5

Pointers

Indirection and addresses

5. Pointers

Any object defined in a program can be referenced through its address. Your computer's memory is organized into bytes, each with its own address. When you declare a variable, a set of bytes is set aside for its use. This is where a variable's value will be stored. The beginning address of these bytes is often referred to as the address of that variable. Normally you will only need to use a variable's name to reference the value held in this memory. However, it is also possible to use the address of this memory, instead of the variable name, to reference that value. Using the address of a variable to reference its value is a pointer operation, and it is usually carried out by a pointer variable.

With a pointer variable, you can reference any object defined in a program. A pointer is a variable that has as its value the address of an object. The address held by a pointer can be used to reference a particular object. In this sense, a pointer can be said to point to an object.

A pointer is often used to reference a variable. In such a case, a pointer will have as its value the address of another variable. Since the pointer is a variable itself, a pointer can be described as a variable that holds the address of another variable. This chapter will cover pointers and addresses and how they are used to reference variables. The first part of the chapter will focus on how pointers are used. Then a detailed examination of how pointers operate using addresses will follow.

It is easier to describe what a pointer is used for than to explain how a pointer works. The use of a pointer can be conceived of in a very straightforward, task-oriented way. However, whenever you use pointers, errors can arise easily. Often, correction of pointer errors requires an understanding of how a pointer operates. This, in turn, requires a technical understanding of how objects are defined and referenced in memory. An understanding of how objects are arranged in memory helps us to understand how addresses can be used to reference those objects.

There are situations in which it makes sense to think of the address itself as a pointer. This is discussed at the end of the chapter. For the purposes of this chapter, the term pointer will refer to a pointer variable. An address will be referred to simply as an address.

Pointers and Object References

A pointer is used as a referencing mechanism. A pointer provides a way to reference an object using that object's address. There are usually three elements involved in this referencing process: a pointer variable, an address, and another variable (see Figure 5.1). The pointer variable holds the address of the other variable. A special operation, called an indirection operation, will use this address to actually reference the other variable.

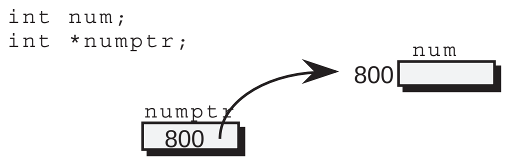

Figure 5.1. Pointer reference and declaration.

A pointer variable is a variable itself. It is declared with a name and a type. Its type consists of two elements: the pointer type specifier-an asterisk-and a data type. The pointer specifier indicates that the variable is a pointer. The data type is the type of object the pointer variable points to, the type of variable it will be used to reference.

The value that a pointer variable holds is an address. An address is different from other kinds of values in that it is used to reference other objects you have declared in your program. Normally, a value is used to reference some real-world value. An integer value is an integer number that can be used in arithmetic calculations. A character value is a character symbol, such as an alphabetic character. An address, however, is the address of a program object. A pointer to an integer holds an address of an integer variable. Figure 5.1 illustrates a pointer declaration and its reference of a variable.

For a pointer to reference another variable, it must first be assigned that variable's address. The address of a variable is obtained with the address operation. The address operator is an ampersand, `&`. When applied to a variable, this operation will result in the address of that variable. This address can then be assigned to a pointer variable. Once the pointer variable has the address of another variable, it can be said to point to that variable.

An indirection operation on the pointer variable will actually reference the pointed-to variable. The indirection operation makes use of both the address held by the pointer and the pointer's data type. The address is used to locate the variable. The data type is used to determine the variable's type. For example, indirection on a pointer to an integer will reference an integer variable, whereas indirection on a pointer to a `float` will reference a floating-point variable. Figure 5.2 displays an address operation and indirection.

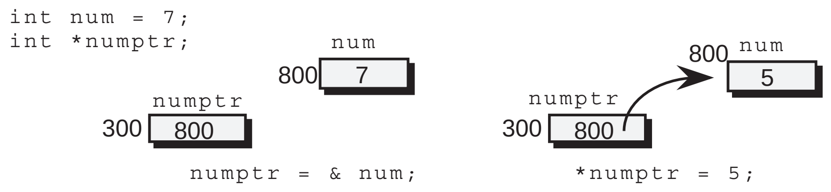

Figure 5.2. Address operation and indirection.

In the **numadd.c** program in Listing 5.1, all these elements come into play. An address operation is used to obtain an address of a variable that is then assigned to a pointer. An indirection on the pointer then references that variable. The pointer variable `numptr` is declared as a pointer to an integer. `numptr` is then assigned the address of the variable `num`. At this point, `numptr` can be said to point to `num`. An indirection operation on `numptr` references the variable `num`, assigning 5 to `num`. In the `printf` statement, the value of `num` is printed out using both the variable `num` and its reference through an indirection operation on `numptr`. Then the address of `num` and the contents of `numptr` are printed out. Notice that they are the same. `numptr` holds the address of `num`.

LISTING 5.1

numadd.c

```
#include <stdio.h>

int main(void)
      {
      int num;
      int *numptr;

      numptr = &num;
      *numptr = 5;

      printf("Num = %d, Indirection on Numptr = %d \n", num, *numptr);
      printf("Address of Num = %p, Address held in Numptr = %p \n", &num, numptr);
      return 0;
      }
```

Output of Listing 5.1:

```
Num = 5, Indirection on Numptr = 5
Address of Num = 800, Address held in Numptr = 800
```

Pointers, Addresses, and Variables

To understand the relationship between pointers and variables, the implementation of each needs to be closely examined. Pointers are declared differently than other variables. A pointer's value and type are designed to provide information with which to reference other variables. An ordinary variable simply represents data values, such as integers or characters. A pointer can be said to represent the variables themselves.

