

## C PROGRAMMING ABBREVIATED

### Mark E. Donaldson

#### PRECEDENCE CHART

Descrip	Oper	Asso	Preced
Function expr	()	Left	Highest
Array expr	[]		
struct indirec	->		
struct memb	.		
Incr/decr	++ --	Right	
One's comp	~		
Unary not	!		
Address	&		
Dereference	*		
Cast	(type)		
Unary plus	+		
Unary minus	-		
Size in bytes	sizeof		
Multiplication	*	Left	
Division	/		
Modulus	%		
Addition	+	Left	
Subtraction	-		
Shift left	<<	Left	
Shift right	>>		
Less than	<	Left	
Less than or =	<=		
Greater than	>		
Greater thanor	>=		
Equal	==	Left	
Not equal	!=		
Bitwise AND	&	Left	
Bitwise XOR	^	Left	
Bitwise OR		Left	
Logical AND	&&	left	
Logical OR		left	
Conditional	? :	Right	
Assignment	= %= += -= *= /= >>= <<= &= ^=  =	Right	
Comma	,	Left	Lowest

#### STANDARD HEADERS

.h File	Purpose
assert.h	putting diagnostics into program
conio.h	Console functions
ctype.h	testing and modifying characters
errno.h	deports error conditions
float.h	describes floating point types
graphic.h	Graphics functions
limits.h	describes integer types
locale.h	formatting numeric values
math.h	mathematical functions
setjmp.h	bypass default () conventions
signal.h	handling signals
stdarg.h	write () with arbitrary num args
stddef.h	common definitions
stdio.h	handling input and output
stdlib.h	general utilities
string.h	handling array of characters
time.h	handling time

#### FUNDAMENTALS

##### • Lexical Issues

C is case sensitive. All reserved words are in lower case. All variables and functions must have a name. **Identifier** is the official word for name. **Identifiers** may be any

length, but most compilers limit the number of significant characters. Identifiers must start with a letter, and may contain letters, digits, and underscores. It may not be a keyword such as **int** or **while**. For example:

```
char element;
int element1;
int print_book_header;
```

##### • Primitive Data Types - Variables

The *primitive data types* are **int**, **float**, and **char**. In logical expressions, **any nonzero value is considered true, and zero is considered false**. Primitive data types may be prefaced by the words **long**, **short**, **signed**, or **unsigned**. **long** types have greater precision than **short** types. **unsigned** types can store only positive values. The default type is **int** so long is short for **long int**.

*All variables must be defined.* To define a variable in C is to request storage for a particular data type. A variable can be **initialized** (assigned a value) when it is defined:

```
int yard, foot, inch; /* defined */
char letter = 'C'; /* defined and initialized */
```

##### • Comments

Comments are enclosed by symbols **/\*** and **\*/**

##### • Program Structure

A C executable source contains preprocessor directives, variable types, and function declarations, in any order provided that an identifier is defined before it is referenced. An executable C program must contain a function called **main**. Execution begins with this function.

#### VARIABLES – CONSTANTS

##### > Declaration Syntax

A declaration statement has the form **type var, var, ...**. Initial values may be specified (initialization) in a declaration as in the statement: **int i = 1, j = 0;**

##### > Integer Constants

An integer constant may be written in decimal: **130 45 88203**. An integer constant that begins with 0 is an octal number: **0130 045**. A sequence of digits preceded by **0X** or **0x** is a hexadecimal number: **0x90A 0xf2**. *Either lowercase a through f, or uppercase A through F is acceptable.* An integer constant may be terminated by **u** or **U** to indicate that it is **unsigned**, or by **l** or **L** to indicate that it is **long**.

##### > Floating point Constants

Floating point constants consists of a string of digits (integral part) followed by a decimal point followed by a string of digits (fractional part) followed by an integer exponent. The integer exponent is **e** or **E** optionally followed by **+** or **-** followed by a string of digits. Either the integral part or fractional part, but not both, may be omitted. A floating point constant may be terminated by **f** or **F** to indicate that it is a **float**, or by **l** or **L** to indicate that it is a **long double**. If a floating

point constant is not terminated with either **f**, **F**, **l**, or **L**, it is of type **double**. Examples:

**2.0 4.3e4 9.21E-9 13.E+4 4e-3 .390**

##### • char Constants

**char** constants are enclosed in single quotes, for example:

```
c = 'L';
```

**internally this assigns the integer value 76 to c (ASCII decimal for L).** **c = 76;** externally and internally assigns the integer value of 76 to c. There is a significant difference between a digit as a character ('3') and a digit as an integer (3). For ASCII the system, the code:

```
char c, d;
c = '3';
d = 3;
```

assigns c the value decimal 51 (the ASCII value of the character 3) and d the value decimal 3.

##### > Special Character Constants

Special character **escape sequence** constants are also delimited by single quotation marks. The following are the most frequently used:

Constant	Meaning
'\a'	Bell rings
'\'	Backslash
'\b'	Backspace
'\f'	Form feed
'\n'	Newline
'\t'	Horizontal tab
'\v'	Vertical tab
'\r'	Carriage return
'\''	Single quote
'\ddd'	Octal constant
'\xhh'	Hexadecimal constant

##### > String Constants

String constants are delimited by double quotation marks. For example

```
x = "This is a string constant"
```

Within a string, C recognizes the escape sequences listed above, as well as **\"** (double quotation marks).

#### FLOW CONTROL – STATEMENTS

##### > Syntax

For all statements, the part designated as action consists of one statement without braces, or *one or more statements enclosed in braces {}*.

##### > The do while Statement

```
do {
    action
    .....
} while (expression);
```

##### > The if Statement

```
if (expression) {
    action
    .....
} or
if (expression) {
```

```

        action 1
        .....
    }
    else {
        action 2
        .....
    }
    or
    if (expression1) {
        action 1
        .....
    }
    else if (expression 2) {
        action 2
        .....
    }
    else if (expression 3) {
        action 3
        .....
    }
    else {
        action 4
        .....
    }
}

```

### > The for Statement

```

for (expr 1; expr 2; expr 3) {
    action
    .....
}

```

### > The switch Statement

```

switch (expression) {
case constant 1:
    statements 1;
case constant 2:
    statements 2;
case constant 3:
    statements 3;
default:
    statements;
}

```

The **goto** statement causes an unconditional transfer to some other part of a program. One place a **goto** statement is useful is exiting from a loop that is contained in several other loops. The **goto** statement requires a label identifier, followed by a colon **identifier**:. Example:

```

for (expression)
    goto out;
out: exit( EXIT_SUCCESS );

```

## OPERATORS

### > Operators & Precedence

See Precedence Chart.

### > Conditional Expressions

Basic syntax is: **expr 1 ? expr 2 : expr 3**;  
 Example: **x = flag ? y : y \* y**; means if flag is true, then x = y. If flag is not true, then x = y \* y.  
 Example: **x = (l > j) ? j : l**; means if l is greater than j, then x = j; else if l is not greater than j, then x = i.

