```
float average = sum/i; // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

3.12 Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name
```

Example:

```
float total = 100;
float & sum = total;
```

total is a **float** type variable that has already been declared; **sum** is the alternative name declared to represent the variable **total**. Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];
int & x = n[10];        // x is alias for n[10]
char & a = '\n';        // initialize reference to a literal
```

The variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant **\n** is stored.

The following references are also allowed:

```
i.  int x;
    int *p = &x;
    int & m = *p;

ii. int & n = 50;
```

The first set of declarations causes **m** to refer to **x** which is pointed to by the pointer **p** and the statement in (ii) creates an **int** object with value 50 and name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)          // uses reference
{
  x = x+10;              // x is incremented; so also m
}
int main()
{
  int m = 10;
  f(m);                  // function call
  .....
  .....
}
```

When the function call **f(m)** is executed, the following initialization occurs:

```
int & x = m;
```

Thus x becomes an alias of m after executing the statement

```
f(m);
```

Such function calls are known as *call by reference*. This implementation is illustrated in Fig. 3.2. Since the variables **x** and **m** are aliases, when the function increments **x, m** is also incremented. The value of **m** becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

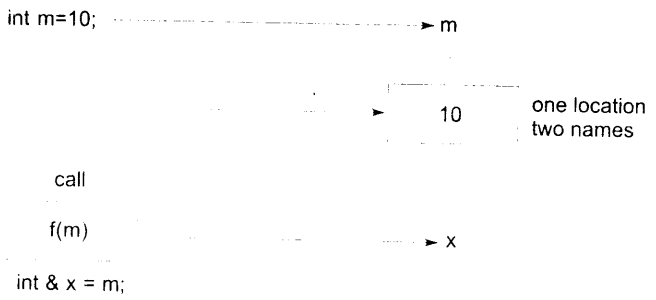


Fig. 3.2 ⇔ *Call by reference mechanism*

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

3.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

```
::        Scope resolution operator
::*       Pointer-to-member declarator
->*       Pointer-to-member operator
.*        Pointer-to-member operator
delete    Memory release operator
endl      Line feed operator
new       Memory allocation operator
setw      Field width operator
```

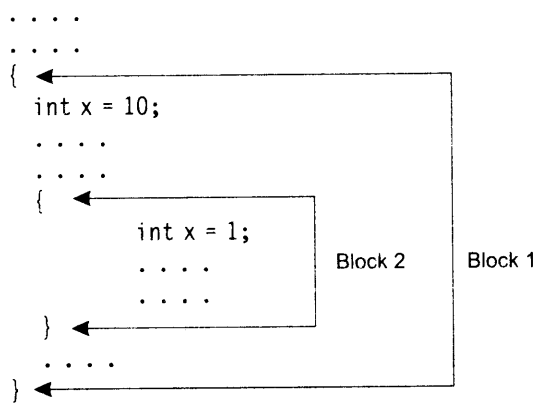
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

3.14 Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
.....
.....
{
   int x = 10;
   .....
   .....
}
.....
.....
{
   int x = 1;
   .....
   .....
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to