In `main()` you also display what you got back from `change()` and the value of `number`.

```
printf("\nIn main, result = %d\tnumber = %d", result, number);
```

Look at the output, though. It demonstrates how vain and pathetic the attempt to change a variable value by passing it to a function has been. Clearly, the variable `number` in `change()` has the value 20 on return from the function. It's displayed both in the function and as a returned value in `main()`. In spite of our transparent subterfuge of giving them the same name, the variables with the name `number` in `main()` and `change()` are evidently quite separate, so modifying one has no effect on the other.

TRY IT OUT: USING POINTERS IN FUNCTIONS

Let's now modify the last example to use pointers, and with a following wind, you should succeed in modifying the value of a variable in `main()`.

```
/* Program 8.5 The change that does   */
#include <stdio.h>

int change(int *pnumber);              /* Function prototype              */

int main(void)
{
  int number = 10;                    /* Starting Value                  */
  int *pnumber = &number;             /* Pointer to starting value       */
  int result = 0;                     /* Place to put the returned value  */

  result = change(pnumber);
  printf("\nIn main, result = %d\tnumber = %d", result, number);
  return 0;
}

/* Definition of the function change() */
int change(int *pnumber)
{
  *pnumber *= 2;
  printf("\nIn function change, *pnumber = %d\n", *pnumber );
  return *pnumber;
}
```

The output from this program looks like this:

```
In function change, *pnumber = 20
In main, result = 20    number = 20
```

How It Works

There are relatively few changes to the last example. You define a pointer in `main()` with the name `pnumber` that is initialized to the address of the variable `number`, which holds the starting value:

```
  int *pnumber = &number;             /* Pointer to starting value       */
```

The prototype of the function `change()` has been modified to take account of the parameter being a pointer:

```
int change(int *pnumber);              /* Function prototype               */
```

The definition of the function `change()` has also been modified to use a pointer:

```
int change(int *pnumber)
{
  *pnumber *= 2;
  printf("\nIn function change, *pnumber = %d", *pnumber );
  return *pnumber;
}
```

This pointer, `pnumber`, has the same name as the pointer in `main()`, although this is of no consequence to the way the program works. You could call it anything you like, as long as it's a pointer of the correct type, because the name is local to this function.

Within the function `change()`, the arithmetic statement has been changed to this:

```
  *pnumber *= 2;
```

Using the `*=` operator isn't strictly necessary, but it makes it a lot less confusing, provided you can remember what `*=` does at this point. It's exactly the same as this:

```
  *pnumber = 2*(*pnumber);
```

The output now demonstrates that the pointer mechanism is working correctly and that the function `change()` is indeed modifying the value of `number` in `main()`. Of course, when you submit a pointer as an argument, it's still passed by value. Therefore, the compiler doesn't pass the original pointer; it makes a copy of the address stored in the pointer variable to hand over to the function. Because the copy will be the same address as the original, it still refers to the variable `number`, so everything works OK.

If you're unconvinced of this, you can demonstrate it for yourself quite easily by adding a statement to the function `change()` that modifies the pointer `pnumber`. You could set it to `NULL`, for instance. You can then check in `main()` that `pnumber` still points to `number`. Of course, you'll have to alter the `return` statement in `change()` to get the correct result.

const Parameters

You can qualify a function parameter using the `const` keyword, which indicates that the function will treat the argument that is passed for this parameter as a constant. Because arguments are passed by value, this is only useful when the parameter is a pointer. Typically you apply the `const` keyword to a parameter that is a pointer to specify that a function will not change the value pointed to. In other words, the code in the body of the function will not modify the value pointed to by the pointer argument. Here's an example of a function with a `const` parameter:

```
bool SendMessage(const char* pmessage)
{
  /* Code to send the message */
  return true;
}
```

The compiler will verify that the code in the body of the function does not use the pmessage pointer to modify the message text. You could specify the pointer itself as const too, but this makes little sense because the address is passed by value so you cannot change the original pointer in the calling function.

Specifying a pointer parameter as const has another useful purpose. Because the const modifier implies that the function will not change the data that is pointed to, the compiler knows that an argument that is a pointer to constant data should be safe. On the other hand, if you do not use the const modifier with the parameter, so far as the compiler is concerned, the function may modify the data pointed to by the argument. A good C compiler will at least give you a warning message when you pass a pointer to constant data as the argument for a parameter that you did not declare as const.

