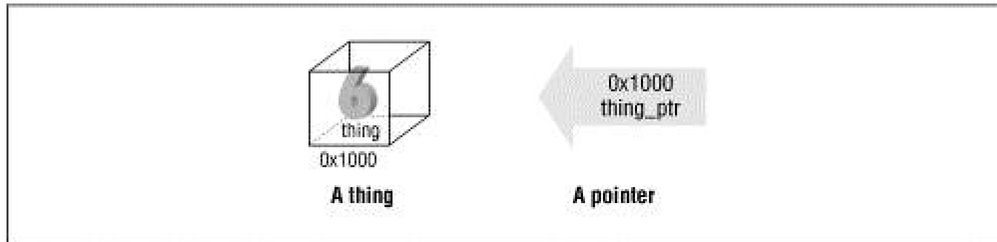
POINTERS AND ARRAYS IN C

There are things and pointers to things. Knowing the difference between the two is very important. This concept is illustrated in Figure 13-1.

Figure 13-1. A thing and a pointer to a thing



In this book, we use a box to represent a thing. The name of the variable is written on the bottom of the box. In this case, our variable is named thing. The value of the variable is 6. The address of thing is 0x1000. Addresses are automatically assigned by the C compiler to every variable. Normally, you don't have to worry about the addresses of variables, but you should understand that they're there.

Our pointer (thing_ptr) points to the variable thing. Pointers are also called *address variables* because they contain the addresses of other variables. In this case, our pointer contains the address 0x1000. Because this is the address of thing, we say that thing_ptr points to thing.

Variables and pointers are much like street addresses and houses. For example, your address might be "214 Green Hill Lane." Houses come in many different shapes and sizes. Addresses are approximately the same size (street, city, state, and zip). So, while "1600 Pennsylvania Ave." might point to a very big white house and "8347 Undersea Street" might be a one-room shack, both addresses are the same size.

The same is true in C. While things may be big and small, pointers come in one size (relatively small).[1]

Many novice programmers get pointers and their contents confused. In order to limit this problem, all pointer variables in this book end with the extension _ptr. You might want to follow this convention in your own programs. Although this notation is not as common as it should be, it is extremely useful.

Many different address variables can point to the same thing. This concept is true for street addresses as well. Table 13-1 lists the location of important services in a small town.

Table 13-1: Directory of Ed's Town USA

Service (variable name)	**Address (address value)**	**Building (thing)**
Fire Department	1 Main Street	City Hall
Police Station	1 Main Street	City Hall
Planning office	1 Main Street	City Hall

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Gas Station	2 Main Street	Ed's Gas Station

In this case, we have a government building that serves many functions. Although it has one address, three different pointers point to it.

As we will see in this chapter, pointers can be used as a quick and simple way to access arrays. In later chapters, we will discover how pointers can be used to create new variables and complex data structures such as linked lists and trees. As you go through the rest of the book, you will be able to understand these data structures as well as create your own.

A pointer is declared by putting an asterisk (*) in front of the variable name in the declaration statement:

```
int thing;      /* define a thing */
int *thing_ptr; /* define a pointer to a thing */
```

Table 13-2 lists the operators used in conjunction with pointers.

Table 13-2: Pointer Operators

Operator	Meaning
*	*Dereference* (given a pointer, get the thing referenced)
&	*Address of* (given a thing, point to it)

The operator ampersand (&) returns the address of a thing which is a pointer. The operator asterisk (*) returns the object to which a pointer points. These operators can easily cause confusion. Table 13-3 shows the syntax for the various pointer operators.

Table 13-3: Pointer Operator Syntax

C Code	Description
thing	Simple thing (variable)
&thing	Pointer to variable thing
thing_ptr	Pointer to an integer (may or may not be specific integer thing)
*thing_ptr	Integer

Let's look at some typical uses of the various pointer operators:

```
int thing; /* Declare an integer (a thing) */
thing = 4;
```

The variable thing is a thing. The declaration int thing does *not* contain an *, so thing is not a pointer:

```
int *thing_ptr;    /* Declare a pointer to a thing */
```

The variable thing_ptr is a pointer. The * in the declaration indicates this is a pointer. Also, we have put the extension _ptr onto the name:

```
thing_ptr = &thing; /* Point to the thing */
```

The expression &thing is a pointer to a thing. The variable thing is an object. The & (address of operator) gets the address of an object (a pointer), so &thing is a pointer. We then assign this to thing_ptr, also of type pointer:

```
*thing_ptr = 5;    /* Set "thing" to 5 */
                   /* We may or may not be pointing */
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

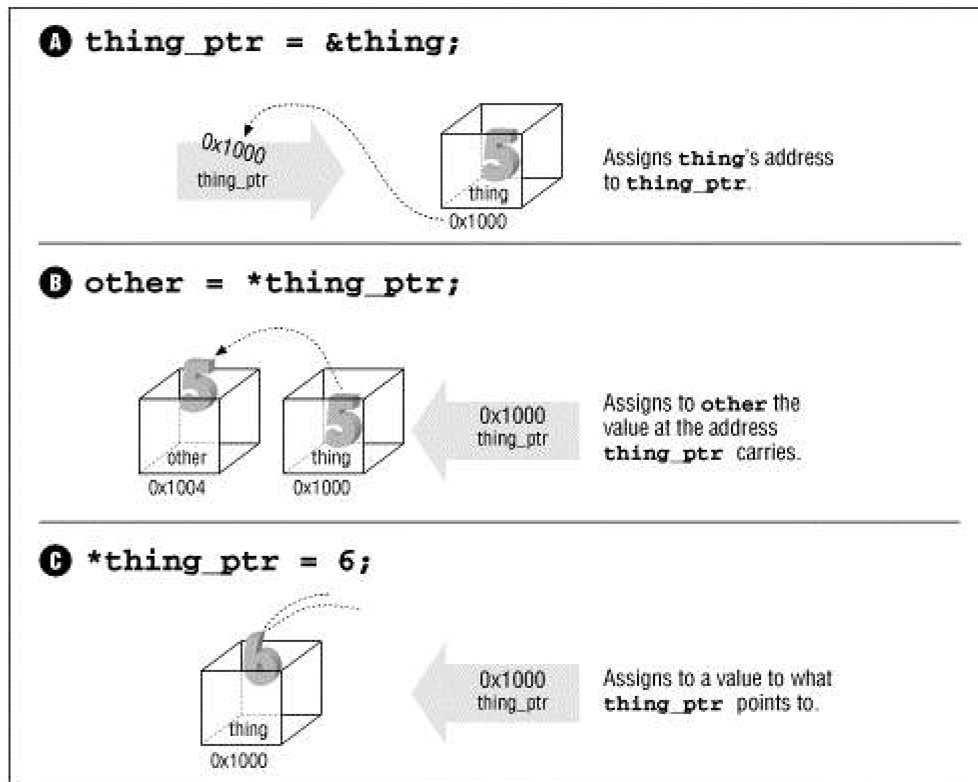
/* to the specific integer "thing" */

The expression *thing_ptr indicates a thing. The variable thing_ptr is a pointer. The * (dereference operator) tells C to look at the data pointed to, not the pointer itself. Note that this points to any integer. It may or may not point to the specific variable thing.

Introduction

These pointer operations are summarized in Figure 13-2.

Figure 13-2. Pointer operations



The following examples show how to misuse the pointer operations:

***thing**

is illegal. It asks C to get the object pointed to by the variable thing. Because thing is not a pointer, this operation is invalid.

&thing_ptr

is legal, but strange. thing_ptr is a pointer. The & (address of operator) gets a pointer to the object (in this case thing_ptr). The result is a pointer to a pointer.

Example 13-1 illustrates a simple use of pointers. It declares one object, one thing, and a pointer, thing_ptr. thing is set explicitly by the line:

```
thing = 2;
```

The line:

```
thing_ptr = &thing;
```

causes C to set thing_ptr to the address of thing. From this point on, thing and *thing_ptr are the same.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Example 13-1: thing/thing.c

```
#include <stdio.h>
int main()
{
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;     /* thing_ptr points to thing_var so */
                   /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
}
```

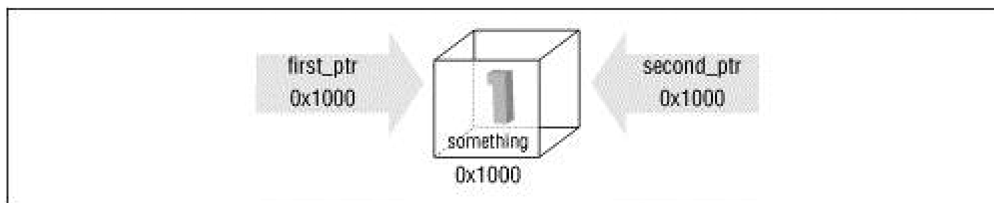
Several pointers can point to the same thing:

```
1:    int something;
2:
3:    int      *first_ptr;     /* one pointer */
4:    int      *second_ptr;    /* another pointer */
5:
6:    something = 1;           /* give the thing a value */
7:
8:    first_ptr = &something;
9:    second_ptr = first_ptr;
```

In line 8, we use the & operator to change something, a thing, into a pointer that can be assigned to first_ptr. Because first_ptr and second_ptr are both pointers, we can do a direct assignment in line 9.

After executing this program fragment, we have the situation shown in Figure 13-3.

Figure 13-3. Two pointers and a thing



You should note that while we have three variables, there is only one integer (something). The following are all equivalent:

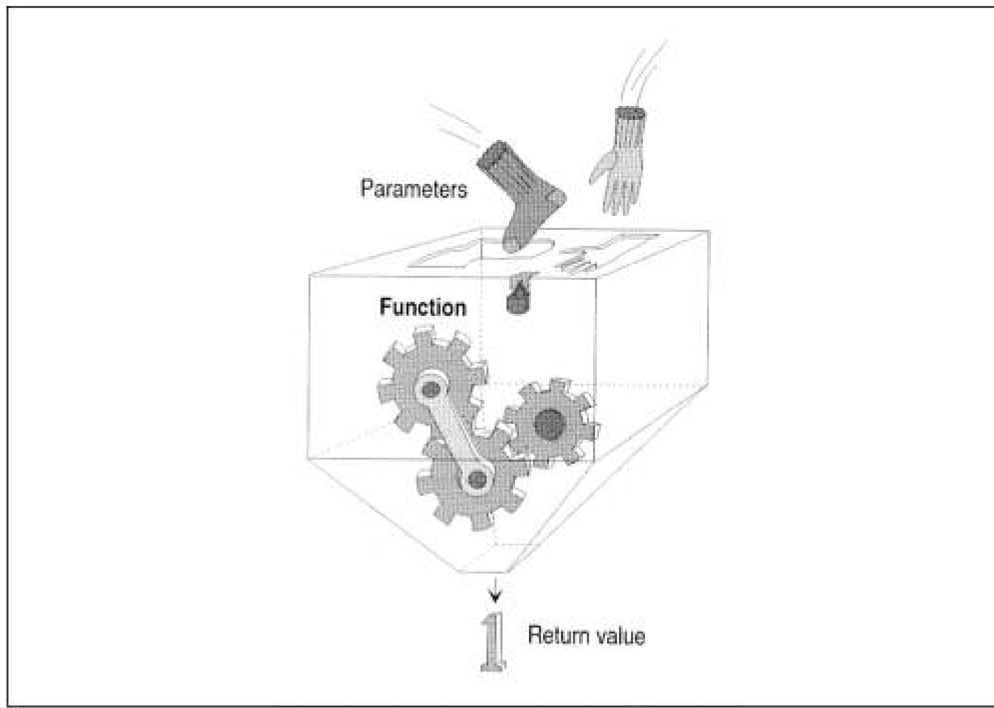
```
something = 1;
*first_ptr = 1;
*second_ptr = 1;
```

Pointers as Function Arguments

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

C passes parameters using "call by value." That is, the parameters go only one way into the function. The only result of a function is a single return value. This concept is illustrated in Figure 13-4.

Figure 13-4. Function call



However, pointers can be used to get around this restriction.

Imagine that there are two people, Sam and Joe, and whenever they meet, Sam can only talk and Joe can only listen. How is Sam ever going to get any information from Joe? Simple: all Sam has to do is tell Joe, "I want you to leave the answer in the mailbox at 335 West 5th Street."

C uses a similar trick to pass information from a function to its caller. In Example 13-2, main wants the function inc_count to increment the variable count.

Passing it directly would not work, so a pointer is passed instead ("Here's the address of the variable I want you to increment"). Note that the prototype for inc_count contains an int *. This format indicates that the single parameter given to this function is a pointer to an integer, not the integer itself.

Example 13-2: call/call.c

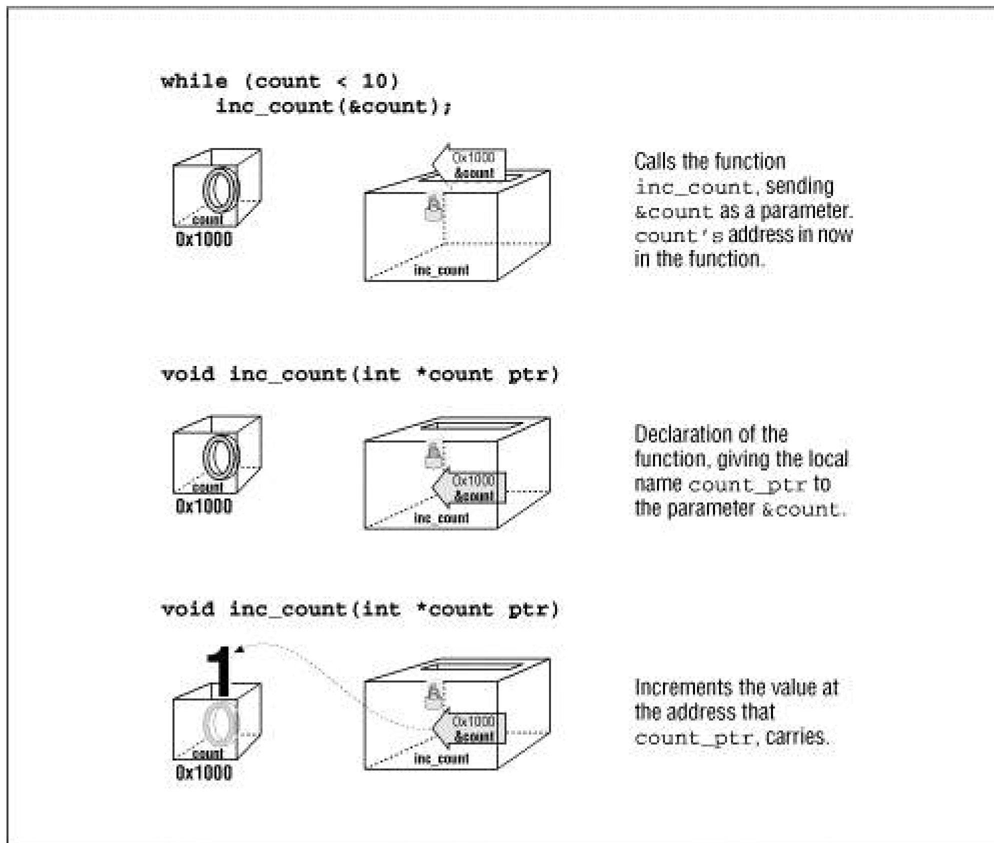
```
#include <stdio.h>
void inc_count(int *count_ptr)
{
   (*count_ptr)++;
}

int main()
{
   int  count = 0;     /* number of times through */

   while (count < 10)
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
        inc_count(&count);

    return (0);
}
```

This code is represented graphically in Figure 13-5. Note that the parameter is not changed, but what it points to is changed.

Figure 13-5. Call of inc_count



Finally, there is a special pointer called NULL. It points to nothing. (The actual numeric value is 0.) The standard include file, *locale.h*, defines the constant NULL. (This file is usually not directly included, but is usually brought in by the include files *stdio.h* or *stdlib.h*.) The NULL pointer is represented graphically in Figure 13-6.

Figure 13-6. NULL

const Pointers

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Declaring constant pointers is a little tricky. For example, the declaration:

```
const int result = 5;
```

tells C that result is a constant so that:

```
result = 10;      /* Illegal */
```

is illegal. The declaration:

```
const char *answer_ptr = "Forty-Two";
```

does *not* tell C that the variable answer_ptr is a constant. Instead, it tells C that the data pointed to by answer_ptr is a constant. The data cannot be changed, but the pointer can. Again we need to make sure we know the difference between "things" and "pointers to things."

What's answer_ptr? A pointer. Can it be changed? Yes, it's just a pointer. What does it point to? A const char array. Can the data pointed to by answer_ptr be changed? No, it's constant.

In C this is:

```
answer_ptr = "Fifty-One";    /* Legal (answer_ptr is a variable) */
*answer_ptr = 'X';           /* Illegal (*answer_ptr is a constant) */
```

If we put the **const** after the * we tell C that the pointer is constant.

For example:

```
char *const name_ptr = "Test";
```

What's name_ptr? It is a constant pointer. Can it be changed? No. What does it point to? A character. Can the data we pointed to by name_ptr be changed? Yes.

