1BEIT105/205 - Programming in C

Module-4

Chapter 6: Functions: ***The General Form of a Function, Understanding the Scope of a Function, Function Arguments, argc and argv—Arguments to main(), The return Statement, What Does main() Return?, Recursion, Function Prototypes, Declaring Variable Length Parameter Declarations, The inline Keyword.***
Chapter 5: Pointers (Contd…): ***Pointers to Functions, C's Dynamic Allocation Functions.***
Textbook 1: Chapter 5, 6

Chapter 5 Pointers (Contd…)

Pointers to Functions, C's Dynamic Allocation Functions.

Pointers to Functions

- A ***function has a physical location in memory that can be assigned to a pointer***. This address is the entry point of the function and it is the address used when the function is called. ***Once a pointer points to a function, the function can be called through that pointer.***
- ***Function pointers also allow functions to be passed as arguments to other functions.*** You obtain the address of a function by using the function's name without any parentheses or arguments.
- To see how this is done, study the following program, which compares two strings entered by the user. Pay close attention to the declarations of ***check( )*** and the function pointer ***"p"***, inside ***main( ).***
- Let's look closely at this program. First, examine the declaration for ***"p"*** in ***main().*** It is shown here:
  int (*p)(const char *, const char *);
- This declaration tells the compiler that ***"p"*** is a pointer to a function that has two ***const char ** parameters, and returns an ***int*** result.

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
    int (*cmp)(const char *, const char *));
int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);
  p = strcmp; /* assign address strcmp to p */
  printf("Enter two strings.\n");
  gets(s1);
  gets(s2);
  check(s1, s2, p);
  return 0;
 }
 void check(char *a, char *b,
    int (*cmp) (const char *, const char *))
 {
  printf(''Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
```

- The parentheses around ***"p"*** are necessary in order for the compiler to properly interpret this declaration. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ.
- Next, examine the ***check( )*** function. It declares ***three parameters***:
  - ***two character pointers***, **"a"** and **"b"**
  - one ***function pointer***, **cmp**.
- Notice that the function pointer is declared using the same format as was ***"p"*** inside ***main( ).*** Thus, ***cmp*** is able to receive a pointer to a function that takes two ***const char ** arguments and

returns an int result. Like the declaration for ***"p"***, the parentheses around the ****cmp*** are necessary for the compiler to interpret this statement correctly.

- When the program begins, it assigns ***"p"*** the address of ***strcmp( ),*** the standard string comparison function. Next, it prompts the user for two strings, and then it passes pointers to those strings along with ***"p"*** to ***check( ),*** which compares the strings for equality. Inside ***check( ),*** the expression ***(*cmp)(a, b)*** calls ***strcmp(***), which is pointed to by ***cmp,*** with the arguments ***"a" and "b".*** The parentheses around ****cmp*** are necessary. ***This is one way to call a function through a pointer***.
- A second, simpler syntax, as shown here, can also be used: ***cmp(a, b);***
- Note that you can call ***check( )*** by using ***strcmp( )*** directly, as shown here: ***check(s1, s2, strcmp);*** This ***eliminates the need for an additional pointer variable***, in this case.
- You can get a better idea of the value of function pointers by studying the expanded version of the previous example, shown next.
- In this version, ***check( )*** can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function. When checking for numeric equality, the string ***"0123"*** will compare equal to ***"123",*** even though the strings, themselves, differ.
- In this program, if you enter a string that begins with a digit, ***compvalues()*** is passed to ***check().*** Otherwise, ***strcmp()*** is used. Since ***check()*** calls the function that it is passed, it can use a different comparison function in different cases.
- Two sample program runs are shown here:

  Enter two values or two strings.
  Test
  Test
  Testing strings for equality.

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void check(char *a, char *b,
   int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);
int main(void)
{
 char s1[80], s2[80];
 printf("Enter two values or two strings.\n");
 gets (s1);
 gets(s2);
 if(isdigit(*sl)) {
    printf(''Testing values for equality.\n");
    check(s1, s2, compvalues);
  }
  else {
    printf("Testing strings for equality.\n");
    check(s1, s2, strcmp);
  }
  return 0;
 }
 void check(char *a, char *b,
    int (*cmp)(const char *, const char *))
 {
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
 int compvalues(const char *a, const char *b)
 {
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

malloc() Function in C

- ***malloc()*** stands for ***Memory Allocation***. It is a dynamic memory allocation function used in C to allocate memory at runtime.
- Defined in header file: ***#include <stdlib.h>***
- It allocates a block of memory in the heap and returns a pointer to the beginning of the block.
- If ***memory allocation fails, it returns NULL***.

General Form

ptr = (castType*) malloc(size_in_bytes);

Parameters:

- ***size_in_bytes*** → number of bytes to allocate.
- ***castType*** → the type of pointer (e.g., int*, float*, etc.).

- Return type: ***void**** (generic pointer)

Example – Allocating Memory for Integers

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *ptr;
    int n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    // Allocating memory using malloc
    ptr = (int*) malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    printf("Memory successfully allocated using
        malloc.\n");

    for (i = 0; i < n; i++) {
        ptr[i] = i + 1;
    }
    printf("The elements are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
// Freeing memory
    free(ptr);
    return 0;
}
```

```
Output
Enter number of elements: 2
Memory successfully allocated using malloc.
The elements are: 1 2
```