■ **Tip** If your function does not modify the data pointed to by a pointer parameter, declare the function parameter as const. That way the compiler will verify that your function indeed does not change the data. It will also allow a pointer to a constant to be passed to the function without issuing a warning or an error message.

To specify the address that a pointer contains as const, you place the const keyword after the * in the pointer type specification. Here is a code fragment containing a couple of examples of pointer declaration that will illustrate the difference between a pointer to a constant and a constant pointer:

```
int value1 = 99;
int value2 = 88;

const int pvalue = &value1;           /* pointer to constant */
int const cpvalue = &value1;          /* Constant pointer    */

pvalue = &value2;                     /* OK: pointer is not constant   */
*pvalue = 77;                         /* Illegal: data is constant */

cpvalue = &value2;                    /* Illegal: pointer is constant  */
*cpvalue = 77;                         /* OK: data is not constant */
```

If you wanted the parameter to the SendMessage() function to be a constant pointer, you would code it as:

```
bool SendMessage(char *const pmessage)
{
  /* Code to send the message    */
  /* Can change the message here */
  return true;
}
```

Now the function body can change the message but not the address in pmessage. As I said, pmessage will contain a copy of the address so you might as well declare the parameter with using const in this case.

TRY IT OUT: PASSING DATA USING POINTERS

You can exercise this method of passing data to a function using pointers in a slightly more practical way, with a revised version of the function for sorting strings from Chapter 7. You can see the complete code for the program here, and then I'll discuss it in detail. The source code defines three functions in addition to `main()`:

```
/* Program 8.6 The functional approach to string sorting*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

bool str_in(char **);                  /* Function prototype for str_in     */
void str_sort(const char *[], int);    /* Function prototype for str_sort   */
void swap( void **p1,  void **p2);     /* Swap two pointers                 */
void str_out(char *[], int);           /* Function prototype for str_out    */

const size_t BUFFER_LEN =  256;
const size_t NUM_P = 50;

/* Function main - execution starts here */
int main(void)
{
  char *pS[NUM_P];                     /* Array of string pointers          */
  int count = 0;                       /* Number of strings read            */

  printf("\nEnter successive lines, pressing Enter at the end of"
                       " each line.\nJust press Enter to end.\n");

  for(count = 0; count < NUM_P ; count++)        /* Max of NUM_P strings   */
    if(!str_in(&pS[count]))                      /* Read a string          */
      break;                                     /* Stop input on 0 return */

  str_sort( pS, count);                          /* Sort strings           */
  str_out( pS, count);                           /* Output strings         */
  return 0;
}

/*******************************************************
*      String input routine                            *
 *  Argument is a pointer to a pointer to a constant   *
 *  string which is const char**                       *
 *  Returns false for empty string and returns true    *
 *  otherwise. If no memory is obtained or if there    *
 *  is an error reading from the keyboard, the program *
 *  is terminated by calling exit().                   *
 *******************************************************/
bool str_in(char **pString)
{
  char buffer[BUFFER_LEN];             /* Space to store input string  */
  if(gets(buffer) == NULL )           /* NULL returned from gets()?   */
  {
    printf("\nError reading string.\n");
    exit(1);                          /* Error on input so exit       */
  }

  if(buffer[0] == '\0')               /* Empty string read?           */
    return false;

  *pString = (char*)malloc(strlen(buffer) + 1);

  if(*pString == NULL)                /* Check memory allocation      */
  {
    printf("\nOut of memory.");
    exit(1);                          /* No memory allocated so exit  */
  }

  strcpy(*pString, buffer);           /* Copy string read to argument */
  return true;
}

/****************************************************
 *      String sort routine                        *
 * First argument is array of pointers to constant  *
 * strings which is of type const char*[].          *
 * Second argument is the number of elements in the *
 * pointer array - i.e. the number of strings       *
 ***************************************************/
void str_sort(const char *p[], int n)
{
  char *pTemp = NULL;                 /* Temporary pointer               */
  bool sorted = false;                /* Strings sorted indicator        */
  while(!sorted)                      /* Loop until there are no swaps   */
  {
    sorted = true;                    /* Initialize to indicate no swaps */
    for(int i = 0 ; i<n-1 ; i++ )
      if(strcmp(p[i], p[i + 1]) > 0)
      {
        sorted = false;               /* indicate we are out of order    */
        swap(&p[i], &p[i+1]);         /* Swap the pointers               */
      }
  }
}

/*******************************************
*      Swap two pointers                  *
* The arguments are type pointer to void* *
* so pointers can be any type*.           *
*******************************************/
```

```
void swap( void **p1,  void **p2)
{
  void *pt = *p1;
  *p1 = *p2;
  *p2 = pt;
}

/*******************************************************
 *      String output routine                          *
 * First argument is an array of pointers to strings  *
 * which is the same as char**                         *
 * The second argument is a count of the number of    *
 * pointers in the array i.e. the number of strings   *
 *******************************************************/
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", p[i]);             /* Display a string           */
    free(p[i]);                       /* Free memory for the string */
    p[i] = NULL;
  }
  return;
}
```