```
name_ptr = "New";           /* Illegal (name_ptr is constant) */
*name_ptr = 'B';            /* Legal (*name_ptr is a char) */
```

Finally, we can put **const** in both places, creating a pointer that cannot be changed to a data item that cannot be changed:

```
const char *const title_ptr = "Title";
```

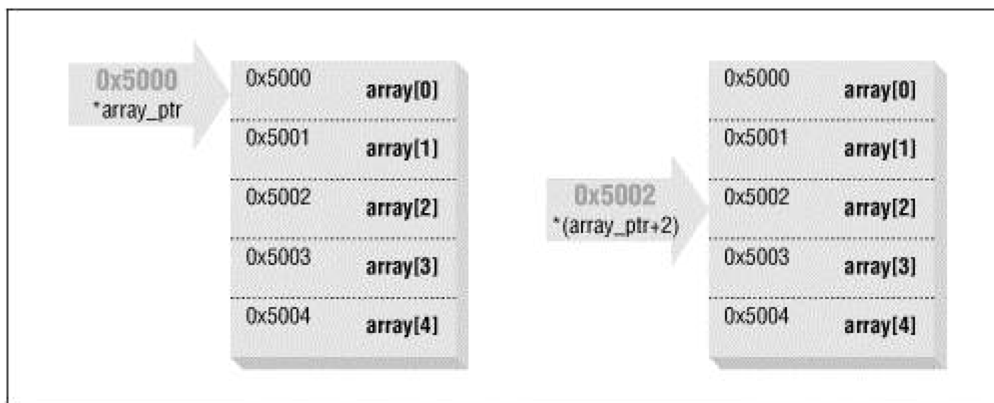
Pointers and Arrays

C allows pointer arithmetic (addition and subtraction). Suppose we have:

```
char array[5];
char *array_ptr = &array[0];
```

In this example, *array_ptr is the same as array[0], *(array_ptr+1) is the same as array[1], *(array_ptr+2) is the same as array[2], and so on. Note the use of parentheses. Pointer arithmetic is represented graphically in Figure 13-7.

Figure 13-7. Pointers into an array



No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

However, (*array_ptr)+1 is *not* the same as array[1]. The +1 is outside the parentheses, so it is added after the dereference. So (*array_ptr)+1 is the same as array[0]+1.
At first glance, this method may seem like a complex way of representing simple array indices. We are starting with simple pointer arithmetic. In later chapters we will use more complex pointers to handle more difficult functions efficiently.
The elements of an array are assigned to consecutive addresses. For example, array[0] may be placed at address 0xff000024. Then array[1] would be placed at address 0xff000025, and so on. This structure means that a pointer can be used to find each element of the array.
Example 13-3 prints out the elements and addresses of a simple character array.
Example 13-3: array-p/array-p.c

```
#include <stdio.h>

#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

NOTE: When printing pointers, the special conversion %p should be used.

When run, this program prints:

```
&array[index] (array+index) array[index]
0x40b0        0x40b0        0x30
0x40b1        0x40b1        0x31
0x40b2        0x40b2        0x32
0x40b3        0x40b3        0x33
0x40b4        0x40b4        0x34
0x40b5        0x40b5        0x35
0x40b6        0x40b6        0x36
0x40b7        0x40b7        0x37
0x40b8        0x40b8        0x38
0x40b9        0x40b9        0x39
```

Characters use one byte, so the elements in a character array will be assigned consecutive addresses. A short int font uses two bytes, so in an array of short int, the addresses increase by two. Does this mean that array+1 will not work for anything other than characters? No. C automatically scales pointer arithmetic so that it works correctly. In this case, array+1 will point to element number 1.
C provides a shorthand for dealing with arrays. Rather than writing:

```
array_ptr = &array[0];
```

we can write:

```
array_ptr = array;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

C blurs the distinction between pointers and arrays by treating them in the same manner in many cases. Here we use the variable array as a pointer, and C automatically does the necessary conversion.

Example 13-4 counts the number of elements that are nonzero and stops when a zero is found. No limit check is provided, so there must be at least one zero in the array.

Example 13-4: ptr2/ptr2.c

```
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
   index = 0;
   while (array[index] != 0)
      ++index;

   printf("Number of elements before zero %d\n",
              index);
   return (0);
}
```

Example 13-5 is a version of Example 13-4 that uses pointers.

Example 13-5: ptr3/ptr3.c

```
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
   array_ptr = array;

   while ((*array_ptr) != 0)
      ++array_ptr;

   printf("Number of elements before zero %d\n",
              array_ptr - array);
   return (0);
}
```

Notice that when we wish to examine the data in the array, we use the dereference operator (*). This operator is used in the statement:

```
while ((*array_ptr) != 0)
```

When we wish to change the pointer itself, no other operator is used. For example, the line:

```
++array_ptr;
```

increments the pointer, not the data.

Example 13-4 uses the expression (array[index] != 0). This expression requires the compiler to generate an index operation, which takes longer than a simple pointer dereference, ((*array_ptr) != 0).

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

The expression at the end of this program, array_ptr - array, computes how far array_ptr is into the array.

When passing an array to a procedure, C will automatically change the array into a pointer. In fact, if you put & before the array, C will issue a warning. Example 13-6 illustrates the various ways in which an array can be passed to a subroutine.

Example 13-6: init-a/init-a.c (continued)

```
#define MAX 10  /* Size of the array */
/********************************************************
 * init_array_1 -- Zeroes out an array.                 *
 *                                                      *
 * Parameters                                           *
 *      data -- The array to zero out.                  *
 ********************************************************/
void init_array_1(int data[])
{
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

/********************************************************
 * init_array_2 -- Zeroes out an array.                 *
 *                                                      *
 * Parameters                                           *
 *      data_ptr -- Pointer to array to zero.           *
 ********************************************************/
void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
int main()
{
    int  array[MAX];

    void init_array_1();
    void init_array_2();

    /* one way of initializing the array */
    init_array_1(array);

    /* another way of initializing the array */
    init_array_1(&array[0]);

    /* works, but the compiler generates a warning */
    init_array_1(&array);

    /* Similar to the first method but  */
    /*    function is different */
    init_array_2(array);
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
   return (0);
}
```

How Not to Use Pointers

The major goal of this book is to teach you how to create clear, readable, maintainable code. Unfortunately, not everyone has read this book and some people still believe that you should make your code as compact as possible. This belief can result in programmers using the ++ and -- operators inside other statements.

Example 13-7 shows several examples in which pointers and the increment operator are used together.

Example 13-7: Bad Pointer Usage

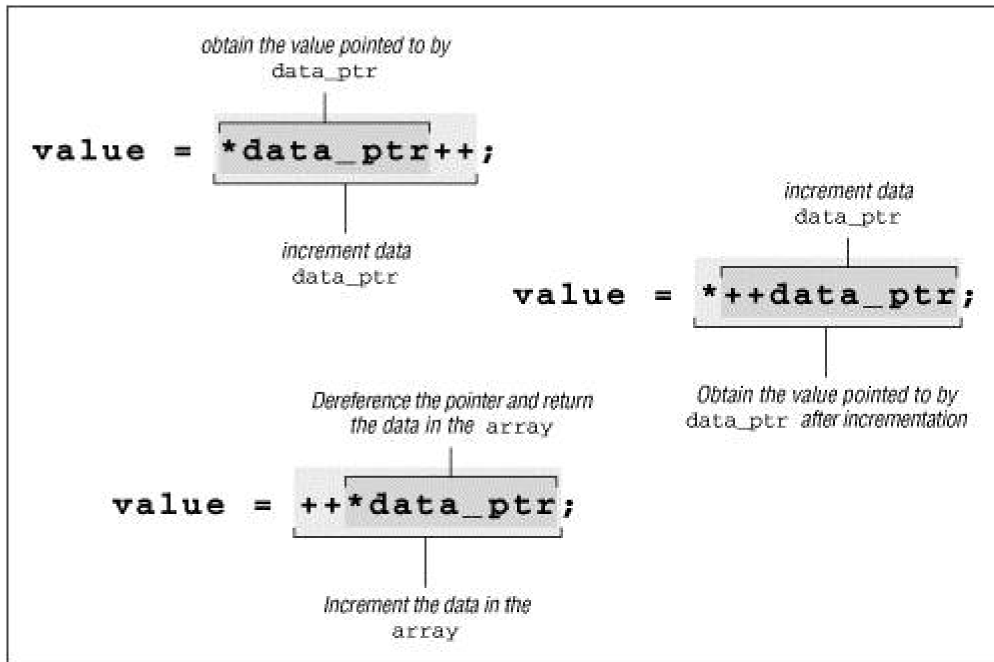
```
/* This program shows programming practices that should **NOT** be used */
/* Unfortunately, too many programmers use them */
int array[10];    /* An array for our data */
int main()
{
   int *data_ptr;   /* Pointer to the data */
   int value;       /* A data value */

   data_ptr = &array[0];/* Point to the first element */
   value = *data_ptr++;  /* Get element #0, data_ptr points to element #1 */
   value = *++data_ptr;  /* Get element #2, data_ptr points to element #2 */
   value = ++*data_ptr; /* Increment element #2, return its value */
                    /* Leave data_ptr alone */
```

To understand each of these statements, you must carefully dissect each expression to discover its hidden meaning. When I do maintenance programming, I don't want to have to worry about hidden meanings, so please don't code like this, and shoot anyone who does.

These statements are dissected in Figure 13-8.

Figure 13-8. Pointer operations dissected

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html



This example is a little extreme, but it illustrates how side effects can easily become confusing.

Example 13-8 is an example of the code you're more likely to run into. The program copies a string from the source (p) to the destination (q).

Example 13-8: Cryptic Use of Pointers

```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

Given time, a good programmer will decode this. However, understanding the program is much easier when we are a bit more verbose, as in Example 13-9.

Example 13-9: Readable Use of Pointers

```
/********************************************************
 * copy_string -- Copies one string to another.         *
 *                                                      *
 * Parameters                                           *
 *      dest -- Where to put the string                 *
 *      source -- Where to get it                       *
 ********************************************************/
void copy_string(char *dest, char *source)
{
    while (1) {
        *dest = *source;

        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;

        ++dest;
        ++source;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    }
}
```

Using Pointers to Split a String

Suppose we are given a string of the form "Last/First." We want to split this into two strings, one containing the first name and one containing the last name.

We need a function to find the slash in the name. The standard function strchr performs this job for us. In this program, we have chosen to duplicate this function to show you how it works.

This function takes a pointer to a string (string_ptr) and a character to find (find) as its arguments. It starts with a **while** loop that will continue until we find the character we are looking for (or we are stopped by some other code below).

```
        while (*string_ptr != find) {
```

Next we test to see if we've run out of string. In this case, our pointer (string_ptr) points to the end-of-string character. If we have reached the end of string before finding the character, we return NULL:

```
        if (*string_ptr == '\0')
           return (NULL);
```

If we get this far, we have not found what we are looking for, and are not at the end of the string. So we move the pointer to the next character, and return to the top of the loop to try again:

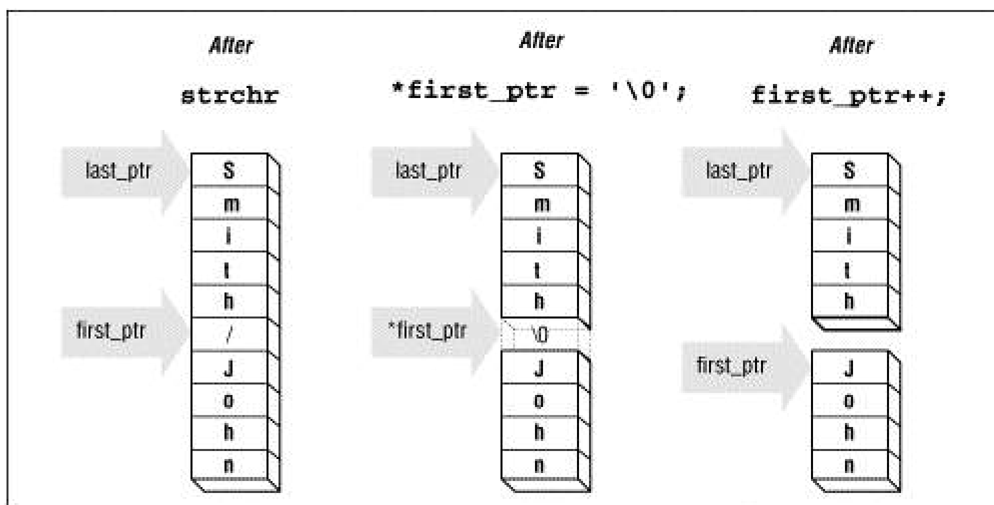
```
        ++string_ptr;
        }
```

Our main program reads in a single line, stripping the newline character from it. The function my_strchr is called to find the location of the slash (/).

At this point, last_ptr points to the first character of the last name and first_ptr points to slash. We then split the string by replacing the slash (/) with an end of string (NUL or \0). Now last_ptr points to just the last name and first_ptr points to a null string. Moving first_ptr to the next character makes it point to the beginning of the first name.

The sequence of steps in splitting the string is illustrated in Figure 13-9.

Figure 13-9. Splitting a string



Example 13-10 contains the full program, which demonstrates how pointers and character arrays can be used for simple string processing.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Example 13-10: split/split.c (continued)

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

/*******************************************************
 * my_strchr -- Finds a character in a string.         *
 *      Duplicate of a standard library function,      *
 *      put here for illustrative purposes             *
 *                                                     *
 * Parameters                                          *
 *      string_ptr -- String to look through.          *
 *      find -- Character to find.                     *
 *                                                     *
 * Returns                                             *
 *      pointer to 1st occurrence of character         *
 *      in string or NULL for error.                   *
 *******************************************************/
char *my_strchr(char * string_ptr, char find)
{
    while (*string_ptr != find) {

       /* Check for end */

       if (*string_ptr == '\0')
           return (NULL);       /* not found */

        ++string_ptr;
    }
    return (string_ptr);        /* Found */
}

int main()
{
    char line[80];      /* The input line */
    char *first_ptr;    /* pointer to the first name */
    char *last_ptr;     /* pointer to the last name */

    fgets(line, sizeof(line), stdin);

    /* Get rid of trailing newline */
    line[strlen(line)-1] = '\0';

    last_ptr = line;    /* last name is at beginning of line */

    first_ptr = my_strchr(line, '/');      /* Find slash */

    /* Check for an error */
    if (first_ptr == NULL) {
        fprintf(stderr,
            "Error: Unable to find slash in %s\n", line);
        exit (8);
    }

    *first_ptr = '\0';  /* Zero out the slash */
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
   ++first_ptr;        /* Move to first character of name */

   printf("First:%s Last:%s\n", first_ptr, last_ptr);
   return (0);
}
```

Question 13-2: Example 13-11 is supposed to print out:

```
Name: tmp1
```

but instead, we get:

```
Name: !_@$#ds80
```

(Your results may vary.) Why?

Example 13-11: tmp-name/tmp-name.c

```
#include <stdio.h>
#include <string.h>

/********************************************************
 * tmp_name -- Return a temporary filename.             *
 *                                                      *
 * Each time this function is called, a new name will   *
 * be returned.                                         *
 *                                                      *
 * Returns                                              *
 *      Pointer to the new filename.                    *
 ********************************************************/
char *tmp_name(void)
{
   char name[30];      /* The name we are generating */
   static int sequence = 0;    /* Sequence number for last digit */

   ++sequence; /* Move to the next filename */

   strcpy(name, "tmp");

   /* But in the sequence digit */
   name[3] = sequence + '0';

   /* End the string */
   name[4] = '\0';

   return(name);
}

int main()
{
   char *tmp_name(void);       /* Get name of temporary file */

   printf("Name: %s\n", tmp_name());
   return(0);
}
```

Pointers and Structures

In Chapter 12, *Advanced Types*, we defined a structure for a mailing list:

```
struct mailing {
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
        char name[60];    /* last name, first name */
        char address1[60];/* two lines of street address */
        char address2[60];
        char city[40];
        char state[2];    /* two-character abbreviation */
        long int zip;     /* numeric zip code */
    } list[MAX_ENTRIES];
```

Mailing lists must frequently be sorted by name and zip code. We could sort the entries themselves, but each entry is 226 bytes long. That's a lot of data to move around. One way around this problem is to declare an array of pointers, and then sort the pointers:

```
/* Pointer to the data */
struct mailing *list_ptrs[MAX_ENTRIES];
int current;    /* current mailing list entry */

    for (current = 0; current = number_of_entries; ++current)
        list_ptrs[current] = &list[current];
    /* Sort list_ptrs by zip code */
```

Now, instead of having to move a 226-byte structure around, we are moving 4-byte pointers. Our sorting is much faster. Imagine that you had a warehouse full of big heavy boxes and you needed to locate any box quickly. You could put them in alphabetical order, but that would require a lot of moving. Instead, you assign each location a number, write down the name and number on index cards, and sort the cards by name.

Command-Line Arguments

The procedure `main` actually takes two arguments. They are called `argc` and `argv`[2]:

```
main(int argc, char *argv[])
{
```

(If you realize that the arguments are in alphabetical order, you can easily remember which one comes first.)

The parameter `argc` is the number of arguments on the command line (including the program name). The array `argv` contains the actual arguments. For example, if the program `args` were run with the command line:

```
args this is a test
```

then:

```
   argc =    5
argv[0] =   "args"
argv[1] =   "this"
argv[2] =   "is"
argv[3] =   "a"
argv[4] =   "test"
argv[5] =   NULL
```

> **NOTE:** The UNIX shell expands wildcard characters like *, ?, and [ ] before sending the command line to the program. See your `sh` or `csh` manual for details.
>
> Turbo C++ and Borland C++ expand wildcard characters if the file *WILDARG.OBJ* is linked with your program. See the manual for details.

Almost all UNIX commands use a standard command-line format. This standard has carried over into other environments. A standard UNIX command has the form:

command options file1 file1 file3 ...

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Options are preceded by a dash (-) and are usually a single letter. For example, the option -v might turn on verbose mode for a particular command. If the option takes a parameter, it follows the letter. For example, the option -m1024 sets the maximum number of symbols to 1024 and -ooutfile sets the output filename to *outfile*.
Let's look at writing a program that can read the command-line arguments and act accordingly. This program formats and prints files. Part of the documentation for the program is given here:

```
print_file [-v] [-llength] [-oname] [file1] [file2] ...
```

where:

-v

specifies verbose options; turns on a lot of progress information messages

-llength

sets the page size to length lines (default = 66)

-oname

sets the output file to name (default = print.out)

file1, file2, ...

is a list of files to print. If no files are specified, the file print.in is printed.

We can use a **while** loop to cycle through the command-line options. The actual loop is:

```
while ((argc > 1) && (argv[1][0] == '-')) {
```

One argument always exists: the program name. The expression (argc > 1) checks for additional arguments. The first one is numbered 1. The first character of the first argument is argv[1][0]. If this is a dash, we have an option.
At the end of the loop is the code:

```
    --argc;
    ++argv;
}
```

This consumes an argument. The number of arguments is decremented to indicate one less option, and the pointer to the first option is incremented, shifting the list to the left one place. (Note: after the first increment, argv[0] no longer points to the program name.)
The **switch** statement is used to decode the options. Character 0 of the argument is the dash (-). Character 1 is the option character, so we use the expression:

```
switch (argv[1][1]) {
```

to decode the option.
The option -v has no arguments; it just causes a flag to be set.
The option -o takes a filename. Rather than copy the whole string, we set the character pointer out_file to point to the name part of the string. By this time we know the following:

```
argv[1][0]   ='-'
argv[1][1]   ='o'
argv[1][2]   = first character of the filename
```

We set out_file to point to the string with the statement:

```
out_file = &argv[1][2];
```

The address of operator (&) is used to get the address of the first character in the output filename. This process is appropriate because we are assigning the address to a character pointer named out_file.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

The -l option takes an integer argument. The library function atoi is used to convert the string into an integer. From the previous example, we know that argv[1][2] is the first character of the string containing the number. This string is passed to atoi.

Finally, all the options are parsed and we fall through to the processing loop. This merely executes the function do_file for each file argument. Example 13-12 contains the print program.

This is one way of parsing the argument list. The use of the **while** loop and **switch** statement is simple and easy to understand. This method does have a limitation. The argument must immediately follow the options. For example, -odata.out will work, but "-o data.out" will not. An improved parser would make the program more friendly, but the techniques described here work for simple programs.

Example 13-12: print/print.c (continued)

```
[File: print/print.c]
/********************************************************
 * Program: Print                                       *
 *                                                      *
 * Purpose:                                             *
 *     Formats files for printing.                      *
 *                                                      *
 * Usage:                                               *
 *     print [options] file(s)                          *
 *                                                      *
 * Options:                                             *
 *     -v          Produces verbose messages.           *
 *     -o<file>    Sends output to a file               *
 *                 (default=print.out).                 *
 *     -l<lines>   Sets the number of lines/page        *
 *                 (default=66).                        *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>

int verbose = 0;        /* verbose mode (default = false) */
char *out_file = "print.out";   /* output filename */
char *program_name;     /* name of the program (for errors) */
int line_max = 66;      /* number of lines per page */

/********************************************************
 * do_file -- Dummy routine to handle a file.           *
 *                                                      *
 * Parameter                                            *
 *     name -- Name of the file to print.               *
 ********************************************************/
void do_file(char *name)
{
   printf("Verbose %d Lines %d Input %s Output %s\n",
      verbose, line_max, name, out_file);
}
/********************************************************
 * usage -- Tells the user how to use this program and  *
 *          exit.                                       *
 ********************************************************/
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
void usage(void)
{
    fprintf(stderr,"Usage is %s [options] [file-list]\n",
                            program_name);
    fprintf(stderr,"Options\n");
    fprintf(stderr,"  -v          verbose\n");
    fprintf(stderr,"  -l<number>  Number of lines\n");
    fprintf(stderr,"  -o<name>    Set output filename\n");
    exit (8);
}
int main(int argc, char *argv[])
{
    /* save the program name for future use */
    program_name = argv[0];

    /*
     * loop for each option
     *   Stop if we run out of arguments
     *   or we get an argument without a dash
     */
    while ((argc > 1) && (argv[1][0] == '-')) {
        /*
         * argv[1][1] is the actual option character
         */
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
            /*
             * -o<name>  output file
             *    [0] is the dash
             *    [1] is the "o"
             *    [2] starts the name
             */
            case 'o':
                out_file = &argv[1][2];
                break;
            /*
             * -l<number> set max number of lines
             */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        /*
         * move the argument list up one
         * move the count down one
         */
        ++argv;
        --argc;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    }

    /*
     * At this point, all the options have been processed.
     * Check to see if we have no files in the list.
     * If no files exist, we need to process just standard input stream.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
          do_file(argv[1]);
          ++argv;
          --argc;
        }
    }
    return (0);
}
```

Programming Exercises

Exercise 13-1: Write a program that uses pointers to set each element of an array to zero.
Exercise 13-2: Write a function that takes a single string as its argument and returns a pointer to the first nonwhite character in the string.

Answers

Answer 13-1: The problem is that the variable name is a temporary variable. The compiler allocates space for the name when the function is entered and reclaims the space when the function exits. The function assigns name the correct value and returns a pointer to it. However, the function is over, so name disappears and we have a pointer with an illegal value.
The solution is to declare name **static**. In this manner, name is a permanent variable and will not disappear at the end of the function.
Question 13-2: After fixing the function, we try using it for two filenames. Example 13-13 should print out:

```
Name: tmp1
Name: tmp2
```

but it doesn't. What does it print and why?
Example 13-13: tmp2/tmp2.c

```
#include <stdio.h>
#include <string.h>

/********************************************************
 * tmp_name -- Returns a temporary filename.            *
 *                                                      *
 * Each time this function is called, a new name will   *
 * be returned.                                         *
 *                                                      *
 * Warning: There should be a warning here, but if we   *
 *     put it in, we would answer the question.         *
 *                                                      *
 * Returns                                              *
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
 *      Pointer to the new filename.                    *
 ******************************************************/
char *tmp_name(void)
{
    static char name[30];       /* The name we are generating */
    static int sequence = 0;    /* Sequence number for last digit */

    ++sequence; /* Move to the next filename */

    strcpy(name, "tmp");

    /* But in the squence digit */
    name[3] = sequence + '0';

    /* End the string */
    name[4] = '\0';

    return(name);
}

int main()
{
    char *tmp_name(void);       /* get name of temporary file */
    char *name1;                /* name of a temporary file */
    char *name2;                /* name of a temporary file */

    name1 = tmp_name();
    name2 = tmp_name();

    printf("Name1: %s\n", name1);
    printf("Name2: %s\n", name2);
    return(0);
}
```

Answer 13-2: The first call to tmp_name returns a pointer to name. There is only one name. The second call to tmp_name changes name and returns a pointer to it. So we have two pointers, and they point to the same thing, name.

Several library functions return pointers to **static** strings. A second call to one of these routines will overwrite the first value. A solution to this problem is to copy the values as shown below:

```
char name1[100];
char name2[100];
strcpy(name1, tmp_name());
strcpy(name2, tmp_name());
```

This problem is a good illustration of the basic meaning of a pointer; it doesn't create any new space for data, but just refers to data that is created elsewhere.

This problem is also a good example of a poorly designed function. The problem is that the function is tricky to use. A better design would make the code less risky to use. For example, the function could take an additional parameter: the string in which the filename is to be constructed:

```
void tmp_name(char *name_to_return);
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

1. This statement is not strictly true for MS-DOS/Windows compilers. Because of the strange architecture of the 8086, these compilers are forced to use both *near* pointers (16 bits) and *far* pointers (32 bits). See your C compiler manual for details.
2. Actually, they can be named anything. However, in 99.9% of programs, they are named argc and argv. When most programmers encounter the other 0.1%, they curse loudly, and then change the names to argc and argv.

UNDERSTANDING POINTERS (for beginners)

INTRODUCTION:

Over a period of several years of monitoring varioustelecommunication conferences on C I have noticed that one of themost difficult problems for beginners was the understanding ofpointers. After writing dozens of short messages in attempts toclear up various fuzzy aspects of dealing with pointers, I set upa series of messages arranged in "chapters" which I could drawfrom or email to various individuals who appeared to need help inthis area.

CHAPTER 1: What is a pointer?

One of the things beginners in C find most difficult tounderstand is the concept of pointers. The purpose of thisdocument is to provide an introduction to pointers and their useto these beginners.
I have found that often the main reason beginners have a problem with pointers is that they have a weak or minimal feelingfor variables, (as they are used in C). Thus we start with adiscussion of C variables in general.

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computerto hold the value of that variable. The size of that blockdepends on the range over which the variable is allowed to vary.For example, on PC's the size of an integer variable is 2 bytes,and that of a long integer is 4 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines.

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

```
int k;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

On seeing the "int" part of this statement the compiler sets aside 2 bytes (on a PC) of memory to hold the value of the integer. It also sets up a symbol table. And in that table it adds the symbol k and the address in memory where those 2 bytes were set aside.

Thus, later if we write:

```
k = 2;
```

at run time we expect that the value 2 will be placed in that memory location reserved for the storage of the value of k. In a sense there are two "values" associated with k, one being the value of the integer stored there (2 in the above example) and the other being the "value" of the memory location where it is stored, i.e. the address of k. Some texts refer to these two values with the nomenclature rvalue (right value, pronounced "are value") and lvalue (left value, pronunced "el value") respectively.

The lvalue is the value permitted on the left side of the assignment operator '=' (i.e. the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the as signment statment, the '2' above. Note that rvalues cannot be used on the left side of the assignment statement. Thus: 2 = k; is illegal.

Okay, now consider:

```
int j, k;
k = 2;
j = 7;    <-- line 1
k = j;    <-- line 2
```

In the above, the compiler interprets the j in line 1 as the address of the variable j (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the j is interpreted as its rvalue (since it is on the right hand side of the assignment operator '='). That is, here the j refers to the value _stored_ at the memory location set aside for j, in this case 7. So, the 7 is copied to the address designated by the lvalue of k.

In all of these examples, we are using 2 byte integers so all copying of rvalues from one storage location to the other is done by copying 2 bytes. Had we been using long integers, we would be copying 4 bytes.

Now, let's say that we have a reason for wanting a variable designed to hold an lvalue (an address). The size required to hold such a value depends on the system. On older desk top computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. Some computers, such as the IBM PC might require special handling to hold a segment and offset under certain circumstances. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Such a variable is called a "pointer variable" (for reasons which will hopefully become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk. In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable definition:

```
int *ptr;
```

ptr is the _name_ of our variable (just as 'k' was the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The "int" says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. Note, however, that when we wrote "int k;" we did not give k a value.If this definiton was made outside of any function many compilers will initialize it to zero.

Simlarly, ptr has no value, that is we haven't stored an address in it in the above definition. In this case, again if the definition is outside of any function, it is intialized to a value #defined by your compiler as NULL. It is called a NULL pointer. While in most cases NULL is #defined as zero, it need not be. That is, different compilers handle this differently. Also note that while zero is an integer, NULL need not be.

But, back to using our new variable ptr. Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary '&' operator and write:

```
ptr = &k;
```

What the '&' operator does is retrieve the lvalue (address) of k, even though k is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer ptr.

Now, ptr is said to "point to" k. Bear with us now, there isonly one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
*ptr = 7;
```

will copy 7 to the address pointed to by ptr. Thus if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are refering to the value of that which ptr is pointing at, not the value of the pointer itself.

Similarly, we could write:

```
printf("%d\n",*ptr);
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

to print to the screen the integer value stored at the address pointed to by "ptr".

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
------------------------------------------------
#include <stdio.h>

int j, k;
int *ptr;


int main(void)
{
  j = 1;
  k = 2;
  ptr = &k;
  printf("\n");
  printf("j has the value %d and is stored at %p\n",j,&j);
  printf("k has the value %d and is stored at %p\n",k,&k);
  printf("ptr has the value %p and is stored at %p\n",ptr,&ptr);
  printf("The value of the integer pointed to by ptr is %d\n",
         *ptr);
  return 0;
}
---------------------------------------
```

To review:

A variable is defined by giving it a type and a name (e.g. int k;)

A pointer variable is defined by giving it a type and a name (e.g. int *ptr) where the asterisk tells the compiler that the variable named ptr is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).

Once a variable is defined, we can get its address by preceding its name with the unary '&' operator, as in &k.

We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '*' operator as
in *ptr.

An "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

CHAPTER 2: Pointer types and Arrays

Okay, let's move on. Let us consider why we need to identify the "type" of variable that a pointer points to, as in:

```
int *ptr;
```

One reason for doing this is so that later, once ptr "points to" something, if we write:

```
*ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by ptr. If ptr was defined as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied. Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integer.

Now, let's say we point our integer pointer ptr at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write:

```
ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 2 to ptr instead of 1, so the pointer "points to" the _next_integer_, at memory location 102. Similarly, were the ptr defined as a pointer to a long, it would add 4 to it instead of

1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures.

Similarly, since ++ptr and ptr++ are both equivalent to

```
ptr + 1
```

(though the point in the program when ptr is incremented may be different), incrementing a pointer using the unary ++ operator, either pre- or post-, increments the address it stores by the amount sizeof(type) (i.e. 2 for an integer, 4 for a long, etc.).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array, i.e. using my_array[0] through my_array[5]. But, we could alternatively access them via a pointer as follows:

```
int *ptr;

ptr = &my_array[0];       /* point our pointer at the first
                             integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
-----------------------------------------------------
#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];     /* point our pointer to the array */
    printf("\n\n");
    for(i = 0; i < 6; i++)
    {
      printf("my_array[%d] = %d   ",i,my_array[i]);   /*<-- A */
      printf("ptr + %d = %d\n",i, *(ptr + i));        /*<-- B */
    }
    return 0;
}
-----------------------------------------------------
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also note how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the the new pointer. Change line B to read:

```
printf("ptr + %d = %d\n",i, *ptr++);
```

and run it again... then change it to:

```
printf("ptr + %d = %d\n",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name, thus in our code where we wrote:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
ptr = &my_array[0];
```

we can write:

```
ptr = my_array;     to achieve the same result.
```

This leads many texts to state that the name of an array is a pointer. While this is true, I prefer to mentally think "the name of the array is a _constant_ pointer". Many beginners(including myself when I was learning) forget that _constant_qualifier. In my opinon this leads to some confusion. For example, while we can write ptr = my_array; we cannot write

```
my_array = ptr;
```

The reason is that the while ptr is a variable, my_array is a constant. That is, the location at which the first element of my_array will be stored cannot be changed once my_array[] has been declared.

Modify the example program above by changing

```
ptr = &my_array[0];     to     ptr = my_array;
```

and run it again to verify the results are identical.

Now, let's delve a little further into the difference between the names "ptr" and "my_array" as used above. We said that my_array is a constant pointer. What do we mean by that? Well, to understand the term "constant" in this sense, let's go back to our definition of the term "variable". When we define a variable we set aside a spot in memory to hold the value of the appropriate type. Once that is done the name of the variable can be interpreted in one of two ways. When used on the left side of the assignment operator, the compiler interprets it as the memory location to which to move that which lies on the right side of the assignment operator. But, when used on the right side of the assignment operator, the name of a variable is interpreted to mean the contents stored at that memory address set aside to hold the value of that variable.

With that in mind, let's now consider the simplest of constants, as in:

```
int i, k;
i = 2;
```

Here, while "i" is a variable and then occupies space in the data portion of memory, "2" is a constant and, as such, instead of setting aside memory in the data segment, it is imbedded directly in the code segment of memory. That is, while writing something like k = i; tells the compiler to create code which at run time will look at memory location &i

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

to determine the value to be moved to k, code created by i = 2; simply puts the '2' in the code and there is no referencing of the data segment.

Similarly, in the above, since "my_array" is a constant, once the compiler establishes where the array itself is to be stored, it "knows" the address of my_array[0] and on seeing:

```
ptr = my_array;
```

it simply uses this address as a constant in the code segment and there is no referencing of the data segment beyond that.

Well, that's a lot of technical stuff to digest and I don't expect a beginner to understand all of it on first reading. With time and experimentation you will want to come back and re-read the first 2 chapters. But for now, let's move on to the relationship between pointers, character arrays, and strings.

==
===

CHAPTER 3: Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In Pascal or (most versions of) Basic, strings are treated differently from arrays. To start off our discussion we will write some code which, while preferred for illustrative purposes, you would probably never write in an actual program. Consider, for example:

```
char my_string[40];

my_string[0] = 'T';
my_string[1] = 'e';
my_string[2] = 'd':
my_string[3] = '\0';
```

While one would never build a string like this, the end result is a string in that it is an array of characters _terminated_with_a_nul_character_. By definition, in C, a string is an array of characters terminated with the nul character. Note that "nul" is _not_ the same as "NULL". The nul refers to a zero as is defined by the escape sequence '\0'. That is it occupies one byte of memory. The NULL, on the other hand, is the value of an uninitialized pointer and pointers require more than one byte of storage. NULL is defined in a header file in your C compiler, nul may not be #defined at all.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
char my_string[40] = {'T', 'e', 'd', '\0',};
```

But this also takes more typing than is convenient. So, C permits:

```
char my_string[40] = "Ted";
```

When the double quotes are used, instead of the single quotes as was done in the previous examples, the nul character ('\0') is automatically appended to the end of the string.

In all of the above cases, the same thing happens. The compiler sets aside an contiguous block of memory 40 bytes long to hold characters and initialized it such that the first 4 characters are Ted\0.

Now, consider the following program:

```
------------------program 3.1-------------------------------------
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void)
{
   char *pA;     /* a pointer to type character */
   char *pB;     /* another pointer to type character */
   puts(strA);   /* show string A */
   pA = strA;    /* point pA at string A */
   puts(pA);     /* show what pA is pointing to */
   pB = strB;    /* point pB at string B */
   putchar('\n');       /* move down one line on the screen */
   while(*pA != '\0')   /* line A (see text) */
   {
     *pB++ = *pA++;     /* line B (see text) */
   }
   *pB = '\0';          /* line C (see text) */
   puts(strB);          /* show strB on screen */
   return 0;
}
--------- end program 3.1 -------------------------------------
```

In the above we start out by defining two character arrays of 80 characters each. Since these are globally defined, they are initialized to all '\0's first. Then, strA has the first 42

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

characters initialized to the string in quotes.