- The code fragment shown here allocates 1,000 bytes of contiguous memory:
  char *p;
  p = malloc(1000); /* get 1000 bytes */
- After the assignment, ***"p"*** points to the first of 1,000 bytes of free memory.
- The next example ***allocates space for 50 integers***. Notice the use of ***sizeof()*** to ensure portability.
  int *p;
  p = malloc(50*sizeof(int));
- Since the heap is not infinite, whenever you allocate memory, you must check the value returned by ***malloc( )*** to make sure that it is not null before using the pointer. Using a null pointer will almost certainly crash your program.
- The proper way to allocate memory and test for a valid pointer is illustrated in this code fragment:

```
p = malloc(100);
if(!p) {
    printf("Out of memory.\n");
    exit (1);
}
```

free( ) Function in C

- The function is the ***opposite of malloc( )*** in that it returns previously allocated memory to the system. ***Once the memory has been freed, it may be reused by a subsequent call to malloc( ).***
- The function ***free( )*** has this prototype: ***void free(void * p);***
- Here, ***"p"*** is a pointer to memory that was previously allocated using ***malloc( ).*** It is critical that you ***never call free( ) with an invalid argument***; this ***will damage the allocation system***.

Important Points

- The memory allocated by ***malloc()*** is uninitialized (contains garbage values).
- Use ***calloc()*** if you want initialized (zero) memory.
- Always check if ***malloc()*** returned NULL.
- Always ***free()*** dynamically allocated memory to prevent memory leaks.

Difference Between malloc(), calloc(), realloc(), and free()

Function	Purpose	Initialization	Syntax
malloc()	Allocate memory	Garbage values	ptr = malloc(size);
calloc()	Allocate multiple blocks	Initialized to 0	ptr = calloc(n, size);
realloc()	Resize allocated memory	Preserves old data	ptr = realloc(ptr, newSize);
free()	Deallocate memory	N/A	free(ptr);

C's Dynamic Allocation Functions

- C's ***dynamic allocation subsystem is used in conjunction with pointers*** to support a variety of ***important programming constructs***, such as ***linked lists*** and ***binary trees***.
- Sometimes you will want to allocate memory using ***malloc( ),*** but operate on that memory as if it were an array, using array indexing. In essence, you may want to create a dynamically allocated array. Since any pointer can be indexed as if it were an array, this presents no trouble.
- For example, the following program shows how you can use a dynamically allocated array to hold a one dimensional array— in this case, a string.

```c
/* Allocate space for a string dynamically, */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  char *s;
  register int t;
  s = malloc(80);

  if(!s) {
    printf(''Memory request failed.\n");
    exit (1);
  }
  gets(s);
  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
  free(s);
  return 0;
}
```

- As the program shows, before its first use, ***"s"*** is tested to ensure that the allocation request succeeded and that a valid pointer was returned by ***malloc( ).*** This is absolutely necessary to prevent accidental use of a null pointer. Notice how the pointer s is used in the call to ***gets( )*** and then indexed as an array to print the string backwards.

- You can ***also dynamically allocate multidimensional arrays***. To do so, you must ***declare a pointer*** that specifies ***array dimension***.
- To see how this works, study the following example, which builds a table of the numbers 1 through 10 raised to their first, second, third, and fourth powers.

```
#include <stdio.h>
#include <stdlib.h>
int pwr(int a, int b);
int main(void)
{
  /* Declare a pointer to an array */
  int (*p)[10];
  register int i, j;
  /* allocate memory to hold a 4 x 10 array */
  p = malloc(40*sizeof(int));
  if(!p) {
    printf(''Memory request failed.\n");
    exit (1);
  }
  for(j=1; j<11; j++)
    for(i=1; i<5; i++)
        p[i-1][j-1] = pwr(j, i);
  for(j=1; j<11; j++) {
    for(i=1; i<5; i++)
        printf("%10d ", p[i-1][j-1]);
    printf ("\n");
  }
  return 0;
 }
pwr(int a, int b)
{
  register int t=1;
  for(; b; b--) t = t*a;
  return t;
}
```

The output produced by this program is

1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

- In ***main( ),*** the pointer ***"p"*** is declared like this: ***int (*p)[10];***
- The parentheses around ****p*** are necessary. This declaration states that ***"p"*** is a pointer to an array of 10 integers. When ***"p"*** is incremented, it will point to the start of the next 10 integers; when decremented, ***"p"*** will point to the previous 10 integers. Thus, ***"p"*** is a pointer to a two-dimensional integer array that has 10 elements in each row. This means that ***"p"*** can be indexed as a two-dimensional array, as the program shows. The ***only difference is that the storage for the array is allocated manually using the malloc( ) statement***, rather than automatically using a normal array declaration statement.
- One final point: You must cast the pointer returned by ***malloc( ),*** as shown here:

p = (int (*)[10]) malloc(40*sizeof(int));

Problems with Pointers

- The classic ***example of a pointer error*** is the ***uninitialized pointer***.
- Consider this program:
- This program assigns the value 10 to some unknown memory location. Since the pointer ***"p"*** has never

```
int main(void)
{
  int x, *p;
  x = 10;
  *p = x; /* error, p not initialized */
    return 0;
 }
```

been given a value, it contains an unknown value when the assignment ***p = x*** takes place. This causes the value of ***"x"*** to be written to some unknown memory location.