Typical output from this program would be the following:

```
Enter successive lines, pressing Enter at the end of each line.
Just press Enter to end.
Mike
Adam
Mary
Steve


Your input sorted in order is:

Adam
Mary
Mike
Steve
```

How It Works

This example works in a similar way to the sorting example in Chapter 7. It looks like a lot of code, but I've added quite a few comment lines in fancy boxes that occupy space. This is good practice for longer programs that use several functions, so that you can be sure you know what each function does.

The whole set of statements for all the functions makes up your source file. At the beginning of the program source file, before you define `main()`, you have your `#include` statements for the libraries that you're using and your function prototypes. Each of these is effective from the point of their occurrence to the end of your file, because they're defined outside of all of the functions. They are therefore effective in all of your functions.

The program consists of four functions in addition to the function `main()`. The prototypes of the functions defined are as follows:

```
bool str_in(const char **);           /* Function prototype for str_in     */
void str_sort(const char *[], int);   /* Function prototype for str_sort   */
void swap( void **p1,  void **p2);    /* Swap two pointers                 */
void str_out(char *[], int);          /* Function prototype for str_out    */
```

The first parameter for the `str_sort()` function has been specified as `const`, so the compiler will verify that the function body does not attempt to change the values pointed to. Of course, the parameter in the function definition must also be specified as `const`, otherwise the code won't compile. You can see that the parameter names aren't specified here. You aren't obliged to put the parameter names in a function prototype, but it's usually better if you do. I omitted them in this example to demonstrate that you can leave them out but I recommend that you include them in your programs. You can also use different parameter names in the function prototype from those in the function definition.

■ **Tip** It can be useful to use longer, more explanatory names in the prototype and shorter names in the function definition to keep the code more concise.

The prototypes in this example declare a function `str_in()` to read in all the strings, a function `str_sort()` to sort the strings, and a function `str_out()` to output the sorted strings in their new sequence. The `swap()` function, which I'll come to in a moment, will swap the addresses stored in two pointers. Each function prototype declares the types of the parameters and the return value type for that function.

The first declaration is for `str_in()`, and it declares the parameter as type `char **`, which is a "pointer to a pointer to a `char`." Sound complicated? Well, if you take a closer look at exactly what's going on here, you'll understand how simple this really is.

In `main()`, the argument to this function is `&pS[i]`. This is the address of `pS[i]`—in other words, a pointer to `pS[i]`. And what is `pS[i]`? It's a pointer to `char`. Put these together and you have the type as declared: `char**`, which is a "pointer to a pointer to `char`." You have to declare it this way because you want to modify the contents of an element in the `pS` array from within the function `str_in`. This is the only way that the `str_in()` function can get access to the `pS` array. If you use only one `*` in the parameter type definition and just use `pS[i]` as the argument, the function receives whatever is contained in `pS[i]`, which isn't what you want at all. This mechanism is illustrated in Figure 8-6.