### > The Cast Operator

The **cast operator** explicitly converts one data type to another data type. The data type to which the value of the original item is converted is written in parentheses to the left of the item. For example, if **x** if of type **int**, the value of the expression **( float ) x** is the original value of **x** converted to **float**. Example of use:

```
average = ( float ) hits / ( float ) at_bats;
```

### > The sizeof Operator

C measures storage in bytes, and it defines one byte to be the amount of memory required to store one character. C provides the **sizeof** operator, whose value is the amount of storage required by an object. **sizeof** is a keyword, and is not a function whose value is determined at run time. **sizeof** is an operator whose value is determined by the compiler. Syntax is: **sizeof ( object )**;. **sizeof ( char )** has a value of 1 on any system. *For the x86 machine sizeof ( int ) is typically 2 for 16 bit systems, 4 for 32 bit systems, and 8 for 64 bit systems.*

### > Bitwise Operators

Bitwise operators allow the programmer to interact directly with the hardware of a particular system. These operators and expressions also make possible a highly efficient use of storage. They work only with integral data types such as **int** and **char**. To use bitwise operators and expressions, the programmer must know several things about the underlying hardware. Assumed here is an 8-bit byte, ASCII character representation, a 16 bit cell for storing an **int**, and two's complement representation:

#### • Bitwise Complement Operator

The **bitwise complement ( or one's complement) operator ~** (tilde) changes each 1 bit in its operand to 0 and changes each 0 bit to 1. Example: If the **int** variable **x** has a value of 6, which in binary is 1111111111111001, the bitwise complement **~x** is -7 (assuming two's complement). The complement does not change 6 into -6. The operator **~** merely complements each bit.

#### • Bitwise Logical Operators

Given two bits **b<sub>1</sub>** and **b<sub>2</sub>**, we can **and** **b<sub>1</sub>** and **b<sub>2</sub>**, we can **or** **b<sub>1</sub>** and **b<sub>2</sub>**, and we can **exclusive or (xor)** **b<sub>1</sub>** and **b<sub>2</sub>**:

b <sub>1</sub>	b <sub>2</sub>	b <sub>1</sub> and b <sub>2</sub>	b <sub>1</sub> or b <sub>2</sub>	b <sub>1</sub> xor b <sub>2</sub>
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

C supports logical operations on bitwise expressions: the **and operator &**, the **or operator |**, and the **exclusive-or operator ^**. Each operator expects two operands.

#### • Bitwise Shift Operators

The **bitwise shift operators** move bits right or left. The first (or left) argument to a shift operator holds the bits to be shifted, and the second (or right) argument tells how far to shift the bits. The left shift operator is

denoted **<<**, and the right shift operator is denoted **>>**. The shift expression as a whole has the data type of the converted first operand. Example:

```

int x = 'A'; /* ASCII for 'A' is decimal 65 */
int y = 2;
x >> y

```

**shifts the bits in x right y positions.** In this case we shift the bits 00000000010000001 right two places. If the leftmost bit is 0, we always fill vacated positions on the left with zeros. As with the left shift operator, the bit's string length remains fixed. In this case, it stays at 16. Thus the value of **x >> y** is 0000000000010000.

## PREPROCESSOR DIRECTIVES

### • Preprocessor

Before a C program is compiled, it is processed by the C preprocessor. **Directives** to the preprocessor begin with **#** and should not be preceded by spaces or tabs.

### • Include Directive

The **#include filename** directive instructs the preprocessor to copy a specified file into the file being processed. If the filename is enclosed in angle brackets **<>**, the file is searched for on the system include path. If it enclosed in double quotes **""**, the file is searched for in the current directory as is normally user defined:

```

#include <stdio.h>
#include "myfile.h"

```

Header files (.h) are normally included as they contain data type and function prototype definitions.

### • Define Directive

The **#define symbol value** directive informs the preprocessor to substitute **value** for every occurrence of **symbol** in the source file:

```
#define MAX 20
```

### • Conditional Directive

Code sequences enclosed between the directives **#ifdef test** and **#endif** are compiled only if test is true. The **directives #ifndef (symbol)** and **#if !defined (symbol)** can be used to conditionally compile code in the event a specified symbol has or has not been the subject of a **#define** directive. Any **#ifdef** directive may also include a **#else** directive. The **#ifndef** directive instructs the preprocessor to define if not yet defined. The **#endif** directive marks the end of the conditional directive.

### • Miscellaneous Directives

The **#line** directive directs the compiler how to number lines and, optionally, which file name to use in reporting an error. The **#line** directive has an optional second argument, a character string, that should be the name of a file. The **#error** directive instructs the compiler to generate an error and to display a corresponding message. The **#pragma** directive gives the compiler implementation-specific instructions. One writes **#pragma** and then the name of the instruction. Example:

```
#include "mydef.h"
#line 25 "myprogram.old"
#if defined(ATT) && defined(IBM)
#error "You can't be on two computers at once!!!"
```

## FUNCTIONS

### ➤ Function Terminology

Any C function can be **invoked** or **called** by another. The **invoking** function may **pass** information to the **invoked** function, and the **invoked** function may **return** information to its invoker. By passing and returning information, *functions communicate with each other*. An invoking function passes information by passing **arguments**. In C, any expression can be an argument. In the invoking function, arguments are evaluated, and the values are passed to the invoked function. An invoked function has **parameters** that catch the information passed to it.

**Function Definition** - Every function has a **header** and a **body**. A function is **defined** by giving its header and body. A function's header consists of:

- The data type returned, or the keyword **void** if a function does not return a value. Specifying the data type is optional, but if omitted, it defaults to **int**.
- The function's name.
- In parentheses, a list of parameters and their data types and names, separated by commas, or the keyword **void** if the function has no parameters.

The right parentheses that terminates the parameter list is *not followed by a semicolon*. There is no limit to the number of parameters a function can have, except that the *number of parameters must be the same as the number of arguments in any invocation of the function*. **Definition example:**

```
char grade( int exam1, int exam2, float ave )
```

**Function Declaration** - A function's **declaration** is different from its definition, and *it is terminated with a semicolon*. A **function prototype** is the form of writing function definitions and declarations, in which the data types are included within parentheses. The function declaration occurs outside all functions and before the first function that invokes it, or inside each function that invokes it. A function declaration outside all functions serves as a declaration for all functions that follow it in the same file. In a declaration, the names that follow the data types of the parameters are optional and are ignored by the compiler. If used, these names need not be the same as the names of the parameters. **Declaration example:**

```
char grade( int mid, int fin, float wt );
```

**Function Invocation** - When a function is invoked, the arguments that are sent to it from the invoking function need not have the same names as its parameters. Data types of the invoking function are not listed in the invocation. If the invoked function is to return a value, the invoking function will typically need to be an assignment argument. A

function is invoked with zero or more arguments. The values of the arguments, which represent information passed from the invoking to the invoked function, are obtained from expressions. The arguments are enclosed in parentheses and separated by commas if there is more than one. Functions can be invoked in two different ways. If a function does not return a value, the function is invoked just by naming it. If a function returns a value, the function's name can appear anywhere a simple variable can appear. **Invocation examples:**

```
letter_grade = grade( mid_term, final, wt );
print_stars( value );
echo_line();
```

### ➤ Arguments and Parameters

Although arguments and parameters may have different names, a function's parameters should match the function's arguments in number and data type. If a function is invoked with two arguments, it should have two parameters. If the arguments are of type **int** and **char**, respectively, the first parameter should be of type **int** and the second of type **char**. When a function is invoked, all the function's arguments are evaluated before control is passed to the invoked function. C does not guarantee the order of evaluation of the arguments however.

