Introduction to C Programming Functions

Motivation and Benefits

- Very often in computer programs there is some code that must be executed multiple times in different places in the program.
- It is also common to need the same code in multiple different programs.
- Encapsulating frequently-used code into functions makes it easy to re-use the code in different places and/or different programs.
- Separating code out into functions also allows the programmer to concentrate on different parts of the overall problem independently of the others.
    - In "top-down" programming a programmer first works on the overall algorithm of the problem, and just assumes that functions will be available to handle individual details.
    - Functions can then be concentrated on independently of the greater context in which they will be used.
- Well-written functions should be general enough to be used in a wide range of contexts, not specific to a particular problem.
- Functions can be stored in libraries for later re-use. Examples of functions we have used include log( ), sqrt( ), abs( ), cos( ), etc.

Syntax and Components

- The general syntax of a function is as follows:

```
return_type function_name( arg_type argument, ... )  {
         local_variable_type  local_variables;
         executable  statement(s);
         return  return_value;
     }
```

- Example:

```
int add( int a, int b ) {
         int sum;
         sum = a +  b;
         return  sum;
     }
```

Return Type

- The "return type" indicates what kind of data this function will return. In the example above,the function returns an int.

- Some languages differentiate between "subroutines", which do not return a value, and "functions", which do. In C there are no subroutines, only functions, but functions are not required to return a value. The correct way to indicate that a function does not return a value is to use the return type "void". (This is a way of explicitly saying that the function returns nothing.)
- If no return type is given, the compiler will normally assume the function returns an int. This can cause problems if the function does not in fact return an int.

Function Name

- The function name is an identifier by which this function will be known, and obeys the same naming rules as applied to variable names (Alphanumeric characters, beginning with alpha, maximum 31 significant characters, etc.)

Formal Parameter List

- Following the function name are a pair of parentheses containing a list of the formal parameters, (arguments) which receive the data passed to the function.
- The ANSI standard requires that the type of each formal parameter to be listed individually within the parentheses as shown in the above example. Even if several parameters are of the same type, each must have its type given explicitly.
- If a function takes no parameters, the parameters may be left empty. The compiler will not perform any type checking on function calls in this case. A better approach is to include the keyword "void" within the parentheses, to explicitly state that the function takes no parameters.

Function Body

- The body of the function is enclosed within curly {} braces, just as the "main" function with which we have been dealing so far, and contains the instructions that will be executed when this function is called.
- The function body starts by declaring all local variables, prior to any executable statements.
  - In C99 variables can be declared any time before they are used, but in general it is still best to declare all variables that will be used at the beginning of the function, or at the beginning of the block in which they are used for very local variables. (See scope below.)
  - There are actually two schools of thought on this issue:
    1. Declaring all variables at the top of the function puts them all together, in one comprehensive list. If you print out the program, then all the variables will print on the same page, making a sort of "checklist" of variables to be taken care of.
    2. If you only work on the computer, and never from printouts, it can be difficult to continuously scroll back and forth between the variable declarations at the beginning of the function and the part of the program you are working on.

Declaring the variables just before you use them keeps the declaration and use on the same screen without scrolling.

3. (As a compromise, you can declare the variables just before use while working on the program, and then move them all up to the top of the function after the program is running

The Return Statement

- The **return** statement exits the called function and returns control back to the calling function.
  - Once a return statement is executed, no further instructions within the function are executed.
- A single return value (of the appropriate type) may be returned.
  - Parentheses are allowed but not required around the return value.
  - A function with a void return type will not have a return value after the return statement.
- More than one return statement may appear in a function, but only one will ever be executed by any given function call.
  - (All returns other than the last need to be controlled by logic such as "if" blocks.)
- If a function does not contain a return statement, most compilers will add one automatically at the end of the routine, and may generate a warning message. The return value, if any, is undefined in this case.
- "main( )" is technically a function, and should return 0 upon successful completion, or a non-zero value otherwise. This is ignored by many programmers, but some compilers will issue warning messages if main() does not contain a return statement.

