LIQUIDNET FAMILY HIGH SCHOOL



C PROGRAMMING LANGUAGE SYLLABUS FOR MPC AND MEC

SENIOR FOUR

TEACHER: **Emile BANGISHAKA**
@Agahozo Shalom Youth Vilage
E mail: ceitmember@yahoo.com

Academic year 2012

CONTENTS

Chapter 1: Introduction to C programming 4
1.1 Expressions and Operators 4
1.2 Assignment Statement 4
1.3 Arithmetic operators 4
1.4 Increment and Decrement Operators 5
1.5 Cast Operators 5
1.6 Bitwise Operators 5
1.7 Comparison 6
1.8 Logical Connectors 7
1.9 Assignment Operators 7
1.10 Characters Special Characters 8
Chapter 2 Variables and Expressions 8
2.1 Variables 8
2.1.1 Global Variables Vs Local Variable 8
2.1.2 External Variables 9
2.1.3 Variable Declaration 9
2.2.4 Variable Types 9
2.1.5 Constants 12
2.2 The C Preprocessor 12
2.3 Input and Output 12
2.3.1 Formatted Output with printf ( ) 13
2.3.2 Basic Structure of a C Program 13
2.3.3 Formatted Output with scanf ( ) 16
2.3.5 putchar (char) ; 16
2.4 Scope, Blocks 17
2.4.1 Blocks and Scope 17
2.4.2 Variable Storage Classes … 17
Chapter 3 - Control Flow 18
3.1 Statements and Blocks 18
3.2 If-Else 18
3.3 Else-If 19
3.4 Switch 20
3.5 Loops - While and For 20
3.6 Loops - Do-While 21
3.7 Break and Continue 21
Chapter 4 Functions 21
4.1 Building Blocks of Programs 21
4.2 Return Value 21

4.3 Function Parameters 22
4.4 Variable Function Parameters 22
4.5 Function Definition and Declaration 22
4.6 Function Prototypes 23
4.6.1 Standard Header Files 24
Chapter 5 Arrays, Pointers and Strings 24
5.1 Pointers are Addresses 24
5.2 Pointers are not Integers 24
5.3 The * and & Operators 25
5.4 Declaring Pointer Variables 25
5.5 Arrays 26
5.5.1 Why use arrays? 27
5.5.2 Limits and the Dimension of an array 27
5.5.3 Arrays and for loops 28
5.5.4 Arrays of More Than One Dimension 29
5.5.5 Arrays and Nested Loops 30
5.5.6 Initializing Arrays 31
5.5.7 Arrays and Pointers 32
5.5.8 Arrays as Parameters 33
5.6 Using Structures 33
5.6.1 Arrays of Structures 35
5.6.2 Structures of Structures 35
5.6.3 Pointers to Structures 37
5.6.4 Pre-initializing Static Structures 37
APPENDICES 39
Appendix I: Maths Library 39

Chapter 1: Introduction to C programming

C (pronounced like the letter C) is a general-purpose computer programming language developed between 1969 and 1973 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system.
Although C was designed for implementing system software, it is also widely used for developing portable application software.
C is one of the most widely used programming languages of all time and there are very few computer architectures for which a C compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++, which began as an extension to C.
C is a relatively ``low-level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

1.1 Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /. Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here. Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

1.2 Assignment Statement

The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like

$y = (m * x) + c$

This assignment will save the value of the expression in variable **y**.

1.3 Arithmetic operators

*	multiplication
/	division
%	remainder after division (modulo arithmetic)
+	addition
-	subtraction and unary minus

The / operator is used for two different operations: integer and floating point division. If both operands of the divide operator are of integral (char, int and its derivatives) type then

integer division is performed. If either operand is float, double or long double then real division is undertaken.

```
int main()
    {
    float a;

    a = 1 / 3;
  printf("%f\n", a);
  return 0;
  }
```

1.4 Increment and Decrement Operators

Increment and decrement operators give a shorthand method of adding/subtracting 1from an object.

++	increment
--	decrement

These operators can be prefix or postfix. An example of the different behaviour of the prefix and postfix forms is given below, but essentially with the prefix form the variable is changed before the value of the expression in which it appears is evaluated, and with the postfix form the variable is modified afterwards.

```
b = 3;
a = b++ + 6;            /* a = 9,  b = 4   */

b = 3;
a = ++b + 6;            /* a = 10, b = 4   */
```

1.5 Cast Operators

Cast operators allow the conversion of a value of one type to another.

(float) sum;	*converts type to float*
(int) agahozo;	*converts type to int*

1.6 Bitwise Operators

Bitwise operators allow manipulation of the actual bits held in each byte of a variable. Each byte consists of a sequence of 8 bits, each of which can store the value 0 or 1

Operator	Operation
~	one's complement
&	bitwise AND
^	bitwise XOR
|	bitwise OR
<<	left shift (binary multiply by 2)
>>	right shift (binary divide by 2)

Examples

Operator	Operand	Result
~	0010111	1101000
<<	0010111	0101110
>>	0010111	0001011

AND	XOR	OR
0 & 0 = 0	0 ^ 0 = 0	0 | 0 = 0
1 & 0 = 0	1 ^ 0 = 1	1 | 0 = 1
0 & 1 = 0	0 ^ 1 = 1	0 | 1 = 1
1 & 1 = 1	1 ^ 1 = 0	1 | 1 = 1

1.7 Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

C notation	Meaning
= =	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Note that == is used in comparisons and = is used in assignments.
Comparison operators are used in expressions like the ones below.

x == y
i > 10
a + b != c

In the last example, all arithmetic is done before any comparison is made. These comparisons are most frequently used to control an if statement or a for or a While loop.

1.8 Logical Connectors

These are the usual And, Or and Not operators.

Symbol	Meaning
&&	And
||	Or
!	Not

They are frequently used to combine relational operators, for example

x < 20 && x >= 10

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state.

