

**Chapter 8: User Defined Functions**  
**Satish Singhal Ph. D.**  
**Version 1.05 (Alpha Version: Not Proof Read)**

**Table of Contents**

Chapter Goals .....	2
Function Machine .....	2
Function Calls .....	4
Transfer of Control during Function Call .....	10
Main Points about Writing Functions .....	13
More on Function Parameters and Function Call Arguments .....	14
Match between Function Parameters and Arguments .....	15
Order of Evaluation of Arguments .....	17
Call By Value Mechanism .....	19
Reference to Variables in C++ .....	21
Pass by Reference Mechanism .....	23
Example of Pass By reference .....	24
Comparison Between Pass-By-Value And Pass-By reference Mechanisms .....	29
More Examples of Pass by Reference .....	32
Passing a const Reference .....	35
Example of Use of Const Reference Passing .....	35
Function Name Overloading .....	40
Example of Function Name Overloading .....	41
Overloading of print Function .....	44
main Function .....	45
Functions with Default Arguments .....	46
print Function with Default cout argument .....	48
Congruency of Function Prototype, Function Header and Function Call .....	50
Data Validation Functions .....	52
string streams .....	54
Incremental Development .....	60
Software Engineering Related Practices .....	60
User Notes .....	60

## **Chapter Goals**

Functions are the main vehicle for modularizing the C++ programs. You will remember from chapter two that we decomposed context diagram into Data Flow Diagram (DFD), where software system was broken down into many components. In software engineering, the best practice is to break down system in a manner, that each component can be coded in the form of a function. Once all functions are written, then they can be combined into a program or software system. This allows a team of programmers to work on a software project, where each programmer just has to know the details of functions they have to write. This allows the software projects to complete faster. There is another important reason for modularization of software using functions. Software engineering has established in that number of “bugs” in the software increase to some power of number of code lines in a module. Therefore, if bigger modules can be broken into smaller ones, by using functions, then number of bugs reduce proportionately. C++ language is ideally suited for modularization. The language design says that:

- Each C++ program must have at least one function, which is called main.
- Program execution always begins with the first line in function main.
- Any other functions must be called by main to do their task.

We discussed concept of functions in chapter four and contrasted it to the concept of functions in mathematics. We expand on that concept further.

Imagine a function called double. In mathematics, we would write it as follows:

$$F(x) = 2 * x;$$

The above definition requires that returned value of  $F(x)$  would always be twice the value of  $x$  for all “legal values” of  $x$ . In this case, it just so happens that all numeric values of  $x$ , that do not cause an overflow are legal. Functions, which are definable, over all possible values of the arguments ( $x$  in this case), are called “total functions”<sup>1</sup>. Notice that the function  $F(y)$  given below is not a total function.

$$F(y) = y/0 ;$$

The reason that  $F(y)$  is not a total function is because it is not definable for the value  $y = 0$ , as you would know from basic mathematics that zero divided by itself is an indeterminate form. For non-zero values of  $y$ ,  $F(y)$  has a constant value of infinity, which is definable but is not given a definite value.

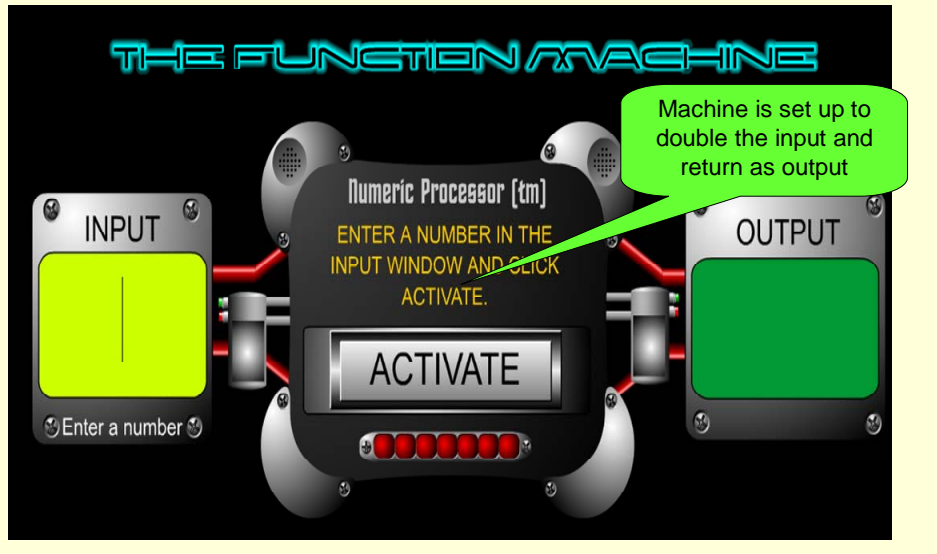
## **Function Machine**

We can imagine all C++ functions to operate like machine or like a black box. (In fact, some software machines are given a special name called “virtual machine”. For example, Java Virtual machine installed in your web browser to enable it to display Java Applets is one example of software machine or virtual machine). Figure 8.1 shows the dynamics of functions or function machine.

---

<sup>1</sup> One should attempt to define all functions that are to be used by the client as “total functions”. We say “attempt”, because at times it may not be possible to do so. If a function is not a “total function”, then limitations and domain of its use must be explicitly outlined in the specification of the function given to the client. Client in this case is the user of the software in which the function is included for direct client use.

## Initial State of Function Machine

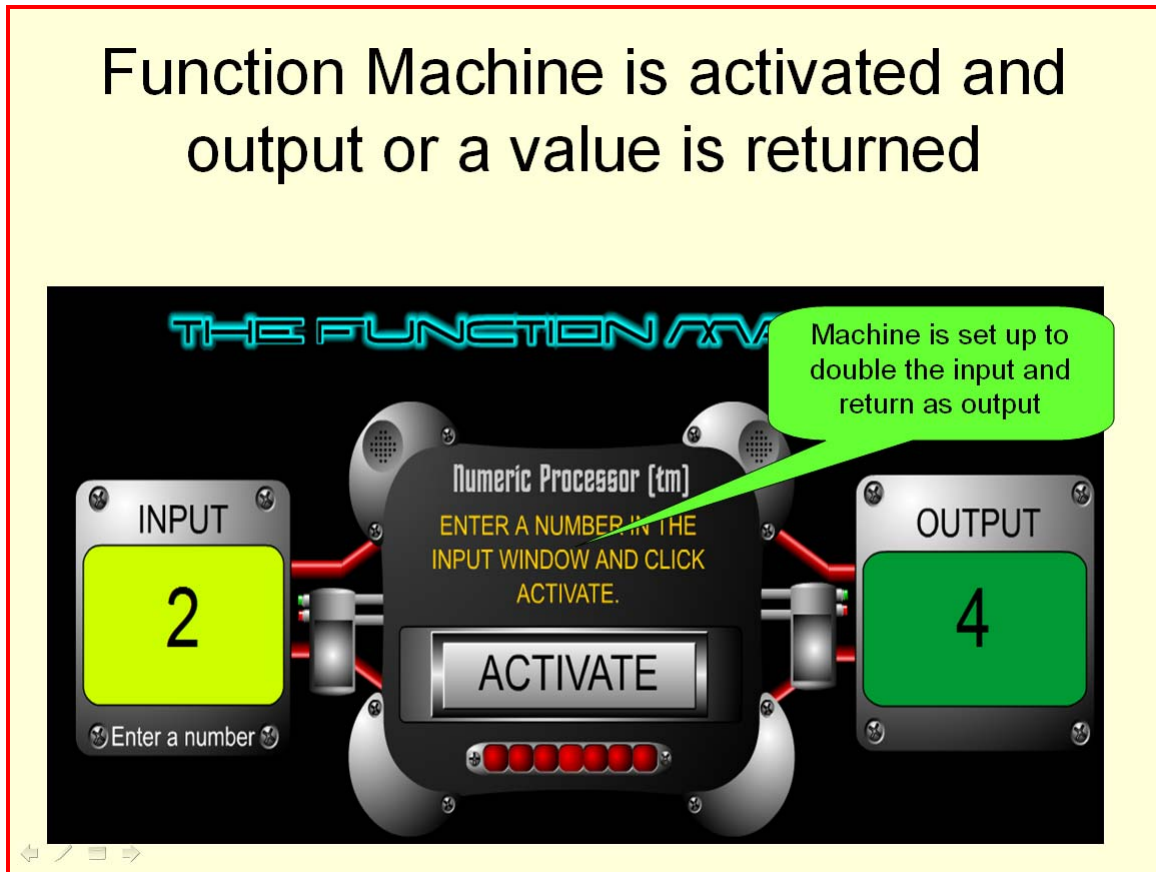


**FIG. 8.1A**

The initial state of function machine, which has the following function definition stored in it (Figure 8.1A):

$$F(x) = 2 * x;$$

When a value is entered in the INPUT box and the “ACTIVATE” switch is pressed then function machine is “churned” and it returns the value generated by the machine, which is twice the value of input. Figure 8.1B shows the value returned when input is two and output generated is four.