- A ***second common error*** is caused by a ***simple misunderstanding of how to use a pointer***. Consider the following:

```
#include <stdio.h>
int main(void)
 {
  int x, *p;
  x = 10;
  p = x;
  printf("%d", *p);
  return 0;
 }
```

- The call to ***printf( )*** does not print the value of ***"x",*** which is 10, on the screen. It ***prints some unknown value*** because the ***assignment p = x; is wrong.*** That statement assigns the value 10 to the pointer ***"p".*** However, ***"p"*** is supposed to contain an address, not a value. To correct the program, write ***p = &x;***

Chapter 6. Functions

The General Form of a Function, Understanding the Scope of a Function, Function Arguments, argc and argv—Arguments to main(), The return Statement, What Does main() Return?, Recursion, Function Prototypes, Declaring Variable Length Parameter Declarations, The inline Keyword. (Pg. 147 – 172)

- ***Functions*** are the ***building blocks of C*** and the ***place where all program activity occurs***.

The General Form of a Function

- The general form of a function is

```
ret-type function-name(parameter list)
{
        body of the function
}
```

- The ***ret-type*** specifies the ***type of data that the function returns***. A ***function may return any type of data except an array***.
- The ***parameter list*** is a ***comma-separated list of variable names and their associated types***. The parameters receive the values of the arguments when the function is called. A function can be without parameters, in which case the parameter list is empty. An empty parameter list can be explicitly specified as such by placing the keyword void inside the parentheses.
- In ***variable declarations***, you can ***declare several variables to be of the same type by using a comma separated list of variable names***. In contrast, all function parameters must be declared individually, each including both the type and name.
- That is, the parameter declaration list for a function takes this ***general form***:

```
f(type varname1, type varname2, . . . , type varnameN)
```

- For example, here are a correct and an incorrect function parameter declaration:

f(int i, int k, int j)	/* correct */
f(int i, k, float j)	/* wrong, k must have its own type specifier */

==========

Understanding the Scope of a Function

- The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. The scopes defined by C were described in ***Module 1 Chapter 2 (The Four C Scopes).***
- Here we will look more closely at ***one specific scope:*** the one ***defined by a function***.
- Each ***function*** is a ***discrete block of code***. Thus, a ***function defines a block scope***. This means that a function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function.
- For instance, you ***cannot use goto to jump into the middle of another function***.
- The code that constitutes the body of a function is hidden from the rest of the program, and ***unless it uses global variables***, it can ***neither affect nor be affected by other parts of the program.***
- Stated another way, the code and data defined within one function cannot interact with the code or data defined in another function because the ***two functions have different scopes***.
- **Scope defines the visibility and accessibility of a function in a C program.**
- ***Functions*** in C can have ***two types of scope***:
  1. ***Global Scope***: The function is declared outside all other functions and can be accessed from any file (if declared extern). It can be called from **anywhere within the same file** (and from other files using ***extern***).
  2. ***Local Scope:*** The function is declared inside another function (nested function concept not supported directly in C, but function prototypes inside a block limit visibility).
- ***Variables that are defined within a function are local variables***. A local variable comes into existence when the function is entered and is destroyed upon exit. Thus, a local variable cannot hold its value between function calls. The ***only exception to this rule*** is ***when the variable is declared with the "static" storage class specifier***. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limit its scope to the function.
- The ***formal parameters*** to a function also fall ***within the function's scope***. This means that a parameter is known throughout the entire function.

```c
#include <stdio.h>

// Global function (can be used anywhere in this file
void globalFunction() {
    printf("This is a GLOBAL function. It can be
        accessed anywhere in this file.\n");
}

// Static (local) function - visible only within this
    file
static void localFunction() {
    printf("This is a STATIC (local) function. It can
        be accessed only in this file.\n");
}

int main() {
  printf("Demonstration of Function Scope in C\n\n");
  globalFunction();  // Accessible
  localFunction(); // Accessible here, but not from
      other files
  return 0;
}
```

- ***All functions have file scope***. Thus, you ***cannot define a function within a function***. This is why C is not technically a block-structured language.