Function Prototypes (a.k.a. Function Declarations)

- When the compiler processes a call to a function, it will check to see that the correct number and types of data items are being passed to the function, and will automatically generate type conversions as necessary. This is known as type checking.
- Type checking is only possible if the compiler already knows about the function, including what kind of data the function is expecting to receive. Otherwise, the compiler has to make assumptions, which can lead to incorrect and erratic behavior if those assumptions are not correct.
- One way of dealing with this situation is to make certain that all functions appear earlier in a file than any calls to them. This works for simple cases, but can make large complex programs hard to follow, and does not work in the cases of (mutually) recursive functions and functions located in separate files or libraries.
- A better approach is to use **function prototypes**. This is a way of declaring to the compiler what data a function will require, without actually providing the function itself.

- Examples:

-
  ```
          int add( int a, int b );
    int  add( int, int );
  ```

- Note that the function prototypes end with semicolons, indicating that this is not a function, but merely a prototype of a function to be provided elsewhere.
- Note also that the variable names are not required in the function prototype. They may be included for clarity if you wish, but the compiler will ignore them. This also means that the variable names used in the function prototype do not need to match those used in the actual function itself.
- For clarity it is generally good style to list all functions that will be used by prototypes at the beginning of the file. Then provide main( ) as the first full function definition, followed by each of the other functions in the order in which the prototypes are listed. (I.e. the prototype list acts as a kind of table of contents for the actual function which appear after main.)
- Function prototypes are often placed in separate header files, which are then included in the routines which need them. For example, "math.h" includes the function prototypes for the C math functions sqrt( ) and cos( ).
- Exercise: Write function prototypes for:

1. A function which takes an int and a float, and returns a double.
   - Answer: `double myfunction( int, float );`
2. A function which takes no arguments and returns no value.
   - Answer: `void yourfunction( void );`

Operation

Calling a Function

- A function is called by using the function name, followed by a set of parentheses containing the data to be passed to the function.
- The data passed to the function are referred to as the "actual" parameters. The variables within the function which receive the passed data are referred to as the "formal" parameters.
- The formal parameters are local variables, which exist during the execution of the function only, and are only known by the called function. They are initialized when the function starts by **copies of** the data passed as actual parameters. This mechanism, known as "pass by value", ensures that the called function can not directly change the values of the calling functions variables. (Exceptions to this will be discussed later.)
- To call a function which takes no arguments, use an empty pair of parentheses.
- Example: total = add(5, 3);
- **VERY IMPORTANT:** It is crucial that the number and types of actual parameters passed match with the number and types of parameters expected by the functions formal parameter list. (If the number of arguments matches but the data

types do not, then the compiler MAY insert some type conversion code if the correct data types are known, but it is safer not to rely on this.)

- **NOTE CAREFULLY:** The actual parameters passed by the calling program and the formal parameters used to receive the values in the called function will often have the same variable names. However it is very important to recognize that they are totally different independent variables (because they exist in different ***scopes***, see below), whether they happen to have the same name or not.
  - Example: In the code below, the variables x, y, z, and x again in main are used to initialize the variables x, z, b, and c in the function. The fact that x and z appear in both main and the function is irrelevant - They are independent variables with independent values.

```
o           void func( double x, double z, double b, double c );
o
o           int main( void ) {
o
o               double x( 1.0 ), y( 2.0 ), z( 3.0 );
o
          func( x, y, z, x );
```

  - The call to the function above initializes the function parameters equivalently to the following **assignment** statements:
    - x in func = x in main
    - z in func = y in main
    - b in func = z in main
    - c in func = x in main

Returning from a Function

- A function may return a single value by means of the return statement.
- Any variable changes made within a function are local to that function. A calling function's variables are not affected by the actions of a called function. (When using the normal pass-by-value passing mechanism for non-arrays. See also the section on alternate passing mechanisms below.)
- The calling function may choose to ignore the value returned by the called function. For example, both printf and scanf return values which are usually ignored.
- The value returned by a function may be used in a more complex expression, or it may be assigned to a variable.
- Note carefully that in the sample program shown below, it is the **assignment statement** which changes the value of y in the main program, **not** the function.