1.9 Assignment Operators

C has many assignment operators. Pascal and FORTRAN have one.

=	assign
+=	assign with add
-=	assign with subtract
*=	assign with multiply
/=	assign with divide
%=	assign with remainder
>>=	assign with right shift
<<=	assign with left shift
&=	assign with bitwise AND
^=	assign with bitwise XOR
|=	assign with bitwise OR

All but the first assignment operator are shorthand methods of modifying a variable or object. For example, the following two statements are equivalent:

```
a = a + 17;
a += 17;
```

The assign with operators come into their own where the specification of the object to be modified is long and complicated, and the chance for error in specifying it twice is large.

1.10 Characters Special Characters

The following special patterns are used to represent a single character in C programs. The leading backslash in the single quotes indicates that more information is to follow.

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

Chapter 2 Variables and Expressions

2.1 Variables

Definition: A variable is a way of referring to a memory location used in a computer program. This memory location holds values- perhaps numbers or text or more complicated types of data like a payroll record.

In C, a variable must be declared before it can be used. Variables can be declared at
The start of any block of code, but most is found at the start of each function. Most
Local variables are created when the function is called, and are destroyed on return
From that function.

2.1.1 Global Variables Vs Local Variable

Local variables are declared within the body of a function, and can only be used within that function. This is usually no problem, since when another function is called, all required data is passed to it as arguments. Alternatively, a variable can be declared globally so it is available to all functions. Modern programming practice recommends against the excessive use of global variables. They can lead to poor program structure, and tend to clog up the available name space.

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

2.1.2 External Variables

Where a global variable is declared in one file, but used by functions from another, then the variable is called an external variable in these functions, and must be declared as such. The declaration must be preceded by the word extern. The declaration is required so the compiler can find the type of the variable without having to search through several source files for the declaration. Global and external variables can be of any legal type. They can be initialized, but the initialization takes place when the program starts up, before entry to the main function.

2.1.3 Variable Declaration

Variables should be declared either outside a function or at the start of a block of code, After the opening {and before any other statements. They must be declared before use and the type must normally be specified (exceptions are external and static declarations where the type *int* is implicit.

```
int       miles, yards;       /* global variables */
main()
{
float     kilometres;     /* local variables */
```

2.2.4 Variable Types

There are a number of 'built-in' data types in C. These are listed below. Where a shorter version of the type name exists, this is given in brackets; essentially the base Type *int* is implicit whenever *short*, *long*, or *unsigned* are used.

```
short int                 (short)
unsigned short int        (unsigned short )
char
unsigned char
signed char
int
unsigned int              (unsigned)
long int                  (long)
unsigned long int         (unsigned long )
float
double
long double
```

The range of values that can be stored in variables of these types will depend on the Compiler (*A compiler is a computer program that transforms human readable source code of another computer program into the machine readable code that a CPU can execute.)* and computer that you are using, but on an IBM PCs and the Borland Turbo C compiler the ranges are:

`short int`	-128 → 127 (1 byte)
`unsigned short int`	0 → 255 (1 byte)
`char`	0 → 255 *or* -128 → +127 [2] (1 byte)
`unsigned char`	0 → 255 (1 byte)
`signed char`	-128 → 127 (1 byte)
`int`	-32,768 → +32,767 (2 bytes)
`unsigned int`	0 → +65,535 (2 bytes)
`long int`	-2,147,483,648 → +2,147,483,647 (4 bytes)
`unsigned long int`	0 → 4,294,967,295 (4 bytes)
`float`	single precision floating point (4 bytes)
`double`	double precision floating point (8 bytes)
`long doubl e`	extended precision floating point (10 bytes)

The ANSI C standard states only that the number of bytes used to store a *long int* is equal to or greater than the number used to store an *int*, and that the size of an *int* is at least as big as the size of a *short int*. The C operator, *sizeof*, reports the number of bytes used to store a variable or a type 3 , and so the above can be rewritten as:

```
sizeof(long int) ≥ sizeof(int) ≥ sizeof(short int)
```

It may be the case, for your compiler and system, that the sizes for all three are the same. The same holds for the three floating point types:

```
sizeof(long double) ≥ sizeof(double) ≥ sizeof(float)
```

If you have an ANSI C compiler, the actual ranges for your system can be found in the header files *limits.h* for integer types and *float.h* for floating point types. ANSI C has the qualifier *signed* which can be applied to any of the integer types (including *char*). *Signed* is the default for all the *int* types, but is dependent on the compiler for *char*.

2.6 Variable Names

Variable and function names (or identifiers) consist of a letter followed by any combination of letters and digits. Underscore (_) is treated as a letter and it is possible have identifiers that begin with an underscore. However, identifiers beginning with an underscore followed by either another underscore or an upper case letter are reserved, and identifiers beginning with an underscore followed by a lower case letter are reserved for file scope identifiers. Therefore, use of underscore as the first letter of an identifier is not advisable.

Upper and lower case letters are distinct, so *agahozo* and *Agahozo* are different Identifiers. ANSI C states that the first 31 characters of an identifier are significant except for objects with *external linkage*. External variables are those declared outside of any function, all functions are external; all external objects (variables and functions) have *external linkage* unless they are declared to be *static*. As external names are used by the linker, which is not part of C, the limits on the number of significant characters are much worse, only six characters and no distinction between upper and lower case. Some linkers will be better than this, but it cannot be relied upon. For this reason, the following is dangerous, as the names of the variables do not differ in the first six characters.

```
int integer1;
int integer2;

int main()
   {
   }
```

List of C keywords that cannot be the variable name:

The following names are reserved by the C language. Their meaning is already defined, and they cannot be re-defined to mean anything else.

auto	*do*	*goto*	*signed*	*unsigned*
break	*double*	*if*	*sizeof*	*void*
case	*else*	*int*	*static*	*volatile*
char	*enum*	*long*	*struct*	*while*
const	*extern*	*register*	*switch*	
continue	*float*	*return*	*typedef*	
default	*for*	*short*	*union*	

Other than these names, you can choose any names of reasonable length for Variables, functions etc. The names must begin with a letter or underscore (letters Are better), and then can contain letters, numbers and underscores.