Type, address, and memory are the three basic elements used to define a variable. A variable is defined and referenced in memory. When a variable is defined, a set of bytes is allocated for its use. The beginning address of those bytes is then associated with the variable's name. Each time the variable's name is used in a statement, the variable's address is used to locate those bytes. These bytes will hold a variable's value. The variable's declared data type is used to interpret that value. An integer variable will hold in its bytes an integer value. Figure 5.3 illustrates a variable's address and memory.

Each time a variable is used, the variable's address is used to locate its memory and its type is used to interpret that memory. An integer variable will use its address to locate its memory and interpret that memory as an integer value. This location and interpretation process references a variable. Address and type are the two pieces of information that a variable needs to reference its memory.

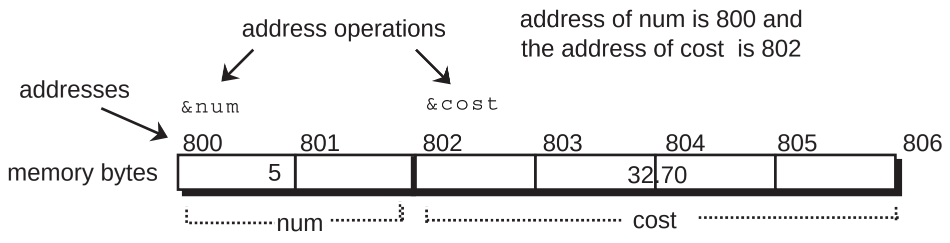

Figure 5.3. Addresses and variables.

In the next example, the integer variable `num` is assigned the integer value 7. The integer value held by the variable `num` is located at a specific address in memory. The integer value itself exists only as a series of binary bits. The `int` type in the `num` variable's declaration indicates that this series of bits is to be interpreted as an integer value.

```
        int num;
        num = 7;
```

A pointer has the same two pieces of information that a variable uses to reference its memory: an address and a type. A pointer variable is declared with a pointer type specifier and a data type. The pointer type is represented by an asterisk, `*`. The asterisk alone does not declare a pointer variable. The asterisk indicates that the value of this variable is an address. This address will be used to locate a variable. The data type placed before the asterisk in the pointer declaration is the type of the variable pointed to by the pointer. The data type is used to reference the memory at that address as a specific kind of variable. In the next example, the asterisk, `*`, indicates that `numptr` is a pointer variable. The data type, `int`, is the type of variable this pointer can reference, an integer.

```
        int *numptr;
```

How, then, does a pointer obtain the address of a variable? First, the address operation has to be applied to the variable. The address operation is literally a unary expression consisting of an address operator and a single operand. The address operator is the ampersand, `&`, and the operand can be any variable. You simply place the ampersand next to the variable. The result of the address operation is the address of the variable. You can then use an assignment operation to assign this address to a pointer variable.

In the next example, the address of `num` is obtained by the address operation and then assigned to the pointer `numptr`. In the **costadd.c** program in Listing 5.2, the address of the cost variable is assigned to the pointer `costptr`. The address held by the `costptr` pointer, as well as the address resulting from an address operation on the cost variable, is printed out. Both addresses are the same.

```
        int num;
        int *numptr;

        numptr = &num;
```

LISTING 5.2

`costadd.c`

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

```
int main(void)
      {
      int num = 5;
      float cost = 32.70;
      int *numptr;
      float * costptr;

      numptr = &num;
      costptr = &cost;

      printf("Address of Num = %p \n", &num);
      printf("Address in Numptr = %p \n",  numptr);
      printf("Address of Cost = %p \n",&cost);
      printf("Address in Costptr = %p \n", costptr);
      return 0;
      }
```

Output of Listing 5.2:

```
      Address of Num = 800,
      Address in Numptr = 800
      Address of Cost = 802,
      Address in Costptr = 802
```

A pointer variable has, as its value, an address, which is simply a number used to reference a location in memory. When an address is used as a pointer value, it is always treated as an address of an object. In C, an address is always used as an address of- an address of a type of object.

An address is similar to an unsigned integer. In C, a pointer can be easily converted to an integer, and vice versa. However, in C, an address is never used as an unsigned integer. It is instead always used as a pointer value: as the contents of a pointer variable, as the result of a pointer expression, or as a constant cast to a pointer value. In this respect, the type of an address is that of a pointer to an object. You can think of an address as not simply an address, but as an address of some type of object.

The change of a pointer variable's value changes the address, but it does not change the data type. In the next example, the pointer variable `numptr` can be assigned many different addresses. However, those addresses will always be considered to be addresses of integers. In the same way, `costptr` may have many different addresses, but those addresses will always be addresses of `float`s. In this sense, one can speak of different types of pointer variables. A pointer variable is literally a variable of type pointer to a type of object. The data type restricts that pointer variable to addresses of that type of object. `costptr` can only be assigned addresses of `float`s; `numptr` can only be assigned addresses of integers.

```
      int * numptr;      int  address
      float * costptr;   float address
```

Two pointer variables can be compared only if their types match. However, as with most operations, you can use a cast expression to generate the same value with a different type. You can apply a cast operation to a pointer that will result in an address of a different type of object from that of the pointer. In the next example, the address held by `costptr` is changed in a cast operation from an address of a `float` to an address of an `int`. The address of an `int` held by `numptr` can now be compared to the address of an `int` generated by the `(int*)` cast operation on `costptr`. `costptr` is still a pointer to a `float`, but the cast operation has taken its value generated a different type of pointer.

```
        int *numptr, *iptr;
        float *costptr;

        if (numptr == iptr)
              printf ("they are equal\n");

        if (numptr ==  (int *) costptr)
              printf ("they are equal\n");
```

Printing out the contents of a pointer variable prints out the address of the variable being referenced by this pointer. The next examples show the different ways you can output addresses. For the sake of these examples, the address held by `numptr` is 534. If the contents of `numptr` are printed out, the output consists of the number 534. This is the address, 534. The output function `printf`, with its conversion capabilities, is usually used to print out addresses. In C, the letter 'p' is a special conversion specifier for pointer values, `%p`. L, l:%lu pointer conversion modifier;

```
printf("Value in numptr is address %p.",numptr);

        Value in numptr is address 534.
```

Indirection Operation

A pointer has the address and data type with which to locate and interpret memory. The actual location and interpretation remain dormant until the address and data type are operated on explicitly. By itself, a pointer is not capable of referencing that memory. You need to use a special operation that can take the information in a pointer and perform the same location and interpretation tasks as a variable. This special operation is called indirection, and it references memory as a variable would. The indirection operation uses the address and its data type to locate and interpret memory.