**FIG. 8.1B**

The process of pressing the switch “ACTIVATE” is analogous to making a function call. You have already learned how to make function calls when you called library functions *pow* and *sqrt* in chapter four to do some mathematical calculations. Additional examples are shown shortly. In essence, all value returning C++ functions work just like a function machine demonstrated above. The client of the function only needs to know as how to use the machine and the legal inputs acceptable by the machine. The client does not need to know as to how the machine works or details of its definition. For example, you were able to use *sqrt* function without actually knowing its source code. This principle is called “Information Hiding” in Software Engineering.

### Function Calls

In the function machine example given above the pressing of switch “ACTIVATE” produced the output from the function machine. In C++, this is called “calling a function”. The main function can call any user defined function and functions can call each other. In this sense, you can think of a C++ program as many tiny function machines assembled in a big machine. C++ language allows user defined functions to call main, but result will be a recursive call to main, which is rarely useful. The Figure 8.2A below gives the details of how to make function calls.

## Function Calls

One function calls another by using the name of the called function next to ( ) enclosing an argument list.

A function call temporarily transfers control from the calling function to the called function.

**FIG. 8.2A**

We will see in an example as how program transfers control from one function to another during a function call. The Figure 8.2B shows the function call syntax.

## Function Call Syntax

FunctionName ( Argument List )

Commas separate arguments passed to the function.

The argument list is a way for functions to communicate with each other by passing information.

The argument list can contain 0, 1, or more arguments, separated by commas, depending on the function.

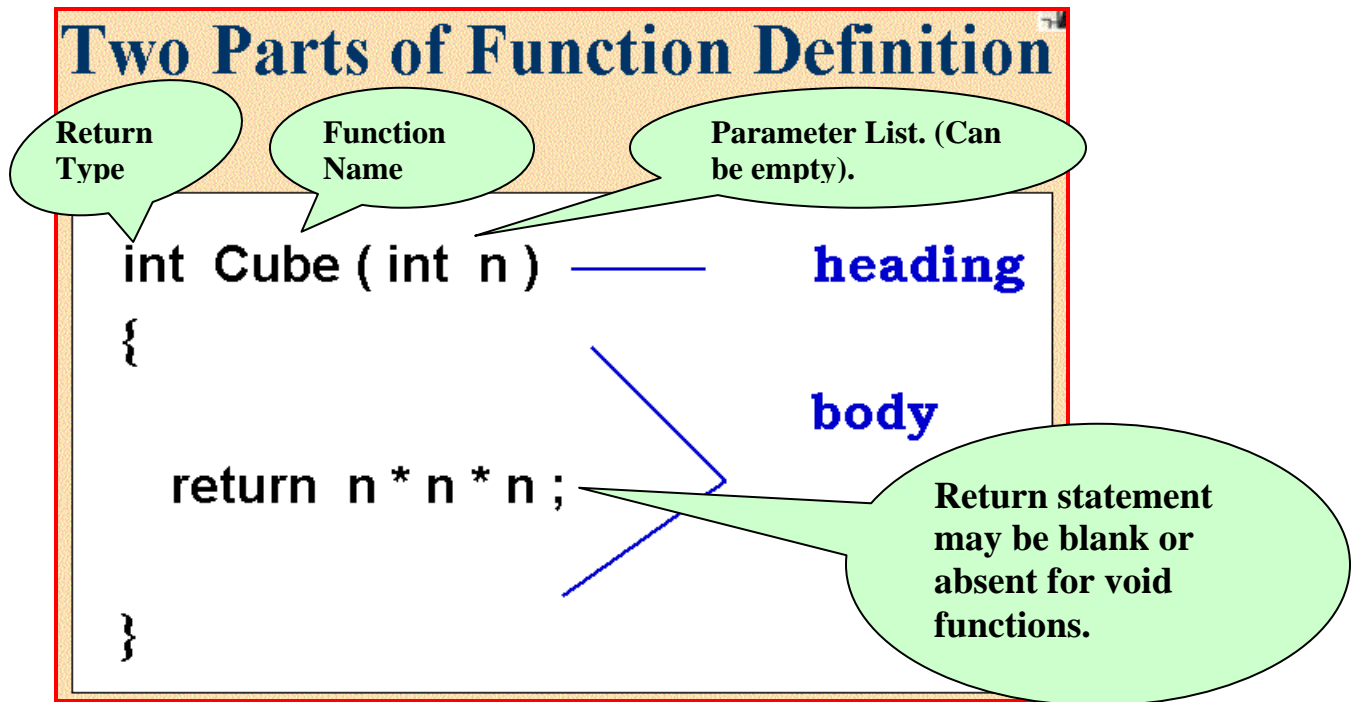
**FIG. 8.2B**

In addition, a value returning function call would be on right side of an expression, whereas the void functions would be written as stand-alone program statements.

One can define a function in two parts – its heading, which has the name and argument list, and the body, which has the function code or definition. For example if a function gets an



integer as an argument and it cubes that integer and returns the cubed value, then the overall function will look like Figure 8.3.



**FIG. 8.3**

The functions in C++ can be two types, the value returning functions or void functions. Figure 8.4 compares void and value returning functions.

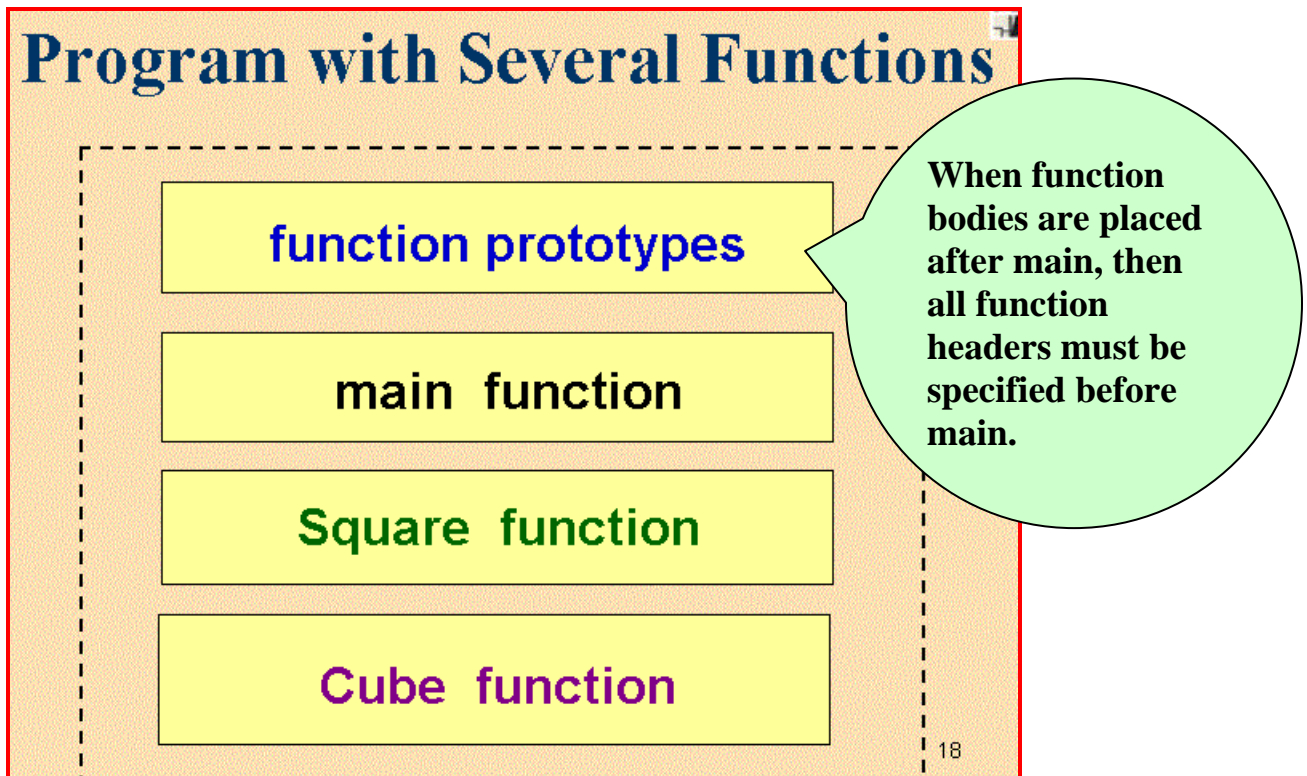
Two Kinds of Functions	
Value-Returning	Void
Always returns a <b>single value</b> to its caller and is called from within an expression.	Never returns a value to its caller, and is called as a <b>separate statement</b> .