Examples of legal variable names include

x	*result*	*outfile*	*newyear*
x1	*x2*	*out_file*	*new_year*

power impetus gamma hi_score

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a Variable name. Most modern ones don't apply this limit though.
The rules governing variable names also apply to the names of functions.

2.1.5 Constants

An integer constant like 1234 is an int. A long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal u or U, and the suffix ul or UL indicates unsigned long. Floating-point constants contain a decimal point (123.4) or an exponent (1e-2) or both; their type is double, unless suffixed. The suffixes f or F indicates a float constant; l or L indicates a long double. The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an unsigned long constant with value 15 decimal.

2.2 The C Preprocessor

The C Preprocessor is a separate step in the compilation process, but not part of the compiler. C Preprocessor is just a text substitution tool and we'll refer to the C Preprocessor as the CPP.

All preprocessor lines always begin with #. The unconditional directives are as follows:

#define – Define a preprocessor macro
#include – Insert a particular header from another file
#undef – Undefine a preprocessor macro

The conditional directives are as follows:

#ifndef – If this macro is not defined
#ifdef – If this macro is defined
#if – Test if a compile time condition is true
#elif – #else an #if in one statement
#else – The alternative for #if
#endif – End preprocessor conditional

2.3 Input and Output

C has no built-in statements for input or output. A library of functions is supplied to perform these operations. The I/O library functions are listed the "header" file <*stdio.h*> you do not need to memorize them, just be familiar with them.

2.3.1 Formatted Output with printf ( )

This function provides for formatted output to the screen.
The syntax is:

```
printf ("format", var1, var2, … ) ;
```

The "format" includes a listing of the data types of the variables to be output and, optionally, some text and control character(s).

- Examples:

```
1.printf("Agahozo Shalom Youth Village ");

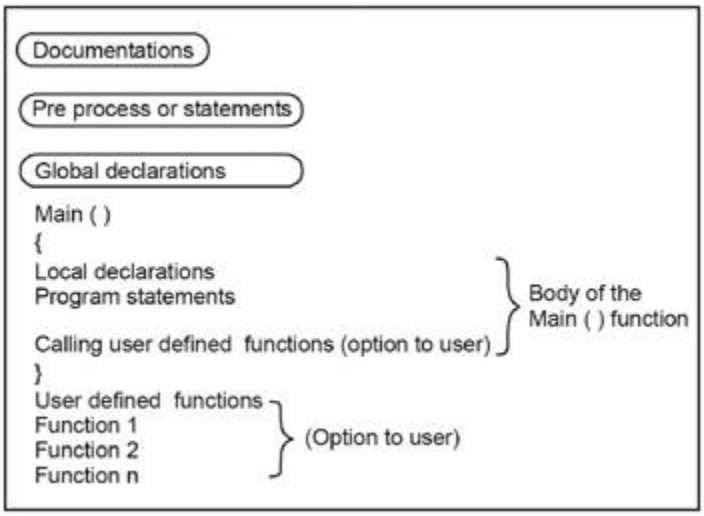
2.float a ;
  int b ;
  scanf ( "%f%d",&a,&b ) ;
     printf ( "You entered %f and %d \n", a, b ) ;
```

The prototype for the *printf ()* is:

Control String Entry	Conversion	What Gets Printed
%d	int	Decimal Integer
%o	Unsigned int	Unsigned Octal
%lo	Unsigned long int	Unsigned Octal
%f	double	Floating Point Value
%c	Unsigned char	Single Character
%s	char	Character String
%e	Double float	Fixed point number

2.3.2 Basic Structure of a C Program

The structure of a C program is a protocol (rules) to the programmer, while writing a C program. The general basic structure of C program is shown in the figure below. The whole program is controlled within main ( ) along with left brace denoted by "{" and right braces denoted by "}". If you need to declare local variables and executable program structures are enclosed within "{" and "}" is called the body of the main function. The main ( ) function can be preceded by documentation, preprocessor statements and global declarations.



Documentations

The documentation section consist of a set of comment lines giving the name of the rogram, the another name and other details, which the programmer would like to use later.

Preprocessor Statements

The preprocessor statement begin with # symbol and are also called the preprocessor directive. These statements instruct the compiler to include C preprocessors such as header files and symbolic constants before compiling the C program. Some of the preprocessor statements are listed below.

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>      } header files
# include <CONIO.h>

# define P L 3.1412.
# define TRVE 1           } Symbolic constants
# define FALSE Ø
```

Global Declarations

The variables are declared before the main ( ) function as well as user defined functions are called global variables. These global variables can be accessed by all the user defined functions including main ( ) function.

The main ( ) function

Each and Every C program should contain only one main ( ). The C program execution starts with main ( ) function. No C program is executed without the main function. The main ( ) function should be written in small (lowercase) letters and it

should not be terminated by semicolon. Main ( ) executes user defined program statements, library functions and user defined functions and all these statements should be enclosed within left and right braces.

Braces

Every C program should have a pair of curly braces ({, }). The left braces indicates the beginning of the main ( ) function and the right braces indicates the end of the main ( ) function. These braces can also be used to indicate the user-defined functions beginning and ending. These two braces can also be used in compound statements.

Local Declarations

The variable declaration is a part of C program and all the variables are used in main ( ) function should be declared in the local declaration section is called local variables. Not only variables, we can also declare arrays, functions, pointers etc. These variables can also be initialized with basic data types. For examples.