The indirection operator is an asterisk. The indirection operation uses the address in a pointer to locate bytes in memory. However, the location part of indirection is only half the operation. The indirection operation then makes explicit use of the pointer's data type to interpret the located memory as a certain type of object. The location and interpretation, taken together, can be thought of as referencing an object. An indirection operation on a pointer to an integer can be thought of as referencing an integer variable. A direct reference using a variable name involves the same kind of location and interpretation of memory as that performed by the indirection operation.

In the next example, there is an indirection operation on the pointer variable `numptr`. `numptr` is first assigned the address of `num`. For the sake of this example, the address of `num` is 534. `numptr` holds the address 534 and has a data type of integer. The indirection operation locates that address and interprets the memory at that address as an integer. The indirection operation can be thought of as referencing the integer variable whose memory is located at address 534. Figure 5.4 illustrates the indirection operation on a pointer variable.

```
int num = 7;
int *numptr = &num;
        printf("%d", *numptr);
```

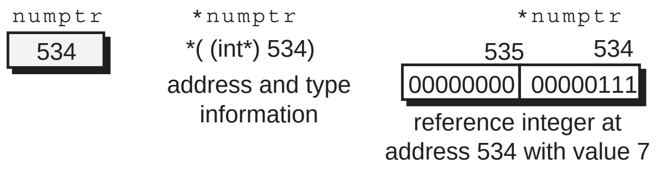
numptr
534
*numptr
((int) 534)
address and type information
*numptr
535
534
00000000
00000111
reference integer at address 534 with value 7

Figure 5.4. Indirection on a pointer to a variable.

Indirection on a pointer to a variable references a variable. This means that you can use indirection not only to interpret memory as some kind of value, but also to assign a value to that memory. Just as you can change the contents of a variable, you can also change the contents of memory referenced by a pointer.

In the next example, the indirection operator works on the address in the pointer variable `numptr`. The indirection operation takes the address in `numptr` (534) and references that memory as an integer variable. The indirection is being used in an assignment expression, assigning the value 3 to the memory at address 534, as Figure 5.5 shows. The indirection operation can be seen as just another variable reference, subject to the same rules as any other variable reference. A variable reference used as the left-hand operand in an assignment operation will assign the right-hand value to that variable. In this case, the variable reference is carried out by an indirection on a pointer, not a variable name.

```
      int num;
      int *numptr = &num;

      *numptr = 3;
```

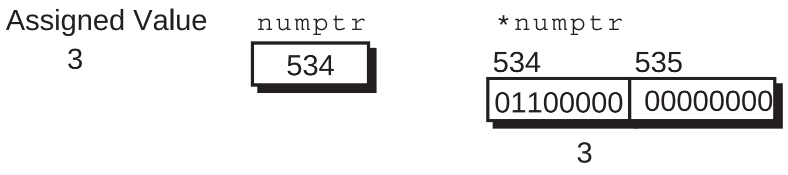

Figure 5.5. Assignment and indirection on a pointer.

In the next example, the indirection operation `*numptr` is operating just like the variable name `num`. The assignment expression `*numptr=3` assigns the value 3 to the memory at 534, just as the assignment expression `num=3` assigns the value 3 to the variable `num`. `res=*numptr` references the value of the memory at 534 and assigns that value to the variable `res`. `res=num` references the value of the variable `num` and assigns it to the variable `res`.

```
      int num, res;
      int *numptr = &num;

      num = 3;
      *numptr = 3;
      res = num;
      res = *numptr;
```

You can use the indirection operation in any kind of expression in which a variable can be used. Within an expression, both variable names and indirection operations can reference data. In the

numind.c program in Listing 5.3, indirection operations are used in several different kinds of expressions. An indirection on `numptr` is used in an arithmetic addition operation. Another indirection is used in a relational expression, `*numptr<5`. Still another indirection is used in a multiplication operation. All of these indirections use the same pointer variable, `numptr`. The address in `numptr` has remained the same throughout the program. This means that all indirections reference the same variable. As long as the address in the pointer variable remains the same, it is as if the same variable is being referenced.

LISTING 5.3

numind.c

```
#include <stdio.h>

void  main(void)
      {
      int num, res;
      int *numptr;

      numptr = &num;
      *numptr = 3;

      res = *numptr  + num;
      if(*numptr < 5)
            res = num * *numptr;

      printf("num= %d,numptr=%p \n",num,numptr);
      printf("res = %d,\n", res);
      }
```

Output of Listing 5.3:

```
      num = 3, numptr = 800
      res = 9,
```

Notice that in the multiplication operation, there are two uses of the asterisk. Placed next to the pointer variable, the asterisk is an indirection operation. Placed between two integers, the asterisk is a multiplication operation. The space between the two asterisks is crucial. Without it, the compiler will interpret the two asterisks as two indirection operations.

There are, then, two kinds of variable references: the variable name and the indirection operation. In the case of pointer variables, the two must not be confused. When used with the indirection operator, the pointer variable is the operand in an indirection operation. Without the indirection operator, the pointer variable is like any other variable name. A pointer variable name by itself simply references a pointer variable, the contents of which are an address, whereas when the pointer variable name is used in an indirection operation, the data object at that address is referenced.