==========

Function Arguments

(Topics covered: Call by Value, Call by Reference - Creating a Call by Reference - Calling Functions with Arrays)

- If a ***function*** is ***to accept arguments***, it ***must declare*** the parameters that will receive the values of the arguments. As shown in the following function, the parameter declarations occur after the function name.

```
int is_in(char *s, char c)
{
  while (*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

- The function ***is_in( )*** has ***two parameters***: ***"s"*** and ***"c".*** This function returns 1 if the character ***"c"*** is part of the string ***"s"***; otherwise, it returns 0. Even though parameters perform the special task of receiving the value of the arguments passed to the function, they behave like any other local variable.
- For example, you can make assignments to a function's formal parameters or use them in an expression.

"Call by Value", "Call by Reference"

- In a computer language there are ***two ways that arguments can be passed to a subroutine***.
- The ***first*** is ***"call by value".*** This ***method copies the value of an argument*** into the formal parameter of the subroutine. In this case, ***changes made to the parameter have no effect on the argument***.
- With few exceptions, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the following program:

```
#include <stdio.h>
int sqr(int x);
 int main(void)
 {
  int t=10;
  printf("%d %d", sqr(t), t);
  return 0;
 }
 int sqr(int x)
 {
  x = x*x;
  return(x);
}
```

Output

```
100 10
```

- In this example, the value of the argument to ***sqr( ), 10,*** is copied into the parameter ***"x"***. When the assignment ***x = x*x*** takes place, only the local variable ***"x"*** is modified. The variable ***"t",*** used to call ***sqr ( ),*** still has the value 10. Hence, the output is 100 10.
- Remember that it is a ***copy of the value of the argument that is passed into a function***. What ***occurs inside the function has no effect on the variable used in the call***.
- ***"Call by reference"*** is the ***second way of passing arguments*** to a subroutine. In this method, the address of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Creating a Call by Reference

- Even though C uses ***"call by value"*** for ***passing parameters***, you ***can create a call by reference by passing a pointer to an argument***, instead of passing the argument itself.
- Since the ***address of the argument is passed to the function***, ***code within the function can change the value of the argument outside the function***.
- ***Pointers are passed to functions just like any other argument***. Of course, you need to declare the parameters as pointer types. For example, the function ***swap( ),*** which exchanges the values of the two integer variables pointed to by its arguments, shows how:

```
void swap(int *x, int *y)
{
  int temp;
  temp =  *x; /* save the value at address x */
  *x = *y;    /* put y into x */
  *y = temp;  /* put x into y */
}
```

- The ***swap( )*** function is able to exchange the values of the two variables pointed to by x and y because their addresses (not their values) are passed. Within the function, the contents of the variables are accessed using standard pointer operations, and their values are swapped.
- Remember that ***swap( )*** (or any other function that uses pointer parameters) must be called with the addresses of the arguments. The following program shows the correct way to call ***swap( ):***

```
#include <stdio.h>
void swap(int *x, int *y);
int main (void)
 {
  int i, j;
  i = 10;
  j = 20;
  printf("i and j before swapping: %d %d\n", i, j);
  swap(&i, &j); /* pass the addresses of i and j */
  printf("i and j after swapping: %d %d\n", i, j);
  return 0;
}

void swap(int *x, int *y)
{
  int temp;
temp = *x;   /* save the value at address x */
  *x = *y;     /* put y into x */
  *y = temp;   /* put x into y */
}
```

Output

```
i and j before swapping: 10 20
i and j after swapping: 20 10
```

- In the program, the variable ***"i"*** is assigned the value 10, and ***"j"*** is assigned the value 20. Then ***swap( )*** is called with the addresses of ***"i"*** and ***"j"***. (The unary operator & is used to produce the address of the variables.) Therefore, the addresses of ***"i"*** and ***"j"***, not their values, are passed into the function ***swap( ).***

Calling Functions with Arrays

- ***Passing arrays as arguments to functions*** because it is an exception to the normal ***call-by-value parameter passing***.
- When an ***array is used as a function argument***, its ***address is passed to a function***. This is an exception to the call-by-value parameter passing convention. In this case, the code inside the

function is operating on, and potentially altering, the actual contents of the array used to call the function.

- For example, consider the function ***print_upper( ),*** which prints its string argument in uppercase:

```c
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);
int main(void)
{
    char s[80];

    printf("Enter a string: ");
    fgets(s, sizeof(s), stdin);

    print_upper(s);
    printf("\ns is now uppercase: %s", s);
    return 0;
}

/* Print a string in uppercase. */
void print_upper(char *string)
{
    int t;
    for(t = 0; string[t] != '\0'; ++t) {
        string[t] = toupper((unsigned char)string[t]
            );
        putchar(string[t]);
    }
}
```

- After the call to ***print_upper( ),*** the contents of array s in ***main( )*** are changed to uppercase.
- The standard library function ***gets( )*** is a classic example of passing arrays into functions. Although the ***gets( )*** in your standard library is more sophisticated, the following simpler version, called ***xgets ( ),*** will give you an idea of how it works.