```
main ()
    {
     int sum=0 ;
     int x ;
     floatf ;
    }
```

Here, the variable sum is declared as integer variable and it is initialized to zero. Other variables declared as int and float and these variables inside any function are called local variables.

Program statements

These statements are building blocks of a program. They represent instructions to the computer to perform a specific task (operations). An instruction may contain an input-output statements, arithmetic statements, control statements, simple assignment statements and any other statements and it also includes comments that are enclosed within /* and */ . The comment statements are not compiled and executed and each executable statement should be terminated with semicolon.

User defined functions

These are subprograms, generally, a subprogram is a function and these functions are written by the user are called user ; defined functions. These functions are performed by user specific tasks and this also contains set of program statements. They may be written before or after a main () function and called within main () function. This is an optional to the programmer.

Now, let us write a small program to display some message shown below.

```
# include <stdio.h>
main()
{
  Printf ("welcome to liquidnet family high school");
}
```

2.3.3 Formatted Output with scanf ( )

This function provides for formatted input from the keyboard.

The syntax is:

scanf ("format" , &var1, &var2, …) ;

The "format" is a listing of the data types of the variables to be input and the & in front of each variable name tells the system WHERE to store the value that is input. It provides the address for the variable.

Example:
```
float a;  int b;
scanf ("%f%d", &a, &b);
```

2.3.4 getchar ( ) ;

This function provides for getting exactly one character from the keyboard.

Example:
```
char ch;
ch =getchar( );
```

2.3.5 putchar (char) ;

This function provides for printing exactly one character to the screen.

Example:
```
char ch;
ch = getchar ( );/* input a character from kbd*/
putchar (ch) ;  /* display it on the screen */
```

2.4 Scope, Blocks

2.4.1 Blocks and Scope

C is a block structured language. Blocks are delimited by { and }. Every block can have its own local variables. Blocks can be defined wherever a C statement could be used. No semi-colon is required after the closing brace of a block.

```
{
int a = 5;
printf("\n%d", a);
{
int  a = 2;
printf("\n%d", a);
}
}
```

Reference to a variable will be to the variable of that name in the nearest enclosing block.

2.4.2 Variable Storage Classes

Auto: The default class. Automatic variables are local to their block. Their storage space is reclaimed on exit from the block.

Register: If possible, the variable will be stored in a processor register. May give faster access to the variable. If register storage is not possible, then the variable will be of automatic class. Use of the register class is not recommended, as the compiler should be able to make better judgment about which variables to hold in registers, in fact injudicious use of register variables may slow down the program.

Static: On exit from block, static variables are not reclaimed. They keep their value. On re-entry to the block the variable will have its old value.

Extern: Allows access to external variables. An external variable is either a global variable or a variable defined in another source file. External variables are defined outside of any function. (Note: Variables passed to a function and modified by way of a pointer are not external variables)

static external: External variables can be accessed by any function in any source file which make up the final program. Static external variables can only be accessed by functions in the same file as the variable declaration.

Chapter 3 - Control Flow

The control-flow of a language specify the order in which computations are performed.

3.1 Statements and Blocks

An expression such as x = 0 or i++ or printf (...) becomes a statement when it is followed by a semicolon, as in

```
x = 0;
  i++;
  printf(...);
```

In C, the semicolon is a statement terminator, Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, While, or for are another There is no semicolon after the right brace that ends a block.

3.2 If-Else

The if-else statement is used to express decisions. Formally the syntax is

```
if (expression)
      statement1
  else
      statement2
```

where the else part is optional. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (expression is zero) and if there is an else part, statement2 is executed instead. Since an if tests the numeric value of an expression, certain coding shortcuts are possible.

The most obvious is writing `if(expression)` instead of `if(expression != 0)` Sometimes this is natural and clear; at other times it can be cryptic.

Because the else part of an if-else is optional, there is an ambiguity when an else if omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if.

For example, in

```
if (n > 0)
      if (a > b)
          z = a;
      else
          z = b;
```

the else goes to the inner if, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0)
 {
        if (a > b)
             z = a;
}
        else
        z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n > 0)
        for (i = 0; i < n; i++)
            if (s[i] > 0) {
                printf("...");
        return i; }
            else
 /* WRONG */    printf ("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the else with the inner if. This kind of bug can be hard to find; it's a good idea to use braces when there are nested ifs.

By the way, notice that there is a semicolon after z = a in

```
    if (a > b)
        z = a;
    else
        z = b;
```

This is because grammatically, a statement follows the if, and an expression statement like ``z = a;" is always terminated by a semicolon.

3.3 Else-If

The construction

```
 if (expression)
        statement
   else if (expression)
        statement
   else if (expression)
        statement
   else if (expression)
        statement
   else
        statement
```

This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces. The last else part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else
        statement
```

can be omitted, or it may be used for error checking to catch an ``impossible'' condition.

3.4 Switch

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

```
    switch (expression) {
        case const-expr: statements
        case const-expr: statements
        default: statements
    }
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

3.5 Loops - While and For

We have already encountered the while and for loops. In

```
  while (expression)
        statement
```

the expression is evaluated. If it is non-zero, statement is executed and expression is re-evaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement. The for statement

```
    for (expr1; expr2; expr3)
        statement
```

is equivalent to

```
  expr1;
    while (expr2) {
        statement
        expr3;
    }
```

3.6 Loops - Do-While

The while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

The syntax of the do is

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

The statement is executed, then expression is evaluated. If it is true, statement is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, do-while is equivalent to the Pascal repeat-until statement.

Experience shows that do-while is much less used than while and for.

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while, and does, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

The continue statement is related to break, but less often used; it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration.

Chapter 4 Functions

The *function* is one of the most basic things to understand in C programming. A *function* is a sub-unit of a program which performs a specific task.

4.1 Building Blocks of Programs

All C programs consist of one or more functions. Functions are the building blocks of A program. All functions are at the same level there is no nesting. One function must be called main.

4.2 Return Value

All functions can return a value, including *main*. Functions can return arithmetic values (`int, float` etc.), structures, unions, pointers or void. If the return type is specified as being void, then no value is returned by the function. Functions cannot return a function or an array.

For functions that will return a value, the `return` statement is used in conjunction with an expression of an appropriate type. This causes immediate termination of the function, with the value of the expression being returned to the caller. `return` can also be used with functions of type `void`, but no return value can be specified.

4.3 Function Parameters

All functions (including main) can accept parameters.
Example:

```
double minimum (double x, double y)
{
if (x < y)
return x;
else
return y;
}
```

The parameter list must be specified, and their types must be declared.
The parameters to the function (the expressions given in the function call) are passed by value only.
It is common to call the variables specified in the function definition parameters and the expressions given in a function call arguments. For example, in the following call of minimum , the expressions a and b * 2 are the arguments to the function. The values of the two expressions will be copied into the parameters x and y.