Sample Program Calling a Function

```
        int add_two( int x ); // Adds 2 to its argument and returns the
result
```

```
         main() { // <---------------------------------------------------------
---- int main( void )

             int y = 5;

             cout << "Before calling any function, y = " << y << endl;

             add_two( y );

             cout <<  "After calling the function once, y = " << y << endl;

             y = add_two( y );

             "After calling the function twice, y = " << y << endl;

             return 0;

         } // main

         int add_two( int x ) { // <-----------------------------------------
int add_two( int )

             cout <<  "In function, x changed from " <<  x;

             x += 2;

             cout <<  " to " <<  x << endl;

             return x;

         } // add_two
```

Output:

```
         Before calling any function, y = 5
         In function, x changed from 5 to 7
         After calling the function once, y = 5
         In function, x changed from 5 to 7
         After calling the function twice, y = 7
```

Data Passing Mechanisms

Pass By Value

- Ordinary data types (ints, floats, doubles, chars, etc) are ***passed by value*** in C/C++, which means that only the numerical value is passed to the function, and used to initialize the values of the functions formal parameters.
- Under the pass-by-value mechanism, the parameter variables within a function receive a ***copy of*** the variables (data) passed to them.
- Any changes made to the variables within a function are local to that function only, and do not affect the variables in main (or whatever other function called the current function.) This is true whether the variables have the same name in both functions or whether the names are different.

Passing Arrays and/or Array Elements

- When one element of an array is passed to a function, it is passed in the same manner as the type of data contained in the array. (I.e. pass-by-value for basic types.)
- However when the entire array is passed, it is effectively passed by reference. (Actually by pointer/address, to be explained completely in the section on pointer variables.)
- See "Passing Arrays to Functions" below.

(Pass by Pointer / Address)

- Pass by pointer / address is another type of data passing that will be covered later under the section on pointer variables.
- (It is often mistakenly termed pass-by-reference in some C textbooks, because the actual pass-by-reference passing mechanism is only available in C++.)
- Pass by pointer / address requires use of the address operator (&) and the pointer dereference operator (*), to be covered later.

(Pass by Reference)

- C++ introduced a new passing mechanism, ***pass by reference***, that is not available in ordinary C, and will not be discussed further in these notes.

Passing Arrays to Functions

- Recall that ordinary data types (ints, floats, doubles, etc.) are passed to functions ***by value***, in which the function receives a copy and all changes are local.
- If an individual element of an array is passed to a function, it is passed according to its underlying data type.
  - So if nums was declared as a one-dimensional array of ints, then passing nums[i] to a function would behave the exact way as passing any other int, i.e. pass-by-value.
- When an entire array is passed to a function, however, it is effectively passed ***by reference***.
  - (It is actually passed by pointer/address, to be covered later, but effectively it is pass-by-reference.)
  - The net result, is that when an entire array is passed to a function, and the function changes variables stored in that array, it **does** affect the data in the calling function's array.
  - Because arrays are passed by reference, there is generally no need for a function to "return" an array. - It merely needs to fill in an array provided by the calling function. (It is possible for functions to return arrays but it requires the use of

pointers and addresses, and frequently dynamic memory allocation, none of which we are ready for yet.)

- In order to ***prevent*** the function from changing the array values, the array parameter can be modified with the keyword ***const.***
- E.g. "`void printArray( const int data[ ], int nValues );`"