```c
char *xgets(char *s)
{
  char ch, *p;
  int t;
  p = s; /* gets() returns a pointer to s */
  for(t=0; t<80; ++t){
    ch = getchar();
    switch(ch) {
      case '\n':
        s[t] = '\0'; /* terminate the string */
        return p;
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
 }
  s[79]  =  '\0';
  return p;
}
```

- The ***xgets( )*** function must be called with a ***char * pointer***. Upon entry, ***xgets( )*** establishes a for ***loop from 0 to 80***. This prevents larger strings from being entered at the keyboard. If more than 80 characters are entered, the function returns.
- Because C has no built-in bounds checking, you should make sure that any array used to call ***xgets ()*** can accept at least 80 characters.

- As you type characters on the keyboard, they are placed in the string. If you type a backspace, the counter t is reduced by 1, effectively removing the previous character from the array. When you press ENTER, a null is placed at the end of the string, signaling its termination.

==========

"argc" and "argv"—Arguments to main()

- Sometimes it is useful to pass information into a program when you run it. Generally, you ***pass information*** into the ***main( )*** function ***via command line arguments***. A ***command line argument*** is the information that ***follows the program's name on the command line of the operating system***.
- For example, when you compile a program, you might type something like the following after the command prompt,

  gcc program_name

  where ***program_name*** is a command line argument that specifies the name of the program you wish to compile.
- ***Two special built-in arguments***, ***"argc"*** and ***"argv",*** are used to receive command line arguments.
- The ***"argc"*** parameter ***holds the number of arguments on the command line and is an integer***. It is always at least 1 because the name of the program qualifies as the first argument.
- The ***"argv"*** parameter is a ***pointer*** to an ***array of character pointers***. Each element in this array points to a command line argument. All ***command line arguments are strings***— any numbers will have to be converted by the program into the proper binary format, manually.
- Here is a ***simple example*** that uses a command line argument. It prints ***"Hello and your name"*** on the screen***, if you specify your name as a command line argument***.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf(''You forgot to type your name.\n");
    exit(1);
  }
  printf("Hello %s", argv[1]);
  return 0;
 }
```

- If you called this program ***name*** and your name were "***Sarathy***", you would type name "***Sarathy***" to run the program. The output from the program would be "***Hello Sarathy***".
- In many environments, ***each command line argument must be separated by a space or a tab***.
- ***Commas, semicolons, and*** the like are not considered separators.
- For example,

  run Spot, run

  is made up of three strings, while ***Herb,Rick,Fred*** is a single string because commas are not generally legal separators.

- You must declare ***"argv"*** properly. The most common method is

  char *argv[];
- The empty brackets indicate that the array is of undetermined length. You can now access the individual arguments by indexing ***"argv".*** For example, ***argv[0]*** points to the first string, which is always the program's name; ***argv[1]*** points to the first argument, and so on.
- Another ***short example*** using command line arguments is the program called ***countdown***, shown here.

- It counts down from a starting value (which is specified on the command line) and beeps when it reaches 0. Notice that the first argument containing the starting count is converted into an integer by the standard function ***atoi( ).*** If the string ***"display"*** is the second command line argument, the countdown will also be displayed on the screen.

```
/* Countdown program. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
int main(int argc, char *argv[])
 {
  int disp, count;
  if(argc<2) {
    printf(''You must enter the length of the
        count\n");
    printf("on the command line. Try again.\n");
    exit(1);
  }
  if(argc==3 && !strcmp(argv[2], "display")) disp=1;
  else disp = 0;
  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);
  putchar('\a'); /* this will ring the bell */
  printf("Done");
  return 0;
 }
```

- Notice that if no command line arguments have been specified, an error message is printed. A program with command line arguments often issues instructions if the user attempts to run the program without entering the proper information.
- ***To access an individual character in one of the command line arguments, add a second index to "argv".*** For example, the next program displays all of the arguments with which it was called, one character at a time:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int t, i;

    for(t = 0; t < argc; ++t) {
        i = 0;

        while(argv[t][i]) {
            putchar(argv[t][i]);
            ++i;
        }
        printf("\n");   // corrected quotes
    }
    return 0;
}
```

- Remember, for ***"argv"***, the first index accesses the string, and the second index accesses the individual characters of the string.
- Usually, you use ***"argc"*** and ***"argv"*** to get initial commands into your program that are needed at start up. For example, command line arguments often specify such things as a filename, an option, or an alternate behavior. Using command line arguments gives your program a professional appearance and facilitates its use in batch files.

- The names ***"argc"*** and ***"argv"*** are traditional but arbitrary. You may name these two parameters to main ( ) anything you like. Also, some compilers may support additional arguments to ***main( ),*** so be sure to check your compiler's documentation.
- When a program does not require command line parameters, it is common practice to explicitly declare ***main( )*** as having no parameters. This is accomplished by using the ***"void"*** keyword in its parameter list.

==========

The "return" Statement (Page 158-164)

Topics covered: Returning from a Function-Returning Values-Returning Pointers-Functions of Type void

- The mechanics of ***"return"*** has two important uses.
- ***First***, it ***causes an immediate exit from the function***. That is, it causes program execution to return to the calling code.
- ***Second***, it ***can be used to return a value***. The following sections examine how the return statement is applied.

Returning from a Function

- A function terminates execution and returns to the caller in ***two ways***.
  1. The first occurs when the last statement in the function has been executed.
  2. The function's ending curly brace (}) is encountered.
- For example, the ***pr_reverse( )*** function in this program simply prints the string I like C backwards on the screen and then returns.----------→