Now, moving into the code, we define two character pointers and show the string on the screen. We then "point" the ponter pA at strA. That is, by means of the assignment statement we copy the address of strA[0] into our variable pA. We now use puts() to show that which is pointed to by pA on the screen. Consider here that the function prototype for puts() is:

```
int puts(const char *s);
```

For the moment, ignore the "const". The parameter passed to puts is a pointer, that is the _value_ of a pointer (since all parameters in C are passed by value), and the value of a pointer is the address to which it points, or, simply, an address. Thus
when we write:

    puts(strA);   as we have seen, we are passing the

address of strA[0]. Similarly, when we write:

    puts(pA);   we are passing the same address, since

we have set pA = strA;

Given that, follow the code down to the while() statement on line A. Line A states:

While the character pointed to by pA (i.e. *pA) is not a null character (i.e. the terminating '\0'), do the following:

line B states: copy the character pointed to by pA to the space pointed to by pB, then increment pA so it points to the next character and pB so it points to the next space.

Note that when we have copied the last character, pA now points to the terminating nul character and the loop ends. However, we have not copied the nul character. And, by definition a string in C _must_ be nul terminated. So, we add the nul character with line C.

It is very educational to run this program with your debugger while watching strA, strB, pA and pB and single stepping through the program. It is even more educational if instead of simply defining strB[] as has been done above, initialize it also with something like:

```
strB[80] = "12345678901234567890123456789012345678901234567890"
```

where the number of digits used is greater than the length of strA and then repeat the single stepping procedure while watching the above variables. Give these things a try!

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Of course, what the above program illustrates is a simple way of copying a string. After playing with the above until you have a good understanding of what is happening, we can proceed to creating our own replacement for the standard strcpy() that comes with C. It might look like:

```
char *my_strcpy(char *destination, char *source)
{
    char *p = destination
    while (*source != '\0')
    {
      *p++ = *source++;
    }
    *p = '\0';
    return destination.
}
```

In this case, I have followed the practice used in the standard routine of returning a pointer to the destination.

Again, the function is designed to accept the values of two character pointers, i.e. addresses, and thus in the previous program we could write:

```
int main(void)
{
  my_strcpy(strB, strA);
  puts(strB);
}
```

I have deviated slightly from the form used in standard C which would have the prototype:

```
char *my_strcpy(char *destination, const char *source);
```

Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. You can prove this by modifying the function above, and its prototype, to include the "const" modifier as shown. Then, within the function you can add a statement which attempts to change the contents of that which is pointed to by source, such as:

```
*source = 'X';
```

which would normally change the first character of the string to an X. The const modifier should cause your compiler to catch this as an error. Try it and see.

Now, let's consider some of the things the above examples have shown us. First off, consider the fact that *ptr++ is to be interpreted as returning the value pointed to by ptr

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

and then incrementing the pointer value. On the other hand, note that this has to do with the precedence of the operators. Were we to write (*ptr)++ we would increment, not the pointer, but that which the pointer points to! i.e. if used on the first character of the above example string the 'T' would be incremented to a 'U'. You can write some simple example code to illustrate this.

Recall again that a string is nothing more than an array of characters. What we have done above is deal with copying an array. It happens to be an array of characters but the technique could be applied to an array of integers, doubles, etc. In those cases, however, we would not be dealing with strings and hence the end of the array would not be _automatically_ marked with a special value like the nul character. We could implement a version that relied on a special value to identify the end. For example, we could copy an array of postive integers by marking the end with a negative integer. On the other hand, it is more usual that when we write a function to copy an array of items other than strings we pass the function the number of items to be copied as well as the address of the array, e.g. something like the following prototype might indicate:

```
void int_copy(int *ptrA, int *ptrB, int nbr);
```

where nbr is the number of integers to be copied. You might want to play with this idea and create an array of integers and see if you can write the function int_copy() and make it work. Note that this permits using functions to manipulate very large arrays. For example, if we have an array of 5000 integers that we want to manipulate with a function, we need only pass to that function the address of the array (and any auxiliary information such as nbr above, depending on what we are doing). The array itself does _not_ get passed, i.e. the whole array is not copied and put on the stack before calling the function, only its address is sent.

Note that this is different from passing, say an integer, to a function. When we pass an integer we make a copy of the integer, i.e. get its value and put it on the stack. Within the function any manipulation of the value passed can in no way effect the original integer. But, with arrays and pointers we can pass the address of the variable and hence anipulate the values of of the original variables.

==
===

CHAPTER 4: More on Strings

Well, we have progressed quite aways in a short time! Let's back up a little and look at what was done in Chapter 3 on copying of strings but in a different light. Consider the following function:

```
char *my_strcpy(char dest[], char source[])
{
    int i = 0;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    while (source[i] != '\0')
    {
      dest[i] = source[i];
      i++;
    }
    dest[i] = '\0';
    return dest;
}
```

Recall that strings are arrays of characters. Here we have chosen to use array notation instead of pointer notation to do the actual copying. The results are the same, i.e. the string gets copied using this notation just as accurately as it did before. This raises some interesting points which we will discuss.

Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array. Thus, the numerical value of the parameter passed is the same whether we use a character pointer or an array name as a parameter. This would tend to imply that somehow:

```
source[i]  is the same as  *(p+i);
```

In fact, this is true, i.e wherever one writes a[i] it can be replaced with *(a + i) without any problems. In fact, the compiler will create the same code in either case. Now,looking
at this last expression, part of it.. (a + i) is a simple addition using the + operator and the rules of c state that such an expression is commutative. That is (a + i) is identical to
(i + a). Thus we could write *(i + a) just as easily as *(a + i).

But *(i + a) could have come from i[a] ! From all of this comes the curious truth that if:

```
char a[20];
int i;
```

writing a[3] = 'x'; is the same as writing

```
3[a] = 'x';
```

Try it! Set up an array of characters, integers or longs,
etc. and assigned the 3rd or 4th element a value using the conventional approach and then print out that value to be sure you have that working. Then reverse the array notation as I have done above. A good compiler will not balk and the results will be identical. A curiosity... nothing more!

Now, looking at our function above, when we write:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
dest[i] = source[i];
```

this gets interpreted by C to read:

```
*(dest + i) = *(source + i);
```

But, this takes 2 additions for each value taken on by i. Additions, generally speaking, take more time than incrementations (such as those done using the ++ operator as in i++). This may not be true in modern optimizing compilers, but one can never be sure. Thus, the pointer version may be a bit faster than the array version.

Another way to speed up the pointer version would be to change:

while (*source != '\0')   to simply   while (*source)

since the value within the parenthesis will go to zero (FALSE) at the same time in either case.

At this point you might want to experiment a bit with writing some of your own programs using pointers. Manipulating strings is a good place to experiment. You might want to write your own versions of such standard functions as:

```
strlen();
strcat();
strchr();
```

and any others you might have on your system.

We will come back to strings and their manipulation through pointers in a future chapter. For now, let's move on and discuss structures for a bit.

==

CHAPTER 5: Pointers and Structures

As you may know, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
struct tag{
    char lname[20];       /* last name */
    char fname[20];       /* first name */
    int age;              /* age */
    float rate;           /* e.g. 12.75 per hour */
    };
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Let's say we have an bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

```
--------------- program 5.1 ------------------
#include <stdio.h>
#include <string.h>

struct tag{
      char lname[20];      /* last name */
      char fname[20];      /* first name */
      int age;             /* age */
      float rate;          /* e.g. 12.75 per hour */
      };

struct tag my_struct;       /* declare the structure m_struct */

int main(void)
{
  strcpy(my_struct.lname,"Jensen");
  strcpy(my_struct.fname,"Ted");
  printf("\n%s ",my_struct.fname);
  printf("%s\n",my_struct.lname);
  return 0;
}
-------------- end of program 5.1 --------------
```

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```
  date_of_hire;
  date_of_last_raise;
  last_percent_increase;
  emergency_phone;
  medical_plan;
  Social_S_Nbr;
  etc.....
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Now, if we have a large number of employees, what we want to do manipulate the data in these structures by means of functions. For example we might want a function print out the name of any structure passed to it. However, in the original C (Kernighan & Ritchie) it was not possible to pass a structure, only a pointer to a structure could be passed. In ANSI C, it is now permissible to pass the complete structure. But, since our goal here is to learn more about pointers, we won't pursue that.

Anyway, if we pass the whole structure it means there must be enough room on the stack to hold it. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure.

For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We define such a pointer with the definition:

```
struct tag *st_ptr;
```

and we point it to our example structure with:

```
st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure?

Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
(*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which st_ptr points to, which is the structure my_struct. Thus, this breaks down to the same as my_struct.age.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
st_ptr->age = 63;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

With that in mind, look at the following program:

```
------------ program 5.2 --------------

#include <stdio.h>
#include <string.h>

struct tag{                    /* the structure type */
       char lname[20];        /* last name */
       char fname[20];        /* first name */
       int age;               /* age */
       float rate;            /* e.g. 12.75 per hour */
       };

struct tag my_struct;         /* define the structure */

void show_name(struct tag *p);    /* function prototype */

int main(void)
{
  struct tag *st_ptr;         /* a pointer to a structure */
  st_ptr = &my_struct;        /* point the pointer to my_struct */
  strcpy(my_struct.lname,"Jensen");
  strcpy(my_struct.fname,"Ted");
  printf("\n%s ",my_struct.fname);
  printf("%s\n",my_struct.lname);
  my_struct.age = 63;
  show_name(st_ptr);          /* pass the pointer */
  return 0;
}


void show_name(struct tag *p)
{
  printf("\n%s ", p->fname);     /* p points to a structure */
  printf("%s ", p->lname);
  printf("%d\n", p->age);
}
-------------------- end of program 5.2 ----------------
```

Again, this is a lot of information to absorb at one time. The reader should compile and run the various code snippets and using a debugger monitor things like my_struct and p while single stepping through the main and following the code down into the function to see what is happening.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

==
===

CHAPTER 6: Some more on Strings, and Arrays of Strings

Well, let's go back to strings for a bit. In the following all assignments are to be understood as being global, i.e. made outside of any function, including main.

We pointed out in an earlier chapter that we could write:

```
char my_string[40] = "Ted";
```

which would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating '\0'.

Actually, if all we wanted to do was store the name "Ted" we could write:

```
char my_name[] = "Ted";
```

and the compiler would count the characters, leave room for the nul character and store the total of the four characters in memory the location of which would be returned by the array name, in this case my_string.

In some code, instead of the above, you might see:

```
char *my_name = "Ted";
```

which is an alternate approach. Is there a difference between these? The answer is.. yes. Using the array notation 4 bytes of storage in the static memory block are taken up, one for each character and one for the nul character. But, in the pointer notation the same 4 bytes required, _plus_ N bytes to store the pointer variable my_name (where N depends on the system but is usually a minimum of 2 bytes and can be 4 or more).

In the array notation, my_name is a constant (not a variable). In the pointer notation my_name is a variable. As to which is the _better_ method, that depends on what you are going to do within the rest of the program.

Let's now go one step further and consider what happens if each of these definitions are done within a function as opposed to globally outside the bounds of any function.

```
void my_function_A(char *ptr)
{
  char a[] = "ABCDE";
  .
  .
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
}

void my_function_B(char *ptr)
{
  char *cp = "ABCDE";
  .
  .
}
```

Here we are dealing with automatic variables in both cases. In my_function_A the automatic variable is the character array a[]. In my_function_B it is the pointer cp. While C is designed in such a way that a stack is not required on those processors which don't use them, my particular processor (80286) has a stack. I wrote a simple program incorporating functions similar to those above and found that in my_function_A the 5 characters in the string were all stored on the stack. On the other hand, in y_function_B, the 5 characters were stored in the data space and the pointer was stored on the stack.

By making a[] static I could force the compiler to place the 5 characters in the data space as opposed to the stack. I did this exercise to point out just one more difference between dealing with arrays and dealing with pointers. By the way, array initialization of automatic variables as I have done in my_function_A was illegal in the older K&R C and only "came of age" in the newer ANSI C. A fact that may be important when one is considering portabilty and backwards compatability.

As long as we are discussing the relationship/differences between pointers and arrays, let's move on to multi-dimensional arrays. Consider, for example the array:

```
char multi[5][10];
```

Just what does this mean? Well, let's consider it in the following light.

```
  char multi[5][10];
  ^^^^^^^^^^^^^
```

If we take the first, underlined, part above and consider it to be a variable in its own right, we have an array of 10 characters with the "name" multi[5]. But this name, in itself, implies an array of 5 somethings. In fact, it means an array of five 10 character arrays. Hence we have an array of arrays. In memory we might think of this as looking like:

```
multi[0] = "0123456789"
multi[1] = "abcdefghij"
multi[2] = "ABCDEFGHIJ"
multi[3] = "9876543210"
multi[4] = "JIHGFEDCBA"
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

with individual elements being, for example:

```
multi[0][3] = '3'
multi[1][7] = 'h'
multi[4][0] = 'J'
```

Since arrays are to be contiguous, our actual memory block for the above should look like:

"0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA"

Now, the compiler knows how many columns are present in the array so it can interpret multi + 1 as the address of the 'a' in the 2nd row above. That is, it adds 10, the number of columns, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add 10*sizeof(int) which, on my machine, would be 20. Thus, the address of the "9" in the 4th row above would be &multi[3][0] or *(multi + 3) in pointer notation. To get to the content of the 2nd element in row 3 we add 1 to this address and dereference the result as in

```
*(*(multi + 3) + 1)
```

With a little thought we can see that:

```
*(*(multi + row) + col)    and
multi[row][col]            yield the same results.
```

The following program illustrates this using integer arrays instead of character arrays.

```
------------------- program 6.1 ----------------------
#include <stdio.h>

#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
  int row, col;
  for (row = 0; row < ROWS; row++)
    for(col = 0; col < COLS; col++)
      multi[row][col] = row*col;
  for (row = 0; row < ROWS; row++)
    for(col = 0; col < COLS; col++)
    {
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
     printf("\n%d  ",multi[row][col]);
     printf("%d ",*(*(multi + row) + col));
    }
  return 0;
}
```

---------------- end of program 6.1 ---------------------

Because of the double de-referencing required in the pointer version, the name of a 2 dimensional array is said to be a pointer to a pointer. With a three dimensional array we would be dealing with an array of arrays of arrays and a pointer to a pointer to a pointer. Note, however, that here we have initially set aside the block of memory for the array by defining it using array notation. Hence, we are dealing with an constant, not a variable. That is we are talking about a fixed pointer not a
variable pointer. The dereferencing function used above permits us to access any element in the array of arrays without the need of changing the value of that pointer (the address of multi[0][0]
as given by the symbol "multi").

EPILOG:

I have written the preceding material to provide an introduction to pointers for newcomers to C. In C, the more one understands about pointers the greater flexibility one has in the writing of code. The above has just scratched the surface of the subject. In time I hope to expand on this material. Therefore, if you have questions, comments, criticisms, etc. concerning that which has been presented, I would greatly appreciate your contacting me using one of the mail addresses cited in the
Introduction.

Q: Okay, I'm kinda new to C, and I was reading that this following example

Q: would not be good to do: main() {

Q: int *iptr;

Q: *iptr = 421;

Q: printf("*iptr = %d\n",*iptr);

Q: }

Q: It was saying that you could get away with it, but in larger programs it

Q: can be a serious problem. It says that it is an uninitialized pointer.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

A: Contrary to what they (the book) told you, you CANNOT get away with uninitialized pointers, period.

Q: How do you initialize pointers?

A: Some insights:

1. A variable, any variable, has, amoing others, two properties called the rvalue and the lvalue. 'l' and 'r' stand for 'left' and 'right'. What is the meaning of these properties. Consider assignment:

```
int l = 2;
int r = 3;

l = r;     <----
```

Conceptually speaking, what basiccaly happens is that the compiler takes the address of 'r' and retrieves the value stored at that address. To obtain the address, it uses the rvalue of 'r'. Now it is clear that rvalues are used at the right hand side of the assignment operator to obtain the address to use.

Then, the value retrieved from 'r' is put in the lvalue of 'l', then, 'l'-s rvalue is used to obtain the address where to store that value.

Definitions:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

rvalue   the attribute of a variable that holds the address where
    that particular variable is stored.

lvalue   the attribute of a variable that holds the value of the
    variable.

Conclusions:

- If you never assign a value to a variables lvalue, that lvalue is undefined!
- The effect of using undefined lvalues is undefined, possibly harmful, and sometimes interesting ;-)
- A variable is a tuple(address, value). See "Aside".
- Assigning to rvalues is the job of the compiler.
- Assigning to lvalues is the job of the programmer.

Aside:

In fact, a variable is tuple(storage, scope, type, address, value):

storage : where is it stored, for example data, stack, heap...

scope : who can see us, for example global, local...

type : what is our type, for example int, int*...

address : where are we located

value : what is our value

2. A pointer is not a second class citizen. It has exactly the same proper-

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

ties as any other variable. The fun thing is that a pointers lvalue is actually the rvalue of another variable, namely the variable it points to. That means that before we can use that lvalue, we first must assign a correct value to it, because if we do not, that lvalue is undefined. That is where the referencing operator '&'. comes into play. Consider:

```
int  v = 3;
int* p;

p = &v;        <----
```

What the &-operator does is a modification of what happens at the left side of the assignment. Basically it tells the compiler not to use 'v'-s rvalue to obtain the address where the lvalue of 'v' is stored, but it tells it to use the rvalue of 'v' as the right hand side of the assignment. It then proceeds as normal, assigning the value obtained to the lvalue of p and storing that at the address contained in the rvalue of p.

What we have now is a p with an rvalue that is equal to the address where p is stored, and an lvalue that is equal to the address where 'v' is stored. Bingo! We initialized a pointer.

Conclusion:

- The meaning of the symbol '&' in the context of a variable is

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

"take-the-address-instead-of-the-value"

3. Now, all we need is a method to obtain the lvalue of 'v' through the pointer p. That is where dereferencing operator '*' comes into play. Consider:

```
int   n;
int   v = 3;
int*  p = &v;

n = *p;        <----
```

Now what happens, is that the *-operator tells the compiler to use the lvalue of p to use as an rvalue to obtain the proper lvalue. What happens exactly is:

- Take the rvalue of p.
- Obtain the lvalue of p.
- Use the lvalue of p as an rvalue to obtain the lvalue of the variable pointed to.

We call this process "dereferencing", that is, the pointer refers to another variable (possibly another pointer) and we follow that reference to arrive at the place we want to be.

Conclusion:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

- The meaning of the symbol '*' in the context of pointers is "use-my-value-as-address".

Well, the answer is there. To intialize a pointer, we must point it to another variable by means of the &-operator. To obtain the value of the variable pointed to, we use the *-operator. However, keep in mind that operators & and * can have a different meaning in other contexts (notably bitwise AND and multiplication).

TABLE OF CONTENTS

Preface

Introduction

Chapter 1: What is a Pointer?

Chapter 2: Pointer Types and Arrays.

Chapter 3: Pointers and Strings

Chapter 4: More on Strings

Chapter 5: Pointers and Structures

Chapter 6: More on Strings and Arrays of Strings

Chapter 7: More on Multi-Dimensional Arrays

Chapter 8: Pointers to Arrays

Chapter 9: Pointers and Dynamic Allocation of Memory

Chapter 10: Pointers to Functions

Epilog

->==

PREFACE

This document is intended to introduce pointers to beginning programmers in the C programming language. Over several years of reading and contributing to various conferences on C including those on the FidoNet and UseNet, I have noted a large number of newcomers to C appear to have a difficult time in grasping the fundamentals of pointers. I therefore undertook the task of

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

trying to explain them in plain language with lots of examples.

The first version of this document was placed in the public domain, as is this one. It was picked up by Bob Stout who included it as a file called PTR-HELP.TXT in his widely distributed collection of SNIPPETS. Since that release, I have added a significant amount of material and made some minor corrections in the original work.

Acknowledgements:

There are so many people who have unknowingly contributed to this work because of the questions they have posed in the FidoNet C Echo, or the UseNet Newsgroup comp.lang.c, or several other conferences in other networks, that it would be impossible to list them all. Special thanks go to Bob Stout who was kind enough to include the first version of this material in his SNIPPETS file.

About the Author:

Ted Jensen is a retired Electronics Engineer who worked as a hardware designer or manager of hardware designers in the field of magnetic recording. Programming has been a hobby of his off and on since 1968 when he learned how to keypunch cards for submission to be run on a mainframe. (The mainframe had 64K of magnetic core memory!).

Use of this Material:

Everything contained herein is hereby released to the Public Domain. Any person may copy or distribute this material in any manner they wish. The only thing I ask is that if this material is used as a teaching aid in a class, I would appreciate it if it were distributed in its entirety, i.e. including all chapters, the preface and the introduction. I would also appreciate it if under such circumstances the instructor of such a class would drop me a note at one of the addresses below informing me of this. I have written this with the hope that it will be useful to others and since I'm not asking any financial remuneration, the only way I know that I have at least partially reached that goal is via feedback from those how find this material useful.

By the way, you needn't be an instructor or teacher to contact me. I would appreciate a note from _anyone_ who finds the material useful, or who has constructive criticism to offer. I'm also willing to answer questions submitted by mail.

Ted Jensen tjensen@netcom.com
P.O. Box 324 1-415-365-8452
Redwood City, CA 94064
Dec. 1995

->==
INTRODUCTION

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

If one is to be proficient in the writing of code in the C programming language, one must have a thorough working knowledge of how to use pointers. Unfortunately, C pointers appear to represent a stumbling block to newcomers, particularly those coming from other computer languages such as Fortran, Pascal or Basic.

To aid those newcomers in the understanding of pointers I have written the following material. To get the maximum benefit from this material, I feel it is important that the user be able to run the code in the various listings contained in the article. I have attempted, therefore, to keep all code ANSI compliant so that it will work with any ANSI compliant compiler. And I have tried to carefully block the code within the text so that with the help of an ASCII text editor one can copy a given block of code to a new file and compile it on their system. I recommend that readers do this as it will help in understanding the material.

->==

CHAPTER 1: What is a pointer?

One of the things beginners in C find most difficult to understand is the concept of pointers. The purpose of this document is to provide an introduction to pointers and their use to these beginners.

I have found that often the main reason beginners have a problem with pointers is that they have a weak or minimal feeling for variables, (as they are used in C). Thus we start with a discussion of C variables in general.

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on PC's the size of an integer variable is 2 bytes, and that of a long integer is 4 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines.

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

```
int k;
```

On seeing the "int" part of this statement the compiler sets aside 2 bytes (on a PC) of memory to hold the value of the integer. It also sets up a symbol table. And in that table it adds the symbol k and the relative address in memory where those 2 bytes were set aside.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Thus, later if we write:

```
k = 2;
```

at run time we expect that the value 2 will be placed in that memory location reserved for the storage of the value of k. In C we refer to a variable such as the integer k as an "object".

In a sense there are two "values" associated with the object k, one being the value of the integer stored there (2 in the above example) and the other being the "value" of the memory location where it is stored, i.e. the address of k. Some texts refer to these two values with the nomenclature rvalue (right value, pronounced "are value") and lvalue (left value, pronounced "el value") respectively.

In some languages, the lvalue is the value permitted on the left side of the assignment operator '=' (i.e. the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the assignment statement, the '2' above. Rvalues cannot be used on the left side of the assignment statement. Thus: 2 = k; is illegal.

Actually, the above definition of "lvalue" is somewhat modified for C. According to K&R-2 (page 197): [1]

"An _object_ is a named region of storage; an _lvalue_ is an expression referring to an object."

However, at this point, the definition originally cited above is sufficient. As we become more familiar with pointers we will go into more detail on this.

Okay, now consider:

```
int j, k;
k = 2;
j = 7;    <-- line 1
k = j;    <-- line 2
```

In the above, the compiler interprets the j in line 1 as the address of the variable j (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the j is interpreted as its rvalue (since it is on the right hand side of the assignment operator '='). That is, here the j refers to the value _stored_ at the memory location set aside for j, in this case 7. So, the 7 is copied to the address designated by the lvalue of k.

In all of these examples, we are using 2 byte integers so all copying of rvalues from one storage location to the other is done by copying 2 bytes. Had we been using long integers, we would be copying 4 bytes.

Now, let's say that we have a reason for wanting a variable

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

designed to hold an lvalue (an address). The size required to
hold such a value depends on the system. On older desk top
computers with 64K of memory total, the address of any point in
memory can be contained in 2 bytes. Computers with more memory
would require more bytes to hold an address. Some computers,
such as the IBM PC might require special handling to hold a
segment and offset under certain circumstances. The actual size
required is not too important so long as we have a way of
informing the compiler that what we want to store is an address.

Such a variable is called a "pointer variable" (for reasons
which hopefully will become clearer a little later). In C when
we define a pointer variable we do so by preceding its name with
an asterisk. In C we also give our pointer a type which, in this
case, refers to the type of data stored at the address we will be
storing in our pointer. For example, consider the variable
declaration:

```
int *ptr;
```

ptr is the _name_ of our variable (just as 'k' was the name
of our integer variable). The '*' informs the compiler that we
want a pointer variable, i.e. to set aside however many bytes is
required to store an address in memory. The "int" says that we
intend to use our pointer variable to store the address of an
integer. Such a pointer is said to "point to" an integer.
However, note that when we wrote "int k;" we did not give k a value.
If this definition was made outside of any function many compilers
will initialize it to zero. Similarly, ptr has no value, that is
we haven't stored an address in it in the above declaration. In
this case, again if the declaration is outside of any function,
it is initialized to a value #defined by your compiler as NULL. It
is called a NULL pointer. While in most cases NULL is #defined
as zero, it need not be. That is, different compilers handle
this differently. Also while zero is an integer, NULL
need not be. However, the value that NULL actually has
internally is of little consequence to the programmer since at
the source code level NULL == 0 is guaranteed to evaluate to
true regardless of the internal value of NULL.

But, back to using our new variable ptr. Suppose now that we
want to store in ptr the address of our integer variable k. To
do this we use the unary '&' operator and write:

```
ptr = &k;
```

What the '&' operator does is retrieve the lvalue (address)
of k, even though k is on the right hand side of the assignment
operator '=', and copies that to the contents of our pointer ptr.
Now, ptr is said to "point to" k. Bear with us now, there is
only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used
as follows:

```
*ptr = 7;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

will copy 7 to the address pointed to by ptr. Thus if ptr "points to" (contains the address of) k, the above statement will set the value of k to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
printf("%d\n",*ptr);
```

to print to the screen the integer value stored at the address pointed to by "ptr".

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
------------------------------------------------
#include <stdio.h>

int j, k;
int *ptr;


int main(void)
{
  j = 1;
  k = 2;
  ptr = &k;
  printf("\n");
  printf("j has the value %d and is stored at %p\n",j,&j);
  printf("k has the value %d and is stored at %p\n",k,&k);
  printf("ptr has the value %p and is stored at %p\n",ptr,&ptr);
  printf("The value of the integer pointed to by ptr is %d\n",
         *ptr);
  return 0;
}
---------------------------------------
```

To review:

A variable is declared by giving it a type and a name (e.g. int k;)

A pointer variable is declared by giving it a type and a name (e.g. int *ptr) where the asterisk tells the compiler that the variable named ptr is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).

Once a variable is declared, we can get its address by preceding its name with the unary '&' operator, as in &k.

We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '*' operator as in *ptr.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

An "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).

References in Chapter 1:

[1] "The C Programming Language" 2nd Edition
B. Kernighan and D. Ritchie
Prentice Hall
ISBN 0-13-110362-8

->==

CHAPTER 2: Pointer types and Arrays

Okay, let's move on. Let us consider why we need to identify the "type" of variable that a pointer points to, as in:

```
int *ptr;
```

One reason for doing this is so that later, once ptr "points to" something, if we write:

```
*ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by ptr. If ptr was declared as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied. Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer ptr at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write:

```
ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 2 to ptr instead of 1, so the pointer "points to" the _next_ _integer_, at memory location 102. Similarly, were the ptr declared as a pointer to a long, it would add 4 to it instead of 1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic", a term which we will come back to later.

Similarly, since ++ptr and ptr++ are both equivalent to ptr + 1 (though the point in the program when ptr is incremented may be different), incrementing a pointer using the unary ++

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

operator, either pre- or post-, increments the address it stores by the amount sizeof(type) where "type" is the type of the object pointed to. (i.e. 2 for an integer, 4 for a long, etc.).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array, i.e. using my_array[0] through my_array[5]. But, we could alternatively access them via a pointer as follows:

```
int *ptr;

ptr = &my_array[0];       /* point our pointer at the first
                             integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
-----------------------------------------------------
#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
   int i;
   ptr = &my_array[0];     /* point our pointer to the first
                              element of the array */
   printf("\n\n");
   for(i = 0; i < 6; i++)
   {
     printf("my_array[%d] = %d   ",i,my_array[i]);   /*<-- A */
     printf("ptr + %d = %d\n",i, *(ptr + i));        /*<-- B */
   }
   return 0;
}
---------------------------------------------------
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the new pointer. Change line B to read:

```
    printf("ptr + %d = %d\n",i, *ptr++);
```

and run it again... then change it to:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
printf("ptr + %d = %d\n",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

```
ptr = my_array;     to achieve the same result.
```

This leads many texts to state that the name of an array is a pointer. While this is true, I prefer to mentally think "the name of the array is the address of first element in the array". Many beginners (including myself when I was learning) have a tendency to become confused by thinking of it as a pointer. For example, while we can write ptr = my_array; we cannot write

```
my_array = ptr;
```

The reason is that the while ptr is a variable, my_array is a constant. That is, the location at which the first element of my_array will be stored cannot be changed once my_array[] has been declared.

Earlier when discussing the term "lvalue" I cited K&R-2 where it stated:

"An _object_ is a named region of storage; an _lvalue_ is an expression referring to an object".

This raises an interesting problem. Since my_array is a named region of storage, why is "my_array" in the above assignment statement not an lvalue? To resolve this problem, some refer to "my_array" as an "unmodifiable lvalue".

Modify the example program above by changing

```
ptr = &my_array[0];     to     ptr = my_array;
```

and run it again to verify the results are identical.

Now, let's delve a little further into the difference between the names "ptr" and "my_array" as used above. Some writers will refer to an array's name as a _constant_ pointer. What do we mean by that? Well, to understand the term "constant" in this sense, let's go back to our definition of the term "variable". When we declare a variable we set aside a spot in memory to hold the value of the appropriate type. Once that is done the name of the variable can be interpreted in one of two ways. When used on the left side of the assignment operator, the compiler interprets it as the memory location to which to move that value resulting

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

from evaluation of the right side of the assignment operator. But, when used on the right side of the assignment operator, the name of a variable is interpreted to mean the contents stored at that memory address set aside to hold the value of that variable.

With that in mind, let's now consider the simplest of constants, as in:

```
int i, k;
i = 2;
```

Here, while "i" is a variable and then occupies space in the data portion of memory, "2" is a constant and, as such, instead of setting aside memory in the data segment, it is imbedded directly in the code segment of memory. That is, while writing something like k = i; tells the compiler to create code which at run time will look at memory location &i to determine the value to be moved to k, code created by i = 2; simply puts the '2' in the code and there is no referencing of the data segment. That is, both k and i are objects, but 2 is not an object.

Similarly, in the above, since "my_array" is a constant, once the compiler establishes where the array itself is to be stored, it "knows" the address of my_array[0] and on seeing:

```
ptr = my_array;
```

it simply uses this address as a constant in the code segment and there is no referencing of the data segment beyond that.

Well, that's a lot of technical stuff to digest and I don't expect a beginner to understand all of it on first reading. With time and experimentation you will want to come back and re-read the first 2 chapters. But for now, let's move on to the relationship between pointers, character arrays, and strings.

->==

CHAPTER 3: Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In BASIC, Pascal, Fortran and various other languages, a string has its own data type. But in C it does not. In C a string is an array of characters terminated with a binary zero character (written as '\0'). To start off our discussion we will write some code which, while preferred for illustrative purposes, you would probably never write in an actual program. Consider, for example:

```
char my_string[40];
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    my_string[0] = 'T';
    my_string[1] = 'e';
    my_string[2] = 'd':
    my_string[3] = '\0';
```

While one would never build a string like this, the end result is a string in that it is an array of characters _terminated_with_a_nul_character_. By definition, in C, a string is an array of characters terminated with the nul character. Be aware that "nul" is _not_ the same as "NULL". The nul refers to a zero as is defined by the escape sequence '\0'. That is it occupies one byte of memory. The NULL, on the other hand, is the value of an uninitialized pointer and pointers require more than one byte of storage. NULL is #defined in a header file in your C compiler, nul may not be #defined at all.

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
    char my_string[40] = {'T', 'e', 'd', '\0',};
```

But this also takes more typing than is convenient. So, C permits:

```
    char my_string[40] = "Ted";
```

When the double quotes are used, instead of the single quotes as was done in the previous examples, the nul character ('\0') is automatically appended to the end of the string.

In all of the above cases, the same thing happens. The compiler sets aside an contiguous block of memory 40 bytes long to hold characters and initialized it such that the first 4 characters are Ted\0.

Now, consider the following program:

```
------------------program 3.1-------------------------------------
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void)
{
   char *pA;     /* a pointer to type character */
   char *pB;     /* another pointer to type character */
   puts(strA);   /* show string A */
   pA = strA;    /* point pA at string A */
   puts(pA);     /* show what pA is pointing to */
   pB = strB;    /* point pB at string B */
   putchar('\n');       /* move down one line on the screen */
   while(*pA != '\0')   /* line A (see text) */
   {
     *pB++ = *pA++;     /* line B (see text) */
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    }
    *pB = '\0';          /* line C (see text) */
   puts(strB);          /* show strB on screen */
   return 0;
}
--------- end program 3.1 -------------------------------------
```

In the above we start out by defining two character arrays of 80 characters each. Since these are globally defined, they are initialized to all '\0's first. Then, strA has the first 42 characters initialized to the string in quotes.

Now, moving into the code, we declare two character pointers and show the string on the screen. We then "point" the pointer pA at strA. That is, by means of the assignment statement we copy the address of strA[0] into our variable pA. We now use puts() to show that which is pointed to by pA on the screen. Consider here that the function prototype for puts() is:

```
int puts(const char *s);
```

For the moment, ignore the "const". The parameter passed to puts is a pointer, that is the _value_ of a pointer (since all parameters in C are passed by value), and the value of a pointer is the address to which it points, or, simply, an address. Thus when we write:

```
puts(strA);
```

as we have seen, we are passing the

address of strA[0]. Similarly, when we write:

```
puts(pA);
```

we are passing the same address, since

we have set pA = strA;

Given that, follow the code down to the while() statement on line A. Line A states:

While the character pointed to by pA (i.e. *pA) is not a nul character (i.e. the terminating '\0'), do the following:

line B states: copy the character pointed to by pA to the space pointed to by pB, then increment pA so it points to the next character and pB so it points to the next space.

When we have copied the last character, pA now points to the terminating nul character and the loop ends. However, we have not copied the nul character. And, by definition a string in C _must_ be nul terminated. So, we add the nul character with line C.