In the first `printf` in the next example, the value of the memory referenced by the indirection operation on `numptr` is printed out. The value of the memory at 534 is 3. In the second `printf`, only the pointer variable name, `numptr`, is used. The pointer variable `numptr` is referenced and its value printed out: the address 534.

```
      printf("%d", *numptr);
      printf("%u", numptr);
```

```
      Output of printf statements:
            3
            534
```

A pointer variable is a variable, and as such you can assign to it different addresses. You can first assign one address to a pointer, and then use another assignment to replace it with another address. In this way you can use the same pointer variable to reference different variables. Indirection operates on the address held by a pointer. If the pointer variable is assigned a new address, the next indirection will reference the variable at that new address. The referencing process relies on the address. When indirection operations work on different addresses, they are, in effect, referencing different variables.

In the **countadd.c** program in Listing 5.4, `numptr` is first assigned the address of `num`. The first indirection references the same memory as that used by `num`. The value 5 is assigned. `numptr` is then assigned a new address, the address of count. The next indirection on `numptr` now references the same memory as that used by the variable count. The value 10 is assigned. An indirection on `numptr` will now reference the value 10.

LISTING 5.4

countadd.c

```
#include <stdio.h>

void  main(void)
      {
      int num, count;
      int *numptr;

      numptr = &num;
      *numptr = 5;
      printf("num = %d,numptr = %p \n", num, numptr);
      numptr = &count;
      *numptr = 10;
      printf("count = %d,numptr = %p \n",count,numptr);
      printf("num = %d, count = %d, *numptr = %d \n",
                                            num, count, *numptr);
      }
```

```
Output of Listing 5.4:

      num = 5, numptr = 800
      count = 10, numptr = 600
      num = 5, count = 10, *numptr = 10
```

Addresses and Memory Maps

The contents of any variable are changed by assigning a new value to that variable. The contents of an integer variable are changed when an integer value is assigned to it. In the same way, the contents of a pointer variable are changed when an address is assigned to it. There are many places from which to obtain addresses. The most common source of an address is another variable. To understand how this works, you will need a rough outline of how memory is organized and used by a program.

Computer memory is a collection of bytes. Each byte has its own address. A pointer can reference any one of these addresses. When a program is loaded into memory from a file, the program is divided into segments. There are four major segments. The first segment is the set of program instructions. This is the program itself. It is usually protected by your operating system. The memory there cannot be referenced or changed. The second segment consists of memory reserved for global and static variables. These are variables that are defined once and remain in existence for the entire run of the program. The third segment is the free and unused portion of memory. Sometimes this is referred to as the heap or allocated memory. This is memory that can be allocated and used during the run of the program. The fourth and last segment is the stack. This is a dynamic part of memory where function variables are defined and function call information is placed. This memory is constantly being used, freed up, and used again in different ways. Figure 5.6 illustrates these four segments.

Addresses can be obtained for all of these segments in many different ways. An address operation can obtain the address of a variable, whether it is global or function-based, referencing memory either in the stack or the data segments. An allocation function such as `malloc` can allocate part of the unused free memory in the third segment. The function will then return the address of that allocated memory. A function name represents the address of program instructions that make up the function, referencing memory in the program segment. You can assign this address of a function to a pointer to a function, as described in Chapter 6. Finally, a pointer cast operation can convert an integer to an address, referencing memory in any of the segments. In this way, an integer can be used to represent a global address, accessing any part of a computer's memory.

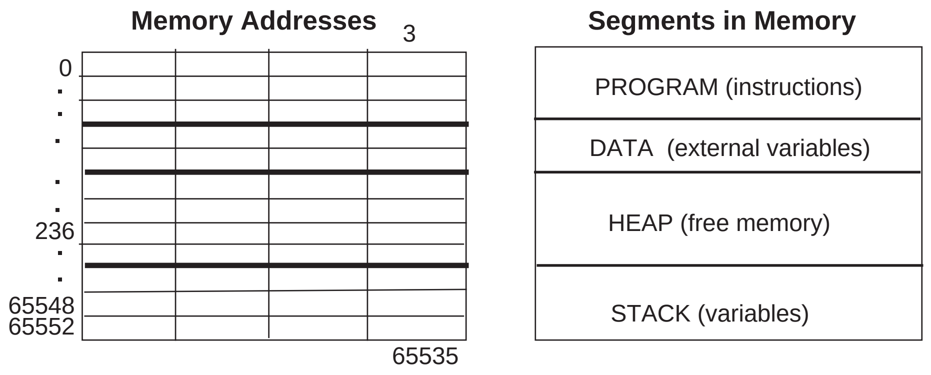

Figure 5.6. Memory map of program organization.

Addresses and Variables: The Address Operation &

When a variable is defined, it is given its own memory. The address of this memory can then be assigned to a pointer. When a pointer is assigned the address of a variable, it will be able to access memory that has been reserved for use by that variable. How, then, can a pointer obtain the address of memory used for a particular variable? tion

The address operation is an expression whose result is the address of a variable. The address operator is a unary operator represented by the ampersand, `&`. The address operator can take a variable name as its operand. In this case, the address operation will result in the address of the memory used for that variable.

```
                    int num = 6;
```

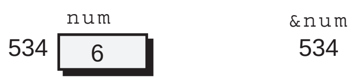

Figure 5.7. Address operation.

In Figure 5.7, the address operation, `&num`, results in the address of the variable's memory. However, an address is always an address of a type of object. In this case, the type associated with the address is the same as the type of the variable. In Figure 5.8, the address operation, `&num`, returns an address, 534, which is the address of an integer. This address can then be assigned to a pointer variable. The pointer variable, in this example, must be a pointer to an integer. The type of the variable and the data type of the pointer must be the same. In Figure 5.8, `numptr` is a pointer to an integer and `num` is an integer.

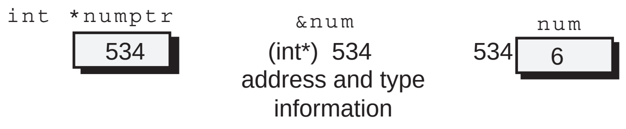

Figure 5.8. Address operation and pointer variable.