**FIG. 8.4**

The main function in C++ can be a void or a value returning function, though if one codes main as a value returning function, then the program is more portable. Overall program may include a combination of value returning and or void functions. My personal

preference is always to code every function as a value returning function (at least return a boolean) so that calling function knows (by the mechanism of return value) that called function reached its end successfully.

Let us assume that we have a C++ program with main and two user-defined functions, which are named square (to return the square of a number) and cube (to return the cube of a number). Then the arrangement of the functions in a C++ program may look something similar to the Figure 8.5.



**FIG 8.5**

Let us say that we need a program to calculate squares and cubes of many numbers. The task of calculating square and cube is best left to two separate functions. The function square will return the squared value of its argument, and function cube will return the cubed value. (Other functions may be included as needed). Listing 8.1 shows main, and functions square, and cube to accomplish the above tasks. We have added the void function print, which prints all the results.

```
#include <iostream>
using namespace std;

int square(int);
int cube(int);
void print(int, int, int);
```

1. Function proto-types.

2. Formal parameter list.

```

int main()
{
    int value = int();
    int square_val = int();
    int cube_val = int();
    bool done = false;
    char input = char();

    while(!done)
    {
        cout<<"Please enter an integer, whose "
              <<"square and cube you need : ";
        cin>>value;

        if(!cin)
        {
            cout<<"Invalid data entered.\n";

            done = false;
            cin.clear();
            cin.ignore(1000, '\n');
        }
        else
        {
            square_val = square(value);
            cube_val = cube(value);
            print(value, square_val, cube_val);
        }
    }

    cout<<"More data? Type [Y]es or [N]o : ";
    cin>>input;
    if(input == 'Y' || input == 'y')
        done = false;
    else
        done = true;
}

cout<<"Thank you for using Square and Cube Program.\n";
return 0;
}
//*****
int square(int value)
{
    return (value*value);
}
//*****

```

### 3. Actual parameter list.



```

int cube(int value)
{
    return (value*value*value);
}
//*****

void print(int value, int square_val, int cube_val)
{
    cout<<"The square of "<<value
        <<" is = "<<square_val<<"\n";
    cout<<"The cube of "<<value
        <<" is = "<<cube_val<<"\n";

    return;
}

```

### Listing 8.1

We see that after including necessary header files, we must declare the proto-types of functions that will be part of our program (bubble #1). The proto-types are necessary, because they tell compiler that a particular function name will be used inside the main and its definition is provided later. This is required by C++ rule, which says that all identifiers must be declared before their use! If function proto-type is not included, then compiler will generate an error, when it reads the statement in the main, where program calls the function. This error would be similar to using a variable, with out first declaring it. The function proto-types include a comma separated parameter list and they are called “formal parameters”. When writing the list of formal parameters it is sufficient to include only the parameter data types.

After providing proto-types and after the body of main function, one can write full function definition in any order, as no particular order is expected. In the main we proceed sequentially up to the statements:

```

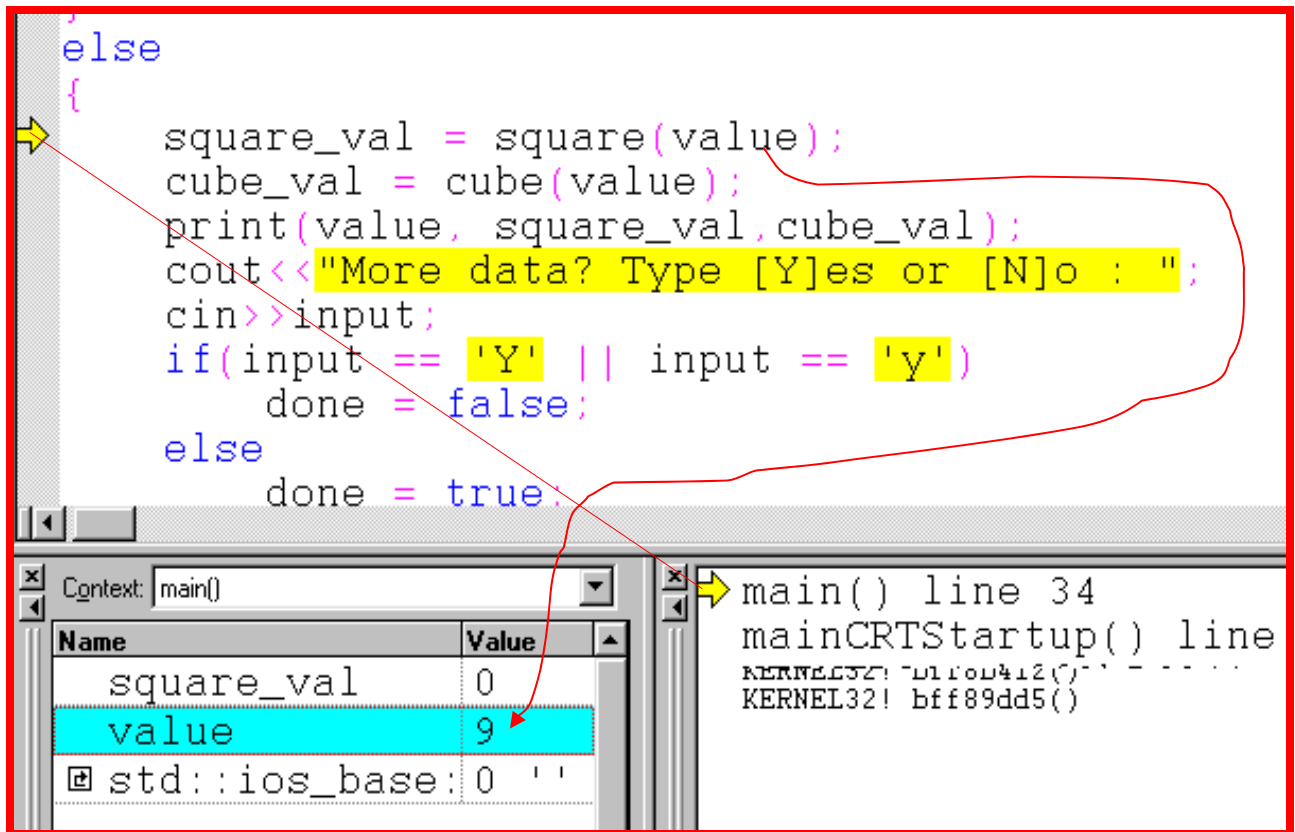
cout<<"Please enter an integer, whose "
    <<"square and cube you need : ";
cin>>value;

```

Based on the condition of input stream the program then makes a selection. If input stream is not in failed state, then the else clause is executed and a call to the function square is made. At that point, the program transfers the control to the function square, but it remembers as to where it was in the main, so that it can come back to the same line, after the execution of function square is over. We discuss transfer of control during a function call further.

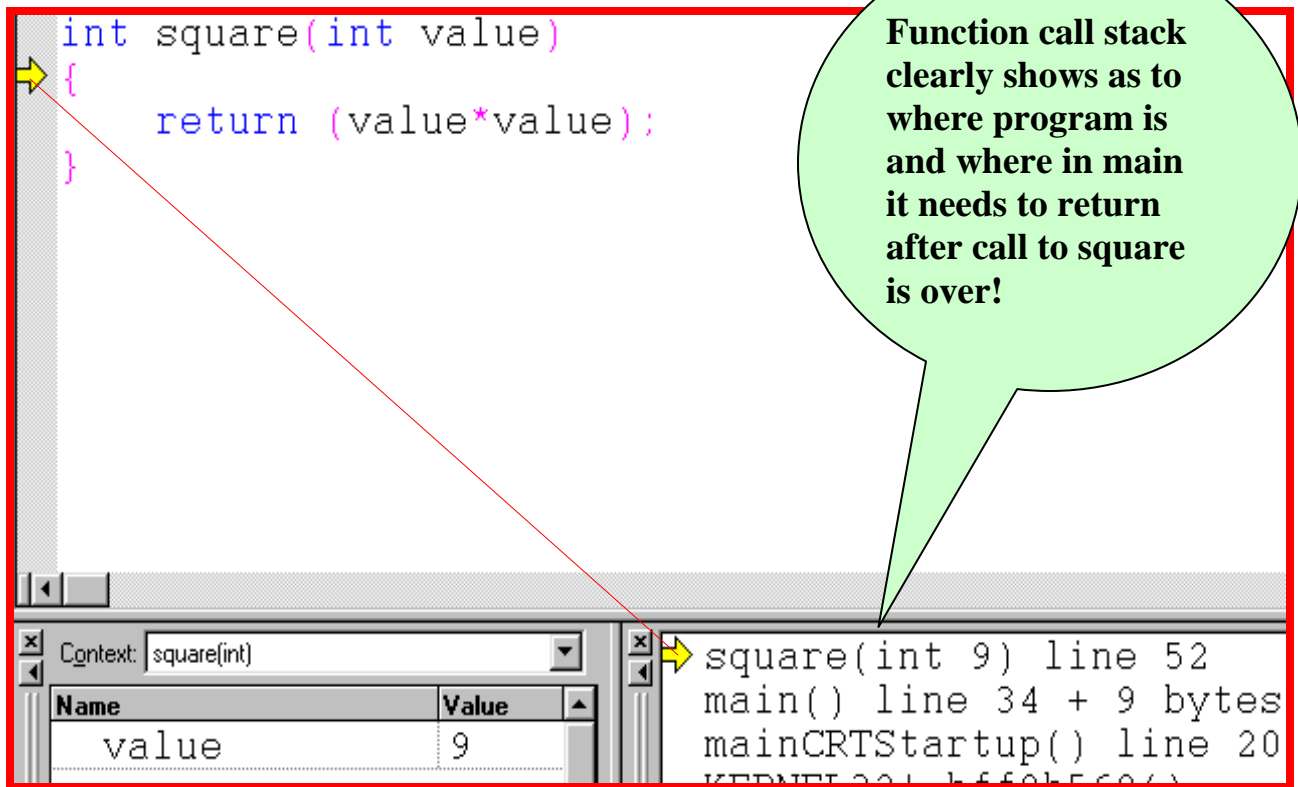
### Transfer of Control during Function Call

We use Visual C++ debugger to show the state of program control, before, during and after the function call (Figures 8.5 to 8.7).