- When an entire array is passed to a function, the size of the array is usually passed as an additional argument.
- For a one dimensional array, the function's formal parameter list does not need to specify the dimension of the array. If such a dimension is provided, the compiler will ignore it.
- For a multi-dimensional array, all dimensions except the first must be provided in a functions formal parameter list. The first dimension is optional, and will be ignored by the compiler.
- **[Advanced:** A partially qualified multi-dimensional array may be passed to a function, and will be treated as an array of lower dimension. For example, a single row of a two dimensional array may be passed to a function which is expecting a one dimensional array. (Question: Why is it possible to pass a row this way, but not a column?)]
- **Examples:** Note the use of arrays and functions in the following sample program. Note that in the calculation of max4, we have passed a two dimensional array containing two rows of three elements as if it were a single dimensional array of six elements. This is cheating, but it happens to work because of the way that the rows of a multidimensional array are stored.

```
/* Program Illustrating the use of Arrays and Functions */

#include <stdlib.h>
#include <stdio.h>

// Finds max in the array
double maxArray( const float numbers[ ], int arraySize );

int main( void ) {

    double array1[ ] = { 10.0, 20.0, 100.0, 0.001 };
    double array2[ 2 ][ 3 ] = { { 5.0, 10.0, 20.0 },
                                { 8.0, 15.0, 42.0 } };

    int sizes[ 2 ] = { 4, 3 };
    double max1, max2, max3, max4, max5;

    max1 = maxArray( array1, 4 );
    max2 = maxArray( array1, sizes[ 0 ] );
    max3 = maxArray( array2[ 1 ], 3 );     // Advanced
    max4 = maxArray( array2[ 0 ], 6 );     // Very Advanced
    max5 = maxArray( array1, -4 ); // Generates an error - returns
0.0;

   printf( "Maximums are %f, %f, %f, %f, and %f\n", max1, max2,
max3, max4, max5 );
```

```
            return 0;

        }

        double maxArray( const double numbers[ ], int arraySize ) {

            /* Function to find the maximum in an array of doubles
               Note the use of the keyword "const" to prevent changing array
data */

            int i;
            double max;

            if( arraySize <= 0 ) {
                return 0.0;
            }

            max = numbers[ 0 ];

            for( i = 1; i < arraySize; i++ )
                max = ( numbers[ i ] > max ) ? numbers[ i ] : max;

            return max;

        }
```

- **New:** In C99 it is possible to declare arrays using variable dimensions, providing the variable has a (positive integer) value at the time the declaration is made. It turns out that this carries over to the declaration of arrays in function parameters as well, which can be particularly useful for multi-dimensional arrays.
- For example, in the prototype:

```
int arrayFunction( int nRows, int nCols, double x[ nRows ],
                   double y[ nRows ][ nCols ] );
```

the variable dimension on x is informative to the human but not necessary for the computer, since we could have declared it as x[ ]. However the nCols dimension on y is very useful, because otherwise the function would have to be written for arrays with pre-determined row sizes, and now we can write a function that will work for arrays with any row length.

Using Arrays to Return Multiple Values from Functions

- Based on what we have learned so far, functions can only return a single value, using the "return" mechanism.
- One way around that limitation is to pass an array to the function, and to let the function fill in the array.
  - The size of the array could range from a single element to as many values as you want the function to return.
  - The difficulty is that all the array elements must be the same type, and do not get separate names.

        - For example, some of the earlier code could be improved to return an error code as well as a result:
        - The maxarray function shown above can now be improved as shown below. Programs using the improved version of maxArray should check the value of errorCode[0] before using the results.
    - Eventually we will learn how to get around this limitation using pointer / address argument passing, and/or structs.
        - (Global variables, discussed below, can also technically get around this limitation, but there are very good reasons for avoiding global variables at all times unless absolutely necessary.)
    - Improved maxarray code:

```
        double maxArray( const double numbers[ ], int arraySize, int errorCode[ ] ) {

            /* Function to find the maximum in an array of doubles
               Note the use of the keyword "const" to prevent changing array data */

            int i;
            double max;

            if( arraySize <= 0 ) {
                errorCode[ 0 ] = -1;  // Errors found in input.  Results invalid.
                return 0.0;
            }

            errorCode[ 0 ] = 0;  // No errors in input

            max = numbers[ 0 ];

            for( i = 1; i < arraySize; i++ )
                max = ( numbers[ i ] > max ) ? numbers[ i ] : max;

            return max;

        }
```

- ## Exercises

Write the following functions:

- double average(const double x[ ], int nValues);
    - Calculates the average of the values in x.
    - nValues is how many elements to calculate
    - Function should return 0.0 if errors are encountered
- double dot(const double x[ ], const double y[ ], int nValues);
    - Calculates the dot product of x and y.
    - nValues is how many elements to calculate
    - Function should return 0.0 if errors are encountered

- int calcSins(const double x[ ], double sinX[ ], int nValues);
  - Calculates the sin of each element of x, and stores the results in sinX.
  - nValues is how many elements to calculate
  - The return value is the actual number of values calculated. It should be equal to nValues if all goes well, or some other value (e.g. zero) if there is a problem.