In the next example, the address operation on the variable `num`, `&num`, results in an address that is the address of the memory used by the variable `num` (534). This address, in turn, becomes the assigned value in an assignment operation. The address resulting from the address operation on `num` is assigned to the pointer variable `numptr`. As a result, `numptr` obtains the address of the variable `num`, 534. (Figure 5.10 later shows how this process is arranged in memory.) The pointer and the integer variables are both variables located at addresses in memory. The pointer variable is assigned the address of the integer variable. The indirection operation references the memory at that address.

```
            int num;
            int *numptr;

            numptr = &num;
```

An indirection operation can reference the same memory used by a variable. In Figure 5.9, the pointer `numptr` holds the address of the variable `num`. An indirection operation on `numptr` will reference the same memory that `num` uses. There are now two ways to reference the memory at address 534: first, with the variable name `num`; and second, with an indirection on address in `numptr`.

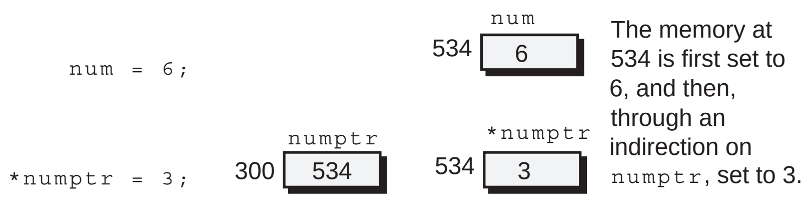
num = 6;
*numptr = 3;
num
534
6
numptr
300
534
*numptr
534
3
The memory at 534 is first set to 6, and then, through an indirection on numptr, set to 3.

Figure 5.9. Indirection and assignments.

In the **numref.c** program in Listing 5.5, `numptr` is assigned the address of `num`. Because of this, the variable name `num` and the indirection on `numptr` can be used interchangeably. Both reference the same memory. The `printf` statement prints out the same value for both `num` and `*numptr` because both access exactly the same memory. That memory was originally set to 2 by the second assignment expression using the variable name `num`. Then it was set to 5 by the third assignment expression using an indirection on the pointer, `*numptr`. The operands in the multiplication operation are actually the same. The first operand uses the variable name to reference the memory of `num`, and the second operand uses an indirection operation to reference the memory of `num`, `*numptr`. The multiplication operation is then 5 * 5, resulting in 25.

LISTING 5.5

numref.c

```
#include <stdio.h>

void  main(void)
      {
      int num, sqnum;
      int *numptr;

      numptr = &num;

      num = 2;
      *numptr = 5;
      sqnum = num * *numptr;

      printf("num = %d,*numptr = %d \n", num, *numptr);
      printf("Square of num = %d \n", sqnum);
      }
```

Output of Listing 5.5:

```
      num = 5, *numptr = 5
      Square of num = 25
```

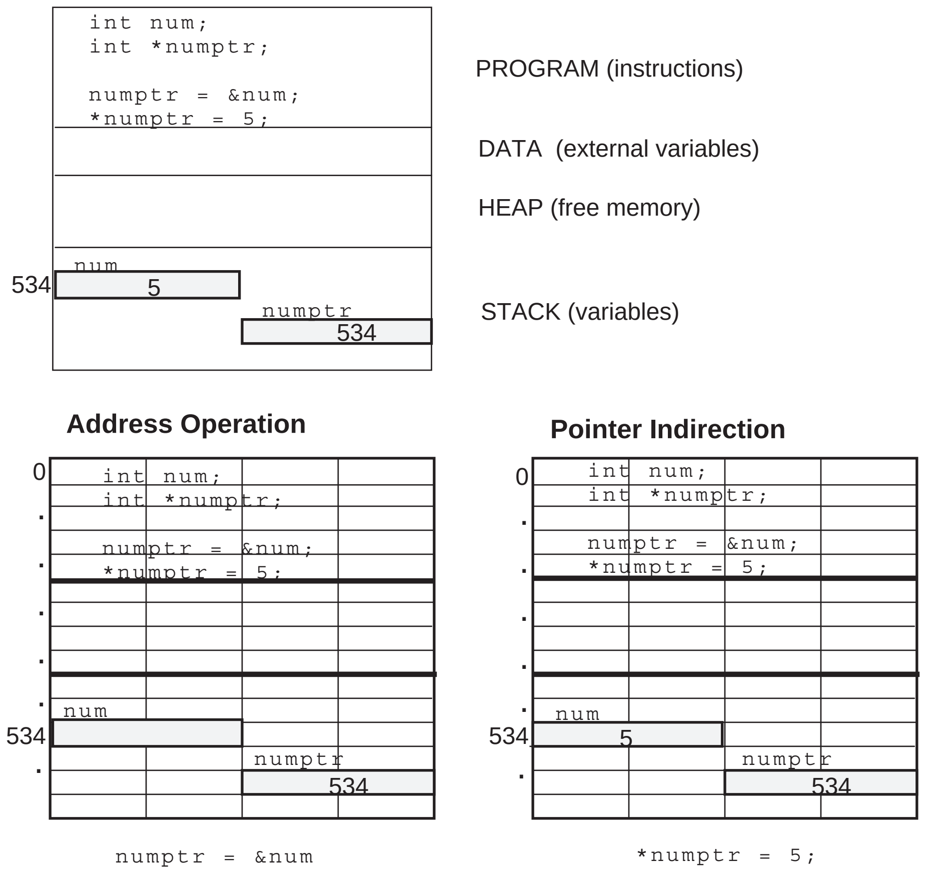

Figure 5.10. Address and indirection operations in memory.

