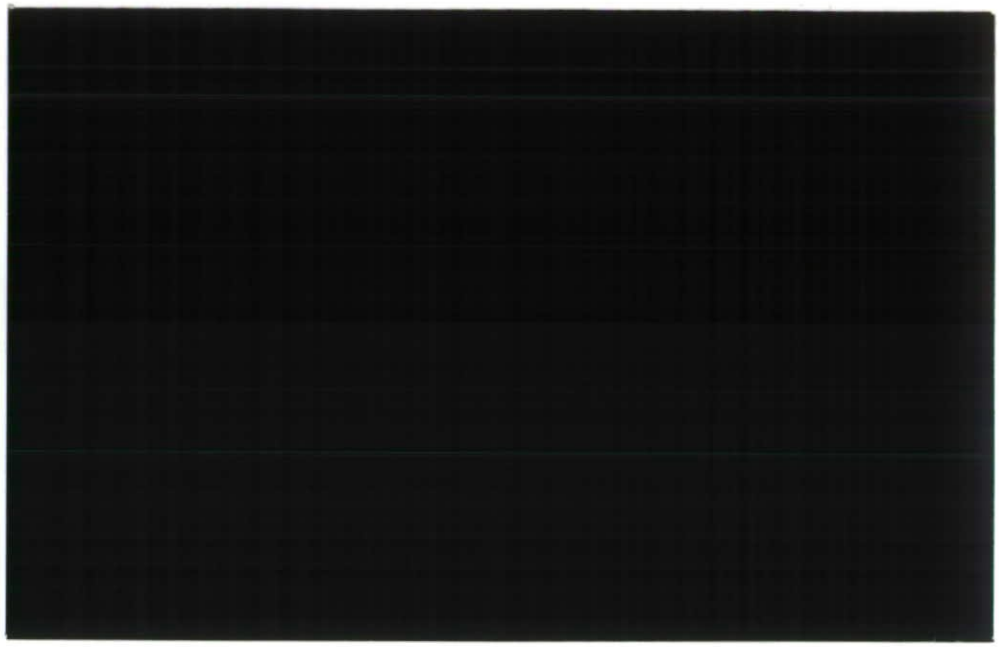
ARCHIVE:

PLEASE DO NOT DESTROY

Institute of Hydrology

1995/096



Institute of Hydrology

C Programming Language Training Course

by
Richard Alexander
(Hydrology Software Section)

Institute of Hydrology
Maclean Building
Wallingford OX10 8BB
United Kingdom

Telephone +44 (0)1491 838800
Facsimile +44 (0)1491 692424

MAY 1995

COPYRIGHT

DISCLAIMER

TRADEMARKS

Course Content

Forward

These notes are aimed at scientists, with some experience of procedural programming, wishing to move to the C language.

Although many existing texts describe the syntax and semantics of the language, few deal with the application of the language to well written software. These note are intended to demonstrate the use of recognised structured techniques applied to the C language.

The notes cover all aspects of the language except for those which are considered to lead to poor structure. They began with a overview of the language and return to cover aspects in more detail. They also cover aspects such as documentation of programs, control flow and functional decomposition.

The language described is that defined in the ANSI standard. For more details on specific parts of the standard the reader is recommend to read "The C Programming Language" by Kernighan and Ritchie.

1 Introduction

1.1 What is C

C is a procedural computer programming language. The ANSI standard provides a *machine independent and unambiguous definition of the language*.

C was originally designed for the development of the UNIX operating system.

1.2 Why learn C?

C is a very popular language. It is well supported with ANSI standard compilers available for most machines (including free ones) and a wide variety of supporting tools. Because of its wide spread use there is also large amount of source code available and documentation on the language.

The language supports many of the features associated with modern languages. These include pointers, dynamic memory allocation, recursion and structures and it is also free format.

C is a fairly low level language, the code can be easily converted to fast efficient code. This however has the side effect of little run-time error checking.

The syntax of the language features *economy* of expression. Careful attention must be paid to ensuring that code remains structured and readable.

C forms the basis of C++ which is a powerful object-oriented language.

1.3 Summary

The main reason for learning C is because of it wide use and support.

2 Fundamentals of C Programs

2.1 Tutorial introduction

2.1.1 Learning the form of a C program

The most basic program in any programming language is usually referred to as the 'Hello World' program and consists of outputting text to the screen. In C this takes the form of:

```
#include <stdio.h>

void main()
{
   printf("Hello World\n");
```

```
);
```

The first line of code includes a header file. The header file describes functions which the programmer wishes to use. Standard libraries are indicated by including the name in angle brackets <>. This tells the compiler to look for the library in a specific location. User defined libraries are enclosed in quotes e.g. `"test.h"`.

The library `stdio.h` contains input and output functions, types and macros.

All C programs have a `main()` function. This is called when the program is first run. By default this returns an integer value, the `void` indicates that we do not wish to return a value.

The curly brackets show the start and end of the function. The code is placed in between these brackets.

`printf` is a library function which is used for output to the screen. The `\n` represents a new line character. The semicolon at the end of the line indicates the end of the statement.

C is a free format language, statements may be split over several lines if desirable.

Source files in C have the extension `.c`. The above program would therefore be saved in a file called `hello.c` for example.

2.1.2 Declaring variables

Variables are used as a place to store data. C contains built in data types for storing numbers and strings. Complicated data types can be built up using arrays and structures.

Variables must be declared before use. Local variables are declared at the top of a block of code. Global variables are declared at the top of a file. A declaration specifies a type and contains one or more variables of that type.

Primitive types are: int, float, double, char.

```
int i;              /* integer number */
float f1,f2;        /* floating point number */
double d1;          /* floating point number, double precision */
char cA, cB;        /* single character */
```

Arrays are indexed lists of values of the same type.

A string is an array of characters:

```
char ac[20];        /* An array of twenty characters */
int an[100];        /* An array of one hundred integers */
```

2.1.3 Arithmetic expressions

C contains built in operators for manipulating numeric variables. These include operators for addition, subtraction, multiplication and division. The assignment operator is a single - character.

```
int nCelcuis;
int nFahr;

nFahr = 55;
nCelcius = (5 * (nFahr-32)) / 9;
```

N.B. The order of the operators. Since integer arithmetic is used, it is important that the multiplication is performed first otherwise values may be lost due to rounding errors.

2.1.4 Designing program flow and control

C contains a number of components for controlling the flow of a program within a function. The most fundamental are the `while` loop and the `if` condition.