### ➤ Call By Value

Every argument to a function is an expression which has a value. C passes an argument to an invoked function by making a copy of the expression's value, storing it in a temporary cell, and making the corresponding parameter this cell's identifier. This method of passing arguments is known as **call by value**.

### ➤ Macros

Macros may be defined with parameters, which act as placeholders for actual arguments. A parameterized macro begins with **define**. Next comes the name of the macro and then parentheses containing its parameters. The parameters are separated by commas. No whitespace is allowed between the macro name and the left parentheses. The macro name and parentheses are followed by the macro's definition. In the code that follows the macro's definition, the preprocessor substitutes each occurrence of the macro by its definition. **Example:**

```
#define print_files( e1, e2, e3 )
    printf( "\n%c\t%c\t%d", (e1), (e2), (e3) )

print_files( char1, char2, num + 1 );
```

### ➤ Recursive Functions

A recursive function is a function that invokes itself. Any C function can invoke itself. **Example:**

```
int fact( int num )
{
    if ( num <= 1 )
        return 1;
    else
```

```
        return num * fact( num - 1 );
}
```

There must be some situations in which a recursive function does not invoke itself or it would invoke itself forever. We call the values for which a recursive function does not invoke itself the **base cases**. Every recursive function must have base cases.

## ARRAYS

An array groups distinct variables of the same type under a single name. A definition of an array reserves one or more cells in memory and associates a name with the cell or cells that the programmer can use to access the cells. **Example:** `int cars[12];`

### ➤ Integer Arrays

An array can be initialized in the definition. The initial values are enclosed in braces and separated by commas. If, in a definition, fewer initial values are given than there are cells, the cells beginning with the first are initialized with the given values. After the initial values supplied are exhausted, each of the remaining cells are initialized to 0. It is an error to supply more initial values than there are cells. If we define and initialize arrays, we can omit the integers that specify the number of cells. **Examples:**

```
int id[4] = { 45, 2, 800, 81 };
int age[] = { 6, 0, 1, 7, 3 };
```

### ➤ Arrays and Pointers

An array's name is a **pointer constant**. Its value is set to the address of the first cell of the array when the array is defined and it cannot be changed thereafter. To access a specific element in an array, we can use the array's name together with an index. The array's name provides the address of its first cell, and the index (beginning from 0) provides the offset from this first cell. **Examples:** `age[0] = 6; age[4] = 3; age = &age[0].`

A **pointer variable** also holds the address of some cell, but unlike a pointer constant, a pointer variable can have its value set through an assignment operator. **Examples:** `int* ptr; ptr = age; /* points to first cell */ ptr = &age[0]; /* point to first cell */ ptr = age[2]; /* point to third cell */`

### ➤ Character Strings as Arrays of Chars

C provides the data type **char** but no data for character strings. Instead, the C programmer must represent a string as an array of characters. The array uses one cell for each character in the string and a final cell to hold the null character `'\0'`, which marks the end of the string. Three main ways to initialize:

```
char stoooge1[4];
char stoooge2[6];
char stoooge3[6] = "Larry";
stoooge1[0] = 'M';
stoooge1[1] = '0';
stoooge1[2] = 'e';
stoooge1[3] = '\0';
scanf( "%s", stoooge2 ); /* type in Curly */
```

### ➤ Arrays As Function Arguments

An array can be passed as an argument from one function to another. Two parameters are required. An array parameter **a** to catch the array passed and a parameter **index n** to catch the index of the last item in the array to be summed. Example:

```
int sum( int[a], int n ) /* function definition */
int sum[12];
x = sum( b, m );
```

### ➤ String Handling Functions

**strcat** and **strncat** are short for string concatenation. The function **strcat** expects two arguments, which should be character strings. It returns the address of the first string. **strncat** expects a third argument, which includes the maximum number of characters to include from the second string. Examples:

```
strcat( string1, string2 );
strncat( string1, string2, 5 );
```

The functions **strcmp** and **strncmp** are short for string compare. The function **strcmp** expects two arguments, compares them, and returns:

- 0 if the two are equal.
- A negative integer if the first string is **lexicographically** less than the second.
- A positive integer if the first string is **lexicographically** greater than the second.

**strncmp** is like **strcmp** except that there is a third argument which specifies the maximum number of characters to be used in the comparisons. Examples:

```
if ( strcmp( string1, string2 ) < 0 )
    strncmp( string1, string2, 5 );
```

The functions **strcpy** and **strncpy** are short for string copy. Each copies all or part of the second argument into the first argument. Each returns the address of the first argument. The function **strncpy** has a third argument which specifies the number of characters to copy. Examples:

```
strcpy( string1, string2 );
strncpy( string1, string2, 5 );
```

The function **strlen** is short for string length. This function expects the address of a string and returns the number of non-null characters up to the null terminator. Example:

```
printf( "Strings is %d\n", strlen( string ) );
```

The functions **strstr**, **strchr**, and **strrchr** are short for string in string, and string has character, respectively. All search a string for a specified component. **strstr** searches for a substring, whereas **strchr** and **strrchr** search for a single character. **strchr** searches for the first occurrence and **strrchr** for the last occurrence. They return the address of the search substring or character, or NULL otherwise. Examples:

```
strstr( "photon spin", "on sp" );
strchr( "photon spin", 'n' );
```

**memcpy** and **memmove** copy **n** bytes from one object to another; **memcmp** compares **n** bytes of one object with **n** bytes of a second object; **memchr** searches for the first occurrence of a character within the first **n** bytes of an object; **memset** copies a character into the first **n** bytes of an object. They do not check for the null terminator.

### ➤ Function to Compute string's Length

```
int length( char string[] );
{
    int count;
    for ( count = 0; string[count] != '\0'; ++count )
        ;
    return count;
}
```

### ➤ Multidimensional Arrays

An array's definition shows how many dimensions it has. An array with one pair of brackets `[]` is one dimensional, and an array with two pairs of brackets `[][]` is two dimensional. Arrays of more than one dimension are **multidimensional arrays**. The total number of cells allocated is determined by multiplying the number within the brackets: `float tolerance[10][50]` has 500 cells. The following example allocates storage for 100 rows of four columns each::

```
/* rows-jobs*/; columns-job attributes */
int job_table[100][4];
```

In functions, a *multidimensional array can be passed like a single dimensional array by giving its name as an argument*. However, in the **declaration**, we must specify the number of cells in all dimensions beyond the first. Examples:

```
void print_table( int jobs[], [4] )
    print_table( job_table );
```

### POINTERS

A variable that holds an address is called a **pointer variable**, or a **pointer**. Example:

```
int l = 10;
int *ptr;
ptr = &l; or int* ptr = &l;
```

We can access the contents of the cell whose address is stored in `ptr` by writing `*ptr`; `*ptr = 10`; Using the `*ptr` is called **dereferencing** the pointer.