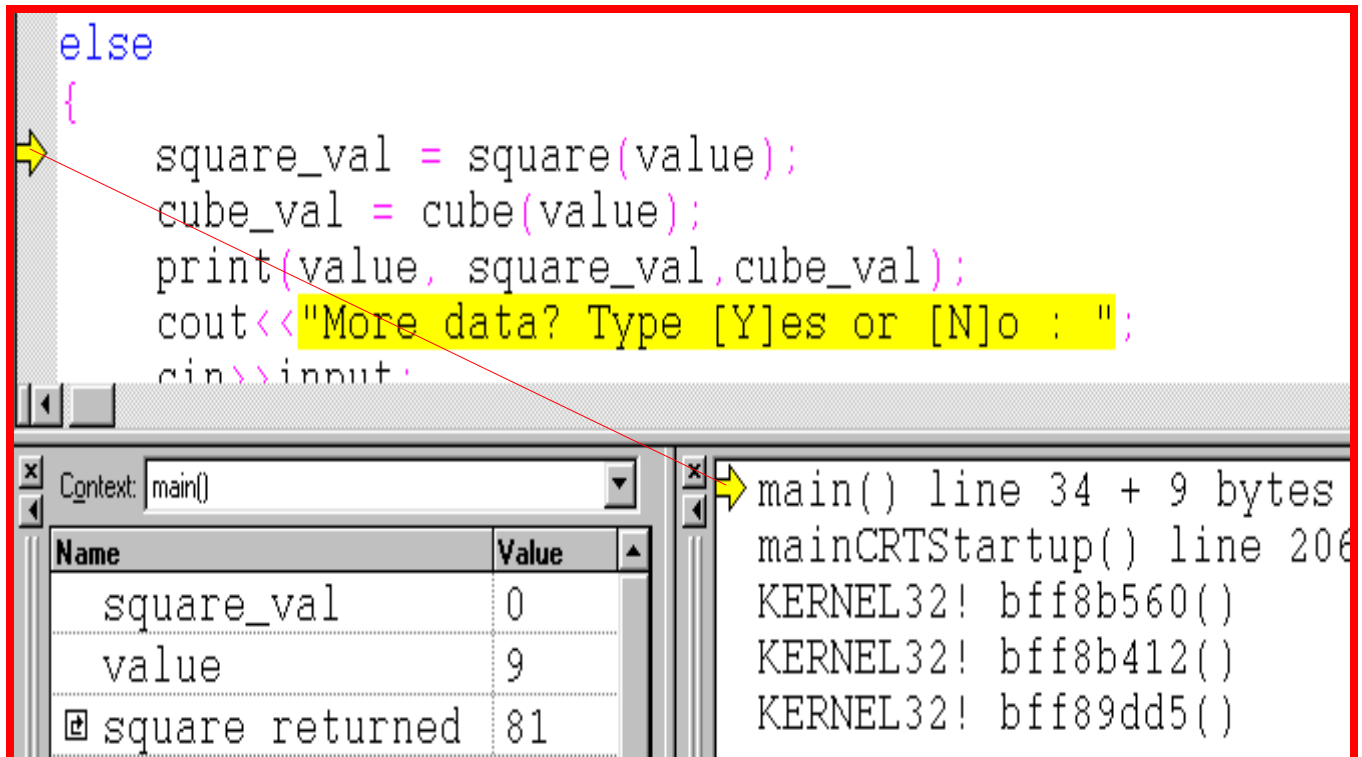
**FIG. 8.5**

The Figure 8.5 shows the situation, just before the control is transferred to the function square by the main. The value being passed to Function Square is nine, and program is executing line number 34 in main function. At this point, by pressing the F11 key on the keyboard, we can see the actual transfer of control to function square (Figure 8.6)



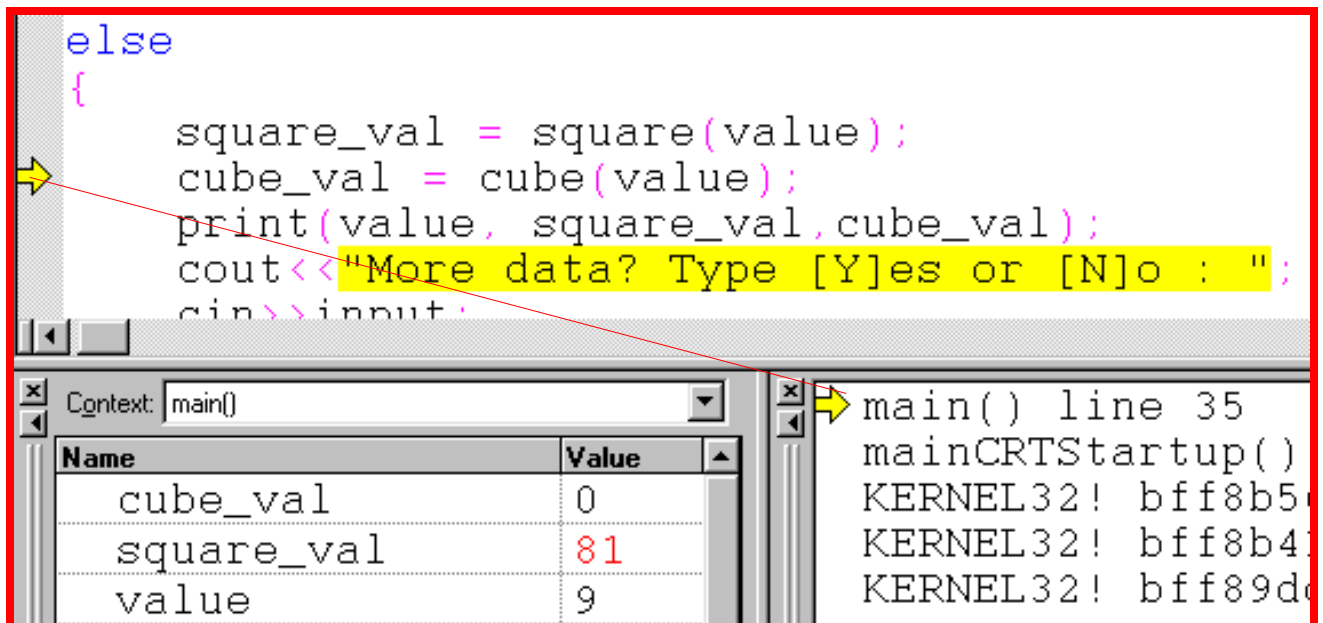
**FIG. 8.6 - Control transferred to function square.**

The line number (52) is actually the line number from the top of the program. However, the statement in right side window clearly shows that control of the program has been transferred to function square. The call stack window also shows that square function receives an integer argument of value nine. The program also remembers that after executing the function square it is supposed to return to main on line number 34. Indeed that is what happens (Figure 8.7).



**FIG. 8.7 - Control transferred back to main to the same line (34) Return value from square is 81**

Program returns to the line number 34 of the main function. The left program window shows that function square returned a value of 81, but this value is not yet deposited into the memory location called square\_val. That will happen when we move to the next line of the program, i.e. e. line 35 (Figure 8.8).



**Figure 8.8**

Now the value of variable `square_val` is changed from zero to 81, as the value returned from the function `square` is deposited into the memory location `square_val`.