It is very educational to run this program with your debugger while watching strA, strB, pA and pB and single stepping through the program. It is even more educational if instead of simply defining strB[] as has been done above, initialize it also with something like:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
strB[80] = "12345678901234567890123456789012345678901234567890"
```

where the number of digits used is greater than the length of strA and then repeat the single stepping procedure while watching the above variables. Give these things a try!

Getting back to the prototype for puts() for a moment, the "const" used as a parameter modifier informs the user that the function will not modify the string pointed to by s, i.e. it will treat that string as a constant.

Of course, what the above program illustrates is a simple way of copying a string. After playing with the above until you have a good understanding of what is happening, we can proceed to creating our own replacement for the standard strcpy() that comes with C. It might look like:

```
char *my_strcpy(char *destination, char *source)
{
    char *p = destination
    while (*source != '\0')
    {
       *p++ = *source++;
    }
    *p = '\0';
    return destination.
}
```

In this case, I have followed the practice used in the standard routine of returning a pointer to the destination.

Again, the function is designed to accept the values of two character pointers, i.e. addresses, and thus in the previous program we could write:

```
int main(void)
{
  my_strcpy(strB, strA);
  puts(strB);
}
```

I have deviated slightly from the form used in standard C which would have the prototype:

```
char *my_strcpy(char *destination, const char *source);
```

Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. You can prove this by modifying the function above, and its prototype, to include the "const" modifier as shown. Then, within the function you can add a statement which attempts to change the contents of that which is pointed to by source, such as:

```
*source = 'X';
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

which would normally change the first character of the string to an X. The const modifier should cause your compiler to catch this as an error. Try it and see.

Now, let's consider some of the things the above examples have shown us. First off, consider the fact that *ptr++ is to be interpreted as returning the value pointed to by ptr and then incrementing the pointer value. On the other hand, this has to do with the precedence of the operators. Were we to write (*ptr)++ we would increment, not the pointer, but that which the pointer points to! i.e. if used on the first character of the above example string the 'T' would be incremented to a 'U'. You can write some simple example code to illustrate this.

Recall again that a string is nothing more than an array of characters, with the last character being a '\0'. What we have done above is deal with copying an array. It happens to be an array of characters but the technique could be applied to an array of integers, doubles, etc. In those cases, however, we would not be dealing with strings and hence the end of the array would not be marked with a special value like the nul character. We could implement a version that relied on a special value to identify the end. For example, we could copy an array of positive integers by marking the end with a negative integer. On the other hand, it is more usual that when we write a function to copy an array of items other than strings we pass the function the number of items to be copied as well as the address of the array, e.g. something like the following prototype might indicate:

```
void int_copy(int *ptrA, int *ptrB, int nbr);
```

where nbr is the number of integers to be copied. You might want to play with this idea and create an array of integers and see if you can write the function int_copy() and make it work.

This permits using functions to manipulate large arrays. For example, if we have an array of 5000 integers that we want to manipulate with a function, we need only pass to that function the address of the array (and any auxiliary information such as nbr above, depending on what we are doing). The array itself does _not_ get passed, i.e. the whole array is not copied and put on the stack before calling the function, only its address is sent.

This is different from passing, say an integer, to a function. When we pass an integer we make a copy of the integer, i.e. get its value and put it on the stack. Within the function any manipulation of the value passed can in no way effect the original integer. But, with arrays and pointers we can pass the address of the variable and hence manipulate the values of the original variables.

->==

CHAPTER 4: More on Strings

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Well, we have progressed quite a way in a short time! Let's back up a little and look at what was done in Chapter 3 on copying of strings but in a different light. Consider the following function:

```
char *my_strcpy(char dest[], char source[])
{
    int i = 0;

    while (source[i] != '\0')
    {
      dest[i] = source[i];
      i++;
    }
    dest[i] = '\0';
    return dest;
}
```

Recall that strings are arrays of characters. Here we have chosen to use array notation instead of pointer notation to do the actual copying. The results are the same, i.e. the string gets copied using this notation just as accurately as it did before. This raises some interesting points which we will discuss.

Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array. Thus, the numerical value of the parameter passed is the same whether we use a character pointer or an array name as a parameter. This would tend to imply that somehow:

```
source[i]  is the same as  *(p+i);
```

In fact, this is true, i.e wherever one writes a[i] it can be replaced with *(a + i) without any problems. In fact, the compiler will create the same code in either case. Thus we see that pointer arithmetic is the same thing as array indexing. Either syntax produces the same result.

This is NOT saying that pointers and arrays are the same thing, they are not. We are only saying that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, which yield identical results.

Now, looking at this last expression, part of it.. (a + i) is a simple addition using the + operator and the rules of c state that such an expression is commutative. That is (a + i) is identical to (i + a). Thus we could write *(i + a) just as easily as *(a + i).

But *(i + a) could have come from i[a] ! From all of this comes the curious truth that if:

```
char a[20];
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
int i;
```

writing   a[3] = 'x';   is the same as writing

```
3[a] = 'x';
```

Try it! Set up an array of characters, integers or longs, etc. and assigned the 3rd or 4th element a value using the conventional approach and then print out that value to be sure you have that working. Then reverse the array notation as I have done above. A good compiler will not balk and the results will be identical. A curiosity... nothing more!

Now, looking at our function above, when we write:

```
dest[i] = source[i];
```

due to the fact that array indexing and pointer arithmetic yield identical results, we can write this as:

```
*(dest + i) = *(source + i);
```

But, this takes 2 additions for each value taken on by i. Additions, generally speaking, take more time than incrementations (such as those done using the ++ operator as in i++). This may not be true in modern optimizing compilers, but one can never be sure. Thus, the pointer version may be a bit faster than the array version.

Another way to speed up the pointer version would be to change:

while (*source != '\0')   to simply   while (*source)

since the value within the parenthesis will go to zero (FALSE) at the same time in either case.

At this point you might want to experiment a bit with writing some of your own programs using pointers. Manipulating strings is a good place to experiment. You might want to write your own versions of such standard functions as:

```
strlen();
strcat();
strchr();
```

and any others you might have on your system.

We will come back to strings and their manipulation through pointers in a future chapter. For now, let's move on and discuss structures for a bit.

->==

CHAPTER 5: Pointers and Structures

As you may know, we can declare the form of a block of data

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
  struct tag{
      char lname[20];       /* last name */
      char fname[20];       /* first name */
      int age;              /* age */
      float rate;           /* e.g. 12.75 per hour */
      };
```

Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

```
--------------- program 5.1 ------------------
#include <stdio.h>
#include <string.h>

struct tag{
      char lname[20];      /* last name */
      char fname[20];      /* first name */
      int age;             /* age */
      float rate;          /* e.g. 12.75 per hour */
      };

struct tag my_struct;       /* declare the structure m_struct */

int main(void)
{
  strcpy(my_struct.lname,"Jensen");
  strcpy(my_struct.fname,"Ted");
  printf("\n%s ",my_struct.fname);
  printf("%s\n",my_struct.lname);
  return 0;
}
-------------- end of program 5.1 --------------
```

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```
  date_of_hire;                 (data types not shown)
  date_of_last_raise;
  last_percent_increase;
  emergency_phone;
  medical_plan;
  Social_S_Nbr;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

etc.....

If we have a large number of employees, what we want to do manipulate the data in these structures by means of functions. For example we might want a function print out the name of the employee listed in any structure passed to it. However, in the original C (Kernighan & Ritchie, 1st Edition) it was not possible to pass a structure, only a pointer to a structure could be passed. In ANSI C, it is now permissible to pass the complete structure. But, since our goal here is to learn more about pointers, we won't pursue that.

Anyway, if we pass the whole structure it means that we must copy the contents of the structure from the calling function to the called function. In systems using stacks, this is done by pushing the contents of the structure on the stack. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
struct tag *st_ptr;
```

and we point it to our example structure with:

```
st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
(*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which st_ptr points to, which is the structure my_struct. Thus, this breaks down to the same as my_struct.age.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
st_ptr->age = 63;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

With that in mind, look at the following program:

```
------------ program 5.2 --------------

#include <stdio.h>
#include <string.h>

struct tag{                     /* the structure type */
       char lname[20];          /* last name */
       char fname[20];          /* first name */
       int age;                 /* age */
       float rate;              /* e.g. 12.75 per hour */
       };

struct tag my_struct;           /* define the structure */

void show_name(struct tag *p);  /* function prototype */

int main(void)
{
  struct tag *st_ptr;           /* a pointer to a structure */
  st_ptr = &my_struct;          /* point the pointer to my_struct */
  strcpy(my_struct.lname,"Jensen");
  strcpy(my_struct.fname,"Ted");
  printf("\n%s ",my_struct.fname);
  printf("%s\n",my_struct.lname);
  my_struct.age = 63;
  show_name(st_ptr);            /* pass the pointer */
  return 0;
}


void show_name(struct tag *p)
{
  printf("\n%s ", p->fname);    /* p points to a structure */
  printf("%s ", p->lname);
  printf("%d\n", p->age);
}
-------------------- end of program 5.2 ----------------
```

Again, this is a lot of information to absorb at one time. The reader should compile and run the various code snippets and using a debugger monitor things like my_struct and p while single stepping through the main and following the code down into the function to see what is happening.

->==

CHAPTER 6: Some more on Strings, and Arrays of Strings

Well, let's go back to strings for a bit. In the following all assignments are to be understood as being global, i.e. made outside of any function, including main.

We pointed out in an earlier chapter that we could write:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
   char my_string[40] = "Ted";
```

which would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating '\0'.

Actually, if all we wanted to do was store the name "Ted" we could write:

```
      char my_name[] = "Ted";
```

and the compiler would count the characters, leave room for the nul character and store the total of the four characters in memory the location of which would be returned by the array name, in this case my_string.

In some code, instead of the above, you might see:

```
     char *my_name = "Ted";
```

which is an alternate approach. Is there a difference between these? The answer is.. yes. Using the array notation 4 bytes of storage in the static memory block are taken up, one for each character and one for the terminating nul character. But, in the pointer notation the same 4 bytes required, _plus_ N bytes to store the pointer variable my_name (where N depends on the system but is usually a minimum of 2 bytes and can be 4 or more).

In the array notation, "my_name" is short for &myname[0] which is the address of the first element of the array. Since the location of the array is fixed during run time, this is a constant (not a variable). In the pointer notation my_name is a variable. As to which is the _better_ method, that depends on what you are going to do within the rest of the program.

Let's now go one step further and consider what happens if each of these declarations are done within a function as opposed to globally outside the bounds of any function.

```
void my_function_A(char *ptr)
{
  char a[] = "ABCDE";
   .
   .
}

void my_function_B(char *ptr)
{
  char *cp = "ABCDE";
   .
   .
}
```

Here we are dealing with automatic variables in both cases. In my_function_A the automatic variable is the character array a[]. In my_function_B it is the pointer cp. While C is designed

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

in such a way that a stack is not required on those systems which don't use them, my particular processor (80286) and compiler (TC++) combination uses a stack. I wrote a simple program incorporating functions similar to those above and found that in my_function_A the 5 characters in the string were all stored on the stack. On the other hand, in my_function_B, the 5 characters were stored in the data space and the pointer was stored on the stack.

By making a[] static I could force the compiler to place the 5 characters in the data space as opposed to the stack. I did this exercise to point out just one more difference between dealing with arrays and dealing with pointers. By the way, array initialization of automatic variables as I have done in my_function_A was illegal in the older K&R C and only "came of age" in the newer ANSI C. A fact that may be important when one is considering portability and backwards compatibility.

As long as we are discussing the relationship/differences between pointers and arrays, let's move on to multi-dimensional arrays. Consider, for example the array:

```
char multi[5][10];
```

Just what does this mean? Well, let's consider it in the following light.

```
    char multi[5][10];
       ^^^^^^^^
```

Let's take the underlined part to be the "name" of an array. Then prepending the "char" and appending the [10] we have an array of 10 characters. But, the name "multi[5]" is itself an array indicating that there are 5 elements each being an array of 10 characters. Hence we have an array of 5 arrays of 10 characters each..

Assume we have filled this two dimensional array with data of some kind. In memory, it might look as if it had been formed by initializing 5 separate arrays using something like:

```
multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

At the same time, individual elements might be addressable using syntax such as:

```
multi[0][3] = '3'
multi[1][7] = 'h'
multi[4][0] = 'J'
```

Since arrays are contiguous in memory, our actual memory block for the above should look like:

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
    ^
    |_____ starting at the address &multi[0][0]
```

Note that I did _not_ write multi[0] = "0123456789". Had I done so a terminating '\0' would have been implied since whenever double quotes are used a '\0' character is appended to the characters contained within those quotes. Had that been the case I would have had to set aside room for 11 characters per row instead of 10.

My goal in the above is to illustrate how memory is laid out for 2 dimensional arrays. That is, this is a 2 dimensional array of characters, NOT an array of "strings".

Now, the compiler knows how many columns are present in the array so it can interpret multi + 1 as the address of the 'a' in the 2nd row above. That is, it adds 10, the number of columns, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add 10*sizeof(int) which, on my machine, would be 20. Thus, the address of the "9" in the 4th row above would be &multi[3][0] or *(multi + 3) in pointer notation. To get to the content of the 2nd element in the 4th row we add 1 to this address and dereference the result as in

```
    *(*(multi + 3) + 1)
```

With a little thought we can see that:

```
    *(*(multi + row) + col)    and
    multi[row][col]            yield the same results.
```

The following program illustrates this using integer arrays instead of character arrays.

```
------------------- program 6.1 ----------------------
#include <stdio.h>

#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
  int row, col;
  for (row = 0; row < ROWS; row++)
    for(col = 0; col < COLS; col++)
      multi[row][col] = row*col;
  for (row = 0; row < ROWS; row++)
    for(col = 0; col < COLS; col++)
    {
      printf("\n%d  ",multi[row][col]);
      printf("%d ",*(*(multi + row) + col));
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    }
  return 0;
}
----------------- end of program 6.1 ---------------------
```

Because of the double de-referencing required in the pointer version, the name of a 2 dimensional array is often said to be equivalent to a pointer to a pointer. With a three dimensional array we would be dealing with an array of arrays of arrays and some might say its name would be equivalent to a pointer to a pointer to a pointer. However, here we have initially set aside the block of memory for the array by defining it using array notation. Hence, we are dealing with a constant, not a variable. That is we are talking about a fixed address not a variable pointer. The dereferencing function used above permits us to access any element in the array of arrays without the need of changing the value of that address (the address of multi[0][0] as given by the symbol "multi").

->==

CHAPTER 7: More on Multi-Dimensional Arrays

In the previous chapter we noted that given

```
 #define ROWS 5
 #define COLS 10

 int multi[ROWS][COLS];
```

we can access individual elements of the array "multi" using either:

```
    multi[row][col]    or  *(*(multi + row) + col)
```

To understand more fully what is going on, let us replace

```
    *(multi + row)    with   X   as in:

    *(X + col)
```

Now, from this we see that X is like a pointer since the expression is de-referenced and we know that col is an integer. Here the arithmetic being used is of a special kind called "pointer arithmetic" is being used. That means that, since we are talking about an integer array, the address pointed to by (i.e. value of)   X + col + 1   must be greater than the address X + col   by and amount equal to sizeof(int).

Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression   multi + row   as used above,   multi + row + 1   must increase by value an amount equal to that needed to "point to" the next row, which in this case would be an amount equal to   COLS * sizeof(int).

That says that if the expression   *(*(multi + row) + col) is to be evaluated correctly at run time, the compiler must

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

generate code which takes into consideration the value of COLS, i.e. the 2nd dimension. Because of the equivalence of the two forms of expression, this is true whether we are using the pointer expression as here or the array expression multi[row][col].

Thus, to evaluate either expression, a total of 5 values must be known:

1) The address of the first element of the array, which is returned by the expression "multi", i.e. the name of the array.

2) The size of the type of the elements of the array, in this case sizeof(int).

3) The 2nd dimension of the array

4) The specific index value for the first dimension, "row" in this case.

5) The specific index value for the second dimension, "col" in this case.

Given all of that, consider the problem of designing a function to manipulate the element values of a previously declared array. For example, one which would set all the elements of the array "multi" to the value 1.

```
void set_value(int m_array[][COLS])
{
  int row, col;
  for(row = 0; row < ROWS; row++)
  {
    for(col = 0; col < COLS; col++)
    {
      m_array[row][col] = 1;
    }
  }
}
```

And to call this function we would then use:

```
set_value(multi);
```

Now, within the function we have used the values #defined by ROWS and COLS which set the limits on the for loops. But, these #defines are just constants as far as the compiler is concerned, i.e. there is nothing to connect them to the array size within the function. row and col are local variables, of course. The formal parameter definition informs the compiler that we are talking about an integer array. We really don't need the first dimension and, as will be seen later, there are occasions where we would prefer not to define it within the parameter definition so, out of habit or consistency, I have not used it here. But, the second dimension _must_ be used as has been shown in the

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

expression for the parameter. The reason is that it is needed in
the evaluation of m_array[row][col] as has been described.
The reason is that while the parameter defines the data type (int
in this case) and the automatic variables for row and column are
defined in the for loops, only one value can be passed using a
single parameter. In this case, that is the value of "multi" as
noted in the call statement, i.e. the address of the first
element, often referred to as a pointer to the array. Thus, the
only way we have of informing the compiler of the 2nd dimension
is by explicitly including it in the parameter definition.

In fact, in general all dimensions of higher order than one
are needed when dealing with multi-dimensional arrays. That is
if we are talking about 3 dimensional arrays, the 2nd _and_ 3rd
dimension must be specified in the parameter definition.

->==

CHAPTER 8: Pointers to Arrays

Pointers, of course, can be "pointed at" any type of data
object, including arrays. While that was evident when we
discussed program 3.1, it is important to expand on how we do
this when it comes to multi-dimensional arrays.

To review, in Chapter 2 we stated that given an array of
integers we could point an integer pointer at that array using:

```
int *ptr;

ptr = &my_array[0];       /* point our pointer at the first
                             integer in our array */
```

As we stated there, the type of the pointer variable must match
the type of the first element of the array.

In addition, we can use a pointer as a formal parameter of a
function which is designed to manipulate an array. e.g.