```c
#include <string.h>
#include <stdio.h>
void pr_reverse(char *s);
int main(void)
{
  pr_reverse("I like C");
  return 0;
 }
void pr_reverse(char *s)
 {
  register int t;
  for(t=strlen(s)-1; t>=0; t--)
    putchar(s[t]);
 }
```

- Once the string has been displayed, there is nothing left for ***pr_reverse( )*** to do, so it returns to the place from which it was called.
- Not many functions use this default method of terminating their execution. ***Most functions rely on the "return" statement to stop execution*** either because a value must be returned or to make a function's code simpler and more efficient.
- A ***function may contain several "return" statements***. For example, the ***find_substr( )*** function in the following program returns the starting position of a substring within a string, or it returns –1 if no match is found. It uses two return statements to simplify the coding.

```c
#include <stdio.h>
int find_substr(char *s1, char *s2);
int main(void)
{
  if(find_substr("C is fun", "is") != -1)
    printf("Substring is found.");
  return 0;
 }
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;
  for(t=0; s1[t]; t++)
    p = &s1[t];
    p2 = s2;
    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* 1st return */
  }
   return -1; /* 2nd return */
}
```

Returning Values

- ***All functions, except those of type void, return a value***. This value is specified by the return statement.
- In C89, if a non-void function executes a return statement that does not include a value, then a garbage value is returned. This is, to say the least, bad practice!
- In C99 (and C++), a non-void function must use a return statement that returns a value. That is, in C99, if a function is specified as returning a value, any return statement within it must have a value associated with it. However, if execution reaches the end of a non-void function (that is, encounters the function's closing curly brace), a garbage value is returned. Although this condition is not a syntax error, it is still a fundamental flaw and should be avoided.
- As long as a function is not declared as void, you can use it as an operand in an expression. Therefore, each of the following expressions is valid:

```
x = power(y);
if(max(x,y) > 100) printf("greater");
for(ch=getchar(); isdigit(ch); ) . . . ;
```

- ***As a general rule, a function call cannot be on the left side of an assignment***.
- A statement such as ***swap(x,y) = 100; /* incorrect statement */*** is wrong. The C compiler will flag it as an error and will not compile a program that contains it.
- When you write programs, your functions will be of three types.
- The ***first type is simply computational***. These functions are specifically designed to perform operations on their arguments and return a value based on that operation. A computational function ***is a "pure" function***. Examples are the standard library functions ***sqrt( ) and sin( ),*** which compute the square root and sine of their arguments.
- The ***second type of function manipulates information and returns a value that simply indicates the success or failure of that manipulation***. An example is the library function ***fclose( ),*** which closes a file. If the close operation is successful, the function returns 0; it returns EOF if an error occurs.
- The ***last type of function has no explicit return value***. In essence, the ***function is strictly procedural and produces no value***. An example is ***exit( ),*** which terminates a program. All functions that do not return values should be ***declared as returning type void***. By declaring a function as void, you keep it from being used in an expression, thus preventing accidental misuse.
- Sometimes, functions that really don't produce an interesting result return something anyway. Consider the following program, which uses the function ***mul( ):-------→***
- In line 1, the return value of ***mul( )*** is assigned to z.
- In line 2, the return value is not actually assigned, but it is used by the ***printf( )*** function.
- Finally, in line 3, the return value is lost because it is neither assigned to another variable nor used as part of an expression.

```
#include <stdio.h>
int mul(int a, int b);
int main(void)
{                          OUTPUT: 200
  int x, y, z;
  x = 10;   y = 20;
  z = mul(x, y);           /* 1 */
  printf("%d", mul(x,y)); /* 2 */
  mul(x, y);               /* 3 */
  return 0;
}
 int mul(int a, int b)
 {
  return a*b;
}
```

Returning Pointers

- ***Pointers*** are ***neither integers nor unsigned integers***. They are the ***memory addresses*** of a certain type of data. One reason for this distinction is that pointer arithmetic is relative to the base type.
- For example, if an integer pointer is incremented, it will contain a value that is four greater than its previous value (assuming 4-byte integers). In general, each time a pointer is incremented (or decremented), it points to the next (or previous) item of its type.

- Since ***the length of different data types may differ***, the ***compiler must know what type of data the pointer is pointing to***. For this reason, a function that returns a pointer must declare explicitly what type of pointer it is returning. For example, you should not use a return type of ***int ** to return a ***char ** pointer! In a few cases, a function will need to return a generic pointer. In this case, the function return type must be specified as ***void *.***
- ***To return a pointer, a function must be declared as having a pointer return type***.
- For example, the following function returns a pointer to the first occurrence of the character c in string s: If no match is found, a pointer to the null terminator is returned.

```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

- Here is a short program that uses ***match( ):***

```
#include <stdio.h>
char *match(char c, char *s); /* prototype */

int main(void)
{
  char s[80], *p, ch;
  gets(s);
  ch = getchar();
  p = match(ch, s);
  if(*p) /* there is a match */
    printf(''%s ", p);
  else
    printf("No match found.");
  return 0;
 }
```

- This ***program reads a string and then a character***. It then searches for an occurrence of the character in the string. If the character is in the string, ***"p"*** will point to that character, and the program prints the string from the point of match. When no match is found, ***"p"*** will be pointing to the null terminator, making ****p*** false. In this case, the program prints No match found.

