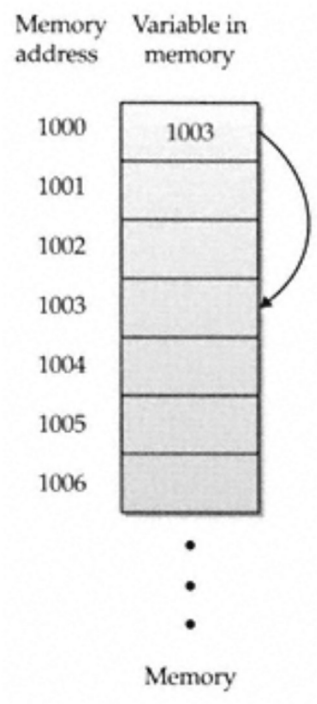
Pointers

What Are Pointers?

A pointer is a variable that holds a memory address. This address is the location of another object (Typically, another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second.



In the above figure one variable points to another.

Pointer Variables

If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

```
type *name;
```

where type is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by name.

when you declare a pointer to be of type int *, the compiler assumes that any address that it holds points to an integer— whether it actually does or not. (That is, an int * pointer always ''thinks" that it points to an int object, no matter what that piece of memory actually contains.) Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.

The Pointer Operators

The pointer operators were discussed in Chapter 2. We will review them here. There are two pointer operators: * and &. The & is a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.) For example,

```
m = &count;
```

places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. You can think of & as returning "the address of." Therefore, the preceding assignment statement can be verbalized as "m receives the address of count."

To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also assume that count has a value of 100. Then, after the preceding assignment, m will have the value 2000.

The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows. For example, if m contains the memory address of the variable count,

```
q = *m;
```

places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m. You can think of * as "at address." In this case, the preceding statement can be verbalized as "q receives the value at address m."

Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
#include <stdio.h>
int main(void)
{
 int x = 99;
 int *p1, *p2;

 p1 = &x;
 p2 = p1;

 /* print the value of x twice */
 printf(''Values at p1 and p2: %d %d\n", *p1, *p2);

 /* print the address of x twice */
 printf("Addresses pointed to by p1 and p2: %p %p", p1, p2);
 return 0;
}
```

```
p1 = &x;
p2 = p1;
```

p1 and p2 both point to x. Thus, both p1 and p2 refer to the same object. Sample output from the program, which confirms this, is shown here.

```
Values at p1 and p2: 99 99
Addresses pointed to by p1 and p2: 0063FDF0 0063FDF0
```

Notice that the addresses are displayed by using the ***%p printf( )*** format specifier, which causes ***printf( )*** to display an address in the format used by the host computer.

Pointer Conversions

One type of pointer can be converted into another type of pointer. There are two general categories of conversion: those that involve ***void **** pointers, and those that don't. Each is examined here.

In C, it is permissible to assign a ***void **** pointer to any other type of pointer. It is also permissible to assign any other type of pointer to a ***void **** pointer. A void * pointer is called a generic pointer. The ***void **** pointer is used to specify a pointer whose base type is unknown. The ***void **** type allows a function to specify a parameter that is capable of receiving any type of pointer argument without reporting a type mismatch.

```
#include <stdio.h>
int main(void)
{
 double x = 100.1, y;
 void *p;

 /* The next statement causes p (which is an void pointer) to point to a
double. */

 p = &x;

 /* The next statement operate as expected. */

 y = *((double *) p); /* attempt to assign y the value x through p */

 /* The following statement will output 100.1. */

 printf("The value of y is: %f", y);

 return 0;
}
```

Pointer Arithmetic

There are only two arithmetic operations that you can use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume ints are 2 bytes long. After the expression

```
p1++;
```

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression

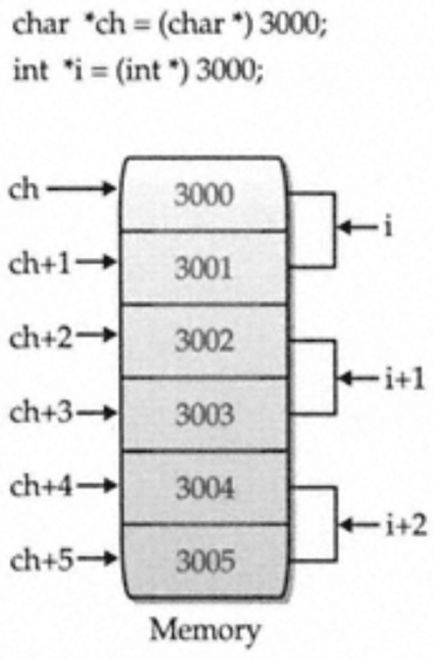
```
p1--;
```

causes p1 to have the value 1998.

You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