A pointer can point to an array as well:

```
float on_time_rate[100];
float* ptr;
ptr = on_time_rate;
```

### ➤ Pointer Arithmetic

Using **array syntax**, an array is accessed using the array's name along with an index to access each element. With **pointer syntax**, cells in an array are accessed using a pointer. Example:

```
char letters[5];
char* ptr;
int count;
/* initialize ptr to address of letter */
ptr = letters;
/* read five characters into letters */
for ( count = 0; count < 5; ++count ){
```

```
*ptr = getchar();
++ptr;
}
/* print the characters in reverse order */
for ( count = 0; count < 5; ++count ){
    --ptr;
    putchar( *ptr );
}
```

### ➤ Array of Pointers

In general, a function that receives an array can calculate the address of any cell in the array because the parameter declaration of the array includes the address of the first cell and the cell size can be deduced from the array type. Example:

```
/* declare function with pointer array args */
void print_names( char* [p], int n );
char* cs[4]; /* define array of pointers */
print_names( cs, 4 ); /* pass pointer array */
void print_names( char* [p], int n )
```

### ➤ Pointers to Functions

A function's name, like an array's name, is a pointer constant. It is possible to define variables and to declare parameters that can contain addresses of functions (pointers to functions). One use of pointers to functions is to pass a function as an argument to another function. To define the variable `ptr` to be of type pointer to a function that has one parameter of type `char` and returns an `int`, we write: `int ( *ptr ) ( char )`; Example:

```
void sum( int ( *ptr ) ( char ) )
```

### STORAGE CLASSES

For any storage class, if both the storage class and the data type are given, the storage class must come first.

### ➤ Scope

A **block** is a section of C code bounded by braces `{}`. The body of a function is a block. A variable can be defined inside a block or outside all blocks. If the definition of a variable is contained in a block, the smallest block that contains the definition is called the **containing block**.

### ➤ auto

When we write **int num**; inside the body of a function, the variable `num` receives the default storage class **auto**. An **auto** variable must be defined inside a function's body. It is legal to specify the storage class **auto** by writing **auto int num**; but it is more common to omit the storage class and write **int num**; Storage for an **auto** variable is allocated when control enters the variable's containing block and is released when control leaves its containing block. An **auto** variable is only visible in its containing block. If an **auto** variable is simultaneously defined and initialized, the initialization is repeated each time storage is allocated. If an **auto** variable is defined but not initialized, the variable has an undefined value when control enters its containing block. Each time a function containing an **auto** is invoked, storage is allocated anew and released when the containing block is left. An **auto** cannot retain its value.

### ➤ extern

The default class for a variable defined outside a function's body is **extern**. An **extern** variable must be defined outside all function bodies. Storage for an **extern** is allocated for the life of the program. If an **extern** variable is simultaneously defined and initialized, it is initialized only once, when storage is allocated. If an **extern** variable is defined but not initialized, the system initializes it to zero once, when storage is allocated. An **extern** variable is visible in all functions that follow its definition.

When an **extern** variable is defined between block after main or other functions, it is only visible to those functions in the program past where it is defined. To extend the visibility of an **extern** variable beyond the functions that follow its definition, we must **declare** it by writing the keyword **extern** (between block), then the data type, and then the variable's name. *Declaring means writing the keyword **extern** between blocks:*

```
extern int count;
```

An **extern** variable has exactly one definition that causes storage to be allocated for the variable and, optionally, an initialization of the variable. An **extern** variable may have several declarations, each making it visible in its containing block, or if the declaration is not contained in a block, the **extern** variable is visible in all functions that follow the declaration. Thus, an **extern** variable may have only one definition, but many declarations.

**A more common reason to declare an extern variable is to make it visible in another file.**

To summarize, the storage class **extern** may be used to allow different functions to access the same variable. Such a variable must be created exactly once by defining the variable outside all blocks. Although an **extern** variable must be defined outside all blocks, there are syntactically legal ways to do so. First, the keyword **extern** must be omitted if the **extern** variable is not initialized at definition time. Second, the keyword **extern** may be either present or absent if the variable is initialized at definition time. Because the keyword **extern** is never required to define an **extern** variable, regardless of whether the **extern** variable is initialized at definition time, it is recommended that the keyword **extern** be omitted when defining **extern** variables. Once defined outside all blocks, an **extern** variable can be made visible in a block by declaring the variable in the block. If the variable is declared outside all blocks, it is visible in each of the functions that follow its declaration. Each declaration must include the keyword **extern**. Although an **extern** variable can be initialized in its definition, it cannot be initialized in any declaration.

### ➤ static

A **static** variable may be defined either inside or outside a function's body. The term **static** must be included in the definition.

Storage for a **static** variable is allocated for the life of a program. If a **static** variable is simultaneously defined and initialized, it is only once when storage is allocated. If a **static** variable is defined but not initialized, the system initializes it to zero once, when storage is allocated. A **static** variable that is defined inside a function's body is visible only in its containing block. *The **static** notation type is particularly useful for when you do not want a loop counter reset even when control of the program has left the function.*

By defining a **static** variable outside any block, we can achieve visibility in all functions that follow the definition just as if the variable were **extern**. *A **static** variable is never visible in more than one file.*

An **auto** or **static** variable that is defined inside a block is visible only in its containing block so we can have identically named yet distinct variables in different functions. A reference to an **auto** or **static** variable defined inside a block that has the same name as an **extern** variable is resolved in favor of the **auto** or **static** variable.

### ➤ register

By defining a **register** variable, the programmer is recommending to the C compiler that a CPU is to be used as the storage cell. The compiler is not required to follow the recommendation. Optimizing compilers try to use registers for loop counters and other variable that are referenced frequently. If the compiler does not heed the **register** recommendation, the storage class defaults to **auto**. A **register** variable may only be defined in a block, although the storage class **register** can be applied to the parameters of a function. Storage for **register** variables is allocated when control enters the block containing the definition and is released when control exits this block. A variable defined as **register** should have as narrow a scope as possible in order to maximize the number of registers available at any time. Because a **register** variable may be stored in a CPU register and not in memory, the address operator **&** cannot be applied.

### ➤ Nested Blocks

C allows us to restrict the visibility of a variable to part of a function by having a block **nested** within another block because a variable defined in a block is visible in its containing block. *It is not legal to nest one function's definition in another functions body.* Moreover, a defined variable may have the same name as another defined variable in the same function as long as it is nested within a block. Although the name is the same, the variable is distinct. C will resolve to the definition found in the smallest block.

### ➤ Storage Classes For Functions

A function, like a variable, has a storage class. It must either be **extern** or **static**. The default storage class for functions is **extern**, which is typically omitted from the function's definition. An **extern** variable can

be visible across functions, even functions in different files. The same is true of an **extern** function, which can be invoked by any other function, whatever its storage class, in any file. To restrict the visibility of a function, you must:

- Explicitly define the storage class as **static**.
- Place the functions that invoke it in the same file, either above or below it.
- Place the functions in which it is not to be visible in a different file from the file that contains it.

If an application requires limited visibility for one or more functions, C can satisfy the requirement with **static** functions housed in a file with only those other functions meant to invoke it.