Functions of Type "void"

- One of ***void's uses is to explicitly declare functions that do not return values***. This prevents their use in any expression and helps avert accidental misuse. For example, the function ***print_vertical()*** prints its string argument vertically down the side of the screen. Since it returns no value, it is declared as void.

```
void print_vertical (char *str)
{
  while (*str)
    printf("%c\n", *str++);
}
```

- An example that uses ***print_vertical( ):***

```c
#include <stdio.h>
void print_vertical(char *str);  /* prototype */
int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);
  return 0;
 }
void print_vertical(char *str)
 {
  while(*str)
    printf(''%c\n", *str++);
}
```

==========

What Does "main()" Return?

- The ***main( )*** function returns an integer to the calling process, which is ***generally the operating system***.
- Returning a value from ***main( )*** is the equivalent of calling ***exit( )*** with the same value.
- If ***main( )*** does not explicitly return a value, the value passed to the calling process is technically undefined.
- In practice, ***most C compilers automatically return 0***, but do not rely on this if portability is a concern.

==========

Recursion

- In C, a ***function can call itself. In this case, the function is said to be recursive***.
- ***Recursion is the process of defining something in terms of itself, and is sometimes called circular definition.***
- A simple example of a recursive function is ***factr( ),*** which computes the factorial of an integer. The factorial of a number n is the product of all the whole numbers between 1 and n.
- For example, 3 factorial is 1 × 2 × 3, or 6. Both factr( ) and its iterative equivalent are shown here:

```c
#include <stdio.h>
// Recursive function to find factorial
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;      // Base case
    else
        return n * factorial(n - 1);   // Recursive call
}
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (num < 0)
        printf("Factorial is not defined for negative numbers.\n");
    else
        printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

```
Output
Enter a number: 4
Factorial of 4 is 24
```

Recursive	Non-recursive

/* recursive */
int factr(int n) {
int answer;
if(n==l) return(1);
answer = factr(n-l)*n; /* recursive call */
return(answer);
}
	/* non-recursive */
	int fact(int n) {
	int t, answer;
	answer = 1;
	for(t=1; t<=n; t++)
	answer=answer*(t);
	return(answer);
	}

- The ***non-recursive*** version of ***fact( )*** should be clear. It uses a loop that runs from 1 to n and progressively multiplies each number by the moving product.
- The operation of the ***recursive factr( )*** is a little more complex. When ***factr( )*** is called with an argument of 1, the function returns 1. Otherwise, it returns the product of ***factr(n-1)*n.*** To evaluate this expression, ***factr( )*** is called with ***n-1***. This happens until n equals 1 and the calls to the function begin returning.
- When a ***function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables***.
- A ***recursive call does not make a new copy of the function***. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.
- ***Recursive functions could be said to "telescope" out and back.***
- ***Recursion seems to offer the possibility of improved efficiency***. Often, recursive routines do not significantly reduce code size or improve memory utilization. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls.
- ***Many recursive calls to a function could cause a stack overrun***. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be exhausted. A stack overrun is what usually causes a program to crash when a recursive function runs wild.
- ***The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms.***
- For example, the ***quicksort algorithm*** is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions.
- Finally, some people seem to think ***recursively more easily than iteratively***.
- ***When writing recursive functions, you must have a conditional statement***, such as an ***"if"***, somewhere to force the function to return without the recursive call being executed. If you don't, the function will never return once you call it.
- ***Omitting the conditional statement is a common error when writing recursive functions***. Use ***printf( )*** liberally during program development so that you can watch what is going on and abort execution if you see a mistake.

==========

Function Prototypes

- ***In modern***, properly written C programs, ***all functions must be declared before they are used***. This is normally ***accomplished*** using a ***function prototype***.
- Function prototypes were not part of the original C language, but were added by C89. Although prototypes are not technically required, their use is strongly encouraged.
- ***Prototypes enable the compiler to provide stronger type checking***.

- When you use prototypes, the compiler can find and report any questionable type conversions between the arguments used to call a function and the type of its parameters. The compiler will also catch differences between the number of arguments used to call a function and the number of parameters in the function.
- The general form of a function prototype is

  type func_name(type parm_name1, type parm_name2, . . . , type parm_nameN);

- The use of parameter names is optional. However, they enable the compiler to identify any type mismatches by name when an error occurs, so it is a good idea to include them.
- The following program illustrates the value of function prototypes. It produces an error message because it contains an attempt to call ***sqr_it( )*** with an integer argument instead of the integer pointer required.