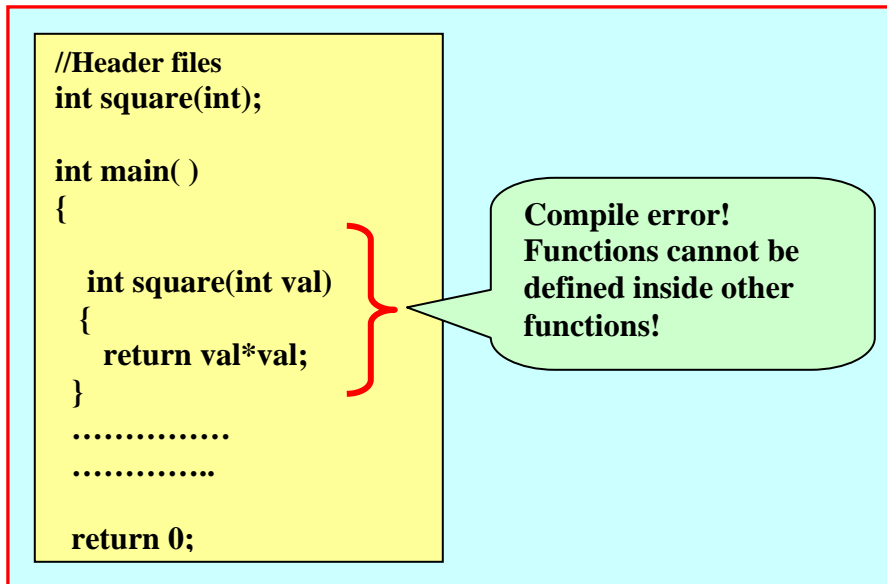
### Main Points about Writing Functions

Now we summarize the main points about writing functions in C++.

- The function prototypes written on top of main are also called function signature.
- The prototype only needs a list of formal parameters as function arguments. This means that it is only enough to specify the data types that will be passed to the function. Variable names are not necessary. However, sometimes variable names are included to clarify the nature of each formal argument.
- When calling a function, then the arguments passed to it, must be actual program data values, which are called “actual arguments”. These actual arguments can be:
  - Initialized variables,
  - Literal constants,
  - Named constants,
  - Call to another value returning function,
  - A mixture of two or more of above in an expression.
- The return statement may be a literal constant (like `main` returns 0), the variable that contains a value, or an expression which function can evaluate to some value before returning it. Expression must be some legal C++ expression.
- A function may contain several return statement (generally subjected to some if-else or switch control structure), but only one of them will be executed per invocation.
- The function prototypes on top of main can be written in any order, and their full definition after the main can be written in any order.

The function prototypes can be declared inside the main, as long as the declaration is done before the function call is made. This however, is not a good practice, because then other functions cannot call the function whose prototype is declared with in main. C++ does not allow functions to be defined inside functions! For example below is a compiler error (Figure 8.9):



**FIG. 8.9**

You have noticed so far that value returning functions are part of expressions. For example in Listing 8.1 call to value returning function square is on the right side of the expression:

```
square_val = square(value);
```

On the other hand void function call is not a part of an expression. The call to a void function stands alone by itself. For example in Listing 8.1 when the void function print is called the call syntax includes a stand-alone statement:

```
print(value, square_val, cube_val);
```

Notice also the contrast between the formal parameters (in the pro-type\_ and actual parameters (in the function call). (See bubbles #2 and #3 in Listing 8.1). The formal parameter list includes data types but actual parameter list includes only the actual data carrying entities.

### **More on Function Parameters and Function Call Arguments**

Function parameters and arguments are two different things but they are related. Their details are shown below in Figure 8.10A.

Classified by Location	
Arguments	Parameters
Always appear in a <b>function call</b> within the calling block.	Always appear in the function <b>heading</b> , or <b>function prototype</b> .

**FIG. 8.10A**

Parameters are the data types and the variable names in the prototype and in function heading, though the prototype does not need the variable names. Arguments are the variable names, literal constants, named constants, or value returned from other functions that appear in the function call statement. Some C++ texts use the term “actual parameters for the arguments”, in which case they call the parameters as “formal parameters”.

### Match between Function Parameters and Arguments

Though function parameters and actual arguments may have different names, they must match in total number, data type, and order in which they are placed with the parenthesis.

Example:

If there is a function prototype, like:

```
float surfaceAreaOfCylinder(float height, int base_radius);
```

then the call to function must of following type:

```
float area = surfaceAreaOfCylinder(10.0, 2);
```

The calls that will cause compile time error are:

- *float area = surfaceAreaOfCylinder (2.0);*
- *float area = surfaceAreaOfCylinder (2.0, “Ask me”);*

One argument left out!

Second argument is of wrong data type!

Contrast the requirement of matching formal parameter data types with the data types of actual argument passed during a function call with the situation when you

buy a metro ticket. The ticket vending machine (Figure 8.10B) would have slots for you to use coins, paper bills, or credit card to purchase your ticket.



**FIG. 8.10B**

You cannot use the slot that accepts paper bill to insert your credit card or use slot that accepts coins to insert paper bills. Thus, think of formal function parameters as slots, which dictate the data types of actual arguments that you use when you call that function. When the data type passed as argument does not match the data type specified in the parameter list then C++ makes the best effort to convert the argument to the parameter data type. If such attempt fails, then it issues a compile time error. For example if we make a call like

```
float area = surfaceAreaOfCylinder (10.0, 2.222);
```

The second argument is actually a double type. C++ will chop off the 0.22 portion of 2.22 and pass it as integer 2 to the function *surfaceAreaOfCylinder*. This will not give

expected results, but it will compile and run. However, as mentioned in chapter four, do not depend upon automatic type conversion! That would be like always expecting US vending machines to accept Canadian coins. As advised in chapter four the implicit contract in using a function is to match data types of formal parameters in type and order. Any deviation from it is breach of the contract with the designer of the function.

### Order of Evaluation of Arguments

C++ evaluates all the function arguments, before the function call is made. However, in the comma-separated list of argument, whether arguments are evaluated left to right or right to left is not guaranteed. For example look at the Listing 8.2 below:

```
#include <iostream>
using namespace std;

void fun(int,int);

void main()
{
    int num1=5;
    fun(++num1,++num1);
}

void fun (int my_int, int your_int)
{
    cout<<my_int<<endl;

    cout<<your_int<<endl;
}
```

### Listing 8.2

If we assume that C++ will evaluate the arguments from left to right, then the value passed to fun as my\_int = 6 and value passed as your\_int = 7. However, C++ does not guarantee that. In fact reverse may be true. See Figure 8.9 below:

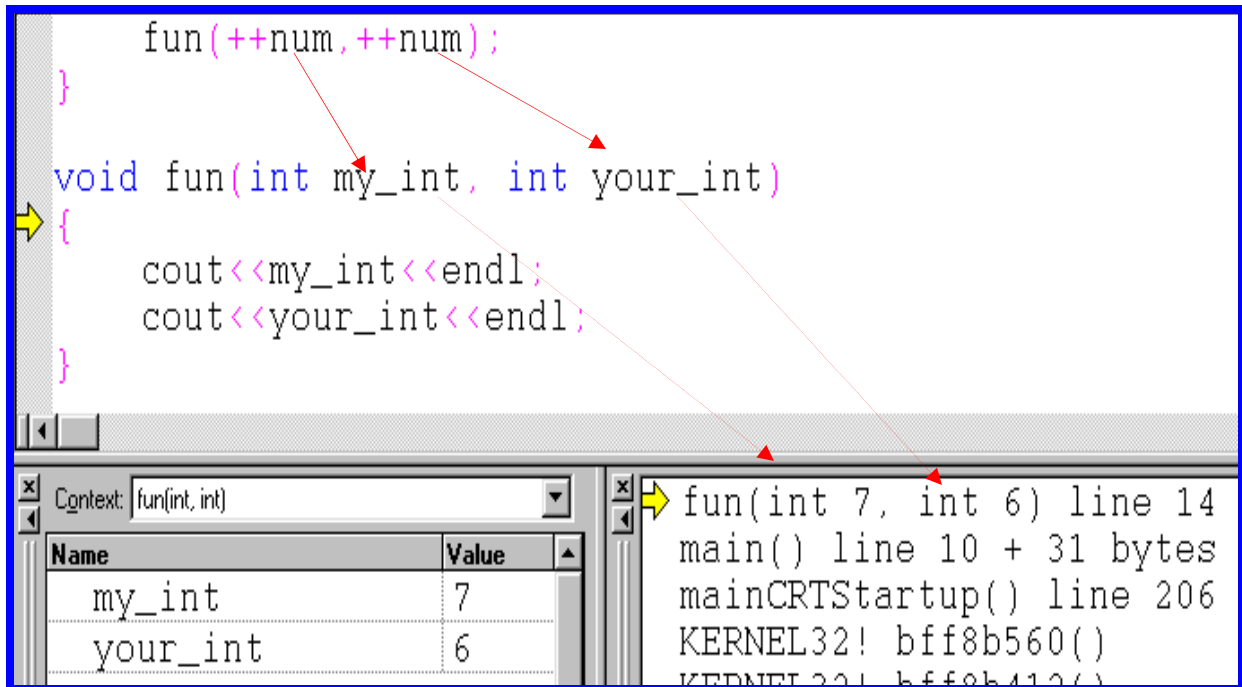
**FIG. 8.8**

Figure 8.8 show that function call was not fun (6,7), rather it was fun (7,6). Since C++ does not guarantee the order in which it will evaluate the arguments, it is best to avoid the code like Listing 8.2. Modifying the Listing 8.2 as Listing 8.3 solves this problem.



```

void main()
{
    int num = 5;
    int val1 = ++num;
    int val2 = ++num;
    fun(val1, val2);
}

void fun(int my_int, int your_int)
{
    cout<<my_int<<endl;
    cout<<your_int<<endl;
}

```

**C++ evaluates arguments explicitly before making the function call.**

Context: fun(int, int)

Name	Value
my_int	6
your_int	7

fun(int 6, int 7) line 16  
main() line 12 + 13 bytes  
mainCRTStartup() line 206  
KERNEL32! bff8b560()

### Listing 8.3

By explicitly evaluating the arguments before passing them to the function solves the problem since C++ now has the evaluated arguments to make the function call.