Given:

```
int array[3] = {'1', '5', '7'};

void a_func(int *p);
```

we can pass the address of the array to the function by making
the call

```
a_func(array);
```

This kind of code promotes the mis-conception that pointers and
arrays are the same thing. Of course, if you have followed this
text carefully up to this point you know the difference between a
pointer and an array. The function would be better written (in
terms of clarity) as a_func(int p[]); Note that here
we need not include the dimension since what we are passing is
the address of the array, not the array itself.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

We now turn to the problem of the 2 dimensional array. As stated in the last chapter, C interprets a 2 dimensional array as an array of one dimensional arrays. That being the case, the first element of a 2 dimensional array of integers is a one dimensional array of integers. And a pointer to a two dimensional array of integers must be a pointer to that data type. One way of accomplishing this is through the use of the keyword "typedef". typedef assigns a new name to a specified data type. For example:

```
typedef unsigned char byte;
```

provides the name "byte" to mean type "unsigned char". Hence

```
byte b[10];     would be an array of unsigned characters.
```

Note that in the typedef declaration, the word "byte" has replaced that which would normally be the name of our unsigned char. That is, the rule for using typedef is that the new name for the data type is the name used in the definition of the data type. Thus in:

```
typedef int Array[10];
```

Array becomes a data type for an array of 10 integers. i.e.

```
Array my_arr;
```

declares my_arr as an array of 10 integers and

```
Array arr2d[5];
```

makes arr2d an array of 5 arrays of 10 integers each.

Also note that Array *p1d; makes p1d a pointer to an array of 10 integers. Because *p1d points to the same type as arr2, assigning the address of the two dimensional array arr2d to p1d, the pointer to a one dimensional array of 10 integers is acceptable. i.e. p1d = &arr2d[0]; or p1d = arr2d; are both correct.

Since the data type we use for our pointer is an array of 10 integers we would expect that incrementing p1d by 1 would change its value by 10*sizeof(int), which it does. That is sizeof(*p1d) is 20. You can prove this to yourself by writing and running a simple short program.

Now, while using typedef makes things clearer for the reader and easier on the programmer, it is not really necessary. What we need is a way of declaring a pointer like p1d without the need of the typedef keyword. It turns out that this can be done and that int (*p1d)[10]; is the proper declaration, i.e. p1d here is a pointer to an array of 10 integers just as it was under the declaration using the Array type. Note that this is different than int *p1d[10]; which would make p1d the name of an

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

array of 10 pointers to type int.

->==

CHAPTER 9: Pointers and Dynamic Allocation of Memory

There are times when it is convenient to allocate memory at run time using malloc(), calloc(), or other allocation functions. Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time. Or it permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures.

When memory is allocated, the allocating function (such as malloc(), calloc(), etc.) returns a pointer. The type of this pointer depends on whether you are using an older K&R compiler or the newer ANSI type compiler. With the older compiler the type of the returned pointer is char, with the ANSI compiler it is void.

If you are using an older compiler, and you want to allocate memory for an array of integers you will have to cast the char pointer returned to an integer pointer. For example, to allocate space for 10 integers we might write:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if(iptr == NULL)
{ .. ERROR ROUTINE GOES HERE .. }
```

If you are using an ANSI compliant compiler, malloc() returns a void pointer and since a void pointer can be assigned to a pointer variable of any object type, the (int *) cast shown above is not needed. The array dimension can be determined at run time and is not needed at compile time. That is, the "10" above could be a variable read in from a data file or keyboard, or calculated based on some need, at run time.

Because of the equivalence between array and pointer notation, once iptr has been assigned as above, one can use the array notation. For example, one could write:

```
int k;
for(k = 0; k < 10; k++
  iptr[k] = 2;
```

to set the values of all elements to 2.

Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the dynamic allocation of multi-dimensional arrays. In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. Depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

to go about our task.

As we have seen, when dynamically allocating a one dimensional array the dimension can be determined at run time. Now, when using dynamic allocation of higher order arrays, we never need to know the first dimension at compile time. Whether we need to know the higher dimensions depends on how we go about writing the code. Here I will discuss various methods of dynamically allocating room for 2 dimensional arrays of integers.

First we will consider cases where the 2nd dimension is known at compile time.

METHOD 1:

One way of dealing with the problem is through the use of the "typedef" keyword. To allocate a 2 dimensional array of integers recall that the following two notations result in the same object code being generated:

```
multi[row][col] = 1;     *(*(multi + row) + col) = 1;
```

It is also true that the following two notations generate the same code:

```
   multi[row]            *(multi + row)
```

Since the one on the right must evaluate to a pointer, the array notation on the left must also evaluate to a pointer. In fact multi[0] will return a pointer to the first integer in the first row, multi[1] a pointer to the first integer of the second row, etc. Actually, multi[n] evaluates to a pointer to that array of integers which makes up the n-th row of our 2 dimensional array. That is, multi can be thought of as an array of arrays and multi[n] as a pointer to the n-th array of this array of arrays. Here the word "pointer" is being used to represent an address value. While such usage is common in the literature, when reading such statements one must be careful to distinguish between the constant address of an array and a variable pointer which is a data object in itself.

Consider now:

```
----------------------------------------------------
#include <stdio.h>
#define COLS 5

typedef int RowArray[COLS];
RowArray *rptr;

int main(void)
{
  int nrows = 10;
  int row, col;
  rptr = malloc(nrows * COLS * sizeof(int))
  for(row = 0; row < nrows; row++)
   for(col = 0; col < COLS; col++)
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
   {
    rptr[row][col] = 17;
   }
}
```

Here I have assumed an ANSI compiler so a cast on the void pointer returned by malloc() is not required. If you are using an older K&R compiler you will have to cast using:

```
rptr = (RowArray *)malloc(.... etc.
```

Using this approach, "rptr" has all the characteristics of an array name and array notation may be used throughout the rest of the program. That also means that if you intend to write a function to modify the array contents, you must use COLS as a part of the formal parameter in that function, just as we did when discussing the passing of two dimensional arrays to a function.

METHOD 2:

In the METHOD 1 above, rptr turned out to be a pointer to type "one dimensional array of COLS integers". It turns out that there is syntax which can be used for this type without the need of typedef. If we write:

```
int (char *xptr)[COLS];
```

the variable xptr will have all the same characteristics as the variable rptr in METHOD 1 above, and we need not use the "typedef" keyword. Here xptr is a pointer to an array of integers and the size of that array is given by the #defined COLS. The parenthesis placement makes the pointer notation predominate, even though the array notation has higher precedence. i.e. had we written

```
int char *xptr[COLS];
```

we would have defined xptr as an array of pointers holding the number of pointers equal to that #defined by COLS. Which is not the same thing at all. However, arrays of pointers have their use in the dynamic allocation of two dimensional arrays, as will be seen in the next 2 methods.

METHOD 3:

Consider the case where we do not know the number of elements in each row at compile time, i.e. both the number of rows and number of columns must be determined at run time. One way of doing this would be to create an array of pointers to type int and then allocate space for each row and point these pointers at each row. Consider:

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

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
int main(void)
{
  int nrows = 5;     /* Both nrows and ncols could be evaluated */
  int ncols = 10;    /* or read in at run time */
  int row, col;
  int **rowptr;
  rowptr = malloc(nrows * sizeof(int *));
  if(rowptr == NULL)
  {
    puts("\nFailure to allocate room for row pointers.\n");
    exit(0);
  }
  printf("\n\n\nIndex   Pointer(hex)   Pointer(dec)   Diff.(dec)");

  for(row = 0; row < nrows; row++)
  {
    rowptr[row] = malloc(ncols * sizeof(int));
    if(rowptr[row] == NULL)
    {
      printf("\nFailure to allocate for row[%d]\n",row);
      exit(0);
    }
    printf("\n%d         %p         %d", row, rowptr[row], rowptr[row]);
    if(row > 0)
      printf("              %d",(int)(rowptr[row] - rowptr[row-1]));
  }
  return 0;
}
```

In the above code rowptr is a pointer to pointer to type int. In this case it points to the first element of an array of pointers to type int. Consider the number of calls to malloc():

```
  To get the array of pointers            1     call
  To get space for the rows               5     calls
                                      -----
                Total                   6     calls
```

If you choose to use this approach note that while you can use the array notation to access individual elements of the array, e.g. rowptr[row][col] = 17;, it does not mean that the data in the "two dimensional array" is contiguous in memory.

But, you can use the array notation just as if it were a continuous block of memory. For example, you can write:

```
  rowptr[row][col] = 176;
```

just as if rowptr were the name of a two dimensional array created at compile time. Of course 'row' and 'col' must be within the bounds of the array you have created, just as with an array created at compile time.

If it is desired to have a contiguous block of memory

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

dedicated to the storage of the elements in the array it can be done as follows:

METHOD 4:

In this method we allocate a block of memory to hold the whole array first. We then create an array of pointers to point to each row. Thus even though the array of pointers is being used, the actual array in memory is contiguous. The code looks like this:

```
----------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

int main(void)
{
  int **rptr;
  int *aptr;
  int *testptr;
  int k;
  int nrows = 5;     /* Both nrows and ncols could be evaluated */
  int ncols = 10;    /* or read in at run time */
  int row, col;
    /* we now allocate the memory for the array */
  aptr = malloc(nrows * ncols * sizeof(int *));
  if(aptr == NULL)
  {
   puts("\nFailure to allocate room for the array");
   exit(0);
  }
    /* next we allocate room for the pointers to the rows */
  rptr = malloc(nrows * sizeof(int *));
  if(rptr == NULL)
  {
   puts("\nFailure to allocate room for pointers");
   exit(0);
  }
    /* and now we 'point' the pointers */
  clrscr();
  for(k = 0; k < nrows; k++)
  {
    rptr[k] = aptr + (k * ncols);
  }
  printf("\n\n\nIndex   Pointer(hex)   Pointer(dec)   Diff.(dec)");

  for(row = 0; row < nrows; row++)
  {
   printf("\n%d         %p         %d", row, rptr[row], rptr[row]);
   if(row > 0)
     printf("              %d",(int)(rptr[row] - rptr[row-1]));
  }
  for(row = 0; row < nrows; row++)
  {
    for(col = 0; col < ncols; col++)
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
    {
      rptr[row][col] = row + col;
      printf("%d ", rptr[row][col]);
    }
    putchar('\n');
  }
  puts("\n\n\n");

    /* and here we illustrate that we are, in fact, dealing with
       a 2 dimensional array in a _contiguous_ block of memory. */

  testptr = aptr;
  for(row = 0; row < nrows; row++)
  {
   for(col = 0; col < ncols; col++)
   {
    printf("%d ", *(testptr++));
   }
   putchar('\n');
  }
  return 0;
}
```

Consider again, the number of calls to malloc()

```
   To get room for the array itself       1      call
   To get room for the array of ptrs      1      call
                                         ----
                 Total                    2      calls
```

Now, each call to malloc() creates additional space overhead since malloc() is generally implemented by the operating system forming a linked list which contains data concerning the size of the block. But, more importantly, with large arrays (several hundred rows) keeping track of what needs to be freed when the time comes can be more cumbersome. This, combined with he contiguousness of the data block which permits initialization to all zeroes using memset() would seem to make the second alternative the preferred one.

As a final example on multidimensional arrays we will illustrate the dynamic allocation of a three dimensional array. This example will illustrate one more thing to watch when doing this kind of allocation. For reasons cited above we will use the approach outlined in alternative two. Consider the following code:

```
-------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
#include <conio.h>
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
int X_DIM=16;
int Y_DIM=8;
int Z_DIM=4;

int main(void)
{
  char ***space;
  char ***Arr3D;
  int x, y, z;
  ptrdiff_t diff;

  /* first we set aside space for the array itself */

  space = malloc(X_DIM * Y_DIM * Z_DIM * sizeof(char));

  /* next we allocate space of an array of pointers, each
     to eventually point to the first element of a
     2 dimensional array of pointers to pointers */

  Arr3D = malloc(Z_DIM * sizeof(char **));

  /* and for each of these we assign a pointer to a newly
     allocated array of pointers to a row */

  for(z = 0; z < Z_DIM; z++)
  {
     Arr3D[z] = malloc(Y_DIM * sizeof(char *));

     /* and for each space in this array we put a pointer to
        the first element of each row in the array space
        originally allocated */

     for(y = 0; y < Y_DIM; y++)
     {
       Arr3D[z][y] = ((char *)space + (z*(X_DIM * Y_DIM) + y*X_DIM));
     }
  }

  /* And, now we check each address in our 3D array to see if
     the indexing of the Arr3d pointer leads through in a
     continuous manner */

  for(z = 0; z < Z_DIM; z++)
  {
    printf("Location of array %d is %p\n", z, *Arr3D[z]);
    for( y = 0; y < Y_DIM; y++)
    {
      printf("  Array %d and Row %d starts at %p", z, y, Arr3D[z][y]);
      diff = Arr3D[z][y] - (char *)space;
      printf("    diff = %d  ",diff);
      printf(" z = %d  y = %d\n", z, y);
    }
    getch();
  }
  return 0;
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

----

If you have followed this tutorial up to this point you should have no problem deciphering the above on the basis of the comments alone. There is one line that deserves a bit of special attention however. It reads:

```
Arr3D[z][y] = ((char *)space + (z*(X_DIM * Y_DIM) + y*X_DIM));
```

Note that here "space" is cast to a character pointer, which is the same type as Arr3D[z][y]. A thing to be careful of, however, is where that cast is made. If the cast were made outside the overall parenthesis as in...

```
Arr3D[z][y] = (char *)(space + (z*(X_DIM * Y_DIM) + y*X_DIM));
```

the code fails. The reason is that the cast, in this case, is not so much to make the types on each side of the assignment operator match, as it is to make the pointer arithmetic work. Recall that when dealing with pointer arithmetic in something like:

```
int *ptr;
ptr = ptr + 1;
```

the second line increments the pointer by sizeof(int), which is 2 on MS-DOS machines. Now looking at the mentioned line, it should be obvious that

```
(z*(X_DIM * Y_DIM) + y*X_DIM))
```

calculates the number of array elements This will turn out to be an arithmetic constant after the calculation. Now since we are dealing with an array of characters the result of the pointer arithmetic which adds this value to the pointer to the start of the array should yield a value equal to the pointer value plus this constant. Were we using an int data type, i.e. casting our "space" pointer to (int *), the actual value by which the pointer would be incremented would be the calculated value times sizeof(int).

->==

CHAPTER 10: Pointers to Functions

Up to this point we have been discussing pointers to data objects. C also permits the declaration of pointers to functions. Pointers to functions have a variety of uses and some of them will be discussed here.

Consider the following real problem. You want to write a function that is capable of sorting virtually any collection of data that can be stored in an array. This might be an array of strings, or integers, or floats, or even structures. The sorting algorithm can be the same for all. For example, it could be a simple bubble sort algorithm, or the more complex shell or quick

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

sort algorithm. We'll use a simple bubble sort for demonstration purposes.

Sedgewick [1] has described the bubble sort using C code by setting up a function which when passed a pointer to the array would sort it. If we call that function bubble(), a sort program is described by bubble_1.c, which follows:

```
/*-------------------- bubble_1.c --------------------*/

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr,10);
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}

void bubble(int a[], int N)
{
 int i, j, t;
 for(i = N-1; i >= 0; i--)
   for(j = 1; j <= i; j++)
     if(a[j-1] > a[j])
       {
         t = a[j-1];
         a[j-1] = a[j];
         a[j] = t;
       }
}
/*---------------------- end bubble_1.c -----------------------*/
```

The bubble sort is one of the simpler sorts. The algorithm scans the array from the second to the last element comparing each element with the one which precedes it. If the one that precedes it is larger than the current element, the two are swapped so the larger one is closer to the end of the array. On the first pass, this results in the largest element ending up at the end of the array. The array is now limited to all elements except the last and the process repeated. This puts the next largest element at a point preceding the largest element. The process is repeated

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

for a number of times equal to the number of elements minus 1. The end result is a sorted array.

Here our function is designed to sort an array of integers. Thus in line 1 we are comparing integers and in lines 2 through 4 we are using temporary integer storage to store integers. What we want to do now is see if we can convert this code so we can use any data type, i.e. not be restricted to integers.

At the same time we don't want to have to analyze our algorithm and the code associated with it each time we use it. We start by removing the comparison from within the function bubble() so as to make it relatively easy to modify the comparison function without having to re-write portions related the actual algorithm. This results in bubble_2.c:

```
/*---------------------- bubble_2.c -------------------------*/
   /* Separating the comparison function */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);
int compare(int m, int n);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr,10);
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}

void bubble(int a[], int N)
{
 int i, j, t;
 for(i = N-1; i >= 0; i--)
   for(j = 1; j <= i; j++)
     if (compare(a[j-1], a[j]))
       {
         t = a[j-1];
         a[j-1] = a[j];
         a[j] = t;
       }
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
int compare(int m, int n)
{
  return (m > n);
}
/*--------------------- end of bubble_2.c -----------------------*/
```

If our goal is to make our sort routine data type independent, one way of doing this is to use pointers to type void to point to the data instead of using the integer data type. As a start in that direction let's modify a few things in the above so that pointers can be used. To begin with, we'll stick with pointers to type integer.