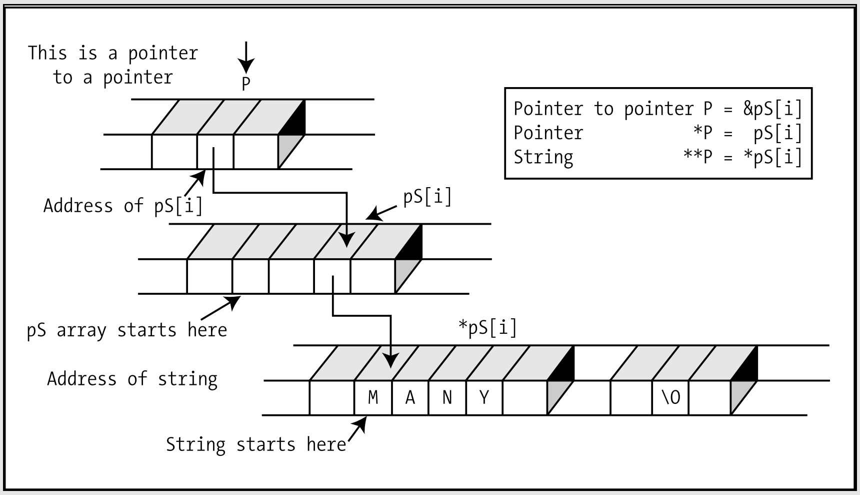

Figure 8-6. *Determining the pointer type*

Of course, type `const char**` is the same as type `const char*[]`, which is an array of elements of type `const char*`. You could use either type specification here.

You can now take a look at the internal working of the function.

The str_in() Function

First, note the detailed comment at the beginning. This is a good way of starting out a function and highlighting its basic purpose. The function definition is as follows:

```
bool str_in(char **pString)
{
  char buffer[BUFFER_LEN];            /* Space to store input string  */

  if(gets(buffer) == NULL)            /* NULL returned from gets()?   */
  {
    printf("\nError reading string.\n");
    return false;                     /* Read error                   */
  }

  if(buffer[0] == '\0')               /* Empty string read?           */
    return false;

  *pString = (char*)malloc(strlen(buffer) + 1);

  if(*pString == NULL)                /* Check memory allocation      */
  {
    printf("\nOut of memory.");
    exit(1);                          /* No memory allocated so exit  */
  }

  strcpy(*pString, buffer);           /* Copy string read to argument */
  return true;
}
```

When the function `str_in()` is called from `main()`, the address of `pS[i]` is passed as the argument. This is the address of the current free array element in which the address of the next string that's entered should be stored. Within the function, this is referred to as the parameter `pString`.

The input string is stored in `buffer` by the function `gets()`. This function returns `NULL` if an error occurs while reading the input, so you first check for that. If input fails, you terminate the program by calling to `exit()`, which is declared in `stdlib.h`. The `exit()` function terminates the program and returns a status value to the operating system that depends on the value of the integer argument you pass. A zero argument will result in a value passed to the operating system that indicates a successful end to the program. A nonzero value will indicate that the program failed in some way. You check the first character in the string obtained by `gets()` against `'\0'`. The function replaces the newline character that results from pressing the Enter key with `'\0'`; so if you just press the Enter key, the first character of the string will be `'\0'`. If you get an empty string entered, you return the value `false` to `main()`.

Once you've read a string, you allocate space for it using the `malloc()` function and store its address in `*pString`. After checking that you did actually get some memory, you copy the contents of `buffer` to the memory that was allocated. If `malloc()` fails to allocate memory, you simply display a message and call `exit()`.

The function `str_in()` is called in `main()` within this loop:

```
  for(count = 0; count < NUM_P ; count++)      /* Max of NUM_P strings   */
    if(!str_in(&pS[count]))                    /* Read a string          */
      break;                                   /* Stop input on 0 return */
```

Because all the work is done in the `str_in()` function, all that's necessary here is to continue the loop until you get `false` returned from the function, which will cause the `break` to be executed, or until you fill up the pointer array `pS`, which is indicated by `count` reaching the value `NUM_P`, thus ending the loop. The loop also counts how many strings are entered in `count`.

Having safely stored all the strings, `main()` then calls the function `str_sort()` to sort the strings with this statement:

```
  str_sort( pS, count );                          /* Sort strings           */
```

The first argument is the array name, `pS`, so the address of the first location of the array is transferred to the function. The second argument is the count of the number of strings so the function will know how many there are to be sorted. Let's now look at how the `str_sort()` function works.