```
/* This program uses a function prototype to
   enforce strong type checking. */
#include <stdio.h>
void sqr_it(int *i); /* prototype */

int main(void)
 {
  int x;
  x = 10;
  sqr_it(x);  /* type mismatch */
  return 0;
 }

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

```
ERROR!
10 |   sqr_it(x);  /* type mismatch */
   |          ^
   |          |
   |          int
 4 | void sqr_it(int *i); /* prototype */
   |             ~~~~~^
```

Original Error: ***sqr_it(x);*** /* type mismatch */ • The function **sqr_it()** expects a pointer (int *), but you passed an **integer**.	✓ **Correct Fix:** ***sqr_it(&x);*** • Pass the **address of x** so the function receives int*.

- A function's definition can also serve as its prototype if the definition occurs prior to the function's first use in the program. For example, this is a valid program:

```
 #include <stdio.h>
/* This definition will also serve
   as a prototype within this program. */
 void f(int a, int b)
{
  printf("%d ", a % b);
}
 int main (void)
 {
  f(10,3);
  return 0;
 }
```

Output

```
1
```

- In this example, since ***f( )*** is defined prior to its use in ***main( ),*** no separate prototype is required. Although it is possible for a function's definition to serve as its prototype in small programs, it is seldom possible in large ones— especially when several files are used. The programs in this book include a separate prototype for each function because that is the way C code is normally written in practice.

- ***The only function that does not require a prototype is main( ) because it is the first function called when your program begins.***
- In C, when a function has no parameters, its prototype uses void inside the parameter list. For example, here is ***f( )'s*** prototype as it would appear in a C program: ***float f(void);*** This tells the ***compiler that the function has no parameters***, and any call to that function that has arguments is an error.
- Function prototypes help you trap bugs before they occur. In addition, they help verify that your program is working correctly by not allowing functions to be called with mismatched arguments.
- One last point: Since ***early versions of C did not support the full prototype syntax, prototypes are technically optional in C***. This is ***necessary to support pre-prototype*** C code.

Old-Style Function Declarations

- In the early days of C, prior to the creation of function prototypes, there was still a need to tell the compiler in advance about the return type of a function so that the proper code could be generated when the function was called. This was accomplished using a function declaration that did not contain any parameter information.
- Using the old-style approach, the function's return type and name are declared near the start of your program, as illustrated here:

```
#include <stdio.h>
double div(); /* old-style function declaration */
int main(void)
{
  printf("%f", div(10.2, 20.0));
  return 0;
 }
double div(double num, double denom)
{
  return num / denom;
}
```

Output
0.510000

- The old-style function type declaration tells the compiler that ***div( )*** returns an object of type ***double***. This allows the compiler to correctly generate code for calls to div( ). It does not, however, say anything about the parameters to ***div( ).***
- The old-style function declaration statement has the following general form:

  type_specifier function_name( );

- ***Notice that the parameter list is empty***. Even if the function takes arguments, none are listed in its type declaration. As stated, the old-style function declaration is outmoded and should not be used for new code.

Standard Library Function Prototypes

- ***Any standard library function used by your program must be prototyped***.
- ***To accomplish*** this, you ***must include the appropriate header*** for each library function.
- All necessary headers are provided by the C compiler. In C, the ***library headers*** are (usually) files that use the ***".h" extension***.
- A header contains ***two main elements***:
  - any definitions used by the library functions and
  - the prototypes for the library functions.
- For example, ***<stdio.h>*** is included in almost all programs in this book because it contains the prototype for ***printf( ).***

==========

Declaring Variable Length Parameter

- You can specify a function that has a variable number of parameters. The most common example is ***printf( ).*** To tell the compiler that an unknown number of arguments will be passed to a function, you must end the declaration of its parameters using three periods. For example, this prototype specifies that ***func( )*** will have at least two integer parameters and an unknown number (including 0) of parameters after that: ***int func(int a, int b, . . .);***
- This form of declaration is also used by a function's definition.
- Any function that uses a variable number of parameters must have at least one actual parameter. For example, this is incorrect: ***int func(. . .); /* illegal */***

==========

Old-Style vs. Modern Function Parameter Declarations

- Early versions of C used a different parameter declaration method than do modern versions of C, including both C89 and C99. This early approach is sometimes called the ***classic form***.
- ***This book uses a declaration approach called the modern form.*** Standard C supports both forms, but strongly recommends the modern form. However, you should know the old-style form because many older C programs still use it.
- The ***old-style function parameter declaration consists of two parts***:
  - a ***parameter list***, which goes inside the parentheses that follow the function name, and
  - the ***actual parameter declarations***, which go between the closing parentheses and the function's opening curly brace.
- The general form of the old-style parameter definition is

```
type parm1;
type parm2;
.
.
.
type parmN;
{
function code
}
```

For example, this modern declaration	will look like this in its old-style form:
float f(int a, int b, char ch) { ***/* . . . */*** ***}***	***float f(a, b, ch)*** ***int a, b;*** ***char ch;*** ***{*** ***/* . . . */*** ***}***

- Notice that the old-style form allows the declaration of more than one parameter in a list after the type name.

==========

The "inline" Keyword

- C99 has added the keyword inline, which applies to functions. By preceding a function declaration with ***"inline",*** you are telling the compiler to optimize calls to the function. Typically, this means that the function's code will be expanded in line, rather than called. However, inline is only a request to the compiler, and can be ignored.

==========******==========