The while loop causes a program to repeat a block of code whilst a condition is true.

```
while (nCelcius > 30)
{
   printf("Its too hot!");
   nCelcius = GetTemperature();
}
```

C also contains a `for` loop, this is however just a variation of a `while` loop.

An `if` condition causes a program to execute a piece of code once if a condition is true and may contain an `else` clause which is executed if the condition was not true.

```
if (nCelcius > 30)
{
   printf("ts too hot";
} else if (nCelcius < 10)
{
  printf("its too cold");
} else
{
   printf("Its just right!");
}
```

2.1.5 Defining and using functions

A function provides a way of breaking a program down into smaller processing units.

The function prototype defines the external interface of the function.

```
int Power(int nNumber, int nPower);
```

It consists of a name, a return value of a specified type and arguments of specified types.

The function implementation contains the actual body of the function.

```
/*************************************************************************
 *
 * int Power(int nNumber, nPower)
 *
 * Simple function to calculate nNumber to the power nPower
 * Nb. This can only handle whole, positive powers
 *
 */

int Power(int nNumber, int nPower)
{
   int i;
   int nResult = 1;

   for (i = 0; i < nPower; i++)
   {
      nResult *= nNumber;
   };

   return nResult;
};
```

The function call consists of a number of parameters passed to the arguments of the function. The value of a function is its return value.

```
int nPower = Power(3,3);
```

2.1.6 Using standard terminal I/O functions

C has standard functions for outputting data to files and the screen and retrieving data from files and the keyboard.

The most common of these are the `printf` and `scanf` family of functions.

`printf` outputs formatted text to the screen.

```
int n = 5;
printf("The value is %i\n",n); /* Outputs "The value is 5" */
```

The first argument to printf specifies the format. `%i` indicates where the next argument's value should be inserted and that it is an integer.

`scanf` retrieves text from the keyboard.

```
float fValue;
printf("Input value: ");
scanf("%f",&fValue);
printf("Square is: %f\n",fValue*fValue);
```

In this case the format specifier indicates that the first value input should be converted to a floating point value. The ampersand in front of the fValue indicates that the value is to be returned. [Pass by address, see 4.2.5]

Similar functions exist from reading and writing to files and strings.

```
FILE *pFile = fopen("test","w");
fprintf(pFile,"Hello\n");
fclose(pFile);
```

2.2 Fundamental types

2.2.1 Integer types, short long, unsigned

The accuracy of these types is implementation dependant. For integers the accuracy determines the range of values they can hold.

Types may be qualified to indicate if they are unsigned or double precision. The default is signed.

```
unsigned int u;      /* 0 to .... */
unsigned char c;

short int n1;    /* at least 16 bits */
short n2;        /* same as short int */
int n3;          /* short int <= int <= long int */
long int l1;     /* at least 32 bits */
long l2;         /* same as long int */
```

2.2.2 Character types

A character is a single byte, capable of holding one character in the local character set.

```
char cTest = 'z';
```

In C, character constants are written in single quotes.

2.2.3 Single and double precision floating point numbers

A floating point variable stores a real number.

`float` is a single precision floating point number
`double` is a double precision floating point number
`long` double is an extra precision floating point number

```
float fValue;
double dValue;
long double ldValue;
```

N.B. A floating point number on a workstation may be held to a greater precision than on a PC. The accuracy for the compiler is defined in `limits.h` and `float.h`.

```
long double ld;
```

2.2.4 Storage classes - auto (default), static, const

Outside functions declarations are static i.e. they are created when the program is first run and exist until it ends. Placing `static` in front of such a variable or function prevents access to it from outside the file it is declared in.

```
/*  Global declaration, cannot be accessed outside file */

static int _nValue;
static int LocalFunction();
```

Placing extern in front of a global variable name means that the declaration is defined elsewhere. Variables are often declared as extern in header files and declared in source files.

```
extern int _nValue;
```

Inside a function or block, variables are automatic by default. They are created when the function or block is entered and are discarded upon exit. Declaring a variable as `static` inside a block means that it retains its value between calls to the function. I.E. only one copy exists.

```
/*****************************************************************************
 *
 * int Random()
 *
 * Returns a pseudo random number
 *
 */

int Random()
{
   static int nRandom;

   nRandom *= 13977;
   nRandom += 8294;

   return nRandom;
};
```

Nb. The meaning of static varies depending on where it is applied. Explicitly declaring an automatic variable (i.e. one inside a function) as `static` causes it to retain its value. Explicitly declaring a static variable (i.e. one outside a function) as `static` prevents access to it from other files.

The qualifier const can be applied to the declaration of any variable to indicate that its value will not be changed. For an array, the const qualifier specifies that the elements will not be altered.

```
const double dE = 2.71828182845905;
const char sMsg[] = "warning: ";
```

2.2.5 Variable naming conventions

Derivative of the Hungarian naming convention

Using prefixes on variable names to indicate their type makes code considerably easier to understand. It also makes thinking up variable names much easier!

TYPE	PREFIX	DECLARATION
integer (16 bit)	n	int
integer (32 bit)	l	long
unsigned integer	u	unsigned int
floating point number	f	float
double precision floating point number	d	double
character	c	char
string	s	char []
boolean	b	int
pointer	p	*
index	i	int
array	a	[]

```
int* pnTree;            /* Pointer to an integer */
char sRiver[MAXLEN];    /* Array of characters */
```

When naming variables try and make them as meaningful as possible without making them excessively long so they become cumbersome or so abbreviated that they becoming meaningless acronyms.

It is convention in C to have variable names in small letters with a capital letter starting each word.

```
int nRiverLocationID;
char sOperatorName[20];
```

2.2.6 Constants

The type of a constant is determined as follows:

Numbers without a decimal place are assumed to be of type integer:
1234 int

Numbers with decimal places are of type double:
123.4 double

A `u` or an `l` appended to the number indicates unsigned and/or long
An `f` indicates the number is single precision floating point

```
125.6l is a long double
```

Hexadecimal numbers are preceded by 0x

```
0xFFFF
```

Character constant are in single quotes

'z'

String constants are in double quotes

```
"Hello\n";
```

These have a null character appended to the end, so the above string would require seven bytes to store - H E L L O being five, one for the new line character and one for the null character. A function to retrieve the length of the string would however return six.

Therefore "z" and 'z' are two different things.