Recursion

- Any function in C may call any other function, ***including itself!***
- When a function calls itself, it is called ***recursion.***
- It is also possible for a set of functions to be ***circularly recursive,*** in which a function does not call itself directly, but does so indirectly through some other function(s). (E.g. function A calls function B which calls function C which calls function A.)
- A very important issue when writing recursive functions is that there be a defined stopping condition that will stop the recursion and prevent it from calling itself forever in an infinitely recursive loop, and that there be a guarantee that once the function is called, it will eventually reach that stopping condition.
- The classic example of recursion is the calculation of factorials.
  - The definition of the factorial of X, denoted as X! , is the product of X * (X - 1) * (X - 2) * . . . * 1.
  - X! can also be defined recursively as:
    - X! = 1 for all X less than or equal to 1
    - X! = X * (X - 1)! for all X greater than 1
  - This can be programmed as:

```
o       long int factorial( int X ) {
o
o               if( X <= 1 )
o               return 1;
o
o            return X * factorial( X - 1 );
        }
```

Character Strings as Arrays of Characters

- The traditional method for handling character strings in C is to use an array of characters.
- A null byte (character constant zero, '\0') is used as a terminator signal to mark the end of the string.
- Ordinary quoted strings ("Please enter a number > ") are stored as null-terminated arrays of characters.
- The string library contains a number of functions for dealing with traditional arrays of characters.
  - (#include <string.h>)

Variable Scope

- Variable scope refers to the range (the scope) in which a variable is defined and in which it exists.
- There are four variable scopes: ordinary local variables, function parameters, global variables, and very local variables.

Ordinary Local Variables

- Most of the variables that we have used so far are ordinary local variables.
- Ordinary local variables are declared inside of a function, and outside of any braced blocks such as while or for loops.
- The scope of these variables (the range in which they exist and are valid) is from the point of their declaration to the end of the function.
- As a general rule, all ordinary variables are normally declared at the very beginning of a function.
    - This was required in traditional C, but is merely a programming style in C99.
    - Programmers who work from printouts, and/or who have extensive programming experience in C, tend to define all their variables at the beginning of each function. This has the advantage, especially when working from printouts, of listing all variables in one place, as a sort of comprehensive inventory or checklist.
    - Programmers who work mostly or exclusively on the screen, particularly those who learned to program originally in C++ and not C, may prefer to declare their variables as close to where they are first used as possible. This has the advantage of keeping variable declarations and their use on the same screen, or at least within a short distance of each other. There is also some argument for improving compiler optimization performance and for avoiding conflicts with other variables having the same names. (See "very local variables" and "variable eclipsing" below.)
    - For this class you may use either style that you wish, provided you are consistent.
- Ordinary local variables are not initialized automatically. They must be initialized by the programmer, or else they will start out with unknown random values. (Possibly zeros in some cases, but you can't count on that.)

Function Parameters

- Function parameters have the same scope as ordinary local variables.
- The only difference is that function parameters get initialized with the values passed to them by the calling function.

Global Variables

- Global variables are declared outside of any function, ordinarily at the very beginning of the file.
- The scope of global variables is from the point of declaration down to the end of the file.
  - (Global variables can be made available to functions in other files also, but that is beyond the scope of these notes.)
- Global variables are accessible to all functions within a file (beyond the point of declaration), without having to be passed.
- Global variables introduce many opportunities for very hard-to-find bugs, as any function can change them and it can often be very difficult to figure out how a global variable is being changed.
- Some systems may initialize globals to zero (see static storage below), but you should not count on that.
- **Global variables should be avoided whenever possible. Beginning programmers should not use global variables.**
  - Globals are most often required when using callback functions, not covered in this course.
  - The introduction of Exceptions in C++ (also not covered in this course) has eliminated much of the former need for global variables.