The str_sort() Function

The function `str_sort()` is defined by these statements:

```
void str_sort(const char *p[], int n)
{
  char *pTemp = NULL;                 /* Temporary pointer              */
  bool sorted = false;                /* Strings sorted indicator       */
  while(!sorted)                      /* Loop until there are no swaps  */
  {
    sorted = true;                    /* Initialize to indicate no swaps */
    for(int i = 0 ; i < n-1 ; i++ )
      if(strcmp(p[i], p[i + 1] ) > 0)
      {
        sorted = false;               /* indicate we are out of order   */
        swap(&p[i], &p[i+1]);         /* Swap the pointers              */
      }
  }
}
```

Within the function, the parameter variable p has been defined as an array of pointers. This will be replaced, when the function is called, by the address for pS that's transferred as an argument. You haven't specified the dimension for p. This isn't necessary because the array is one dimensional. The address is passed to the function as the first argument, and the second argument defines the number of elements that you want to process. If the array had two or more dimensions, you would have to specify all dimensions except the first. This would be necessary to enable the compiler to know the shape of the array. The first parameter is const because the function does not change the strings, it simply rearranges their addresses.

You declare the second parameter, n, in the function str_sort() as type int, and this will have the value of the argument count when the function is called. You declare a variable named sorted that will have the value true or false to indicate whether or not the strings have been sorted. Remember that all the variables declared within the function body are local to that function.

The strings are sorted in the for loop using the swap() function that interchanges the addresses in two pointers. You can see how the sorting process works in Figure 8-7. Notice that only the pointers are altered. In this illustration, I used the input data you saw previously, and this input happens to be completely sorted in one pass. However, if you try this process on paper with a more disordered sequence of the same input strings, you'll see that more than one pass is often required to reach the correct sequence.

Note that there is no return statement in the definition of the str_sort() function. Coming to the end of the function body during execution is equivalent of executing a return statement without a return expression.

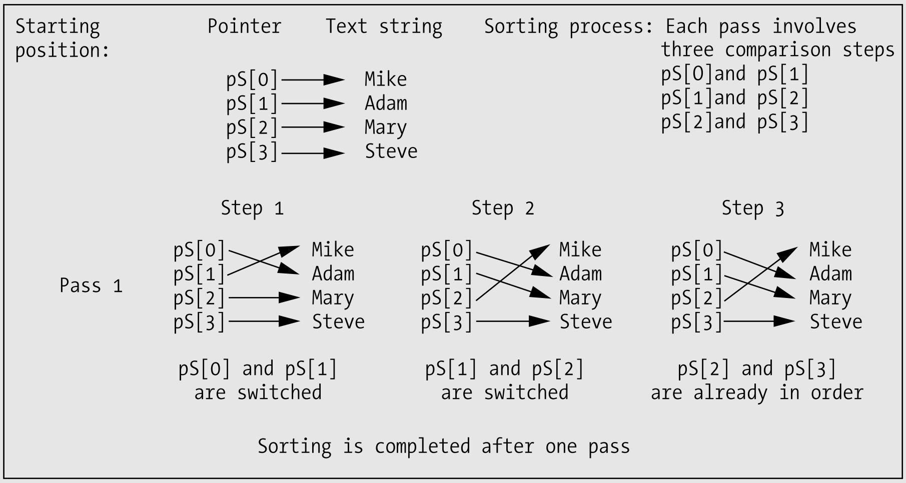
Starting position:
Pointer
Text string
Sorting process: Each pass involves three comparison steps
pS[0]and pS[1]
pS[1]and pS[2]
pS[2]and pS[3]
pS[0] pS[1] pS[2] pS[3]
Mike Adam Mary Steve
Step 1
Step 2
Step 3
Pass 1
pS[0] and pS[1] are switched
pS[1] and pS[2] are switched
pS[2] and pS[3] are already in order
Sorting is completed after one pass

Figure 8-7. *Sorting the strings*

The swap() Function

The swap() function called by sort() is a short utility function that swaps two pointers:

```
void swap( void **p1,  void **p2)
{
  void *pt = *p1;
  *p1 = *p2;
  *p2 = pt;
}
```

As you know, pointers are passed by value, just like any other type of argument so to be able to change a pointer, you must pass a pointer to a pointer as the function argument. The parameters here are of type `void**`, which is “pointer to `void*`”. Any pointer of the form `type*` will convert to type `void*` so this function can swap two pointers of any given type. The swapping process is simple. The address stored at the location specified by `p1` is stored in `pt`. The address stored in `p2` is transferred to `p1`. Finally `p2` is set to the original address from `p1`, now in `pt`.