### Call By Value Mechanism

As mentioned before, every argument to a function is from the followings: a literal constant, a named constant, a variable, an expression, or another function call. Before function call is made, a value of the argument is evaluated. Thus, every argument has a value. One way in which C++ passes the argument value to the function is by making a copy of the value of the argument. This mechanism is called “call by value” mechanism. We can clearly see this mechanism, if we can print out the address of the variables using address operator & in main and in the invoked function. Listing 8.4 does that.

```

#include <iostream>
using namespace std;
void fun(int,int);
void main()
{
    int num = 5;
    int val1 = ++num;
    int val2 = ++num;
    cout<<"The memory address of val1 = "
         <<(long)(&val1)<<endl;
    cout<<"The memory address of val2 = "
         <<(long)(&val2)<<endl;
    fun(val1,val2);
}

void fun(int my_int, int your_int)
{
    cout<<"The memory address of my_int = "
         <<(long)(&my_int)<<endl;
    cout<<"The value of my_int = "
         <<my_int<<endl;
    cout<<"The memory address of your_int = "
         <<(long)(&your_int)<<endl;
    cout<<"The value of your_int = "
         <<your_int<<endl;
}

```

```

The memory address of val1 = 7077360
The memory address of val2 = 7077356
The memory address of my_int = 7077272
The value of my_int = 6
The memory address of your_int = 7077276
The value of your_int = 7

```

#### Listing 8.4

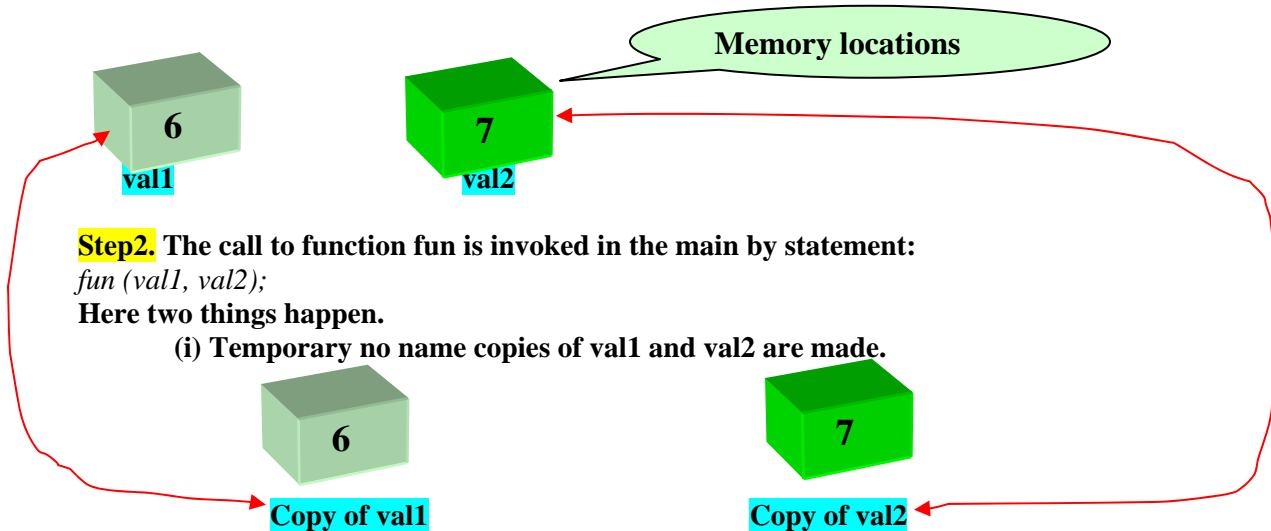
We can see that the memory addresses of num1 and num2 in main are 7077360 and 7077356 respectively. However, the memory addresses of my\_int and your\_int in function fun are 7077272 and 7077276 respectively<sup>2</sup>. The memory address is the actual location in RAM, where the data are stored. Different locations in RAM for val1 and my\_int indicate that they are different variables, even though their values are same. Little bit later, when we have learned about references, we will see that if function fun was coded to accept

<sup>2</sup> The memory addresses could be different each time you run this code. Therefore, the actual values are not important. The important thing is that values are different in main function and in the called function.

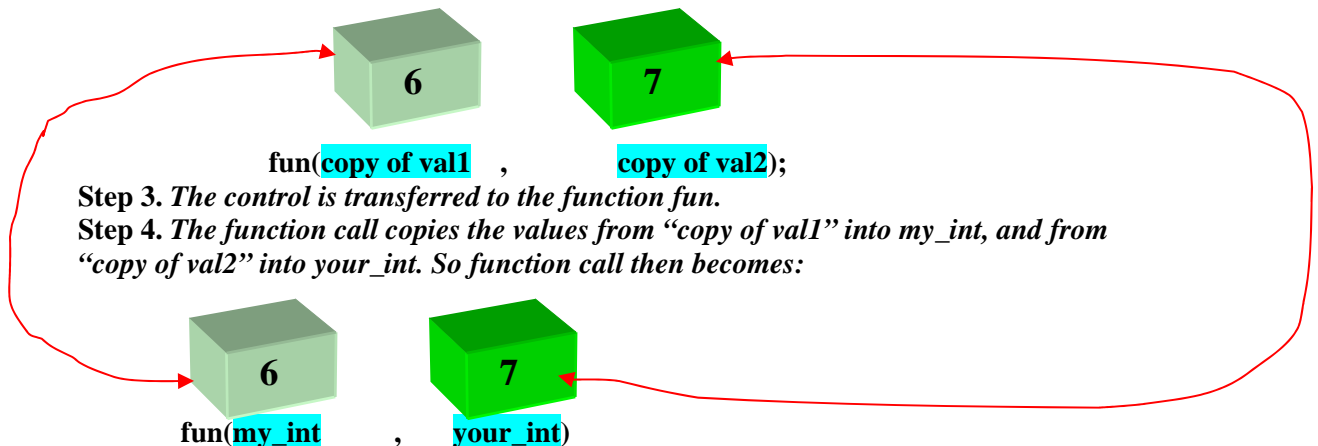
references to 2 integers and in making the function call to fun we pass the reference to val1 and val2, then actual variables are passed to the called function, not just their copies.

Does this experiment throw light on as to how the pass by value mechanism of function calls work? It does indeed. The steps below show the overall mechanism.

**Step1.** *val1 and val2 have values of 6 and 7.*



(ii) Copies are handed over to the function call.  
 The function call transmutes as follows:



**Step 5.** The code with in function fun is executed with values of my\_int as 6 and of your\_int as 7.

## Reference to Variables in C++

In C++, we have a mechanism available, by which we can give additional names to an existing variable. This mechanism is called assigning a reference. For example, let us view the code fragment given below:

```
int  int1 = 66;

int& int2 = int1;
```

Creates int2 as another name for int1. Now either name can be used.

In the code fragment above int2 becomes an alias for int1. The memory location, which was originally called int1, gets an additional name int2. Either name can, now be used to have read or write access to memory location int1. How can we prove that code fragment like above assigns two names to the same memory location (Listing 8.5)?

```
#include <iostream>
using namespace std;

int main()
{
    int int1 = 66;
    int & int2 = int1;

    cout<< "int1 = " << int1 << endl;
    cout<< "int2 = " << int2 << endl;

    int2 = 33;

    cout<< "int1 = " << int1 << endl;
    cout<< "int2 = " << int2 << endl;

    cout<< "The address if int1 = "
         << (long) (&int1) << endl;

    cout<< "The address if int2 = "
         << (long) (&int2) << endl;

    return 0;
}
/**/
```

1. int1 is given an additional name called int2.