The escape characters in C are:

\a (bell)	\\ backslash
\b backspace	\? question mark
\f formfeed	\' single quote
\n newline	\" double quote
\r carriage return	\0 null character
\t horizontal tab	\000 octal number
\v vertical tab	\xhh hex number

enumerations

These are a list of constant integer values which are assigned names:

```
enum boolean {FALSE,TRUE};
```

By default enumerations start at zero so FALSE would be 0 and TRUE 1.

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT,
NOV, DEC};
```

This would allow JAN to be used in place of the number 1 making code much more meaningful.

```
if (nMonth == JAN)
{
   nDays = 31;
};
```

These are usually placed at the top of a source file.

2.2.7 Initialisation

A variable may be initialised in its declaration.

```
static double fPI = 3.1415927;
int nValue = 5;
```

Variables are initialised when they are created. Therefore static variables inside a function are initialised once only when the program is first run. Automatic variables are initialised each time the function is called.

A variable does not have to be initialised. Static variables are initialised to zero. The value of an uninitialised automatic variable is undefined. It is therefore good practice to initialise variables.

2.3 Operators

2.3.1 Numeric Operators

C supports the following numerical operators:

Binary operators:

+ addition
- subtraction
* multiplication
/ division
% modulus (remainder!)

Precedence rules are as for mathematics:

```
int nA = 7;
int nB = 12;
int nC;

5+2*7 == 19;  /* TRUE */
nC = nA * nB;
```

When using constants in mathematical equations it is often preferable to define them at the top of a file.

```
#define PI 3.1415927

int GetArea(float fRadius)
{
   return PI * fRadius * fRadius;
};
```

2.3.2 Relational Operators

Relational operators are used to determine if conditions are true or false. Nb. In C the equality operator is ==.

It is a common mistake to use the assignment operator in the place of the equality operator.

Relational operators return 1 if successful and 0 if unsuccessful.

Other operators include the:

inequality operator !=
greater than >
less than <
less than or equal to <=
greater than or equal to >=

Several conditions may be combined together using logical operators:

&&	logical AND
||	logical OR
!	logical NOT

```
if (a <= b && b != 0)
{
   /* ... */
}
```

Care should be made not to confuse these with the bitwise operators.

Nb. That floating point numbers should never be compared directly for equality. This is because rounding errors may have occurred in calculations. It is standard to compare their difference to a fixed value.

```
#define PRECISION 0.00001

if (fabs(f1-f2) < PRECISION)
{
   /* ....*/
}
```

2.3.3 Assignment operators

An assignment causes the variable on the left hand side of the argument to take the value on the right hand side.

```
a = b * 2;
```

Assignment can be made between variables which are both numeric, both pointers or both structures. [See 2.3.6 for type conversion rules]

The assignment operator may be combined with binary operators.

```
n = n * 5; /* Equivalent functionality */
n *= 5;
```

C contains many such abbreviations of syntax, it is important however to make sure that the code still remains readable.

2.3.4 Increment and Decrement Operators

The increment operator increases the value of the operand by one.

```
int n = 5;
n++;
/* n now has the value 6 */
```

The operator may be prefix or postfix. If it is prefix then the value is increment first, then used. Otherwise it is used and then incremented.

```
int n = 5;
a = n++; /* a is set to 5, n becomes 6 */

int n = 5;
a = ++n; /* n become 6, n is set to 6 */
```

As this can lead to confusion it is often best to avoid this feature.

The decrement operator works likewise.

```
n--;
```

2.3.5 Bitwise operations

Bitwise operators work on char, int and long values, signed or unsigned.

The operators are:

&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

For example the binary value 01101101 ANDed with
00001111 would return

00001101

The main use of this is for setting and identifying the state of individual bytes.

```
#define SHOW_BORDER          0x0001
#define SHOW_TITLE           0x0002
#define SHOW_LEGEND          0x0004
#define SHOW_THICKLINES      0x0008
#define SHOW_PATTERNEDLINES  0x0010

/* Create flag */

wFlag = SHOW_BORDER | SHOW_TITLE | SHOW_LEGEND;

/* Check for flag */

if (wFlag & SHOW_TITLE)
{
   /*....*/
}
```

2.3.6 Type Conversions

When an operator has operands of different types, they are converted to a common type. Automatic conversions are those that convert data without loss of data.

```
int n = 5;
double d = n; /* Conversion int to double */
```

A cast can be used to explicitly convert a value from one type to another. This feature can be abused and should therefore be used with caution.

```
double d = 5.6;
int n = (int)d;  /* n = 5 */
```

Nb. The cast doesn't change the value of the argument it is applied to.

2.4 Conditional program execution

2.4.1 if, else statement

An `if` statement tests the parenthesised condition, and if the condition is true (non-zero), executes the following statement. An `if` statement may have an optional `else` statement which is executed if the condition is false (zero).

```
/* Allocate memory */

np = (int *)malloc(size);

/* If memory allocated successfully then set first value */

if (np != NULL)
{
   *np = 5;
} else
{
   printf("Unable to allocate memory\n");
}
```

`If` statements may be nested:

```
if (condition)
{
} else if (condition)
{
} else
{
}

if (condition)
{
   if (condition)
   {
   } else
   {
   }
}
```

The parenthesis after `if` statements are not compulsory but without them nested statements may be ambiguous.

2.4.2 switch, case, default

The `switch` statement tests whether an expression matches one of a number of constant integer values.

```
enum {TYPE_TSGRAPH, TYPE_TSTABLE, TYPE_DPGRAPH};

/* ... */

switch (iType)
{
   case TYPE_TSGRAPH :
   {
      DrawTSGraph();
      break;
   }

   case TYPE_TSTABLE :
   {
      DrawTSTable();
      break;
   }

   case TYPE_DPGRAPH :
   {
      DrawDPGraph();
      break;
   }

   default :
   {
      printf("Type not supported\n");
   };
}
```

The default operation is executed if no match is found.

The break statement causes the program to leave the switch loop after the match is found otherwise the default statement would always be executed as well as any match.

More than one case may be matched to a statement:

```
switch (cInput)
{
   case 'Q' : case 'X' :
   {
      bExitProgram = TRUE;
      break;
   }
}
```

2.5 Loops and iteration

2.5.1 while loop

A `while` statement implements a conditional loop. There are two forms of the `while` loop.

The first tests the condition before the loop is first entered, if the condition is true then the loop is entered. It is re-tested when the body of the loop has executed. If true the body is re-executed. This continues until the test becomes false.