### TYPE QUALIFIERS

Every variable has a data type (type specifier), such as **char** or **int**, and a storage class, such as **auto** or **extern**. A variable may also have one or two type qualifiers of **const** and **volatile**. A type qualifier is optional in a variable's definition or a parameter's declaration. If a type qualifier occurs in a variable's definition, it comes:

- After the storage class
- Before the data type

If a type qualifier occurs in a parameter's declaration, it also comes before the data type. Any combination of storage class, data type, and type qualifier is possible. A variable or parameter can have both type qualifiers. When type qualifiers occur together they may occur in any order. Examples:

```
volatile float mass;  
float area(const float mass );  
const volatile int mass = 3;  
volatile const double pi;
```

*Type qualifiers are only recommendations to the compiler about whether to optimize.*

### ➤ const

A **const** variable or parameter is a constant in the sense that it cannot occur as an **lvalue**:

- The left hand side of an assignment operation.
- The target of an increment or decrement operation.

A **const** variable may be initialized in its definition. A **const** array behaves as any array of **const** variables in that no variable in the array may occur as an **lvalue** after the array has been initialized in its definition.

*The type qualifier **const** can be used in two distinct ways with pointers.* First, **const** can be used to make the pointer's reference **const**. Here, the keyword **const** applies to **char\*** rather than **ptr**.

```
const char* ptr = s; /* reference is to char */  
*ptr = 'F'; /* error */  
ptr = 'F'; /* OK */
```

Second, the keyword **const** can be used to make the pointer itself **const**. Here, **const** applies to **ptr** rather than to the cell that holds the variable:

```
char* const ptr = s;
```

```
*ptr = 'F' ; /* OK */
ptr = 'F'; / error */
```

The **#define** directive can be used to define a macro constant. Seen by the preprocessor, so it is replaced before the compiler sees it. A **const** variable cannot be used to specify an array's size, and it cannot be used as **case** values in the **switch** construct.

In C, **const** has two main uses. First, **const** variables are used as substitutes for nonparameterized macros whenever the substitution is legal. Second, **const** pointer parameters are used to protect data accessible to a function that is to be read but not written. When large amounts of data are passed to a function, it is typically not done by value, but rather as a pointer to the data to save time and space. However, the protection of call by value is lost because the function that receives the address of the data can alter the data. **const** pointer parameters are thus used to obtain the efficiency of passing pointers and the protection afforded by call by value.

#### ➤ volatile

The type qualifier **volatile** indicates that a variable's storage cell might be referenced and have its value changed by something outside the program that defines the variable. For example, a routine by the operating system itself may access the storage and change the value.

### INPUT/OUTPUT

#### ➤ Files

Some functions that perform standard I/O identify the file to read or write by using a **file pointer**, which can store the address of information required to access the file. To define a file pointer:

```
FILE* fp;
FILE *fin, *fout;
fin = fopen( "infile.dat, "r" );
```

**FILE\*** is a data type like **char** or **int**. The functions **fopen** and **fclose** are used to open and close files. **fopen** expects two arguments: the name of the file given a char string, and the mode (binary add b):

Mode	If File Exists	Not Exist
"r"	open reading	Error
"w"	open new	Creates new
"a"	open append	Creates new
"r+"	open read/write	Error
"w+"	open new r/w	Creates new
"a+"	open appen r/w	Creates new
"wb"	open binary w	Creates new
"r+b"	open binary r/w	Error

When a file is opened, a **file position marker** is set to a location in the file and identifies next place to read/write (beginning or end based on mode).

#### ➤ Characters

**fgetc** expects a single argument; a pointer to file to read. **getc** same as **fgetc** but used as macro. **getchar** expects no argument, returns next char from stdin, or returns EOF.

**getchar** skips no white space and always reads the next character (unlike **scanf**), and provides common interface to utilities. **fputc** expects two arguments: a character to write and a pointer to the file to write. **putc** same as **fputc** but used as macro. **putchar** expects single argument of a char to write:

```
while ( ( c = getchar ( ) ) != EOF )
    putchar( c );
    or
*ptr = getchar;
```

Reading Characters	Examples
	FILE* fp;
	Char c;
fgetc	c = fgetc( fp );
getc	c = getc( fp );
getchar	c = getchar();
Writing Characters	Examples
	FILE* fp;
	char c;
fputc	fputc( c, fp );
putc	putc( c, fp );
putchar	putchar( c );

#### ➤ Strings

**fgets** expects three arguments: address of an array in which to store a char string, the **max** characters to store, and a pointer to a file to read. If **max** is the specified maximum number of chars to store, **fgets** reads characters from the file into the array until:

- **max** - 1 characters have been read;
- all characters up to and including the next newline character have been reached;
- EOF is reached.

**fgets** never stores more than **max** chars including '\n' and '\0'. If you format a file so each line represents a record, **fgets** can be used to read one whole record. **fputs** expects two arguments: the address of a null terminated char string and a pointer to a file:

```
fgets( record, MAX + 1, fptr);
fputs( record, stdout);
```

**gets** expects one argument, the address of an array in which to store string. It needs no file pointer. **Max** chars not specified and it reads until newline or EOF encountered. **puts** writes a string to stdout and expects only one argument; the address of the null terminated string.

```
char ans[2];
gets( ans );
puts ( ans);
```

Function	Src/dest	Newline	Returns
fgets	Any file	Reads	s& /NUL
gets	stdin	!read	s& /NUL
fputs	Any file	!append	!-/EOF
puts	stdout	appends	!-/EOF

#### ➤ Formatted Input/Output

Function	Input Source
scanf( fmat_str, ptr... )	stdin
fscanf( file_ptr, fmat_str,ptr )	A file
sscanf( array, fmat_str, ptr )	Array of char
Function	Output Dest
printf( fmat_str, arg1.... )	Stdout
fprintf( file_ptr, fmat_str,arg )	A file
sprintf( array, fmat_str, arg )	Array of char

### Scanning:

Each scanning function expects a **format string** and an **address list**, and the **items must correspond**. The **address list contains addresses of storage cells**. The **format string may contain any of the following**:

- White space which can be a combination of blanks, tabs, and newlines.
- Characters other than white space or % that must match the next non-white space characters in the input.
- A conversion code composed of characters in the following order:
  1. percent sign %.
  2. Optional assignment suppression operator \* which prevents the scanned expression from being stored in a variable.
  3. Optional positive number to specify **max field width**.
  4. Code that determines how input is converted for storage in variable.

Except for codes %[], %c, and %n, the scanning functions skip white space in input. A terminating '\0' is not added when the %c code is used but is added when the %s code is used. Each **scanning function returns EOF** if encounters end-of-file before conversion, and the **number of successful conversions stored**.

```
scanf( "%c", &charvariable); & for %d%f too.
scanf( "%s", stringvariable);
```

### Printing:

Each printing function expects a **format string** and an **argument list**. Argument list may contain any legal C expression **including function call**. **Items** in format string and argument list **must correspond**. Printing functions **return number of characters written or neg integer for error**. Format string may contain the following:

- Characters copied to output.
- Conversion code of characters in the following order:
  1. percent sign %.
  2. optional flags.
  3. optional positive number for minimum field length.
  4. optional . which separates field width from the optional positive number to specify precision.
  5. optional positive number to specify precision.
  6. code that determines how output is converted for printing.