The str_out() Function

The last function called by `main()` is `str_out()`, which displays the sorted strings:

```
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", p[i]);             /* Display a string           */
    free(p[i]);                       /* Free memory for the string */
    p[i] = NULL;
  }
  return;
}
```

In this function, the parameter `n` receives the value of `count` and the strings are displayed using `n` as the count for the number of strings. The `for` loop outputs all the strings. Once a string has been displayed, the memory is no longer required, so you release the memory that it occupied by calling the library function `free()`. You also set the pointer to `NULL` to avoid any possibility of referring to that memory again by mistake.

You’ve used a `return` statement at the end of the function, but you could have left it out. Because the return type is `void`, reaching the end of the block enclosing the body of the function is the equivalent of `return`. (Remember, though, that this isn’t the case for functions that return a value.)

Returning Pointer Values from a Function

You’ve seen how you can return numeric values from a function. You’ve just learned how to use pointers as arguments and how to store a pointer at an address that’s passed as an argument. You can also return a pointer from a function. Let’s look first at a very simple example.

TRY IT OUT: RETURNING VALUES FROM A FUNCTION

You’ll use increasing your salary as the basis for the example, as it’s such a popular topic.

```
/* Program 8.7 A function to increase your salary */
#include <stdio.h>

long *IncomePlus(long* pPay);          /* Prototype for increase function */

int main(void)
{
  long your_pay = 30000L;             /* Starting salary                */
  long *pold_pay = &your_pay;         /* Pointer to pay value           */
  long *pnew_pay = NULL;              /* Pointer to hold return value   */
  pnew_pay = IncomePlus( pold_pay );
  printf("\nOld pay = $%ld", *pold_pay);
  printf("   New pay = $%ld\n", *pnew_pay);
  return 0;
}

/* Definition of function to increment pay */
long *IncomePlus(long *pPay)
{
  *pPay += 10000L;                       /* Increment the value for pay      */
  return pPay;                           /* Return the address               */
}
```

When you run the program, you'll get this output:

```
Old pay = $40000   New pay = $40000
```

How It Works

In `main()`, you set up an initial value in the variable `your_pay` and define two pointers for use with the function `IncomePlus()`, which is going to increase `your_pay`. One pointer is initialized with the address of `your_pay`, and the other is initialized to `NULL` because it's going to receive the address returned by the function `IncomePlus()`.

Look at the output. It looks like a satisfactory result, except that there's something not quite right. If you overlook what you started with ($30,000), it looks as though you didn't get any increase at all. Because the function `IncomePlus()` modifies the value of `your_pay` through the pointer `pold_pay`, the original value has been changed. Clearly, both pointers, `pold_pay` and `pnew_pay`, refer to the same location: `your_pay`. This is a result of the statement in the function `IncomePlus()`:

```
return pPay;
```

This returns the pointer value that the function received when it was called. This is the address contained in `pold_pay`. The result is that you inadvertently increase the original amount that you were paid—such is the power of pointers.

TRY IT OUT: USING LOCAL STORAGE

Let's look at what happens if you use local storage in the function `IncomePlus()` to hold the value that is returned. After a small modification, the example becomes this:

```
/* Program 8.8 A function to increase your salary that doesn't   */
#include <stdio.h>

long *IncomePlus(long* pPay);          /* Prototype for increase function */

int main(void)
{
  /* Code as before…                 */
  return 0;
}
```

```
/* Definition of function to increment pay */
long *IncomePlus(long *pPay)
{
  long pay = 0;                       /* Local variable for the result */

  pay = *pPay + 10000;                /* Increment the value for pay   */
  return &pay;                        /* Return the address            */
}
```

How It Works

You will probably get a warning message when you compile this example. When I run this I now get the following result (it's likely to be different on your machine and you may even get the correct result):

```
Old pay = $30000   New pay = $27467656
```

Numbers like $27,467,656 with the word "pay" in the same sentence tend to be a bit startling. You would probably hesitate before complaining about this kind of error. As I said, you may get different results on your computer, possibly the correct result this time. You should get a warning from your compiler with this version of the program. With my compiler, I get the message "Suspicious pointer conversion". This is because I'm returning the address of the variable `pay`, which goes out of scope on exiting the function `IncomePlus()`. This is the cause of the remarkable value for the new value of pay—it's junk, just a spurious value left around by something. This is an easy mistake to make, but it can be a hard one to find if the compiler doesn't warn you about the problem.