```
minimum(a, b*2);
```

4.4 Variable Function Parameters

All function parameters are passed by value. To make a function alter a variable, the address of the variable must be passed, i.e. pass the variable by reference.

```
int max(int a, int b, int *c);
int main(){
int x = 4, y = 5, z;
max(x, y, &z);          /* generate a pointer to z */
return 0;}
int max(int a, int b, int *c)
{if (a > b)
*c = a;                 /* *c modifies the variable */
else                    /* whose address was passed */
*c = b;                 /* to the function */
}
```

4.5 Function Definition and Declaration

A function definition is where the function name, parameters, code and return type are specified. A function declaration is where the name and return type of a function are given. The definition of a function includes a declaration of that same function implicitly.

A function can be declared many times (as long as the declarations declare the function

to be of the same type) but can only be defined once. Declarations of functions are sometimes necessary to appease the compiler, which always assumes, if the information is not available, that all functions return *int*.

```
        /* declaration of minimum() */
        double minimum(double x, double y);
         int main(){
printf("%f\n", minimum(1.23, 4.56));
return 0;}
/* definition of minimum()  */
double minimum(double x, double y)
double x, y;
{ if (x < y)
 return x;
else
 return y;}
```

The problem could also be solved by placing the function definition before the call to the function.

4.6 Function Prototypes

The ANSI C standard introduces function prototypes. An example is given below. It also allows function definitions to be written in the same form as the new prototypes.

```
double minimum(double, double);
/* prototype  of minimum() */
int main()
{
printf("%f\n", minimum(1.23, 4.56));
return 0;
}
double minimum(double x, double y)
/* definition of minimum() */
{
if (x < y)
 return x;
 else
 return y;
}
```

The use of prototypes or the new style function definition allows the compiler to check that the parameters to a function are sensible (not necessarily the same), as well as checking the return type of the function; but only if the definition or prototype appears before the function call.

ANSI C draws a distinction between the following two statements. The first is a function declaration stating that fred takes an, as yet, unspecified number of parameters. The second is a function prototype which states that jim takes no parameters.

```
double fred();     /* declaration */
double jim(void); /* prototype */
```

4.6.1 Standard Header Files

Prototypes of the library functions are given in several standard header files. For example,

`stdio.h`	contains prototypes for printf , scanf , putchar and
`getchar.h`	Other standard header files are:
`assert.h`	assertions
`ctype.h`	character class tests
`float.h`	system limits for floating point types
`limits.h`	system limits for integral types
`math.h`	mathematical functions
`setjmp.h`	non-local jumps
`signal.h`	signals and error handling
`stdarg.h`	variable length parameter lists
`stdlib.h`	utility functions; number conversions, memory allocation, exit and system , Quick Sort string.h string functions
`time.h`	date and time functions

To include these standard header files in your code, use the preprocessor directive #include and place angle brackets around the name of the file, e.g. #include <math.h>

Chapter 5 Arrays, Pointers and Strings

5.1 Pointers are Addresses

A pointer is really the address of something in memory. Pointers are needed if it is required that a function change the value of an object (though global variables could be used). For example, the `scanf` function takes pointers as arguments.

5.2 Pointers are not Integers

It is possible to treat pointers as though they are integers since they hold actual addresses in memory e.g 7232. `printf, scanf` , assignment and some of the arithmetic operators work with pointers, but in many cases such operations should not be used as the results will differ from one machine to another.

5.3 The * and & Operators

& gives the address of something in memory, that is it generates a pointer to the object
* gives what is pointed at by a pointer (called dereferencing)

For example, assuming the following declaration:

```
int i = 17;
```

Then the expression &i generates a pointer to the variable i. We can then find out what this pointer is pointing at by applying the * operator to the newly generated pointer; thus *&i should have the value 17.

Decisions chose

5.4 Declaring Pointer Variables

To declare a pointer variable we must state the type of object that the pointer will point at. To declare a pointer to an integer, we write:

```
int *ptr;
```

which says that the type of `*ptr` is int, that is the object that we find when following the pointer ptr is an integer. Therefore the type of ptr is pointer to

`int, or int * .`

When declaring several pointers in a comma separated list, it is necessary to place a ***** before the identifier of `each` pointer:

```
int *ptr1, *ptr2, *ptr3;
```

this declares three pointers to integers, whereas

```
int *ptr1, ptr2, ptr3;
```

declares one pointer variable and two integer variables. When using initialisers with pointer variables, it is the variable itself that is being initialiased (i.e the address the pointer holds) and not anything at the other end of the pointer:

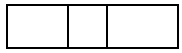
```
main(){
int i = 17;           /* ordinary int variable */
int *pi = &i; /* a pointer to an integer
      initialis ed to point to ipi now holds address of i */
*pi = 25; /* assign to what is pointed at
          by the pointer,i now contains the value 25*/
printf("%d",*pi) /* prints the value of what is in
    the address pi contains e.g 25*/
printf("%d2,pi) /* prints the contents of pi
    e.g 6582, (the address of i) */
return 0;
        }
```

5.5 Arrays

Rows and tables of storage. Arrays are a convenient way of grouping a lot of ariables under a single variable name. Arrays are like pigeon holes or hessboards, with each compartment or square acting as a storage place; they can be one dimensional, two dimensional or more dimensional! An array is defined using square brackets []. For example: an array of three integers called "triplet" would be declared like this:

```
int triplet[3];
```

Notice that there is no space between the square bracket [and the name of the array. This statement would cause space for three integers type variables to be created in memory next to each other as in the diagram below.



The number in the square brackets of the declaration is referred to as the `index' or `subscript' of the array and it must be an integer number between 0 and (in this case) 2. The three integers are called elements of the array and they are referred to in a program by writing:

```
triplet[0]

triplet[1]

triplet[2]
```

Note that the indicies start at zero and run up to one less than the number which is placed in the declaration (which is called the dimension of the array.) The reason for this will become clear later. Also notice that every element in an array is of the same type as every other. It is not (at this stage) possible to have arrays which contain many different data types. When arrays are declared inside a function, storage is allocated for them, but that storage space is not initialized: that is, the memory space contains garbage (random values). It is usually necessary, therefore, to initialize the array before the program truly begins, to prepare it for use. This usually means that all the elements in the array will be set to zero.

5.5.1 Why use arrays?

Arrays are most useful when they have a large number of elements: that is, in cases where it would be completely impractical to have a different name for every storage space in the memory. It is then highly beneficial to move over to arrays for storing information for two reasons:

- The storage spaces in arrays have indicies. These numbers can often be related to variables in a problem and so there is a logical connection to be made between an array an a program.
- In C, arrays can be initialized very easily indeed. It is far easier to initialize an array than it is to initialize twenty or so variables.

The first of these reasons is probably the most important one, as far as C is concerned, since information can be stored in other ways with equally simple initialization facilities in C.

5.5.2 Limits and the Dimension of an array

C does not do much hand holding. It is invariably up to the programmer to make sure that programs are free from errors. This is especially true with arrays. C does not complain if you try to write to elements of an array which do not exist! For example:

```
char array[5];
```

is an array with 5 elements. If you wrote:

```
array[7] = '*';
```

C would happily try to write the character ***** at the location which would have corresponded to the seventh element, had it been declared that way. Unfortunately this would probably be memory taken up by some other variable or perhaps even by the operating system. The result would be either:

- The value in the incorrect memory location would be corrupted with unpredictable consequences.
- The value would corrupt the memory and crash the program completely! On Unix systems this leads to a memory *segmentation fault*.

The second of these tends to be the result on operating systems with proper memory protection. Writing over the bounds of an array is a common source of error. Remember that the array limits run from zero to the size of the array minus one.

5.5.3 Arrays and for loops

Arrays have a natural partner in programs: the for loop. The `for` loop provides a simple way of counting through the numbers of an index in a controlled way. Consider a one dimensional array called `array`. A for loop can be used to initialize the array, so that all its elements contain zero:

```
#define SIZE  10;
main ()
{ int i, array[SIZE];
for (i = 0; i < SIZE; i++)
   {
   array[i] = 0;
   }
}
```

It could equally well be used to fill the array with different values. Consider:

```
#define SIZE  10;
main ()
{ int i, array[size];
for (i = 0; i < size; i++)
   {
   array[i] = i;
   }
}
```

This fills each successive space with the number of its index:

index	0	1	2	3	4	5	6	7	8	9
element contents	0	1	2	3	4	5	6	7	8	9

The `for` loop can be used to work on an array sequentially at any time during a program, not only when it is being initialized. The example listing below shows an example of how this might work for a one dimensional array, called an Eratosthenes sieve. This sieve is an array which is used for weeding out prime numbers, that is: numbers which cannot be divided by any number except 1 without leaving a remainder or a fraction. It works by filling an array with numbers from 0 to some maximum value in the same way that was shown above and then by going through the numbers in turn and deleting (setting equal to zero) every multiple of every number from the array. This eliminates all the numbers which could be divided by something exactly and leaves only the prime numbers at the end.

5.5.4 Arrays of More Than One Dimension

There is no limit, in principle, to the number of indicies which an array can have. (Though there is a limit to the amount of memory available for their storage.) An array of two dimensions could be declared as follows:

```
float numbers [SIZE][SIZE];
```

`SIZE` is some constant. (The sizes of the two dimensions do not have to be the same.) This is called a two dimensional array because it has two indicies, or two labels in square brackets. It has (`SIZE * SIZE`) or size-squared elements in it, which form an imaginary grid, like a chess board, in which every square is a variable or storage area.

0	1	2	3	4	5	6	7	8	… (Up to Size)
1									
2									
3									
4									
5									
6									
7									

.

.

```
(Up to size)
```

Every element in this grid needs two indicies to pin-point it. The elements are accessed by giving the coordinates of the element in the grid. For instance to set the element 2,3 to the value 12, one would write:

```
array[2][3] = 12;
```

The usual terminology for the two indicies is that the first gives the row number in the grid and that the second gives the column number in the grid. (Rows go along, columns hold up the ceiling.) An array cannot be stored in the memory as a grid: computer memory is a one dimensional thing. Arrays are therefore stored in rows. The following array:

1	2	3
4	5	6
7	8	9

Another way of saying that arrays are stored row-wise is to say that the second index varies fastest, because a two-dimensional array is always thought of as...