Example	Meaning
printf( "%d", var )	Decimal output
printf( "%o", var )	Octal output
printf( "%x", var )	Hexadecimal output
printf( "%c", var )	Character output
printf( "%ld", var )	Long decimal output
printf( "%f", var )	Float output
printf( "%u", var )	Unsigned output
printf( "%7c", var )	Field width 7
printf( "%-7c", var )	Left justify width 7
printf( "%12c", var )	Field width 12
printf( "%7.3f", var )	3 decimal places
printf( "%-7.3f", var )	Left justify

To convert lower case to upper, and upper case to lower:

```
toupper( string );  
tolower( string );
```

### ➤ Unformatted Input/Output

**fwrite** writes binary data (blocks) without any formatting to a file opened in binary mode. **fread** used for reading unformatted binary data from a file. **fwrite** expects four arguments: **fwrite**( array, output\_size, output\_count, file\_ptr ); It tries to write output\_count items, each of size output\_size bytes, from array to the file pointed to by file\_ptr. It returns number of items written.

```
fwrite( vol, sizeof( float ), Size, fp );
```

writes in binary form the float array vol of size Size to the file pointed to by fp.

**fread** expect four arguments: **fread**( array, input\_size, input\_count, file\_ptr ); It tries to read input\_count items, each of size input\_size bytes, into array from the file pointed to by file\_ptr. It returns the number of items read.

```
fread( vol, sizeof( float ), Size, fp );
```

reads the binary data written by **fwrite** into float array vol.

### ➤ Moving Around In Files

Base position = **SEEK\_SET** (beginning of file), **SEEK\_CUR** (current location of file position marker), **SEEK\_END** (end of file).

**fseek**, **ftell**, and **rewind** are functions to determine or change the location of the **file position marker**. **fseek** header can be written: **int fseek**( FILE\* file\_pointer, long offset, int base\_position );

```
fseek( file_pointer, 0, SEEK_SET );
```

sets the file position marker one byte beyond the first byte. **rewind** resets the file position marker in the file specified by file\_pointer to the beginning of the file. **rewind** header can be written: **void rewind**( FILE\* file\_pointer );

```
rewind( file_pointer );
```

**ftell** returns the location of the file position marker as an offset in bytes from the first position in the file specified by file\_pointer. For binary files, **ftell** can be used to determine the number of characters entered under program control from the beginning of the file to the file position marker. For text files, **ftell** returns a value that **fseek** can use latter to move the file position marker to the position given by **ftell**. In effect, **ftell** can provide a position to which **fseek** latter can return the file position marker.

### STRUCTURES

A **structure** aggregates variables of different data types in order to represent an object such as an element. A structure must be **declared, defined, and initialized**.

**Declaration:** The declaration creates the data type and tells the system how much

storage needs to be allocated whenever an instance of the structure is defined. A **structure declaration** begins with the keyword **struct** and is followed by a **tag**, or user supplied name.

```
struct element{  
    char name[10];  
    char symbol[5];  
    float aWgt;  
    float mass;  
};
```

In the example, the **tag** is element. The **members**, name, symbol, aWgt, and mass, are enclosed in braces. It is common to **declare** tagged structures in a header file and include when needed.

#### **Definition:**

Can be defined as other variables:

```
struct element e1, e2, e3, e4, es[30];
```

Or can be combined with declaration after the ending brace:

```
} e1, e2, e3, e4, es[30];
```

#### **Initialization:**

To access a structure member, write the structure variable name, a period (**member operator**), and the member name:

```
e1.mass = 3.0;    or for an array member,  
es[6].mass += 2.0;
```

Or a structure variable can be initialized at definition time with values for members enclosed in braces:

```
e1 = { "hydrogen", "H", 3.0, 2.2 };
```

Or with **stringcopy**:

```
strcpy( e1.name, "hydrogen" );  
To initialize an array:
```

```
es[3] =  
{ { "hydrogen", "H", 3.0, 2.2 },  
  { "carbon", "C", 2.0, 2.3 },  
  { "oxygen", "O", 4.0, 2.4 } };
```

Assignment operator can be used to initialize entire structure variable or variable members of same data type:

```
struct element {  
    .....  
} e1, e2  
e2.mass = e1.mass;    or  
e2 = e1; (assuming e1 has been initialed).
```

➤ **typedef Construct:** Provides a synonym for either a built-in or user-defined data type:

```
typedef <data type> <user-provided name>;  
or typedef int element means element becomes a type int.
```

With structures:

```
typedef struct <tag> { members }  
struct element {  
    .....  
} element e1, e2, e3, e4, es[3];
```

➤ **Pointers to Structures:**

```
typedef struct element {  
    .....  
} element e1, e2, e3, e4, es[3];  
element ptr;
```

```
ptr = &e1  
ptr.mass = 2.2; or (*ptr).mass = 2.2; or  
ptr -> mass = 2.2, where -> is pointer operator. Here, e1.name, (*ptr).name, ptr -> name, *(ptr -> name), *ptr -> name are all equiv. Same if add + 1 (second byte).
```

A **self-referential structure** is a structure that includes among its members a pointer to itself: **struct e1\* ptr;**

A *structure or structure member can be passed to a function by value or by pointer:*

```
void e1( struct element );    (declare)  
e2 = e1( e1 );                (call)  
void e1( struct element )    (define) or;  
  
void e1( struct* element );  
ptr = &e1;                    (declare)  
e1( ptr );                    (call)  
void e1( struct* element )    (define)
```

### UNIONS

The storage referenced by a union variable can hold data of different types subject to the restriction that at any one time, the storage holds data of a single type. For a union variable, the system allocates an amount of storage large enough to hold its largest member. The declaration and definition of unions are syntactically the same as for structures:

```
union numbers num1;  
num1.letter = "A";  
num1.number = 5529;
```

Each new assignment cancels the previous assignment.

### BIT FIELDS

Purposes: 1) economize on storage of structure's members; 2) enable access to individual bits of storage.

```
struct example1 {  
    unsigned int field1 : 4;  
    unsigned int field2 : 8;  
    unsigned int field3 : 4;  
};
```

### ENUMERATED TYPES

Enumerated type is a data type with user-specified values. Syntax is similar to structures and unions. To declare: 1) write keyword **enum**; 2) optional identifier of the enumerated type (tag) followed by; 3) list of names, enclosed in braces and separated by commas, that are permissible values for this data type. By default, the first value is associated with 0, then 1, then 2, etc.

```
enum status { single, married, divorced };  
enum status1, status2, status3;  
status1 = married;
```

### DATA STRUCTURES

#### ➤ Storage Allocation

**malloc()** and **calloc()** are used to allocate storage at **run time** instead of **compile time**:

```
ptr = malloc( sizeof( int ) );
```

The value returned by **malloc** is copied into a variable defined to point to the type of storage allocated. **calloc** expects two

arguments of the number of storage cells to allocate and the size of them. **calloc** is used for contiguous storage cells that may be processed by taking offsets from a starting address:

```
ptr = calloc( 20, sizeof( int ) );
```