Try combining the two `printf()` statements in `main()` into one:

```
printf("\nOld pay = $%ld   New pay = $%ld\n", *pold_pay, *pnew_pay);
```

On my computer it now produces the following output:

```
Old pay = $30000   New pay = $40000
```

This actually looks right, in spite of the fact that you know there's a serious error in the program. In this case, although the variable `pay` is out of scope and therefore no longer exists, the memory it occupied hasn't been reused yet. In the example, evidently something uses the memory previously used by the variable `pay` and produces the enormous output value. Here's an absolutely 100 percent cast-iron rule for avoiding this kind of problem.

Cast-Iron Rule Never return the address of a local variable in a function.

So how should you implement the `IncomePlus()` function? Well, the first implementation is fine if you recognize that it does modify the value at the address that is passed to it. If you don't want this to happen, you could just return the new value for the pay rather than a pointer. The calling program would then need to store the value that's returned, not an address.

If you want the new pay value to be stored in another location, the `IncomePlus()` function could conceivably allocate space for it using `malloc()` and then return the address of this memory. However, you should be cautious about doing this because responsibility for freeing the memory would then be left to the calling function. It would be better to pass two arguments to the function, one being the address of the initial pay and the other being the address of the location in which the new pay is to be stored. That way, the calling function has control of the memory.

Separating the allocation of memory at runtime from the freeing of the memory is sometimes a recipe for something called a **memory leak**. This arises when a function that allocates memory dynamically but doesn't release it gets called repeatedly in a loop. This results in more and more of the available memory being occupied, until in some instances there is none left so the program crashes. As far as possible, you should make the function that allocates memory responsible for releasing it. When this is not possible, put in place the code to release the memory when you code the dynamic memory allocation.

You can look at a more practical application of returning pointers by modifying Program 8.6. You could write the routine `str_in()` in this example as follows:

```
char *str_in(void)
{
  char buffer[BUFFER_LEN];            /* Space to store input string */
  char *pString = NULL;               /* Pointer to string           */

  if(gets(buffer) == NULL)            /* NULL returned from gets()?  */
  {
    printf("\nError reading string.\n");
    exit(1);                          /* Error on input so exit       */
  }

  if(buffer[0] == '\0')               /* Empty string read?          */
    return NULL;

  pString = (char*)malloc(strlen(buffer) + 1);

  if(pString == NULL)                 /* Check memory allocation     */
  {
    printf("\nOut of memory.");
    exit(1);                          /* No memory allocated so exit  */
  }
  return strcpy(pString, buffer);     /* Return pString              */
}
```

Of course, you would also have to modify the function prototype to this:

```
char *str_in(void);
```

Now there are no parameters because you've declared the parameter list as `void`, and the return value is now a pointer to a character string rather than an integer.

You would also need to modify the `for` loop in `main()`, which invokes the function to

```
  for(count=0; count < NUM_P ; count++)         /* Max of NUM_P strings      */
    if((pS[count] = str_in())== NULL)           /* Stop input on NULL return */
      break;
```

You now compare the pointer returned from `str_in()` and stored in `pS[count]` with `NULL`, as this will indicate that an empty string was entered or that a string was not read because of a read error. The example would still work exactly as before, but the internal mechanism for input would be a little different. Now the function returns the address of the allocated memory block into which the string has been copied. You might imagine that you could use the address of `buffer` instead, but remember that `buffer` is local to the function, so it goes out of scope on return from the function. You could try it if you like to see what happens.

Choosing one version of the function `str_in()` or the other is, to some extent, a matter of taste, but on balance this latter version is probably better because it uses a simpler definition of the parameter, which makes it easier to understand. Note, however, that it does allocate memory that has to be released somewhere else in the program. When you need to do this it's best to code the function that will release the memory in tandem with coding the function that allocates it. That way, there's less risk of a memory leak.