Very Local Variables

- Variables declared within a { braced block }, such as a loop, if, or switch construct are termed ***very local variables.***
- The scope of very local variables is from the point of declaration to the end of the block in which they are declared.
  - (They should normally be declared at the beginning of the block.)
- Programmers sometimes declare very local variables for temporary variables that are only needed within a particular block.
  - This is usually done to avoid name conflict problems with other variables having the same name declared in a more general scope. (See eclipsing below.)
- Very local variables can make code hard to read and understand, and should only be used when there is a very good reason to do so.
- One common use of very local variables is in the code fragment:

```
for( int i = 0; i < limit; i++ ) {
```

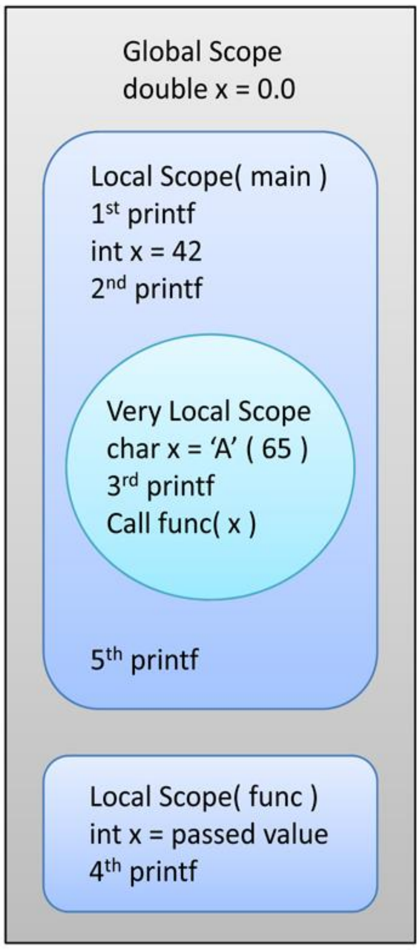
- In this case the loop counter i exists within the body of the loop, and ceases to exist when the loop exits.
- There will be no conflict between this variable i and any other declared at a more general scope. (See below.)
- If you choose the variable declaration style of declaring all ordinary local variables at the beginning of the function, then any very local variables

you declare should be declared at the beginning of the block in which they are defined.

Variable Eclipsing

- If the same variable name is used in multiple scopes (e.g. global, local, very local), **and** the scopes overlap,then in the regions of overlap the more specific variable will ***eclipse,*** or hide the more general variable(s).
- When the more specific variable goes out of scope, then the next more general variable becomes visible again, unchanged by whatever may have occurred while it was eclipsed.
- Example. In the code below the comments indicate which X will be printed at which locations.

```
void func( int );

double x( 0.0 ); // Global scope

int main( void ) {

    printf( "First x = %f\n", x ); // prints the global, 0.0

    int x = 42;   // Ordinary local

    printf( "Second x = %d\n", x ); // prints the ordinary local, 42

    if( x > 40 ) {

        char x( 'A' ); // Very local, value = 65

        printf( "Third x = %c\n", x ); // Prints very local, 'A'

        func( x ); // Passes the very local char, converted to an int.

    }
    printf( "Fifth  x = %d\n", x ); // Ordinary local 42 again

    return 0;
}

void func( int x ) { // local parameter

 printf( "Fourth x = %d\n", x ); // Local parameter, 65

    return;
}
```



Program Output:

```
First x = 0.0
Second x = 42
Third x = A
Fourth x = 65
Fifth x = 42
```

Storage Class

- Variables can be in one of four storage classes, depending on where in computer memory they are stored.

Automatic Variables

- Automatic variables, (a.k.a. auto variables) are stored on a data structure known as "the stack".
- The stack grows and shrinks as a program executes.
- In particular, when a new function is entered, space is allocated on the stack to store all of the local variables for that function. (Actually space is allocated for each variable at the time when it first goes into scope, i.e. when it is declared.)

- More importantly, when the function exits, the stack space allocated to that function is freed up, and becomes available for other uses. (The stack shrinks.)
- Local variables, function parameters, and very local variables are ordinarily auto variables stored on the stack.
- Any data stored in auto variables is lost when the variables go out of scope, i.e. when a function exits. The next time the variable comes back into scope (i.e. when the function gets called again), the variable is allocated new space, and re-initialized if an initialization is given.