```
POSITION pos = GetHeadPosition(list);
while (pos != NULL)
{
   value = GetNext(list, pos);
};
```

The alternative is the do-while loop. In this case the condition is first tested after the loop has been executed once. [This is equivalent to *repeat until* in BASIC].

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

2.5.2 for loop

Conventionally the `for` loop is used to repeat a loop a predetermined number of times with an index.

```
int i;
for (i = 0; i < 100; i++)
{
   statements;
};
```

This cause the loop to be repeated 100 times. The value of 'i' will be start at 0 and be incremented each time the loop is executed. On the last iteration of the loop it will have the value 99. The value of i will be 100 upon exiting the loop.

The for loop continues whilst the condition is true. A common mistake is to test for the end condition.

To increment in stages of 2, `i++` would be replaced with `i += 2`.

The following `for` and `while` statements are equivalent.

```
for (initialise;  condition; statement)
{
  /* .... */
}

initialise;
while(condition)
```

```
{
   /*...*/
   statement;
}
```

Although `for` loops may be used for purposes other than predetermined increments, it is usually more readable to use a `while` loop.

3 Functions and Program Structure

3.1 Functions

3.1.1 Declaring functions

A function provides a convenient way to encapsulate some computation. In other words they hide the implementation of a process replacing it with a description of what is to be done.

A C function is based on a mathematical function. It has parameters and a return value.

$f(x) = x^2$

$y = f(5)$

[Nb. However that C is an imperative language. Functions in C are not pure functions, they can have side effects they can change values outside the scope of the function. Also the values of variables inside the function can change]

The standard function `pow(double x, double y)` returns the value of x to the power y. We are not interested in how it is implemented, just how to use it. This is defined in the standard library `math.h`

```
dValue = pow(5,3);
```

The following example shows a typical function implementation:

```
typedef int BOOL;
enum {FALSE, TRUE};
enum {JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};
int _anMonthLen[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};

/*****************************************************************
 *
 * int IsDateValid(int nYear, int nMonth, int nDay)
 *
 * Function for testing the validity of date arguments.
 *
 * Returns 1 if the date is valid e.g. 22/1/1995 and 0 if it is
 * invalid e.g. 29/2/1981
 *
 */

int IsDateValid(int nYear, int nMonth, int nDay)
```

```
{
   BOOL bValid = TRUE;

  /* Ensure the month number is between 1 and 12 */

   if( nMonth < JAN || nMonth > DEC )
   {
      bValid = FALSE;
   }

  /* If the month is February calculate the number of days in the
   * month for this year */

   if(bValid && nMonth == FEB)
   {
      if (IsLeapYear(nYear))
      {
        _anMonthLen[2] = 29;
      } else
      {
        _anMonthLen[2] = 28;
      }
   }

   /* Ensure that the day number does not exceed the maximum number
    * of days in the month */

   if(bValid && nDay < 1 || nDay > _anMonthLen[nMonth] )
   {
      bValid = FALSE;
   } else
   {
      bValid = TRUE;
   }

   return bValid;
}

/*****************************************************************************
 *
 * int IsLeapYear(int nYear)
 *
 * Returns non-zero if the given year is a leap year, zero otherwise
 *
 */

BOOL IsLeapYear(int nYear)
{
   return !(nYear % 4) && ((nYear % 100) || (!(nYear % 400)));
};
```

Functions are also used to provide a way of reducing repetitions of similar pieces of code. If a program performs the same or similar function in more than one place in a program then using a function can reduce the repetition.

3.1.2 Function Return Values

The value of a function is its return value. It is often a good idea to return `1` if successful and `0` if unsuccessful. These can be defined to `TRUE` and `FALSE`.

```
enum {FALSE, TRUE};
```

This will allow the return value to be checked using boolean operations:

```
if (bPrinter && IsPrint())
{
    printf("printed successfully");
}
```

A program should always have a single return statement at the bottom of the function. This will make the code easier to follow and ensure that any files that are opened or memory allocated are released at the end.

This can be achieved by setting a boolean flag at the start of the function to TRUE. If an error occurs in the function the flag is set to FALSE and no more processing occurs. The de-initialisation functions are still called at the end. The boolean value is then returned.

```
int IsReadFile()
{
   int bOK = TRUE;

  /* Open file */

   FILE* pFile = fopen("filename","r");

   if (pFile == NULL)
   {
      bOK = FALSE;
   }

 /* Process data */

   if (bOK)
   {

   }

 /* Tidy up */

   if (pFile != NULL)
   {
      fclose(pFile);
   }

return bOK;
};
```

Nb. The test that the file was opened successfully before a call is made to close it.

3.1.3 Recursive Functions

Recursion is a very powerful facility which allows something to be defined in terms of itself. In C a function may call itself directly or indirectly.

```
int factorial(int n)
{
  if (n > 0)
```

```
  {
     return factorial(n-1);
  } else
  {
     return 1;
  }
}
```

This is particularly useful for certain structures which are recursive in their definition such as lists and trees.

3.1.4 Command line argument

When calling a C program from the command line, it is possible to append arguments after it. For example the command cd (change directory) takes the name of the directory as an argument.

In C, the functions `main` retrieves two optional parameters, the first being the number of command line arguments, the second being an array of pointers to the arguments. The first argument, number zero, is the name of the program itself, therefore there will always be at least one argument.

```
main(int nArgCount, char *apsArgs[])
{

  char sFileName[MAXFILELENGTH];
  int handle;
  char ch;

 /* If no file name is supplied is supplied then request the user for
  * one otherwise use the argument supplied
  */

  if (nArgCount == 1)
  {
   GetFileName(sFileName);

  } else
  {
    strcpy(filename,apsArgs[1]);
  };

/* ... */
};
```

3.1.5 Variable length arguments

When declaring functions it is possible to append three full stops ... as the last argument to allow any number of arguments for the function.

This is used in standard library functions such as printf and scanf

```
/* Declaration*/

int fprintf(FILE* stream, const char *format, ...);
```

```
/* Call to functions */

fprintf(pFile, "%s", sName);
```

Although apparently useful this suffers from the problem of no type checking and should be avoided if at all possible.

[This means that it is possible to inadvertently access memory outside the current scope without warning]

3.2 Program Structure

3.2.1 Scope rules

The functions and external variables that make up a C program may split over several files.

The scope of a name is the part of the program within which the name can be used. For the purposes of good program structure, the scope of names should be restricted as far as possible. This is because the greater the availability of a function or variable, the more complex the potential interactions between different parts of the program.

[Object-oriented languages such as C++, are built around the concept of encapsulation of data and functions - i.e. limiting scope]

The scope rules are as follows:

The scope of an external variable [i.e. one outside a function] lasts from the point at which it is declared to the end of the file.

If an external variable is defined in a different source file or it is to be referred to before it is defined then an extern declaration must be made. This states that a variable of this type and name is declared elsewhere.