2. Name int2 is used to change the value stored in memory location int1.

```

int1 = 66
int2 = 66
int1 = 33
int2 = 33
The address of int1 = 7077364
The address of int2 = 7077364

```

### Listing 8.5

We see that in Listing 8.5, the variable *int1* is given an additional name using the assignment,

```
int2 & = int1; // Reference variable must be initialized in same line!
```

Printing the value of *int1* with both names (*int1* & *int2*) gives the same result (66). Then we do something interesting. We make the assignment,

```
int2 = 33;
```

If *int2* would have been a different variable, then the above statement will not change the value stored in memory location *int1*. In next two lines, we print out the values stored under name *int1* & *int2*, and they both turn out to be 33. It means that the name *int2* and *int1* are attached to same memory location. We finally prove this hypothesis by printing the address of the memory location *int1* and *int2* and they both are same value of 7077364. That proves that name *int1* and *int2* refer to same memory location. The value of addresses of *int1* and *int2* may change with different invocations of program in Listing 8.5. The important point is that they have the same address.

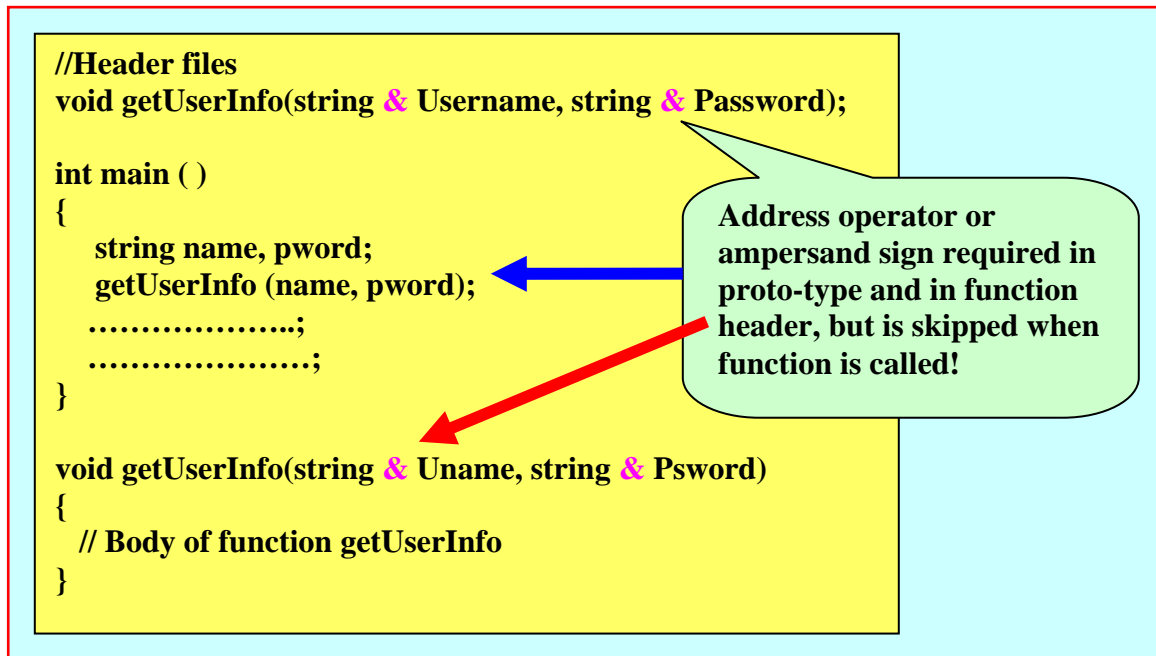
### Pass by Reference Mechanism

Often it is not so useful to assign an additional name to a variable the way we have done in Listing 8.6. However, this reference assigning is used in a very powerful way, when functions accept variables passed to them by reference. The argument list of functions is altered somewhat, when the variables are passed to it by reference. We put an ampersand sign (&) next to the data type being passed in the proto-type and in the function header. The syntax of making actual function call remains unchanged. Let us look at the syntax rules for passing by reference a bit more closely. Imagine that you have an account on an Internet web site. Website may use a function *getUserInfo* to get your username and password. The proto-type of function *getUserInfo* may appear as one of the followings:



```
void getUserInfo(string &, string & );
void getUserInfo(string & Username, string & Password);
```

Both versions of proto-type for function *getUserInfo* are syntactically equivalent. However, the second version is more descriptive. Let us now show the syntax of full definition and call to function *getUserInfo* (Figure 8.9A).

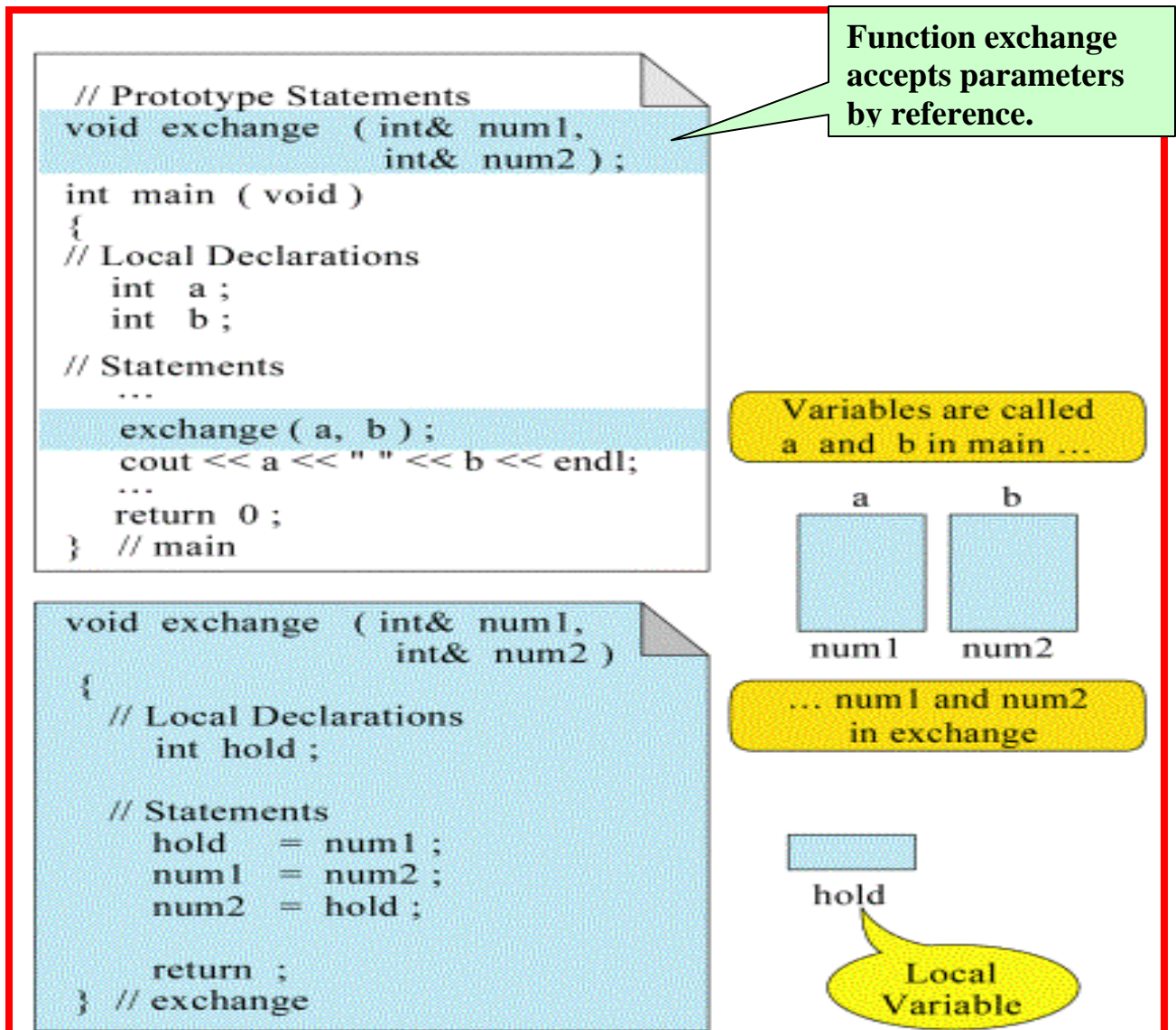


**FIG. 8.9A**

Notice that when passing a parameter by reference, the address or ampersand operator (&) is required after the data type in the function proto-type and in the function header. However, only the names of variables are enough when calling the function. Therefore, the syntax of making function calls does not change compared to when calling a function that takes parameters by value. In Figure 8.9A, we show the function *getUserInfo* taking both parameters by reference. This is not a requirement. Decision as to how many and which parameters must be passed by reference and which others by value depends upon the design of function and what task function is required to perform.

#### Example of Pass By reference

We show the mechanism of passing parameters to functions by reference in the Figure 8.9B.