```
p1 = p1 + 12;
```

makes p1 point to the 12th element of p1's type beyond the one it currently points to.Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You can subtract one pointer from another in order to find the number of objects of their base type that separate the two. All other arithmetic operations are prohibited. Specifically, you cannot multiply or divide pointers; you cannot add two pointers; you cannot apply the bitwise operators to them; and you cannot add or subtract type float or double to or from pointers.



All pointer arithmetic is relative to its base type (assume 2-byte integers)

Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, p1 has been set to the address of the first array element in str. To access the fifth element in str, you could write

```
str[4]
```

or

```
*(p1+4)
```

Both statements will return the fifth element. Remember, arrays start at 0. To access the fifth element, you must use 4 to index str. You also add 4 to the pointer p1 to access the fifth element because p1 currently points to the first element of str. (Recall that an array name without an index returns the starting address of the array, which is the address of the first element.)

Multiple Indirection

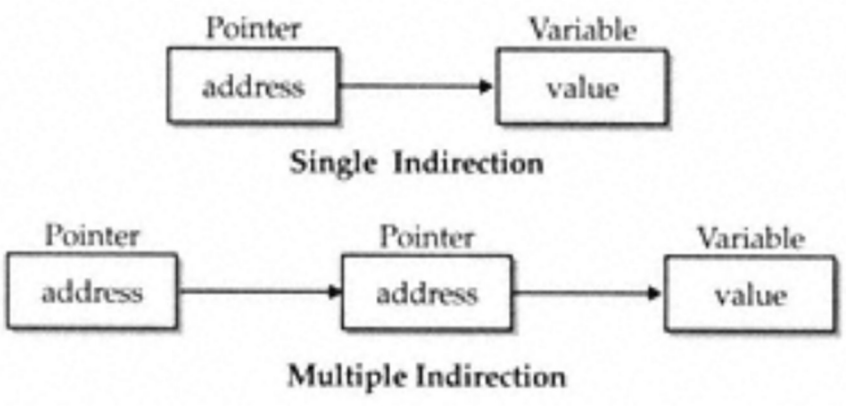
You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection, or pointers to pointers. Pointers to pointers can be confusing. Figure 5-3 helps clarify the concept of multiple indirection. As you can see, the value of a normal pointer is the address of the object that contains the desired value. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the desired value.

Multiple indirection can be carried on to whatever extent desired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors.

A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk in front of the variable name. For example, the following declaration tells the compiler that ***newbalance*** is a pointer to a pointer of type ***float***:

```
float **newbalance;
```

You should understand that ***newbalance*** is not a pointer to a floating-point number but rather a pointer to a ***float*** pointer.



To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include <stdio.h>

int main(void)
{
int x, *p, **q;

x = 10;
p = &x;
q = &p;

printf("%d", **q); /* print the value of x */

return 0;
}
```

Here, ***p*** is declared as a pointer to an integer and ***q*** as a pointer to a pointer to an integer. The call to ***printf( )*** prints the number ***10*** on the screen.

Structures

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A structure declaration forms a template that can be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called members. (Structure members are also commonly referred to as elements or fields.)

Usually, the members of a structure are logically related. For example, the name and address information in a mailing list would normally be represented in a structure. The following code fragment shows how to declare a structure that defines the name and address fields. The keyword struct tells the compiler that a structure is being declared.

```
struct student
{
 char name[30];
 int regdno;
 int marks[5];
 char branch[20];
}Ram,Shyam,Ghanshyam;
```

Notice that the declaration is terminated by a semicolon. This is because a structure declaration is astatement. Also, the structure tag ***student*** identifies this particular data structure and is its type specifier.

The general form of a structure declaration is

```
struct tag {
 type member-name;
 type member-name;
 type member-name;
 .
 .
 .
} structure-variables;
```

where either tag or structure-variables may be omitted, but not both.

Accessing Structure Members

Individual members of a structure are accessed through the use of the . operator (usually called the dot operator). For example, the following statement assigns the regd number 12345678 to the ***regdno*** field of the structure variable ***Ram*** declared earlier:

```
Ram.regdno = 12345678;
```

The object name (in this case, Ram) followed by a period and the member name (in this case, regdno) refers to that individual member. The general form for accessing a member of a structure is

```
object-name.member-name
```

Therefore, to print the regdno on the screen, write

```
printf("%d", Ram.regdno);
```

This prints the ***regdno*** contained in the ***regdno*** member of the structure variable ***Ram***.

In the same fashion, the character array ***Ram.name*** can be used in a call to ***gets( )***, as shown here:

```
gets(Ram.name);
```

This passes a character pointer to the start of name.

Structure Assignments

The information contained in one structure can be assigned to another structure of the same type using a single assignment statement. You do not need to assign the value of each member separately. The following program illustrates structure assignments:

```
#include <stdio.h>

int main(void)
{
struct student
{
char name[30];
int regdno;
int marks[5];
char branch[20];
}Ram,Shyam,Ghanshyam;

Ram.regdno = 12345678;

Shyam = Ram; /* assign one structure to another */

printf("%d", Shyam.regdno);

return 0;
}
```

After assignment ***Shyam.regdno*** will contain 12345678.