```
extern int _nUserID;
```

Variables declared inside a block cannot be seen outside.

```
int func()
{
   int nValue;
}
```

External variables and functions declared as static may not be accessed from outside the file in which they are declared.

Functions are global unless they are declared as static.

```
static int LocalFunction();
```

[See 2.2.4]

3.2.2 Block structure

A block defines a section of code, it usually follows a control statement e.g. `for`, `if`, `while` or a function.

Using spaces instead of tabs, resolves problem of changing editors, three spaces indentation is recommended.

Align matching braces vertically and with start of control statement

```
if (/*.....*/)
{
   while (/*....*/)
   {

   }
}
```

Avoid a single line following a control statement. It is a common mistake to add another line and forget to add the braces.

```
for (int i = 0; i < 5; an[i++]=0);  /* AVOID*/

for (int i =0; i < 5; i++)      /*AVOID*/
   an[i++];

for (int i = 0; i < 5; i++)      /* OKAY*/
{
   an[i] = 0;
}
```

Variables may be declared as local to a block.

3.2.3 Header files

These files contain information which is to be shared by several source files. Their name is usually appended with an 'h' e.g. "test.h". They contain function prototypes for functions which are declared in one source file and used in an another. Data structures may be declared in a header file. Also shared constant defintions are declared in header files. Global variables are also declared.

[Variables must be declared as `extern` otherwise if the header file is included by more than source file then the same variable would be defined by more than one place].

Header files are included into a source file using a `#include` command.

```
#include "test.h"
```

This should be thought of as the statement `#include "test.h"` being replaced by the contents of the file `test.h` at compile time.

```
/*************************************************************************
```

```
 *
 * Copyright (c) 1992 Institute of Hydrology
 *
 * Project      : SWIPS
 * File         : SWIPS.H
 * Author       : Richard Alexander
 * Date         : 26th December 1992
 *
 * Abstract     :
 *
 * Main Include file for SWIPS.
 *
 */

#ifndef _SWIPS_H_
#define _SWIPS_H_

#include <windows.h>


/* Declared in SWP_INIT.C
 */

extern char _far szSysPath[];
extern char _far szSysIni[];       /* Initialisation file */

/* Declared in SWIPS.C
 */

extern HANDLE hAccel;              /* resource handle of accelerators */
extern HWND hWndMDIClient;         /* handle of MDI Client window */
extern HWND hInst;                 /* Program instance */
extern HWND hWndMain;              /* Handle of main window */

/* Help types
 */

#define HLP_NEWPROJECT  0
#define HLP_OPENPROJECT 1
#define HLP_IMPORT      2
#define HLP_EXPORT      3

/* Global initialisation functions
 */

int InitialiseMenu(void);
int InitialiseVariables(BOOL bStartup);

/* END OF SWIPS.H
 */

#endif
```

The statement `#define identifier substitution-text` declares a macro. Whenever identifier is found it is replaced by the substitution text. This is a poweful facility which should be used carefully as there is no syntax checking. Enumerations should be used in preference where possible.

Include files may themselves include further files. To prevent an infinite recursion of files including each other, at the top of file the line `#ifndef NAME` checks that a unique name has not already been defined. If it has not, it is defined, and the rest of

the file is included. The corresponding `#endif` is placed at the bottom of the include file.

3.2.4 Comments!!

Why Use Comments

Programming languages describe the process required to perform a function - HOW. The purpose of a comment is to describe in English WHAT the function does.

Guidelines

Write the comments BEFORE writing the proceeding code. This will focus the mind on what you are about to write, also the *I'll add the comments afterwards* attitude usually results in no comments.

Comments form documentation of code. Descriptions of functions should allow another user to use the function without having to examine the source code to see how it is implemented.

C Comments

The ANSI standard C comment is delimited by `/*` and `*/`

File headers

Should contain a copyright message, the date written, the author and the file name, an abstract describing the contents of the file and also an edit history of changes since the file was first written.

```
/*************************************************************************
 * creading.cpp Implementation file for class storing readings
 *
 * Copyright (c) 1994 Institute of Hydrology
 * Author:  Richard Alexander
 * Date:    2nd June 1994
 *
 * Edit History
 *
 * 3/5/95 RDA Changed so that time series data grouped into years of
 *            data rather than days
 */
```

Function headers

Should describe the use of the procedure (as opposed to its implementation) and describe the parameters and whether they are supplied, returned or both. They should also describe the return value.

```
/*************************************************************************
 * int GetSigFigs(double dValue)
 *
 * Determines the number of significant figures that a value should
 * be displayed to
```

```
 *
 * E.g. 140.5 would return the value 4
 *
 * Arguments
 *
 * double dValue
 *         o The value for which the number of characters is
 *           required.
 *
 * Returns the number of significant figures required to display the
 * value
 */
```

Code

Should be divided into blocks with a comment describing WHAT each block does and if necessary how it works.

```
/* Clear the current contents of the clipboard, and set
 * the data handle to the new string.
 */

     if (OpenClipboard())
     {
          EmptyClipboard();
          SetClipboardData(CF_TEXT, hData);
          CloseClipboard();
     }
```

3.2.5 Defensive Programming

Although it appears to be extra work, adding additional code in case things go wrong (and they will!) will save time in the long run.

An `assert` function asserts that a statement is correct. If an assertion is false then a message appears on the screen indicating where the assertion has failed, making debugging much easier.

Compilers often have the option of compiling in debug or release mode. A program compiled in debug mode will allow the programmer to step through the code at run time user a debugging tool, it will also enable assertions and allow the programmers to place code which only exists in the debug version.

```
#include <assert.h>

double CalculateMean(double dTotal, int nValues)
{
 double dMean;

 /* Calculate mean value, ensure that the number of values is not
  * zero.  Nb. This is one of the few cases when using an
  * (in)equality operator on a floating point number is acceptable
  */

    if (nValues != 0)
    {
         dMean = dTotal/nValues;
```

```
    } else
    {
        dMean = 0;
        assert(0); /* Always display assertion*/
    }
    return dMean;
}
```

Other checks include always asserting that pointers passed as parameters are not NULL.