Incrementing Pointers in a Function

When you use an array name as an argument to a function, a *copy* of the address of the beginning of the array is transferred to the function. As a result, you have the possibility of treating the value received as a pointer in the fullest sense, incrementing or decrementing the address as you wish. For example, you could rewrite the `str_out()` function in Program 8.6 as follows:

```
void str_out(char *p[] , int n)
{
  printf("\nYour input sorted in order is:\n\n");
  for(int i = 0 ; i<n ; i++)
  {
    printf("%s\n", *p);              /* Display a string           */
    free(*p);                        /* Free memory for the string */
    *p++ = NULL;
  }
  return;
}
```

You replace array notation with pointer notation in the `printf()` and `free()` function calls. You wouldn't be able to do this with an array declared within the function body, but because you have a copy of the original array address, it's possible here. You can treat the parameter just like a regular pointer. Because the address you have at this point is a copy of the original in `main()`, this doesn't interfere with the original array address `pS` in any way.

There's little to choose between this version of the function and the original. The former version, using array notation, is probably just a little easier to follow. However, operations expressed in pointer notation often execute faster than array notation.

Summary

You're not done with functions yet, so I'll postpone diving into another chunky example until the end of the next chapter, which covers further aspects of using functions. So let's pause for a moment and summarize the key points that you need to keep in mind when creating and using functions:

- C programs consist of one or more functions, one of which is called `main()`. The function `main()` is where execution always starts, and it's called by the operating system through a user command.
- A function is a self-contained named block of code in a program. The name of a function is in the same form as identifiers, which is a unique sequence of letters and digits, the first of which must be a letter (an underline counts as a letter).

- A function definition consists of a header and a body. The header defines the name of the function, the type of the value returned from the function, and the types and names of all the parameters to the function. The body contains the executable statements for the function, which define what the function actually does.
- All the variables that are declared in a function are local to that function.
- A function prototype is a declaration statement terminated by a semicolon that defines the name, the return type, and the parameter types for a function. A function prototype is required to provide information about a function to the compiler when the definition of a function doesn't precede its use in executable code.
- Before you use a function in your source file, you'd either define the function or declare the function with a function prototype.
- Specifying a pointer parameter as `const` indicates to the compiler that the function does not modify the data pointed to by the function.
- Arguments to a function should be of a type that's compatible with the corresponding parameters specified in its header. If you pass a value of type double to a function that expects an integer argument, the value will be truncated, removing the fractional part.
- A function that returns a value can be used in an expression just as if it were a value of the same type as the return value.
- *Copies* of the argument values are transferred to a function, not the original values in the calling function. This is referred to as the **pass-by-value mechanism** for transferring data to a function.
- If you want a function to modify a variable that's declared in its calling function, the address of the variable needs to be transferred as an argument.

That covers the essentials of creating your own functions. In the next chapter, you'll add a few more techniques for using functions. You'll also work through a more substantial example of applying functions in a practical context.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 8-1. Define a function that will calculate the average of an arbitrary number of floating-point values that are passed to the function in an array. Demonstrate the operation of this function by implementing a program that will accept an arbitrary number of values entered from the keyboard, and output the average.

Exercise 8-2. Define a function that will return a string representation of an integer that is passed as the argument. For example, if the argument is 25, the function will return `"25"`. If the argument is –98, the function will return `"-98"`. Demonstrate the operation of your function with a suitable version of `main()`.

Exercise 8-3. Extend the function that you defined for the previous example to accept an additional argument that specifies the field width for the result, and return the string representation of the value right-justified within the field. For example, if the value to be converted is –98 and the field width argument is 5, the string that is returned should be `"  -98"`. Demonstrate the operation of this function with a suitable version of `main()`.

Exercise 8-4. Define a function that will return the number of words in a string that is passed as an argument. (Words are separated by spaces or punctuation characters. Assume the string doesn't contain embedded single or double quotes—that is, no words such as "isn't.") Define a second function that will segment a string that's passed as the first argument to the function into words, and return the words stored in the array that's passed as the second argument. Define a third function that will return the number of letters in a string that's passed as the argument. Use these functions to implement a program that will read a string containing text from the keyboard and then output all the words from the text ordered from the shortest to the longest.