It is just as easy to have more than one pointer variable set to the same variable's address. In the **varptrs.c** program in Listing 5.6, both `numptr` and `iptr` are assigned the address of `num`. An indirection on either `numptr` or `iptr` will access the same memory as that used by `num`. First, both `numptr` and `iptr` are set to the address of `num`. An indirection on `numptr` changes the memory used by `num` to 5. The next indirection on `iptr` changes `num`'s memory again, this time to 10. The multiplication operation works through two different pointers, `numptr` and `iptr`, but references the same memory, that of `num`. Then values of `num`, the indirection on `numptr`, and the indirection on `iptr` are printed out. Then the address of `num` and the addresses in the pointer variables `numptr` and `iptr` are printed out. Notice that these addresses are all the same. The value of `sqnum` will be the result of the multiplication operation 10 * 10 - 100.

LISTING 5.6

varptrs.c

```
#include <stdio.h>

void  main(void)
      {
      int num, sqnum;
      int *numptr;
      int * iptr;

      numptr = &num;
      iptr = &num;

      *numptr = 5;
      *iptr = 10;
      sqnum = *iptr * *numptr;

      printf("num = %d,*numptr = %d,*iptr = %d \n",
                                          num, *numptr, *iptr);
      printf("&num = %p,numptr = %p,iptr = %p\n",
                                          &num, numptr, iptr);
      printf("Square of num = %d \n", sqnum);
      }
```

Output of Listing 5.6:

```
      num = 10, *numptr = 10, *iptr = 10
      &num = 800, numptr = 800, iptr = 800
      Square of num = 100
```

An error will occur if a programmer were to forget to type in the address operator, `&`, when the address of a variable is called for. This is a simple mistake that is very easy to make. In the next example, the programmer has left out the `&` before the variable `num` in the assignment operation. The value of the variable `num` is assigned to `numptr`, not the address of `num`. `numptr` ends up with the address 6, instead of the address 534, the address of `num`. Some compilers may balk at this assignment. Others may simply convert the integer to a pointer and make the assignment, issuing only a warning. Figure 5.11 show the result of a pointer being assigned the value of a variable instead of its address.

```
      int num;
      float *numptr;

      num = 3;
      numptr = num; /* should be:  numptr = &num */
```

int *numptr
6
num
534
6

Figure 5.11. Pointer with value of variable instead of address.

Addresses and Allocated Memory: `malloc`

A program can call a function to cut out and set aside parts of the unused free memory in the heap segment for use by pointers. You can then use a pointer to access this memory. In this way, parts of this unused memory can be allocated and used as if they were variables.

For this free memory to be usable, it must first be allocated. The allocation function `malloc` allocates parts of this unused memory. `malloc` takes an argument that is the number of bytes it is to allocate. The number of bytes depends on the type of variable those bytes are to be used for. Types are of different sizes and vary from system to system. A common configuration is `int` = 2 bytes, float = 4 bytes, char = 1 byte. The programmer must make sure that there are enough bytes allocated for the type of object needed. In the next example, `malloc` has set aside 2 bytes (the usual size for an integer) for the program's use.

```
        malloc(2);
```

However, on some systems an integer may be 4 bytes. How can you make sure what the actual size of a variable is? The sizeof operator does this automatically for you. The sizeof operator can be applied either to the names of objects or to types. When it is applied to types, the type must be enclosed in parentheses. The operation `sizeof(int)` gives the correct size of an integer in bytes for your system.

In Figure 5.12, the result of the sizeof operation is used as the argument for `malloc`. The sizeof operation will be performed first, resulting in the number of bytes for an integer. Assuming that the size of an integer is 2 bytes, `malloc` has then set aside 2 bytes for the program's use.

```
        malloc(sizeof(int));
```

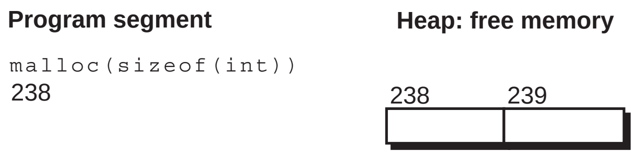

Figure 5.12. malloc function.

`malloc` returns to the program an address of the first byte of all the bytes set aside. An address is always an address of a type of object. However, `malloc` does not know for what type of object the memory is allocated. It only receives an integer value that is the number of bytes to be allocated. The address it returns is a generic address whose type is specified by a generic pointer type. In C, a generic pointer is a pointer to void. The address returned from `malloc` is an address of an unknown type of object, a void object. An address whose type is a pointer to void can be assigned to any pointer variable of any type, with no conversions or casts required.

The address that `malloc` returns can then be assigned to a pointer variable. In Figure 5.13, the address returned by `malloc` is assigned to `numptr`. For the sake of this example, the address consists of the address location 238, and it is cast to the address of an integer. To hold this address, a pointer variable with a pointer type of `int` is needed.

```
      int *numptr;

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

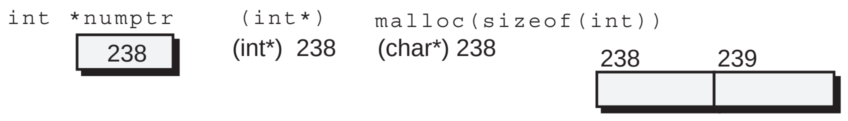

Figure 5.13. malloc and pointer variable.

Notice that `numptr` is a pointer variable. As a variable, it resides in the stack segment. However, its contents consist of the address 238. The address 238 is in the heap segment. A connection has been made between a pointer variable in the stack segment and memory in heap segment. The pointer holds the address of the allocated memory. It is through the pointer variable that your program can use that allocated memory. In the **allocptr.c** program in Listing 5.7, `numptr` obtains its address from `malloc`. `costptr` obtains its address from an address operation on cost. Their addresses, being from different segments, will be very different. Figure 5.14 illustrates how memory is allocated by `malloc` and its address assigned to `numptr`.

LISTING 5.7

`allocptr.c`

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

void  main(void)
      {
      float cost = 7.25;
      float *costptr;
      int *numptr;

      numptr = (int *) malloc(sizeof(int));
      *numptr = 5;
      costptr = &cost;

      printf("*numptr = %d, numptr=%p\n",*numptr,numptr);
      printf("cost = %f,costptr = %p \n", cost, costptr);
      free(numptr);
      }
```