```
int GetIndex(int* pValues, int nValue)
{
     assert(pValues != NULL);
    /* ...*/
}
```

When allocating memory or opening or writing to a file, always check that the operation was successful.

[C++ exception handling makes this much easier!]

When freeing memory or closing files check that the file was actually open.

[See return values for example 3.1.2]

Additional code may be added to check that a function has worked in the debug version. E.G. If a function sorts a list then check the list is sorted. Compilers may define a macro such as _DEBUG indicating if a program is compiled in debug mode.

```
int SortList(char* aList[], int nLength)
{
   int i;
   /* sort list ....*/

#ifdef _DEBUG
  /* check sort successfully */

   for (i = 0; i < nLength-1; i++)
   {
       assert(aList[i] < aList[i+1];
   }
#endif
};
```

Once a program has been developed, it will be compiled using release mode. This will allow compiler optimisations making the code run faster, remove debug information making it smaller, it will also remove all assertions and debug code.

4 Arrays, structures and pointers

4.1 Arrays

4.1.1 Declaring and accessing

An array is an indexed list of values of the same type. In C, arrays are of fixed size unless they are declared dynamically [see 4.2.6].

The declaration of an array consists of the type of the values, followed by the name of the array with the size of the array in square brackets.

```
int anValues[20];
```

The array size must be a constant value.

Arrays may be initialised with a list of constant values inside curly braces.

```
int anValues[20] = {5,7,9,12,15};
```

Any values which are not assigned a value will be set to zero. [The values contained in unitialised, automatically declared arrays is undefined]

Arrays are accessed using an index to the value required. Indexes are always based on zero so an array of twenty values has indices from 0 to 19.

```
for (i = 0; i < 20; i++)
{
  nTotal = nTotal + anValues[i];
};
```

Only one value within an array may be accessed at a time.

4.1.2 Arrays of characters

Strings are defined as arrays of characters.

```
char sName[32];
```

It is important to ensure that the string length is sufficient to hold the possible values. C has no run-time array bounds checking [although some compilers have the option to implement this] therefore it is even more important within the code to ensure the end of the array is not overrun.

Arrays of characters may be initialised with string constants.

```
char sInitialisationFile[] = "test.ini";
```

In this case the empty square brackets indicate that the size of the array will be calculated by the compiler.

Entire arrays cannot be assigned

```
sInitialisationFile = "test.ini"; /* WRONG !!! */
```

A standard library of functions exists <string.h> for manipulating strings. The function strcpy copies the contents of one string into another.

```
strcpy(sIntialisationFile,"test.ini");
```

Formatted input/output functions sscanf and sprintf also exist for strings. [See 5.2]

The length of strings can be determined by searching for a null terminator character. When writing into a string array, the length of the array is usually passed as an argument.

4.1.3 Multidimensional

Arrays may be declared to any dimension. These are rectangular.

```
char sDBExtensions[10][4] =
{
    "RAW","EQU","CAL","FGN","DTH","LOG","PRJ","LOC","CVL","RFL"
};
```

In this case the first value specifies the number of strings, the second the length of each string. The value of four for the string length allows for a null terminator.

The size of both dimensions of the array must be specified.

Accessing a multidimensional array is as follows:

```
sDBExtension[1][0] == 'E';  /* TRUE */
```

A multidimensional array should be thought of as an array of arrays. The following example indexes the second sub-array. It uses a library function strcmp to compare two strings. This returns zero if the two strings are the same.

```
strcmp(sDBExtension[1],"EQU") == 0; /* TRUE */
```

[Many standard library functions return zero if successful, this is counter-intuitive being the boolean value FALSE. It is a hangover from UNIX, caution is therefore necessary when using return values from library functions.]

Multidemensional arrays can be initialised using sublevels of curly braces:

```
int an[3][2] = {{2,3},{4,3},{2,1}};
```

4.1.4 Passing as arguments

Arrays are passed by address in C. This means that instead of a copy of the array being passed, like other arguments, it is the original array that can be changed directly. This is for reasons of efficiency as arrays can be very large.

```
int a[10];
func(a);
```

```
int func(int a[10])
{
   a[0] = 5;  /* Modify original array directly */
};
```

The size of the array does not have to be specified for the last dimension. Since there is no checking that the size of the array passed as an argument and that of the function parameter match, it is sensible to not to specify the length of the array and to pass it as an argument.

```
int a[5];
func(a,sizeof(a)/sizeof(int));

int func(int a1[], int n)
{
};
```

The `sizeof` function determines the size of an object in bytes. Dividing by the size of each element in the array determines the number of elements in the array. It is sensible to use the `sizeof` function rather than specify the actual size twice as this ensures consistency.

The `sizeof` function cannot be used on the parameter of the function to determine the length of the array. This is because arrays are passed by address, the size of parameter `a1` is the size of the address.

In many cases, it arrays it is more intuitive to manipulated arrays inside functions using pointers.

```
long a1[50];
func(a1);

void func(long* p1)
{
   while (*p1 != -999)
   {
      p1++;
   }
}
```

Since arrays are reference by address, the name of the array is therefore equivalent to its address.

```
&a == a; /* TRUE */
```

To pass an array by value, it must be placed inside a structure. [See 4.3]

For multi-dimensional arrays, only the last dimension may not be specified. The other dimension should match.

```
int a[5][10];
func(a);

void func(int a[5][])
{
```

```
);
```

Since an array is passed by address, an array cannot be returned directly from a function unless it is placed in a structure. Normally arrays modified in a function are declared in the calling function.

4. 2 Pointers

4.2.1 Purpose

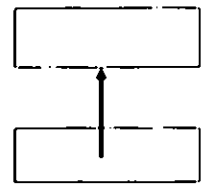
A pointer is a variable that contains the address of a variable. They allow access to dynamically allocated memory and are useful for manipulating arrays.

4.2.2 Declaration

The value of a pointer is an address

```
int n;
int* pn = &n;
```

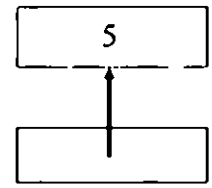
The type of a pointer is 'pointer to' the type of value it holds the address of (points to). The above example declares a pointer to integer an integer value.



4.2.3 Operations

The value a pointer points to is accessed using an asterisk

```
*pn = 5;
```



Pointers to arrays or dynamically allocated memory may be accessed using array indices

```
int an[5];

/* Nb.  the address (&) operator is not required as arrays are
 * always referred to by their address
 */

pn = an;

an[2] = 5;
```

The address of pointers may be modified

```
pn++;
pn = pn+5;
```

The main use of this is when searching through an array.