**free()** is used to release run-time storage. It expects one argument, a pointer to storage allocated by **malloc** or **calloc**:

```
free( ptr );
```

### ➤ **Linked Lists**

A linked list consists of ordered nodes:

$N_1, N_2, N_3, \dots, N_n$

Together with the address of the first node,  $N_1$ . Each node has several fields, one of which is an address field. The address field of node  $N$  contains the address of the following node in the list. The address field of the last node contains the NULL value. Functions can be written to find, add, or delete nodes within the linked list. A **self referential structure** is often used to link the nodes:

```
typedef struct element {
    char name[10];
    struct element* next;
} element;
void print_element( const element* ptr );
main()
{
    element element1, element2, *start;
    element1.next = &element2; /* link next */
    start = &element1;
    print_elements( start );
}
void print_element( const element* ptr )
element *current, *first;
current = first = malloc( sizeof( element ) );
.....
ptr = ptr -> next;
```

### ➤ **Stacks and Queues**

A **stack** is a list in which insertions and deletions occur at the same end. The end of the stack at which the insertions and deletions occur is the **top**. A stack insertion is called a **push** and a stack deletion is called a **pop**. This is last-in, first-out (**LIFO**) behavior:

```
23 87 2 10 9
pop pop
2 10 9
push 88
88 2 10 9
```

A **queue** is a list in which insertions occur at one end (**rear**) and deletions occur at the other end (**front**). This is first-in, first-out (**FIFO**) behavior. Think of line of people going into building.

### ➤ **Graphs and Trees**

**Graphs** are drawn with dots (**vertices**) and lines (**edges**). A graph consists of a set of  $V$  of vertices (nodes) and a set of  $E$  of edges (arcs) such that each edge  $e$  in  $E$  is associated with an unordered pair of distinct

vertices. For each distinct pair of vertices  $v$  and  $w$ , there is at most one edge associated with  $v$  and  $w$ . If edge  $e$  is associated with the pair of vertices  $v$  and  $w$ , we write  $e = (v, w)$ . An edge  $e$  in a graph that is associated with the pair of vertices  $v$  and  $w$  is said to be incident on  $v$  and  $w$ , and  $v$  and  $w$  are said to be adjacent vertices. When values are assigned to the edges in a graph it is called a **network** or **weighted graph**.

A **tree** is a special type graph. To **traverse** a tree is to visit each node in the tree. The order in which traversing occurs depends upon the traversing algorithm used. The three main algorithms are **preorder**, **inorder**, and **postorder**.

### **ASSERTIONS**

An **assertion** is a condition that must always be true at some point in the program's execution. Assertions, which are embedded in the code, are checked for correctness when the program is running. When the assertion fails, the system detects the error, terminates the program, and issues an error message. Assertions are also useful for checking **preconditions** (an expression that must be true at entrance to a function) and **postconditions** (an expression that must be true at exit from a function). C supports assertions through macros and the syntax is:

```
assert( condition );
```

where condition is an expression that is either true (nonzero), or false (zero). A loop invariant is an assertion in a loop, that is, a condition that must always be true at a point in the loop. Example:

```
void sel_sort( int a[], int n )
{
    int min_ind, i, j;
    for ( i = 0; i < n - 1; i++ ) {
        assert( i >= 0 );
        assert( i < n );
        min_ind = i;
        j = i + 1;
        do {
            assert( j >= 0 );
            assert( j < n );
            assert( min_ind >= 0 );
            assert( min_ind < n );
            j = i + 1;
            if ( a[j] < a[min_ind] )
                min_ind = j;
        } while ( ++j < n );
        assert( min_ind >= 0 );
        assert( min_ind < n );
        swap( &a[i], &a[min_ind] );
        assert( sorted(a, i, n ) );
    }
}
```

C makes it possible to disable assertions by defining the macro **NDEBUG(NO DEBUG)** just before the **#include <assert.h>** directive:

```
#define NDEBUG
#include <assert.h>
```

Assertions are frequently used during program development and then disabled for commercial distribution. Using assertions, it is possible to give formal definition of what it

means for a data structure to be correct, and thus, to give mathematical proof that the code is correct.

### **EXCEPTION HANDLING**

An **exception** is an unusual event that occurs during processing. For example, an exception occurs if the result of a floating point computation is too large or small to store in a floating point cell. Computer systems generate a **signal** to indicate the occurrence of an exception. When a signal is generated, the operating system responds to the signal by aborting the program or generating an error message. A C program can detect signals and then invoke its own functions, known as **exception handlers**, to respond to the exception in a user defined way. The header file **signal.h** contains macros whose values are integers that represent signals. The list of macros are as follows:

Macro	Meaning
<b>SIGABRT</b>	Abnormal termination
<b>SIGINT</b>	Interrupt
<b>SIGILL</b>	Illegal instruction
<b>DIGFPE</b>	Illegal arithmetic instruction
<b>SIGSEGV</b>	Illegal storage access
<b>SIGTERM</b>	Request program termination

The library function **signal** can be used to catch a signal. The first argument to **signal** is one of the mnemonics above and the second is the function that handles the exception. This function can be user written, or **SIG\_DFL** or **SIG\_IGN** which are defined in **signal.h**.

Macro	Meaning
<b>SIG_DFLT</b>	Terminates program when specified signal is caught
<b>SIG_IGN</b>	Ignores a specified signal

A C program that traps a signal may handle it in one of three ways:

- Terminate itself if the response to the signal is set to **DIG\_DFL**.
- Ignore the signal if the response to the signal is set to **SIG\_IGN**.
- Invoke a user-defined function that responds to the signal in its own way.

Examples:

```
/* Ignore keyboard generated interrupts */
signal( SIGINT, SIG_IGN ); or
```

```
signal( SIGINT, handler );
void handler( int sig )
/* print cautionary message */
printf( "\nPlease do not hit BREAK " );
printf( "as this terminates the program. " );
printf( "You must restart. " );
exit( EXIT_SUCCESS );
```

### ➤ **Nonlocal Jumps**

C supports nonlocal jumps, which override program control. A **nonlocal jump** occurs when an invoked function does not return to its invoker. C implements nonlocal jumps with the library functions **setjmp** and **longjmp**, each of which expects an argument **jmp\_buf** (defined in **setjmp.h**). Function **setjmp** set the jump point or the destination for the nonlocal jump. It expects

a single argument of type **jmp\_buf** and returns 0, unless the return is from a call to **longjmp**, in which case, it returns the value passed to **longjmp**. Example:

```
void jumper( void )
{
    longjmp( buf, 1 );
    ....
}
```

Long jumps may be used as a part of exception handling. Long jumps override normal program control and thereby complicate the control logic. They should be written with caution.

### GRAPHICS SUPPORT

The programmer writes to video display using smallest indivisible units that can be referenced. In **text mode** these units are characters, and in **graphics mode** these units are pixels. The standard C function library supports text mode only.