Output of Listing 5.7:

```
      *numptr = 5, numptr = 65000
      cost = 7.25, costptr = 700
```

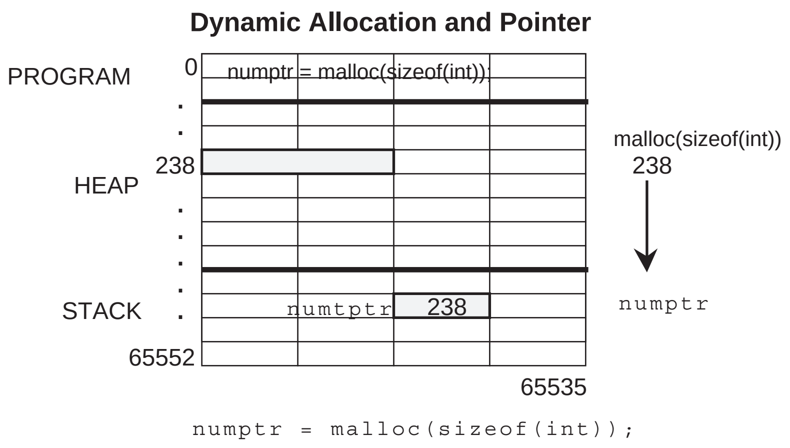
Dynamic Allocation and Pointer
PROGRAM
0
numptr = malloc(sizeof(int));
238
HEAP
STACK
65552
65535
malloc(sizeof(int))
238
numptr
numtptr
238
numptr = malloc(sizeof(int));

Figure 5.14. Dynamic allocation of memory and assignment of address to declared variable.

An indirection operation on `numptr` can then reference the memory at 238 as an integer variable. In Figure 5.15, the memory at 238 was referenced as an integer variable and assigned the value 3.

```
        *numptr = 3;
```

The memory that has been obtained from the heap has not been designated for any other use, as was the memory used for variables. In a sense, what has happened here is a kind of simulation of a variable declaration. A variable declaration designates memory and then uses type and address information to reference that memory. The same thing happens in this situation, using only a pointer. `malloc` designates the memory, and the pointer holds the address and type information with which to reference that memory. An indirection on that pointer will reference that memory as an object. In the program in Listing 5.7, an indirection on `numptr` references an integer object, even though no integer object has been declared. The memory at the address held by `numptr` is treated as the memory of an integer object.

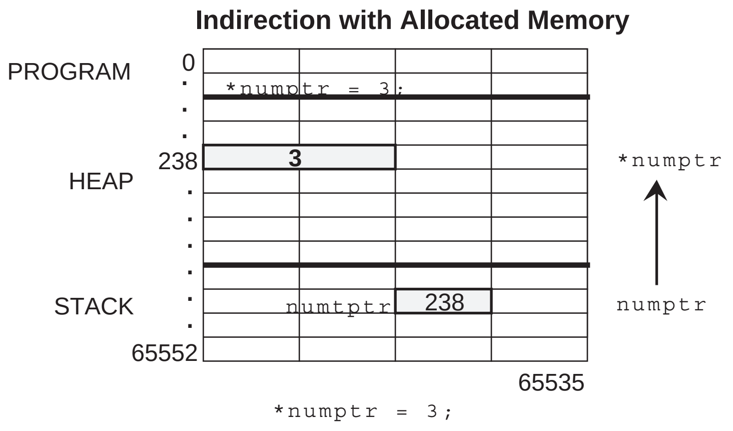

Figure 5.15. Indirection on allocated memory using a pointer variable.

The memory that malloc allocated is reserved and cannot be reallocated until it is freed. The function free will free allocated memory. In Listing 5.7, the function free is called with the pointer numptr to free the memory at the address held by numptr. That memory could then be reallocated in a subsequent call to malloc.

The function declarations for malloc and free are held in a header file called **stdlib.h**. This file needs to be included in any program that uses malloc and free. To do this, you have to place a preprocessor include command for **stdlib.h** at the head of your program, just as you did for the **stdio.h** file: #include <stdlib.h>.

Addresses and Global Memory: Integer Conversions

Addresses are equivalent to unsigned integer values. For example, in a computer with 64,000 bytes of memory, each byte will be referenced by its own address. The 1000th byte will be referenced by the number 1000. However, an address in C is never represented as simply an address. An address can only be represented as an address of a type of object. In other words, an address can only be represented as a pointer. In this sense, an address cannot be written as an integer. It is possible, however, to take an integer and convert it to a pointer. In this case, the integer becomes an address of a type of object. This conversion is effected with a cast.

The type in such a cast operation is a pointer to an object. The cast for a pointer to an integer is (int*). The type in this cast is literally the type used in the declaration of a pointer to an integer-the keyword int and the asterisk symbol: int*.

In the next example, the address resulting from the cast of the integer is assigned to a pointer variable. The pointer variable then uses the address to reference the memory at address 143. An attempt to assign the integer 143 directly to a pointer variable without using a cast may or may not succeed. Some compilers may assume an implied conversion and proceed with the assignment, giving only a warning. However, an attempt to use the integer 143 directly in an indirection operation, without first casting it as a pointer, is invalid.

```
void  main(void)
            {
            int *ptr;

            ptr = (int *) 143;
            *ptr = 5;
            }
```

In next example, a preprocessor define directive is used to define a symbolic constant for the pointer conversion. This makes for a clearer style. The preprocessor will replace the symbolic constant IPTR with the replacement string (int*) 143 before the program is compiled. Symbolic constants provide a way to easily represent global addresses in your program.

```
#define IPTR  (int*) 143

int main(void)
            {
            int *ptr;

            ptr = IPTR;
            *ptr = 5;
            }
```

There is no restriction on the indirection operation in regard to memory. It may use the same memory a variable uses. It may use memory allocated by special allocation functions. It may even use memory that has not been allocated at all. In this respect, an indirection operation can use an address to access memory located anywhere. It can be used to access memory that has been reserved for other variables, the program itself, or even, in some cases, the operating system.