```
char sName[] = "Hello there!";
char* pc = sName;

/* Search for a space */

while (*pc != ' ' && *pc != '\0')
{
   pc++;
}
```

Nb. The declaration of a pointer `int* p` (or `int *p`) should be distinguished from the accessing of the value a pointer points to `*p`.

4.2.4 NULL Pointer

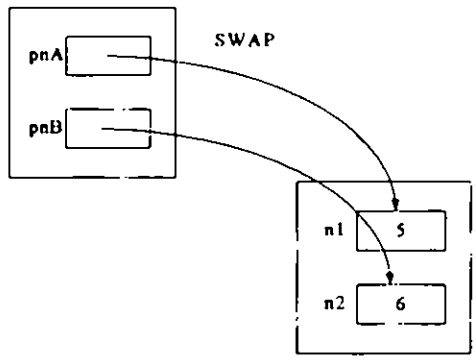
The NULL pointer is defined in `stdio.h`. Pointers may be assigned to NULL to indicate that they do not point to anything. Functions often return NULL as an error value.

4.2.5 Passing as arguments

It is often desirable to modify the values of arguments to functions. Since arguments are passed by value (i.e. copies are passed) by default, to modify the original it is necessary to pass a pointer to it.

```
void swap(int *pnA,int *pnB)
{
   int nTemp = pnA;
   *pnA = *pnB;
   *pnB = nTemp;
}

int n1 = 5;
int n2 = 6;
swap (&n1,&n2);
```



In this example the values `n1` and `n2` are to be swapped. The function `swap` must therefore change the actual values rather than copies of them. This is done through pointers.

4.2.6 Dynamic memory allocation

It isn't always possible or desirable to determine the size or number of arrays or structures at compile time. This is where dynamic memory allocation is used. This feature allows a specific size block of memory to be obtained.

Memory allocated is manipulated through a pointer. A cast is necessary to convert the data to the correct type.

```
int* plValues = (long*)malloc(nValues*sizeof(long));
```

This allocates a block of memory large enough to hold `n` values of size `long`. These values can now be accessed as for an ordinary array.

```
for (i = 0; i < nValues; i++)
{
   plValues[0] = 0;
}
```

`malloc` allocates unitialised memory of a specific size. It returns `NULL` if unable to allocate the memory.

`calloc` allocates memory initialised to zero and takes the two arguments, the size of each element and the number of elements.

Always check that memory was allocated successfully before using it.

Memory allocated with `malloc` or `calloc` persists until a call is made to release it using `free`. Therefore every call to allocate memory must have a corresponding call to release it.

```
if (plValues != NULL)
{
   free(plValues);
};
```

4.2.7 Function pointers

Function pointers allow the function to be called to be determined at runtime. The alternative to this is to have a `switch` statement with all the alternative functions to be called.

The capability of detemining functions at run-time is potentially very powerful. C++ exploits this in a type safe format with virtual functions. In C, function pointers are used infrequently.

4. 3 Structures

4.3.1 Purpose

A structure is a collection of one or more variables, possibly of different types. They enable related data to be grouped together. For example personal details such as name, address and phone numbers could be grouped into one entity.

4.3.2 Declaring and assigning

A structure definition contains an optional name (the structure tag) and a list of members with types.

```
struct DETAILS
{
   char sName[20];
   int nAge;
   char sAddress[50];
};
```

A variable may be declared of this type, either by placing the names immediately after the type definition, in which case the structure tag is optional:

```
struct
{
   char sName[20];
   int nAge;
   char sAddress[50];
} details1, details2;
```

If a tag is supplied then the instances may be declared thus:

```
struct DETAILS details1, details2;
```

Structure variables may be initialised at declaration:

```
struct DETAILS fred = {"Fred",55,"37 T Lane"};
```

Individual members are accessed using the "." operator.

```
details.nAge = 5;
```

Entire structures may be assigned, they may also be passed as arguments and returned from functions.

Structures may be placed inside arrays and arrays inside structures. Dynamically allocated memory may be cast to the size of a structure.

Since structures may be quite large, they are often passed as arguments to functions by address. Structure members are accessed through a pointer to the structure using to -> operator.

```
func(&details1);

int func(struct DETAILS* pDetails)
{
   pDetails->nAge = 10;
};
```

4.3.3 Typedef

The facility allows the creation of new data type names.

```
typedef int BOOL;
BOOL bAccept;
```

When used with structures this removes the need to precede the name of every declaration of a structure with the word `struct`.

```
typedef
{
   int nDay;
   int nMonth;
   int nYear;
} DATE;

typedef struct
{
   long lSite;
   DATE date;
   int nReadings;
   double fReadings[50];
} READING;

READING reading;
```

4.4 Unions

4.4.1 Purpose

A `union` is a variable that may hold objects of different type and sizes at different times. They provide a way to manipulate different kinds of data in a single area of storage. It allows a single variable to hold any one of several types.

4.4.2 Declaring and assigning

A union is declared thus.

```
union u_tag
{
   int nVal;
   float fVal;
   char *pcVal;
} u;
```

The variable `u` will be large enough to hold the largest of the three types.

```
u.nVal = 5;
```

Care must obviously taken to ensure that if a value of one type is stored in a union then it is retrieved as a variable of the same type. Unions may be placed inside structures which could have a flag indicating the type of the value stored.

The exception to this is where unions are used to convert data from one type to another.

```
union lxb
{
   long lu;              /* The long  */
   unsigned char bu[4];  /* The bytes */
} lxb;
```

```
lxb.bu[0] = 'A';
lxb.bu[1] = 'B';
/* ... */

long l = lxb.lu;
```

Like structures, unions may be assigned and passed as arguments.

5 Standard Libraries

5.1 Input Output

5.1.1 Standard I/O and Streams

C uses streams for input and output. There are three built in streams:

`stdout`	- output to the screen
`stdin`	- input from the keyboard
`stderr`	- error output to the screen

These may be redirected when the program is run so that, for example, output goes to a file. This is operating system dependant.

```
c:\> test > output.txt
```

C has many functions for manipulating streams defined in `stdio.h`. These allow the retrieval and output of characters and strings to and from streams.

```
fgetc(FILE *pStream);          /* Get a character from a stream */
fgets(char *ps, int n, FILE *pStream); /* Get a string */

fputc(int c, FILE* pStream); /* Output a character to a stream */
fputs(const char *ps, FILE* pStream) /* Output a string */

fprintf(FILE* pStream, const char *psFormat, ...);
fscanf(FILE* pStream, const char* psFormat, ...);
```

Functions also exist for output directly to `stdout` and reading directly from `stdin`. These are equivalent to passing `stdin` or `stdout` as arguments to the above functions.