```
array[row][column]
```

So for every row stored, there will be lots of columns inside that row. That means the column index goes from 0..SIZE inside every row, so it is changing faster as the line of storage is followed.
A three dimensional array, like a cube or, could also be defined in the same kind of way:

```
double cube [SIZE][SIZE][SIZE];
```

or with different limits on each dimension:

```
short  notcubic[2][6][8];
```

Three dimensional arrays are stored according to the same pattern as two dimensional arrays. They are kept in computer memory as a linear sequence of variable stores and the last index is always the one which varies fastest.

5.5.5 Arrays and Nested Loops

Arrays of more than one dimension are usually handled by nested for loops. A two dimensional array might be initialized in the following way:

```
main ()
{ int i,j;
  float array[SIZE1][SIZE2];

for (i = 0; i < SIZE1; i++)
    {
    for (j = 0; j < SIZE2; j++)
       {array[i][j] = 0;}
    }
}
```

In three dimensions, three nested loops would be needed:

```
main (){
int i,j,k;
  float array[SIZE1][SIZE2][SIZE3];
for (i = 0; i < SIZE1; i++)
   {
   for (j = 0; j < SIZE2; j++)
      {
      for (k = 0; k < SIZE3; k++)
         { array[i][j][k] = 0;}
      }
   }
}
```

5.5.6 Initializing Arrays

Arrays can be initialized in two ways. The first way is by assigning every element to some value with a statement like:

```
array[2] = 42;
array[3] = 12;
```

or perhaps with the aid of one or more for loops. Because it is tedious, to say the least, not to mention uneconomical, to initialize the values of each element to as different value, C provides another method, which employs a single assignment operator = and curly braces `{ }`. *This method only works for static variables and external variables.*

Recall that arrays are stored row-wise or with the last index varying fastest. A 3 by 3 array could be initialized in the following way:

```
static int array[3][3] =
                {
                      {10,23,42},
                      {1,654,0},
                      {40652,22,0}
                };
```

The internal braces are unnecessary, but help to distinguish the rows from the columns. The same thing could be written:

```
int array[3][3] =
            {
            10,23,42,
            1,654,0
            40652,22,0
            };
```

Take care to include the semicolon at the end of the curly brace which closes the assignment.

Note that, if there are not enough elements in the curly braces to account for every single element in an array, the remaining elements will be filled out with zeros. Static variables are always guaranteed to be initialized to zero anyway, whereas auto or local variables are guaranteed to be garbage: this is because static storage is created by the compiler in the body of a program, whereas auto or local storage is created at run time.

5.5.7 Arrays and Pointers

The information about how arrays are stored was not included just for interest. There is another way of looking at arrays which follows the BCPL (Basic Combined Programming Language) idea of an array as simply a block of memory. An array can be accessed with pointers as well as with `[]` square brackets.

The name of an array variable, standing alone, is actually a pointer to the first element in the array.

For example: if an array is declared

```
float numbers[34];
```

then `numbers` is a pointer to the first floating point number in the array; `numbers` is a pointer in its own right. (In this case it is type `pointer to `float'.) So the first element of the array could be accessed by writing:

```
numbers[0] = 22.3;
```

or by writing

```
*numbers = 22.3;
```

For character arrays, which are dealt with in some depth in chapter 20, this gives an alternative way of getting at the elements in the array.

```
char arrayname[5];
char *ptr;

for (ptr = arrayname; ptr <= arrayname+4; ptr++)
   {
   *ptr = 0;
   }
```

The code above sets the array `arrayname` to zero. This method of getting at array data is not recommended by this author except in very simple computer environments. If a program is running on a normal microcomputer, then there should be few problems with this alternative method of handling arrays. On the hand, if the microcomputer is multi-tasking, or the program is running on a larger system which has a limited manager, then memory ceases to be something which can be thought of as a sequence of boxes standing next to one another. A multi-tasking system shares memory with other programs and it takes what it can find, where it can find it. The upshot of this is that it is not possible to guarantee that arrays will be stored in one simple string of memory locations: it might be scattered around in different places. So `ptr = arrayname + 5;`

might not be a pointer to the fifth character in a character array. This could be found instead using the & operator. A pointer to the fifth element can be reliably found with:

```
ptr = &(arrayname[5]);
```

5.5.8 Arrays as Parameters

What happens if we want to pass an array as a parameter? Does the program copy the entire array into local storage? The answer is no because it would be a waste of time and memory. Arrays can be passed as parameters, but only as variable ones. This is a simple matter, because the name of the array is a pointer to the array. The Game of Life program above does this. Notice from that program how the declarations for the parameters are made.

```
main ()
{
char array[23];
function (array);
.....
}
function (arrayformal)
char arrayformal[23];
{
}
```

Any function which writes to the array, passed as a parameter, will affect the original copy. Array parameters are always variable parameters

5.6 Using Structures

How does a program use the variables which are locked inside structures? The whole point about structures is that they can be used to group data into sensible packages which can then be treated as single objects. Early C compilers, some of which still exist today, placed very severe restrictions upon what a program could do with structures. Essentially, the members of a structure could be assigned values and pointers to individual structures could be found. Although this sounds highly restrictive, it did account for the most frequent uses of structures. Modern compilers allow more flexible use of structures: programs can assign one structure variable to another structure variable (provided the structures match in type); structure variables can be passed, whole, as parameters to functions and functions can return structure values. This makes structures extremely powerful data objects to have in a program. A structure is assigned to another structure by the following statements.

```
struct Personal x,y;
x = y;
```

The whole bundle of members is copied in one statement! Structures are passed as parameters in the usual way:

```
function (x,y);
```

The function then has to be declared:

```
function (x,y)
struct PersonalData x,y;
{  }
```

Finally, a function which returns a structure variable such as:

```
{ struct PersonalData x,function();
x = function();
}
```

would be declared in the following way:

```
struct PersonalData function ()
{  }
```

Notice that the return type of such a function must also be declared in the function which calls that it, in the usual way. The reader will begin to see that structure names account for a good deal of typing! The typedef statement is a very good way of reducing this burden.

The members of a structure are accessed with the `.` dot character. This is a structure *member operator*. Consider the structure variable `x`, which has the type `struct PersonalData`. The members of `x` could be assigned by the following program:

```
main ()
{
struct PersonalData x;
FillArray ("Some name", x.name);
FillArray ("Some address", x.address);
x.YearOfBirth = 1987;
x.MonthOfBirth = 2;
x.DayOfBirth = 19;
}
```

where `FillArray()` is a hypothetical function which copies the string in the first parameter to the array in the second parameter. The dot between the variable and the names which follow implies that the statements in this brief program are

talking about the members in the structure variable `x`, rather than the whole collective bundle. Members of actual structure variables are always accessed with this dot operator. The general form of a member reference is:

structure variable.member name

This applies to any type of structure variable, including those accessed by pointers. Whenever a program needs to access the members of a structure, this dot operator can be used. C provides a special member operator for pointers, however, because they are used so often in connection with structures.

5.6.1 Arrays of Structures

Just as arrays of any basic type of variable are allowed, so are arrays of a given type of structure. Although a structure contains many different types, the compiler never gets to know this information because it is hidden away inside a sealed structure capsule, so it can believe that all the elements in the array have the same type, even though that type is itself made up of lots of different types. An array would be declared in the usual way:

```
int i;
struct PersonalData x,array[size];
```

The members of the arrays would then be accessed by statements like the following examples:

```
array[i] = x;
array[i] = array[j];
array[i].YearOfBirth = 1987;
i = array[2].MonthOfBirth;
```

5.6.2 Structures of Structures

Structures are said to nest. This means that structure templates can contain other structures as members. Consider two structure types:

```
struct first_structure
   {
   int value;
   float number;
   };
```

and

```
struct second_structure
   {
   int tag;
   struct first_structure fs;
   }
x;
```

These two structures are of different types, yet the first of the two is included in the second! An instance of the second structure would be initialized by the following assignments. The structure variable name is `x`:

```
x.tag = 10;
x.fs.value = 20;
x.fs.number = 30.0;
```

Notice the way in which the member operator `.` can be used over and over again. Notice also that no parentheses are necessary, because the reference which is calculated by this operator is worked out from left to right. This nesting can, in principle, go on many times, though some compilers might place restrictions upon this nesting level. Statements such as:

```
  variable.tag1.tag2.tag3.tag4 = something;
```

are probably okay (though they do not reflect good programming). Structures should nest safely a few times.

A word of caution is in order here. There is a problem with the above scheme that has not yet been addressed. It is this: what happens if a structure contains an instance of itself? For example:

```
struct Regression
   {
   int i;
   struct Regression tag;
   }
```

There is simply no way that this kind of statement can make sense, unless the compiler's target computer has an infinite supply of memory! References to variables of this type would go on forever and an infinite amount of memory would be needed for every variable. For this one reason, it is forbidden for a structure to contain an instance of itself. What is not forbidden, however, is for a structure to contain an instance of a pointer to its own type (because a pointer is not the same type as a structure: it is merely a variable which holds the address of a structure). Pointers to structures are quite invaluable, in fact, for building data structures such as linked lists and trees.

5.6.3 Pointers to Structures

A pointer to a structure type variable is declared by a statement like:

```
struct Name *ptr;
```

`ptr` is then, formally, a pointer to a structure of type `Name` only. `ptr` can be assigned to any other pointer of similar type and it can be used to access the members of a structure. It is in the second of these actions that a new structure operator is revealed. According to the rules which have described so far, a structure member could be accessed by pointers with the following statements:

```
struct PersonalData *ptr;
(*ptr).YearOfBirth = 20;
```

This says let the member `YearOfBirth` of the structure pointed to by ptr, have the value 20. Notice that `*ptr`, by itself, means the contents of the address which is held in ptr and notice that the parentheses around this statement avoid any confusion about the precedence of these operators. There is a better way to write the above statement, however, using a new operator: `->`. This is an arrow made out of a minus sign and a greater than symbol and it is used simply as follows:

```
struct PersonalData *ptr;
ptr->YearOfBirth = 20;
```

This statement is identical in every way to the first version, but since this kind of access is required so frequently, when dealing with structures, C provides this special operator to make the operation clearer. In the statements above, it is assumed that `ptr` has been assigned to the address of some pre-assigned structure: for example, by means of a statement such as:

```
  ptr = &x;
```

where `x` is a pre-assigned structure.

5.6.4 Pre-initializing Static Structures

In the chapter on arrays it was shown how static and external type arrays could be initialized with values at compile time. Static and external structures can also be pre-assigned by the compiler so that programs can set up options and starting conditions in a convenient way.

A static variable of type `PersonDat` (as in the example programs) could be declared and initialized in the same statement:

```
#define NAMESIZE 20
#define ADDRESSSIZE 22
struct PersonDat
   {
   char *name;
   char *address;
   int YearOfBirth;
   int MonthOfBirth;
   int DayOfBirth;
   };
main ()
{
static struct PersonalData variable =
       {
       "Alice Wonderment",
       "Somewhere in Paradise",
       1965,
       5,
       12
       };
/* rest of program */
}
```

The items in the curly braces are matched to the members of the structure variable and any items which are not initialized by items in the list are filled out with zeros.

APPENDICES

Appendix I: Maths Library

These functions require double parameters and return double values unless otherwise stated. It is important to include `math.h`.

`ABS(x)`
: Return absolute (unsigned) value. (macro)

`fabs(x)`
: Return absolute (unsigned) value. (Function)

`ceil(x)`
: Rounds up a "double" variable

`floor(x)`
: Rounds down (truncates) a "double" variable.

`exp(x)`
: Find exponent

`log(x)`
: Find natural logarithm

`log10(x)`
: Find logarithm to base 10

`pow(x,y)`
: Raise x to the power y

`sqrt(x)`
: Square root

`sin(x)`
: Sine of (x in radians)

`cos(x)`
: Cosine of (x in radians)

`tan(x)`
: Tangent of (`x` in radians)

`asin(x)`
: Inverse sine of `x` in radians

`acos(x)`
: Inverse cosine of `x` in radians

`atan(x)`
: Inverse tangent of `x` in radians

`atan2(x,y)`
: Inverse tangent of `x/y` in radians

`sinh(x)`
: Hyperbolic sine

`cosh(x)`
: Hyperbolic cosine

`tanh(x)`
: Hyperbolic tangent