This pointer conversion of an integer constant is as close as C comes to an explicit pointer constant. Integer values have corresponding integer constants. Pointer values do not. Though, technically, an address is equivalent to an integer, an address is always an address of a type of object, never just a bare address. To use an integer as an address, you first need to convert it to a pointer.

Addresses as Pointer Values: Pointer Expressions

You will find that the term "pointer" is used in several different ways. A pointer is defined as a variable that holds the address of another variable. At the same time, the term pointer is also used to refer to the address itself. A closer look at the pointer type may be helpful in clearing up this confusion. A pointer type consists of a data type and the pointer type specifier, the asterisk. The pointer type of an integer pointer is `int*`. The asterisk is the actual pointer specifier. The data type is the type of object the pointer points to. A pointer type always consists of both the pointer specifier (the asterisk) and the data type (the type of object it points to). A pointer is always a pointer to a type of object.

A pointer type is used to declare pointer variables. As such, pointer variables are referred to as pointers. However, there are expressions that result in addresses. The type of such an address is that of a pointer type. An address is not an integer, nor is there any special address type. Since this address's type is a pointer type, an address can also be referred to as a pointer value. It is an address of a type of object. This pointer value has not only address information, but also type information. This means that the address resulting from a pointer expression has the information with which an object can be referenced. Such an address can be used to locate memory, and the address's pointer data type can be used to interpret that memory. An address with a pointer type, a pointer value, bears both address and type information. The indirection operation can be performed directly on it. The indirection operation is not restricted to pointer variables. It operates on any pointer value, whether that address is held in a pointer variable or is the result of a pointer expression.

The notion of a pointer expression is new. A pointer expression is an expression that results in an address. The address operation is a pointer expression whose result is a pointer value: an address of a variable. The `malloc` function call is a pointer expression that results in a pointer value: an address of allocated memory. A cast operation on an integer is a pointer expression whose result is a pointer value: an address designated by the integer value and cast to a pointer type. All of these addresses are pointer values that have pointer types through which they can reference specific types of objects.

The indirection operation can work directly on these addresses. In the next example, there are three different indirections directly on addresses. In the first case, the address operation on `num` results in the address of `num`. An indirection on that address references an integer at `num`'s memory. The value 5 is assigned to it.

In the second case, the function call to `malloc` results in an address, which is then cast to an integer address. An indirection on this address references the memory at that address as an integer. The value 5 is also assigned to it.

In the third example, a cast operation on the integer 143 results in an address of a `float`. The indirection operation references the memory at address 143 as a `float` variable. The value 7.25 is assigned to it.

```
        *(&num) = 5;
```

```
      *( (int*) malloc(sizeof(int)) ) = 5;
      *((float*)143 ) = 7.25;
```

This cast operation can also be written using a define command like that in the previous example.

```
      #define FPTR  (float *) 143

            *(FPTR) = 5;
```

There are other kinds of pointer expressions, most notably those used to manage arrays. The basic concept, however, is the same. The pointer expression results in an address that can then be used in an indirection operation to reference an object.

Chapter Summary: Pointers

Pointer variables hold the addresses of objects. These addresses can be used to reference an object. A pointer variable is declared with a data type and a pointer type specifier, the asterisk. Pointers are declared to reference one data type. One can speak of different types of pointers. A pointer to an integer will reference only integers. A pointer to a `float` can only reference `float`s.

An address is always an address of a type of object. It holds not only location information, but also type information. The address of a variable is obtained with the address operation. This address can then be assigned to a pointer variable. An indirection operation can then use this address to reference the variable. An indirection operation on a pointer that holds the address of a variable is just as valid a reference to that variable as the variable's own name. Such an indirection operation can be used anywhere a variable's name can be used, even in an assignment operation.

Addresses can be obtained in several different ways. An address can be obtained from a variable using the address operation. An address can also be obtained from free memory using the `malloc` function. An address can be obtained by converting an integer to an address with a cast operation.

An address is an address of a type of object, not simply a numeric location. An address holds both type and location information. For this reason, the indirection operation works on any address, whether it is held in a pointer variable or is the result of some operation that results in an address. In this sense, an address can be thought of as a pointer itself.

TABLE 5.1 Pointers and Addresses

Pointer Declaration	type * pointer name ;
	`int *numptr;`
	`float *costptr;`

Address Operation	&variable
	`numptr = &num;`
	`*(&num) = 5;`

Allocated Memory	(type *) malloc(sizeof(type)) `numptr = (int*) malloc (sizeof(int));`

Integer Conversion	(type *) integer
	`numptr = (float *) 143;`
	`*((float*) 143) = 7.25;`

Exercises

1. Modify the following program by adding more variables and pointers to variables. Assign the address of each variable to a pointer. Then use indirection on the pointer in assignment statements. Print out both pointer indirections and addresses in pointers. Print out variables and addresses of variables.

```
#include <stdio.h>

void  main(void)
      {
      int num;
      int *numptr;

      numptr = &num;
      *numptr = (3 * 5) / 2;

      printf("Indirection = %d, Address = %p\n ",
                        *numptr, numptr);
      printf("Num= %d, Num Address= %p\n ",num,&num);
      }
```

2. Modify the following program by adding pointers of different types. Assign values and print out their addresses. Don't forget to use indirection when you have to.

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

int main(void)
      {
      int *numptr;

      numptr = malloc(sizeof(int));

      *numptr =  (3 * 5) / 2;
      printf("Numptr Indirection = %d \n", *numptr);
      printf("Num Address = %p\n ", numptr);
      return 0;
      }
```