```
fprintf(stdout,"Hello\n");  /* is equivalent to */
printf("Hello\n");
```

5.1.2 Formatted Output

The output function printf translates internal values into characters.

```
int printf(char *pFormat, ...);
```

`printf` converts, formats and prints its arguments on the standard output under the control of the format string. It returns the number of characters printed.

Ordinary characters in the format statement are printed straight out.

Conversion specifications begin with a `%` and end with a conversion character.

```
%[flags][width][.precision][{h|l}]type
```

Flags are:

-	left adjustments
+	always print sign
space	if no sign prefix with space
0	pad with leading zeros
width	minimum field width
precision	maximum characters for string
	number of decimal places for float
	maximum number of digits for int
h	short
l	long

The conversion characters are:

d,i	int
o	octal
x	hex
u	unsigned int
c	char
s	char *
f	double
e	double (exponential)
%	% character

```
void OutputTime(TIME *pTime)
{
   printf("%02:u%02u",pTime->uHours,pTime->uMinutes);
};
```

This example displays the time as 09:34 where the hours and minutes field are at least two characters wide and padded with leading zeros if necessary.

The number and type of arguments are determined from the format. If these do not match the argument supplied, no compiler error is given.

5.1.3 Formatted Input

`scanf` reads characters from the standard input, interprets them according to their format specifications and stores the results in the remaining arguments.

It returns the number of successfully matched and assigned input items. EOF is returned if the end of file was met.

```
int scanf(char* pFormat, ...);
```

The arguments must be pointers as the values are to be returned.

In the format specification, blanks and tabs are ignored. Ordinary characters must match the next non-white characters in the input stream.

```
%[*][width][{h|l}]type
```

*	assignment suppression
width	maximum number of characters to be read in
h	short
l	long

The types available are:

d	int *
i	int * (may be octal or hex)
o	int * (octal)
u	unsigned int *
x	int * (hex)
c	char * (character)
s	char * (string)
e,f,g	float *

```
/* Input a date delimited by any character e.g. 05/07/1992 */

if (sscanf(szDate,"%u%*c%u%*c%u",&uDay, &uMonth, &Year) != 3))
{
   printf("Invalid date");
}
```

Nb. The it is the address of the arguments uDay, uMonth and uYear which is passed to the function as values are to be returned.

5.1.4 File Access

Streams are opened using the fopen command. The first argument gives the file name, the second the mode - whether it is being opened for reading/writing.

N.B. Since the backslash character has special meaning in C it must be quoted. This does not affect UNIX path names.

```
FILE *pOpen = fopen("\\tmp\\test.out","w");
```

The function returns a pointer to NULL if the file cannot be opened.

Modes available are:

"r" open for reading
"w" open for writing
"a" append, open file for writing to end

A + symbol e.g. "a+" indicates the file should be opened for reading and writing. A call to fflush must be made before switching mode. It is unusual to open files for both reading and writing at the same time.

A b character as part of the mode e.g. "rb" opens the file in binary mode. This causes no conversions to be made to the file upon opening. Under DOS on, opening a file in text mode all newline characters are prefixed with a linefeed character. Upon closing a DOS file in text mode the newline characters are removed.

Streams must be closed (except built in ones) before exiting the program.

```
if (pFile != NULL)
{
   fclose(pFile);
}
```

5. 2 String functions

5.2.1 Using

Strings may be manipulated using sprintf and scanf.

```
int StringAsDate(char *psDate, DATE *pDate)
{
   BOOL bOK = TRUE;

  /* Retrieve date from string */

   if (sscanf(psDate,"%u%*c%u%*c%u",
              &pDate->uDay, &pDate->uMonth, &pDate->uYear) != 3)
   {
      bOK = FALSE;
   } else
   {

/* Ensure date is valid */

      if (IsValidDate(pDate) == FALSE)
      {
         bOK = FALSE;
      }
   }
   return bOK;
};
```

A standard library also exists string.h for handling null terminated strings.

strcpy	copies one string into another
strcat	appends one string to the end of another

Variants of these `strncpy` and `strncat` take the length of the destination string as an argument. This is safer and should be used in preference.

`strcmp` compares two strings and returns
0 if they are equal,
<0 if the first comes before the second in the alphabet and
> 0 otherwise

`strlen` returns the length of the string excluding the null terminating character

When using strings, always ensure that the array is sufficiently large to hold the result.

6 Compiling under UNIX

The C compiler on the Silicon graphics is an ANSI standard compiler.

The UNIX compiler is called `cc`.

The compiler on the SUNS is non-standard. The GNU C compiler is available however and this is standard. To use GNU compiler on the SUN workstations, first type 'SETUP GCC' then use 'gcc' instead of 'cc'

The cc command also links the files. To compile a number of source files type:

cc file1.c file2.c

By default the UNIX compiler creates an output file called a.out. This can be overridden by using the -o option.

cc -o name.out file1.c file2.c

7 Common mistakes in C

These are examples of mistakes everyone makes in C, they are very difficult spot even by an experienced programmer so be aware of them.

Some of these are picked up by the compiler but may give misleading error messages. Others aren't.

1) Semicolon at the end of for statement means that the loop is only executed once.

```
int i;
int a = 0;
for (i = 0; i < 5; i++);
{
    a++;
};
```

2) Assignment operator instead of equality

```
if (a = b)
{
   /* ... */
}
```

3) Missed () off call to function with no arguments.

```
BOOL _bValid = TRUE;
int IsValid()
{
   return _bValid;
}

if (IsValid)
{

}
```

4) #defines are a textual substitution should be `#define I 7` with no semicolon

```
#define I=7
#define I 7;
```

5) Missing semicolon, compiler tries to make 'int' an instance of the struct.

```
struct INTEX;
{
   int i;
}

int nValue;
```

6) Returning a pointer to array which is destroyed on return from function

```
char *GetString()
{
  char sString[10] = "Hello";
  return sString;
}
```

7) Not passing address of scanf second argument, not passing value to printf second argument

```
scanf("%i",nValue);     /* should be &nValue */
printf("%i",&nValue);   /* should be nValue */
```

8) Cannot assign directly to a string

```
char s[20];
s = "Hello"; /* WRONG */
```

8 Bibliography

The C Programming Language, 2nd Edition Kernighan and Ritchie.

COMP.LANG.C Newsgroup: Frequently asked questions.