#### > Video Display

A system dependent **x, y (0, 0)** coordinate system (normally from the upper left corner of screen) is used to specify the position of a pixel on the video display. Random access to the display is provided by specifying the coordinates of a pixel. The system maintains a **current pixel position**. Relative access to the display is obtained by using the current pixel position. The largest **x** and **y** values define the resolution of the display device. A graphics system I supported by various library functions whose responsibilities include:

- Control
- Error Handling
- Drawing
- Text output
- Color
- Status

#### > Control

It is necessary to obtain a graphics driver and to put the system into graphic mode before drawing on and writing to the video display. The header file **graphics.h** must be included to provide interface to the graphics library. The function **initgraph** is called to initialize the graphics system:

```
initgraph( &driver, mode, "..." );
```

where **driver**, of type **int**, is initialized to the specific driver to use **DETECT**. The system will select an appropriate driver. The variable **mode**, also of type **int**, requests the mode in which the graphics system is to be placed unless **driver** is set to **DETECT**, in which case the graphics system is placed in the mode of highest resolution. The third argument, a C string, specifies the path for **initgraph** to follow to look for the graphics driver. If the string is the null string "", **initgraph** searches in the current directory. The function **closegraph()** frees all dynamic storage used by the graphics system and restores the video display to the mode it was in before the call to **initgraph**. Example:

```
void f( void );
{
```

```
int driver = DETECT, mode;
initgraph( &driver, &mode, "" );
closegraph();
}
```

#### > Error Handling

Error handling functions identify graphics errors and provide error messages. The function **graphresult** returns an **int** to signal whether an error occurred in the graphics operation. If no error occurred, **graphresult** returns **grOk**. If an error occurred, **graphresult** returns a value different from **grOk**. The function **grapherrormsg** returns a pointer to a string that contains a message appropriate to the error code returned by **graphresult**.

#### > Drawing

Drawing functions create and display figures, fill existing figures with shading or color, and simultaneously draw and fill figures:

```
circle( x, y, r );
line( xfrom, yfrom, xto, yto );
linere( deltax, deltay );
bar( left, top, right, bottom );
moveto( x, y ); /* changes pixel position */
```

#### > Text Output

Text output functions are provided to display text in graphics mode. They make it possible to use different fonts and to control justification of the output string relative to the current pixel position. Example:

```
outtextxy( x, y, "..." );
```

#### > Color and Status

Color functions identify current colors and to set colors, and are highly dependent on the hardware. Status functions are provided to determine the current environment, such as mode and current pixel position. **x = getmaxx()** returns the maximum legal **x** coordinate of the video display. This value gives the horizontal resolution of the display. **getmy** returns the maximum **y** coordinate of the video display, the vertical resolution.

### FUNCTIONS CHART

Math Functions	
<b>abs</b>	Absolute value of an int
<b>acos</b>	Arccosine
<b>asin</b>	Arcsine
<b>atan</b>	Arctangent
<b>atof</b>	Convert string to double
<b>atoi</b>	Convert string to int
<b>atol</b>	Convert sign to long
<b>ceil</b>	Ceiling
<b>cos</b>	Cosine
<b>cosh</b>	Hyperbolic cosine
<b>exp</b>	e <sup>x</sup>
<b>fabs</b>	Absolute value of double
<b>floor</b>	Floor
<b>labs</b>	Absolute value of a long
<b>log</b>	log <sub>e</sub> x
<b>log10</b>	log <sub>10</sub> x
<b>pow</b>	x <sup>y</sup>
<b>rand</b>	Generate a random integer
<b>sin</b>	Sine
<b>sinh</b>	Hyperbolic sine
<b>sqrt</b>	Square root
<b>srand</b>	Seed random num generator
<b>tan</b>	Tangent

<b>tanh</b>	Hyperbolic tangent
Memory Allocation Functions	
<b>calloc</b>	Allocate storage
<b>free</b>	Free storage
<b>malloc</b>	Allocate storage
Input/Output Functions	
<b>fclose</b>	Close a file
<b>feof</b>	Check end-of-file
<b>fgetc</b>	Read a character from file
<b>fgets</b>	Read a string from file
<b>fopen</b>	Open a file
<b>fprintf</b>	Write formatted output to file
<b>fputc</b>	Write a character to file
<b>fread</b>	Read several items
<b>fscanf</b>	Read formatted input
<b>fseek</b>	Move within a file
<b>ftell</b>	Find position within a file
<b>fwrite</b>	Write several items
<b>getc</b>	Read a character
<b>getch</b>	Read a character
<b>getchar</b>	Read a character
<b>gets</b>	Read a string
<b>printf</b>	Write formatted output
<b>putc</b>	Write a character
<b>putchar</b>	Write a character
<b>puts</b>	Write a string
<b>rewind</b>	Move to beginning of file
<b>scanf</b>	Read formatted input
<b>sprintf</b>	Write formatted output
<b>sscanf</b>	Read formatted input
<b>ungetc</b>	Return character to buffer
Type and Conversion Functions	
<b>atof</b>	Convert string to double
<b>atoi</b>	Convert string to int
<b>atol</b>	Convert string to long
<b>isalnum</b>	Alphanumeric?
<b>isalpha</b>	Alphanumeric character?
<b>iscntrl</b>	Control character?
<b>isdigit</b>	Decimal digit?
<b>isgraph</b>	Nonblank, printable character
<b>islower</b>	Lowercase character?
<b>isprint</b>	Printable character?
<b>ispunct</b>	Punctuation character?
<b>isspace</b>	Space character?
<b>isupper</b>	Uppercase character?
<b>isxdigit</b>	Hexadecimal character?
<b>tolower</b>	Conv uppercase to lowercase
<b>toupper</b>	Conv lowercase to uppercase
String Functions	
<b>memchr</b>	Find leftmost char in object
<b>memcmp</b>	Compare objects
<b>memcpy</b>	Copy object
<b>memmove</b>	Copy object
<b>strcat</b>	Concatenate strings
<b>strchr</b>	Find leftmost char in string
<b>strcmp</b>	Compare strings
<b>strcpy</b>	Copy strings
<b>strcspn</b>	Complement span
<b>strlen</b>	Length of string
<b>strncat</b>	Concatenate strings
<b>strncmp</b>	Compare strings
<b>strpbrk</b>	Find break character
<b>strrchr</b>	Find rightmost char in string
<b>strspn</b>	Span
<b>strstr</b>	Find substring
Miscellaneous Functions	
<b>bsearch</b>	Binary search
<b>clearerr</b>	Clear EOF & error indicators
<b>difftime</b>	Compute time differences
<b>exit</b>	Terminate program
<b>longjmp</b>	Restore environment

<b>lsearch</b>	Linear search
<b>qsort</b>	Quicksort
<b>setjmp</b>	Set jump
<b>signal</b>	Invoke func to handle signal
<b>system</b>	Execute a command
<b>time</b>	Find time
<b>Nonstandard Borland Graphics Functions</b>	
<b>bar</b>	Draw and fill rectangle
<b>circle</b>	Draw circle
<b>closegrap</b>	Close graphics system
<b>getmaxx</b>	Get max x-coord on screen
<b>getmaxy</b>	Get max y-coord on screen
<b>graphresu</b>	Describe last graphics error
<b>grapheror</b>	Give error message
<b>initgraph</b>	Initialize graphics system
<b>line</b>	Draw line (absolute coord)
<b>linere1</b>	Draw line (relative coord)
<b>moveto</b>	Change current pixel position
<b>outtextxy</b>	Write string