```
/*----------------------- bubble_3.c -------------------------*/
#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr,10);
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}

void bubble(int *p, int N)
{
 int i, j, t;
 for(i = N-1; i >= 0; i--)
   for(j = 1; j <= i; j++)
     if (compare(&p[j-1], &p[j]))
       {
         t = p[j-1];
         p[j-1] = p[j];
         p[j] = t;
       }
}

int compare(int *m, int *n)
{
  return (*m > *n);
}
/*------------------ end of bubble3.c -------------------------*/
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

Note the changes. We are now passing a pointer to an integer (or array of integers) to bubble(). And from within bubble we are passing pointers to the elements of the array that we want to compare to our comparison function. And, of course we are dereferencing these pointer in our compare() function in order to make the actual comparison. Our next step will be to convert the pointers in bubble() to pointers to type void so that that function will become more type insensitive. This is shown in bubble_4.

```
/*------------------ bubble_4.c ----------------------------*/
#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr,10);
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}

void bubble(int *p, int N)
{
 int i, j, t;
 for(i = N-1; i >= 0; i--)
   for(j = 1; j <= i; j++)
     if (compare((void *)&p[j-1], (void *)&p[j]))
       {
         t = p[j-1];
         p[j-1] = p[j];
         p[j] = t;
       }
}

int compare(void *m, void *n)
{
  int *m1, *n1;
  m1 = (int *)m;
  n1 = (int *)n;
  return (*m1 > *n1);
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
/*------------------ end of bubble_4.c ---------------------*/
```

Note that, in doing this, in compare() we had to introduce the casting of the void pointer types passed to the actual type being sorted. But, as we'll see later that's okay. And since what is being passed to bubble() is still a pointer to an array of integers, we had to cast these pointers to void pointers when we passed them as parameters in our call to compare().

We now address the problem of what we pass to bubble(). We want to make the first parameter of that function a void pointer also. But, that means that within bubble() we need to do something about the variable t, which is currently an integer. Also, where we use t = p[j-1]; the type of p[j-1] needs to be known in order to know how many bytes to copy to the variable t (or whatever we replace t with).

Currently, in bubble_4.c, knowledge within buffer() as to the type of the data being sorted (and hence the size of each individual element) is obtained from the fact that the first parameter is a pointer to type integer. If we are going to be able to use bubble() to sort any type of data, we need to make that pointer a pointer to type void. But, in doing so we are going to lose information concerning the size of individual elements within the array. So, in bubble_5.c we will add a separate parameter to handle this size information.

These changes, from bubble4.c to bubble5.c are, perhaps, a bit more extensive than those we have made in the past. So, compare the two modules carefully for differences.

```
/*---------------------- bubble5.c ---------------------------*/
#include <stdio.h>
#include <string.h>

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(void *p, size_t width, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  bubble(arr, sizeof(long), 10);
  putchar('\n');
  for(i = 0; i < 10; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
void bubble(void *p, size_t width, int N)
{
 int i, j;
 unsigned char buf[4];
 unsigned char *bp = p;
 for(i = N-1; i >= 0; i--)
   for(j = 1; j <= i; j++)
     if (compare((void *)(bp + width*(j-1)), (void *)(bp + j*width)))  /* 1 */
       {
/*         t = p[j-1];   */
        memcpy(buf, bp + width*(j-1), width);
/*         p[j-1] = p[j];   */
        memcpy(bp + width*(j-1), bp + j*width , width);
/*         p[j] = t;   */
        memcpy(bp + j*width, buf, width);
       }

}

int compare(void *m, void *n)
{
  long *m1, *n1;
  m1 = (long *)m;
  n1 = (long *)n;
  return (*m1 > *n1);
}
/*--------------------- end of bubble5.c ---------------------*/
```

Note that I have changed the data type of the array from int to long to illustrate the changes needed in the compare() function. Within bubble I've done away with the variable t (which we would have had to change from type int to type long). I have added a buffer of size 4 unsigned characters, which is the size needed to hold a long (this will change again in future modifications to this code). The unsigned character pointer *bp is used to point to the base of the array to be sorted, i.e. to the first element of that array.

We also had to modify what we passed to compare(), and how we do the swapping of elements that the comparison indicates need swapping. Use of memcpy() and pointer notation instead of array notation work towards this reduction in type sensitivity.

Again, making a careful comparison of bubble5.c with bubble4.c can result in improved understanding of what is happening and why.

We move now to bubble6.c where we use the same function bubble() that we used in bubble5.c to sort strings instead of long integers. Of course we have to change the comparison function since the means by which strings are compared is different from that by which long integers are compared. And,in bubble6.c we have deleted the lines within bubble() that were commented out in bubble5.c.

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
/*--------------------- bubble6.c ---------------------*/
#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N);
int compare(void *m, void *n);

int main(void)
{
  int i;
  putchar('\n');
  for(i = 0; i < 5; i++)
  {
    printf("%s\n", arr2[i]);
  }
  bubble(arr2, 20, 5);
  putchar('\n\n');
  for(i = 0; i < 5; i++)
  {
    printf("%s\n", arr2[i]);
  }
  return 0;
}

void bubble(void *p, int width, int N)
     {
      int i, j, k;
      unsigned char buf[MAX_BUF];
      unsigned char *bp = p;
      for(i = N-1; i >= 0; i--)
        for(j = 1; j <= i; j++)
        {
          k = compare((void *)(bp + width*(j-1)), (void *)(bp + j*width));
          if (k > 0)
            {
             memcpy(buf, bp + width*(j-1), width);
             memcpy(bp + width*(j-1), bp + j*width , width);
             memcpy(bp + j*width, buf, width);
            }
        }
    }

int compare(void *m, void *n)
{
   char *m1 = m;
   char *n1 = n;
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
   return (strcmp(m1,n1));
}
/*------------------- end of bubble6.c ---------------------*/
```

But, the fact that bubble() was unchanged from that used in bubble5.c indicates that that function is capable of sorting a wide variety of data types. What is left to do is to pass to bubble() the name of the comparison function we want to use so that it can be truly universal. Just as the name of an array is the address of the first element of the array in the data segment, the name of a function decays into the address of that function in the code segment. Thus we need to use a pointer to a function. In this case the comparison function.

Pointers to functions must match the functions pointed to in the number and types of the parameters and the type of the return value. In our case, we declare our function pointer as:

```
   int (*fptr)(const void *p1, const void *p2);
```

Note that were we to write:

```
    int *fptr(const void *p1, const void *p2);
```

we would have a function prototype for a function which returned a pointer to type int. That is because in C the parenthesis () operator have a higher precedence than the pointer * operator. By putting the parenthesis around the string (*fptr) we indicate that we are declaring a function pointer.

We now modify our declaration of bubble() by adding, as its 4th parameter, a function pointer of the proper type. It's function prototype becomes:

```
  void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
```

When we call the bubble(), we insert the name of the comparison function that we want to use. bubble7.c illustrate how this approach permits the use of the same bubble() function for sorting different types of data.

```
/*------------------- bubble7.c ------------------*/
#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
int compare_string(const void *m, const void *n);
int compare_long(const void *m, const void *n);
int main(void)
{
  int i;
  puts("\nBefore Sorting:\n");
  for(i = 0; i < 10; i++)              /* show the long ints */
  {
    printf("%ld ",arr[i]);
  }
  puts("\n");
  for(i = 0; i < 5; i++)                /* show the strings */
  {
    printf("%s\n", arr2[i]);
  }
  bubble(arr, 4, 10, compare_long);         /* sort the longs */
  bubble(arr2, 20, 5, compare_string);     /* sort the strings */
  puts("\n\nAfter Sorting:\n");
  for(i = 0; i < 10; i++)            /* show the sorted longs */
  {
    printf("%d ",arr[i]);
  }
  puts("\n");
  for(i = 0; i < 5; i++)           /* show the sorted strings */
  {
    printf("%s\n", arr2[i]);
  }
  return 0;
}

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *))
     {
      int i, j, k;
      unsigned char buf[MAX_BUF];
      unsigned char *bp = p;
      for(i = N-1; i >= 0; i--)
        for(j = 1; j <= i; j++)
        {
          k = fptr((void *)(bp + width*(j-1)), (void *)(bp + j*width));
          if (k > 0)
            {
             memcpy(buf, bp + width*(j-1), width);
             memcpy(bp + width*(j-1), bp + j*width , width);
             memcpy(bp + j*width, buf, width);
            }
        }
     }

int compare_string(const void *m, const void *n)
{
   char *m1 = (char *)m;
   char *n1 = (char *)n;
   return (strcmp(m1,n1));
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
}

int compare_long(const void *m, const void *n)
{
  long *m1, *n1;
  m1 = (long *)m;
  n1 = (long *)n;
  return (*m1 > *n1);
}
/*---------------- end of bubble7.c ----------------*/
```

Pointers and arrays - Storage and parameter passing
==

When dealing with arrays, it may help to think about the task of the compiler. Given:

type array[num];

'array' is a reference to the beginning of the block of memory allocated for the array. The amount of memory needed to store the array is num * sizeof(type). The compiler figures out how to index through the array by multiplying the subscript times the sizeof the type and adding the result to a pointer to the base of the block. Thus, for:

float fAry[10];

fAry can be thought of as a float* that points to the beginning of the block allocated to the array. We find the first element (subscript 0), by calculating (subscript * sizeof(type) + base) = (0 * sizeof(float) + fAry). Obvioulsy, the first element is at the base. The second element is four bytes in from the base at (fAry + 4 * sizeof(float)). And so on.

So, the key for the compiler to be able to generate proper indexing through the array, is that the compiler know the size of the type contained in the array. If the type is float, then element 0 is at base, and element 1 is at base+4. If the type is a structure of 50 bytes, then element 1 would be at base+50.

For a two-dimensional array, the block of memory is contiguous and the left-most index is the major increment. The array ary[5][10] can be looked at as an array containing 5 units of ary[10]. The elements ary[0][0] through ary[0][9] are contiguous, and the next element is ary[1][0]. Here, to increment through the ten elements of ary[0][0] thruough ary[0][9], the compiler must again know the size of the type contained in the array. But to increment to the beginning of the second set of ten elements, the compiler must know both the size of the type and the size of the minor

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

increment. For int ary[i][j], the beginning of ary[i] is at ary + i * (max j) * sizeof(type), which would be ary + i * 10 * 2. So the second element would start at 20 bytes offset from the base of the block.

At a simple level, there is no real difference between char* cPtr and char[] cAry. Both labels cPtr and cAry are references to the beginning of a block of memory allocated to store elements of type char. However, there are some distinct differences in the way memory is allocated for them. Let us consider the following declarations being made as local (automatic) variables:

```
char *ptr1;
char *ptr2 = "method 1";
char ptr3[9] = "method 2";
char ptr4[] = "method 3";
```

The size of a pointer, sizeof(void*), is either 2 or 4 bytes depending on the memory model. We will assume the large memory model, where pointers are 4 bytes.

In the first case, 4 bytes are allocated on the stack for ptr1. However, no memory has been allocated to which ptr1 might refer, and the value of ptr1 is undefined. ptr1 is called an uninitialized pointer.

In the second case, 4 bytes are allocated for ptr2 on the stack, and 9 bytes are allocated in the data segment. The bytes in the data segment are initialized with the string literal "method 1" (allowing one for the null terminator), and ptr2 is initialized to point to the 9 bytes in the data segment.

In the third case, when the function is called, 9 bytes are allocated on the stack, and those 9 bytes are initialized with the values of the characters in the string literal "method 2". Where do the bytes come from? The string literal has been stored in the data segment. When your function is called, the space is allocated on the stack and the string literal is copied into those bytes. Using this method, process time is lost to copy the bytes, and during the execution of your function, the string actually exists in two places, the original copy in the data segment, and the local copy on the stack.

The fourth case works just like the third, except the compiler figures out for you the length of the literal.

When declared as global variables, there are some distinct differences. ptr1 is still an uninitialized pointer, but the 4 bytes allocated to hold the pointer are now in the data segment instead of on the stack. The 4

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

bytes for ptr2 are also in the data segment; so now we have a 4-byte pointer in the data segment that point to a 9-byte block, also in the data segment, which holds the string literal "method 2". For ptr3 and ptr4, space is never allocated on the stack, there is only one copy ever of the literal, and using the variables ptr3 or ptr4 is the same as manipulating a pointer to the string.

When declared as local variables, the most efficient is the method used for ptr1. Declared as global variables, the most efficient are ptr3 and ptr4.

Now let's look at passing these arrays to functions. int* iptr points to an integer. If I pass iptr to a function, then that function can use iptr to access and modify the value pointed to by iptr. But, the function only gets a copy of iptr, so if the function modifies its copy of the pointer, the calling function will neither know nor be affected. This is important when passing an uninitialized pointer to a function that will allocate memory and initialize the pointer to point to that block of memory. Since this requires the pointer itself to be modified, we must tell the function where the pointer is, or pass a pointer to the pointer.

There is a subtlety here with respect to arrays. If I declare a function:

```
void foo( int fooAry[10] );
```

then it would seem that 2 * 10 = 20 bytes are going to be passed on the stack. However, in the case of arrays, what actually gets passed is a pointer to the array - just sizeof(void*) bytes. Also, since the function actually receives a pointer to the array, and not a local copy, if function foo modifies the array, then the calling routine could end up with corrupted data. So, the following call to function foo could result in mainAry being modified in foo!

```
main() {
  int mainAry[10];
  foo( mainAry );
}
```

Although they are similar, the same rule does not hold for structs. A struct is passed by value, so passing a struct of 50 bytes will allocate 50 bytes on the stack.

Is the following example correct and will the array iAry be modified?

```
void foo( int* iptr ) {
  iptr[2] = 5;
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

```
main() {
  int iAry[10];
  foo( iAry );
}
```

Yes. Since the compiler knows the type, it knows how to index through the array in function foo. Howver, we must ensure that we don't try to make it index beyond the tenth element because this is all the memory that has been allocated.

The following shows how to pass a two-dimensional array:

```
void foo1( float ary[5][10] ) {}
void foo2( float ary[][10] ) {}

main() {
  float fAry[10][10];
  foo1( fAry );
  foo2( fAry );
}
```

Notice that foo1 and foo2 both accomplish the same thing. The compiler does not need to know the value of the major (left-most) dimension. However, could we declare this?

```
void foo3( float **ary ) {}
```

No. How would the compiler know how to index into the array? It's easy enough to figure out where ary[0][0] through ary[0][9] are; but, how does the compiler know where ary[1][0] is? It must know the extent of the second dimension. However, the above function definition is correct. Only, ary is a pointer to a pointer to type float, which is more commonly viewed as an array of pointers to type float. So, ary[0] is of type float*, a pointer to one float, or an array of floats. Allocating the array of pointers, which are 4 bytes each, to each point to a single float, also 4 bytes, would not be efficient. If we used it as a reference to an array of floats, then it might be useful; but, we would either have to make arrays refered to by each of the pointers in the array the same predetermined length, or have some means of finding the end of each of those arrays. The usefulness of such a data structure is not readily apparent for floats, but for arrays of type char (or strings), it makes more sense.

I can implement two-dimensional arrays of type char in the same fashion and treat them as an array of strings. However, strings are often of different

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

lengths. Consider an array of 10 strings where the longest string is 10 characters. A two-dimensional array would be:

```
#define maxLen 11
#define maxItems 10
char myString[maxItems][maxLen];
```

The overall size of the array myStrings is 10 * 11 * sizeof(char) = 110 bytes. If the strings are not all 10 characters long, there will be wasted space. Of course, this may not be significant. But, there is another way of accomplishing this. Consider the following:

```
char *myStrings1[maxItems];
char *myStrings2[];
char **myStrings3;
```

The first is an a array of 10 pointers to type char. Each of the 10 pointers in the array are uninitialized, and must be initialized using some form of dynamic memory allocation such as calls to malloc.

myStrings2 is a pointer to an array of pointers. But, no memory has been allocated to hold that array. So to use it, we must first allocate space to hold the array. Then we can initailize the elements of the array, each of them a pointer to type char, by a separate allocation and initialization for each of them. myStrings3, a pointer to a pointer to type char, works the same as myStrings2. Just as int *iptr can be looked at as an array of integers, so can char **myStrings3 be viewed as a pointer to a pointer to type char, or, an array of pointers to type char. Thus to use it we must first allocate space to hold the array:

```
myStrings = ( char** ) malloc( nItems * sizeof( char * ) );
```

Lets say that nItems = 10 and we are still using the large memory model. The amount of allocated space is 10 * 4 = 40 bytes, which is just enough to hold ten char pointers. Now we must allocate space for each of the strings, and initialize the corresponding pointer in the array to reference that allocated block of memory. In our example, we will do this in a loop:

```
for( i=0; i<nItems; i++)
   myStrings[i] = ( char* ) malloc( maxLen + 1 );
```

Why is this possibly better than just declaring myStrings[nItems][maxLen]? One reason is that it moves the structure into the far heap (in our memory model) where there is more memory than in the data segment or on the stack. Another is that using this technique, we can create a two-dimensional array (an array of strings or a multiple dimensional array of any type) that is

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

effectively >64K. As long as neither the block that holds my array of pointers nor any of the blocks allocated for each of those pointers exceeds 64K, I can successfully access data that is in a structure format which totals well over a segment, or even several segments. Another is that I don't have to allocate the same amount of space for each of the strings. By definition, a string is terminated with a null ('\0'). Since we have a convenient method of finding the end of the array (unlike the example for floats), this is easy to use for strings.

So, now lets say we want to pass this array of strings to a function. In the first case, we only want to use the strings to display them.

```
void Show( char **strings )
{
  puts( string[1] );
}
```

The above will display the second string in my array of strings. This method of passing the strings will also allow us to modify the contents of each of the strings, or even initialize each of the pointers in the array, since it is actually passed a pointer which refers to the base of the array that holds the pointers to each of the strings. Now suppose I want to initialize the pointer to the actual array. In my main function, I have merely declared char **myStrings, and I want to allocate space for the array of pointers and for each of the items to which those pointers will point, all in a function. We might do that like this.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

void foo( char ***ary, int nItems, int maxLen ) {
   int i;
   *ary = ( char** ) malloc( nItems * sizeof( char* ) );
   for( i=0; i<nItems; i++ )
      (*ary)[i] = ( char* ) malloc( maxLen + 1 ); /* sizeof( char ) = 1 */
   strcpy( (*ary)[0], "The wonderful world of pointers!" );
}

void main() {
  char **myStrings;
  foo( &myStrings, 5, 40 );
  puts( myStrings[0] );
}
```

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html

************************************ Good Luck Helpmate User

No license: PDF produced by PStill (c) F. Siegert - http://www.this.net/~frank/pstill.html