```

The output from this program is:
This is str1 after str2 is added – James Nightingale

strcpy() function

The strcpy() function is used to copy one string to another string. It takes two strings as arguments. The second string is copied to replace the first string. The first string array size must be large enough to hold the second string.

```
#include <iostream.h>
#include <string.h> //This is library for the strcpy function
void main(void)
{
char str2[35] = {"This is the string to be copied."};
char str1[40] = {'\0'}; //This initialises the string to null (empty)
	cout << str2<< endl;
	cout << str1 <<endl;
	strcpy(str1,str2);
	cout << str1 << endl;
	cout << str2 << endl;
}
```

The output from this program is:
This is the string to be copied.

This is the string to be copied.
This is the string to be copied.

The string pointed to by the second argument (str2) is copied into the array str1 (the first argument).
The blank line is the result of the first printing of str1 because the string array was initialised to null.

gets() function

The **gets**() function is in the **stdio.h** library. This function gets a string from the standard input device. It reads characters until a '\n' character is entered. It takes all the characters before but not including the newline ('\n') character and tacks on a null character ('\0') and gives the string to the calling program. It is the programmer's responsibility to make sure that the string declared for the input is long enough to take the string input.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h> /* needed for the strlen function */
void main(void)
{
int i;
// declare a string to take 29 characters plus null
char strinput[30];
	cout << "Enter a string of characters" << endl;
	gets(strinput);
	// output each character in the string with a space
	for(i=0;i<strlen(strinput);i++)
		cout << strinput[i] << " ";
	cout << endl << strinput << endl;
}
```

The gets() function works differently in different compilers. Some compilers store the carriage return character '\n' at the end of the string before the null character '\0' and some do not store the carriage return character.
This can lead to problems if a program is compiled on a different computer with a different compiler, as the results from the program may be different.
C++ is not a standardised language, which is why compilers can do different things for the same pre-defined function.

puts() function

The **puts()** function is in the **stdio.h** library. The puts() function outputs a string. Each string outputted by puts() goes on a new line.

```
#include <iostream.h>
#include <stdio.h>
#define DEFSTR "I am a defined string."
void main(void)
{
char str1[] = "A character array was initialised.";
      puts("I am a string argument.");
      puts(DEFSTR);
      puts(str1);
}
```

The output from this program is:
I am a string argument.
I am a defined string.
A character array was initialised.

Conversion functions

The **atoi()** and **atof()** functions are in the **stdlib.h** library. These functions can be used to validate data input. If the program inputs the data as a string and then uses atoi() or atof() to convert the data to numeric a check can be made to ensure that the data is numeric.

atoi()

This function is passed a string, which it converts, to an integer. It returns the converted integer value of the string, or 0 if the string cannot be converted.

atof()

This function is passed a string, which it converts, to a floating point (double). It returns the converted value of the string, or 0 if the string cannot be converted.

```
#include <iostream.h>
// Must include this library if using atoi or atof
#include <stdlib.h>
void main(void)
{
int num1;
double num2;
char str1[10];
      cout << "\nEnter an integer number " << endl;
      cin >> str1; // try entering letters instead of numbers
      num1=atoi(str1); // Convert string data to integer
      // test to ensure that could convert string to integer
```

```
        if (num1==0)
              cout << "Could not convert string data" << endl;
        else  cout << num1;
        cout << "\nEnter a float number ";
        cin >> str1;
        num2=atof(str1); // Convert string data to float
        if (num2==0)
              cout << "Could not convert string data"  << endl;
        else  cout << num2;
}
```

When running this program try entering letters instead of numbers.

toupper() function

The **toupper**() function is in the **ctype.h** or the **stdlib.h** library depending on which compiler you use. The toupper() function takes a character and returns it as uppercase. If the character is already uppercase no change is made.

```
#include <iostream.h>
#include <stdio.h>
#include <ctype.h> // or stdlib.h
#include <string.h>
void main()
{
        int i;
        char upperch[30];
        /* declare a string to take 29 characters plus null */
// The static keyword means the string will be initialised as empty
        static char str[30];
// Copy the string 'Test string' into str
        strcpy(str,"Test string");
        for(i=0;i<strlen(str);i++)
        {
              //change each character in the string to uppercase
              // and put in a new string upperch
              upperch[i] = toupper(str[i]);
              // output each character and a space
              cout << upperch[i] << " ";
        }
        cout << endl << str << endl;
}
```

tolower() function

The **tolower()** function is in the **ctype.h** library or the **stdlib.h** library depending on which compiler you use. The tolower() function takes a character and returns it as lowercase. If the character is already lowercase no change is made.

C++ streams libraries

While all the stdio library I/O functions (such as printf and scanf) are still available, C++ provides a group of classes and functions for I/O defined in the iostream library. To access these, the program must have the directive #include <iostream.h>.

The C++ stream mechanism is more efficient and flexible. Formatting output is simplified by extensive use of overloading. The same operator can be used to output both predefined and user-defined data types.

Standard I/O

C++ provides predefined stream objects as follows:

cin	Standard input, usually the keyboard, corresponding to stdin in C
cout	Standard output, usually the screen, corresponding to stdout in C
cerr	Standard error output, usually the screen, corresponding to stderr in C

You can redirect these standard streams from and to other devices and files. (In C, you can redirect only stdin and stdout.)

The standard error output, cerr, can be used as a separate stream object to output error messages. When used as a separate stream object for all error messages, the error messages can be redirected without affecting the output to other stream objects e.g. the output using cout.

A simplified view of the iostream hierarchy, from primitive to specialized, is as follows:

streambuf	Provides methods for memory buffers
ios	Handles stream state variables and errors
istream	Handles formatted and unformatted character conversions from a streambuf
ostream	Handles formatted and unformatted character conversions to a streambuf
iostream	Combines istream and ostream to handle bi-directional operations on a single stream

Questions 1

1. Write a program that inputs a string from the keyboard and outputs the number of characters in the string. The program should keep looping until the character '0' is entered.

2. Write a program that inputs a string from the keyboard and outputs the number of words in the string. The program should loop to accept input 5 times.

3. Write a program that inputs a character that can be A, B, C or D, followed by a number that can be from 1000 to 1500 (inclusive). The program should loop to accept input 10 times. Use cerr to output any error messages.

4. Write a program that inputs a string, copies the string to another string and changes the characters in the second string to uppercase. Output the contents of the second string. The program should loop to accept input 5 times.

5. Write a program that sets up the strings "Station ", "Watford", "Coventry" and "Birmingham". Use string concatenation to add the town names to the "Station " string and output the three new strings created.

6. Write a program to control two ventilation units VentA and VentB. The program should loop until the temperature –1 is input. The temperature should be input via the keyboard and the ventilation units should be controlled as show below.

Temperature	**VentA**	**VentB**
<=30	Closed	Closed
>30 and < 35	Open	Closed
>=35	Open	Open

   The state of the ventilation units should be output after each change of temperature.

7. Write a program that accepts a string as input and then displays the following menu:

   MENU

   1. Convert to lowercase
   2. Convert to uppercase
   3. Output the number of characters
   4. Output the number of words
   5. Exit

   Enter the option required:

   The appropriate action should be taken when a menu option is input e.g. if option 1 is selected the input string should be converted to lowercase and then output.

[This page is intentionally blank]