Static Variables

- Static variables are stored in a separate storage area known as "the heap".
- Space for static variables is allocated one time only, before main( ) begins, and never expires.
- Global variables are normally static. Other variables may be declared static.
- In particular, if function variables are declared as "static", then they are only initialized once, and retain their values between function calls. (The variables can still go out of scope, but when they come back into scope they will still retain their previous values.)
- **Example:** The following code will generate the output shown below the code:

```
void staticExampleFunction( void );

int main( void ) {

    for( int i = 0; i < 5; i++ )
         staticExampleFunction( );

    return 0;

} // main

void staticExampleFunction( void ) {

    int normalInt = 0;
    static int staticInt = 0;

    printf( "The normal int = %d.  The static int = %d.\n",
++normalInt, ++staticInt );

    return;
}
```

Output:

```
The normal int = 1.  The static int = 1.
The normal int = 1.  The static int = 2.
The normal int = 1.  The static int = 3.
The normal int = 1.  The static int = 4.
The normal int = 1.  The static int = 5.
```

- Static variables can be used to count how many times a function is called, or to perform some special behavior the first time a function is called. (Declare a static variable "firstTime" initialized to true. If firstTime is true, do the special code and then set firstTime to false.)
- **Variation:** The static keyword applied to a global variable makes it global to this file only, and not visible from other files, (in a multi-file development project.)

Extern Variables

- The "extern" keyword applied to a variable indicates that it is declared and allocated space in some other file.
- A declaration with the word "extern" is like a function prototype - It tells the compiler of the existence of the variable, without actually creating it or allocating any space for it.
- All such variables must be declared exactly once, (i.e. in one file only of a multi-file development project) without the "extern", so that space can be allocated for it.
- Exterm variables are global in the file in which they are declared without "extern", but may be either local or global in other files.
- Extern will be covered more fully under the topic of multi-file development.

Register Variables

- The keyword "register" suggests to the compiler that the given variable be stored in one of the CPU ***registers***, (for faster access), instead of in regular memory.
- Register variables act as auto variables, except they do not have an "address", and so cannot be referred to by pointer variables or by the address operator, &.
- Loop counters are the most common and obvious use of register variables.
- Modern optimizing compilers have elminated most need for the keyword register.

Summary of Variable and Parameter Declaration Qualifiers

The following example shows all possible qualifiers for variables and function parameters, and how those qualifiers affect the variables in three key areas:

33. Storage duration, indicating whether the item continues to exist when it goes out of scope (static storage), or whether it is re-created and re-initialized every time that it goes into scope (auto storage.)
34. Scope, indicating whether the item is available to the remainder of the file (file scope), or only through the remainder of the block in which it is defined (block scope.)
35. Linkage, indicating whether the item is also accessible from other files (external linkage) or whether it is private to multiple functions only within this file (internal linkage), or whether it is accessible only within a block (no linkage, i.e. none.)

```
int a;
extern int b;
static int c;

void f( int d, register int e ) {
     auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}
```

Name	Storage Duration	Scope	Linkage
a	static	file	external
b	static	file	??? - see below
c	static	file	internal
d	auto	block	none
e	auto	block	none
f	auto	block	none
g	auto	block	none
h	auto	block	none
i	static	block	none
j	static	block	??? - see below
k	auto	block	none

??? - The linkage for b and j depends on their original declaration, but are normally external

Inline Functions (C99 only)

There is also one additional qualifier that can be applied to functions only: ***inline***

The ordinary function call-and-return mechanism involves a certain amount of overhead, to save the state of the original function on the stack, create stack space for a return address and the local (auto) variables needed for the new function and its parameters, transfer control to the new function, do the work, store the return value back on the stack, clean up the stack, and then transfer control back to the original calling function.

For certain small fast functions, this overhead can add significantly to the processing time of the function, often greatly surpassing the effort needed to perform the work of the function.

Therefore the ***inline*** qualifier applied to a function suggests to the compiler to simply copy the instructions for the function into this location in the calling function's code, instead of invoking all of the overhead of the call and return process.

An alternative to inlined functions is parameterized macros, implemented with #define, which are covered elsewhere.