**FIG. 8.9B**

The proto-type and definition of function exchange is shown (Figure 8.9B). The purpose of function exchange is to take two integer values as arguments and swap the values stored in them. If in call by reference, the actual parameters, rather than their copies are being sent to a function, then the function must be able to change the values of the variable sent to it. In function exchange, we create a local variable called hold, which is used temporarily to store the value of one of the variable, while the exchange of variable values is made. The actual code and results for Figure 8.9B are shown in Listing 8.6.

```

//
#include <iostream>
using namespace std;

void exchange(int& num1, int& num2);

int main()
{
    int int1 = 900;
    int int2 = -400;

    cout<<"Before call to function exchange"
    <<" the value of int1 = "<<int1<<endl;
    cout<<"Before call to function exchange"
    <<" the value of int2 = "<<int2<<endl;

    exchange(int1,int2);

    cout<<"After call to function exchange"
    <<" the value of int1 = "<<int1<<endl;
    cout<<"After call to function exchange"
    <<" the value of int2 = "<<int2<<endl;

    return 0;
}
//*****
void exchange(int& num1, int& num2)
{
    int hold = int();
    hold = num1;
    num1 = num2;
    num2 = hold;

    return;
}
//**/

```

```

Before call to function exchange the value of int1 = 900
Before call to function exchange the value of int2 = -400
After call to function exchange the value of int1 = -400
After call to function exchange the value of int2 = 900

```

### Listing 8.6

Why was function exchange able to change the values of variables in the main function?  
That is because both int1 and int2 were passed by reference to the exchange function. The

function received actual variables, rather than their copies, though names were changed to num1 and num2. Having the actual variables passed to the function allowed it to change their values and reflect such change in main function as well. We can show that if we remove the ampersand sign from the int arguments to function exchange, this swap of values will not work. This is shown in Listing 8.7.

```
#include <iostream>
using namespace std;

void exchange(int num1, int num2);

int main()
{
    int int1 = 900;
    int int2 = -400;

    cout<<"Before call to function exchange"
    <<" the value of int1 = "<<int1<<endl;
    cout<<"Before call to function exchange"
    <<" the value of int2 = "<<int2<<endl;

    exchange(int1,int2);

    cout<<"After call to function exchange"
    <<" the value of int1 = "<<int1<<endl;
    cout<<"After call to function exchange"
    <<" the value of int2 = "<<int2<<endl;

    return 0;
}

void exchange(int num1, int num2)
{
    int hold = int();
    hold = num1;
    num1 = num2;
    num2 = hold;
    cout<<"In function exchange, the value of"
    <<" num1 = "<<num1<<endl;
    cout<<"In function exchange, the value of"
    <<" num2 = "<<num2<<endl;
    return;
}
```

```

Before call to function exchange the value of int1 = 900
Before call to function exchange the value of int2 = -400
In function exchange, the value of num1 = -400
In function exchange, the value of num2 = 900
After call to function exchange the value of int1 = 900
After call to function exchange the value of int2 = -400

```

### Listing 8.7

The above listing and its results show that though function exchange does swap values of num1 and num2, the results are strictly local. The swapped values never are transmitted to the function main. That is because the values passed to exchange () in Listing 8.7 were merely copies and not actual variables themselves.

How can we prove that the variables passed to exchange in Listing 8.6 were actual variables and not their copies. Very simply, we can print the memory addresses of the variables in the main and then in the function exchange. If values are the same, then the actual variables, rather than their copies were passed. We show this in Listing 8.8.

```

#include <iostream>
using namespace std;
void exchange(int& num1, int& num2);
int main()
{
    int int1 = 900;
    int int2 = -400;

    cout<<"The address of int1 in main = "
    <<(long)(&int1)<<endl;
    cout<<"The address of int2 in main = "
    <<(long)(&int2)<<endl;
    exchange(int1,int2);

    return 0;
}
//*****
void exchange(int& num1, int& num2)
{
    cout<<"The address of num1 in function excahnge = "
    <<(long)(&num1)<<endl;
    cout<<"The address of num2 in function exchange = "
    <<(long)(&num2)<<endl;

    return;
}

```



```

The address of int1 in main = 7077364
The address of int2 in main = 7077360
The address of num1 in function exchange = 7077364
The address of num2 in function exchange = 7077360

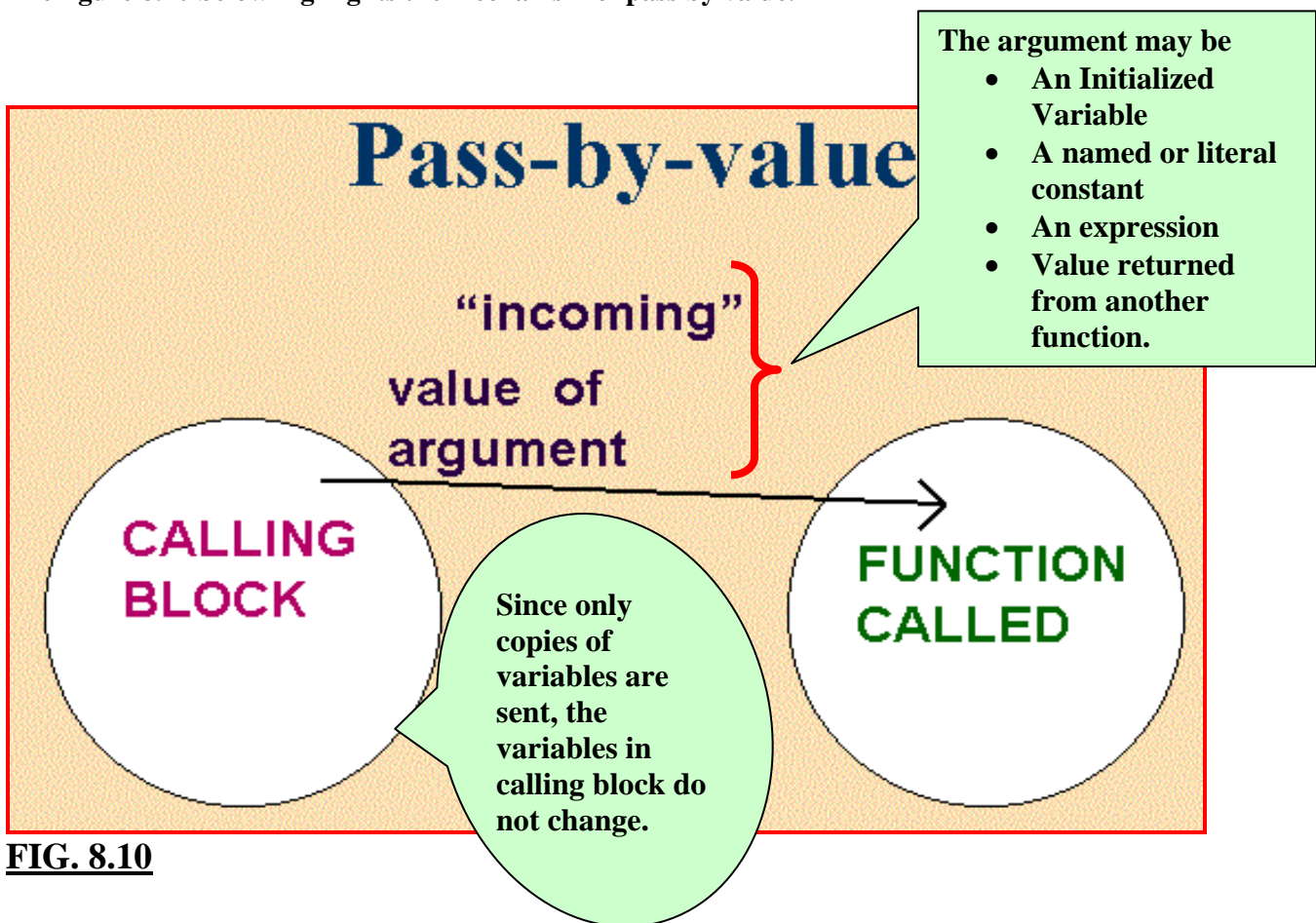
```

**Listing 8.8**

We clearly see that the memory location of int1 in main is same that is of num1 in function exchange. It holds true for int2 and num2 as well. **That would mean that int1 and num1 is same variable, with two names!** In addition, int2 and num2 also refer to same variable.

### **Comparison Between Pass-By-Value And Pass-By reference Mechanisms**

As described above, the functions in C++ can accept parameters by two different mechanisms. One mechanism passes parameters by their values and other by reference. The figure 8.10 below highlights the mechanism of pass by value.

**FIG. 8.10**

In pass-by-value mechanism, the parameters passed may be, initialized variables, named or literal constants, expressions that C++ can evaluate, or return values from other functions. Taking example of call to function `square ( )` in Listing 8.1, the followings are all legal calls to function `square`:

```
int val = 5;  
square_val = square (val);
```

```
square_val = square (5);
```

```
square_val = square ( int ( sqrt (25.0) ) );
```

**cmath will  
be required.**

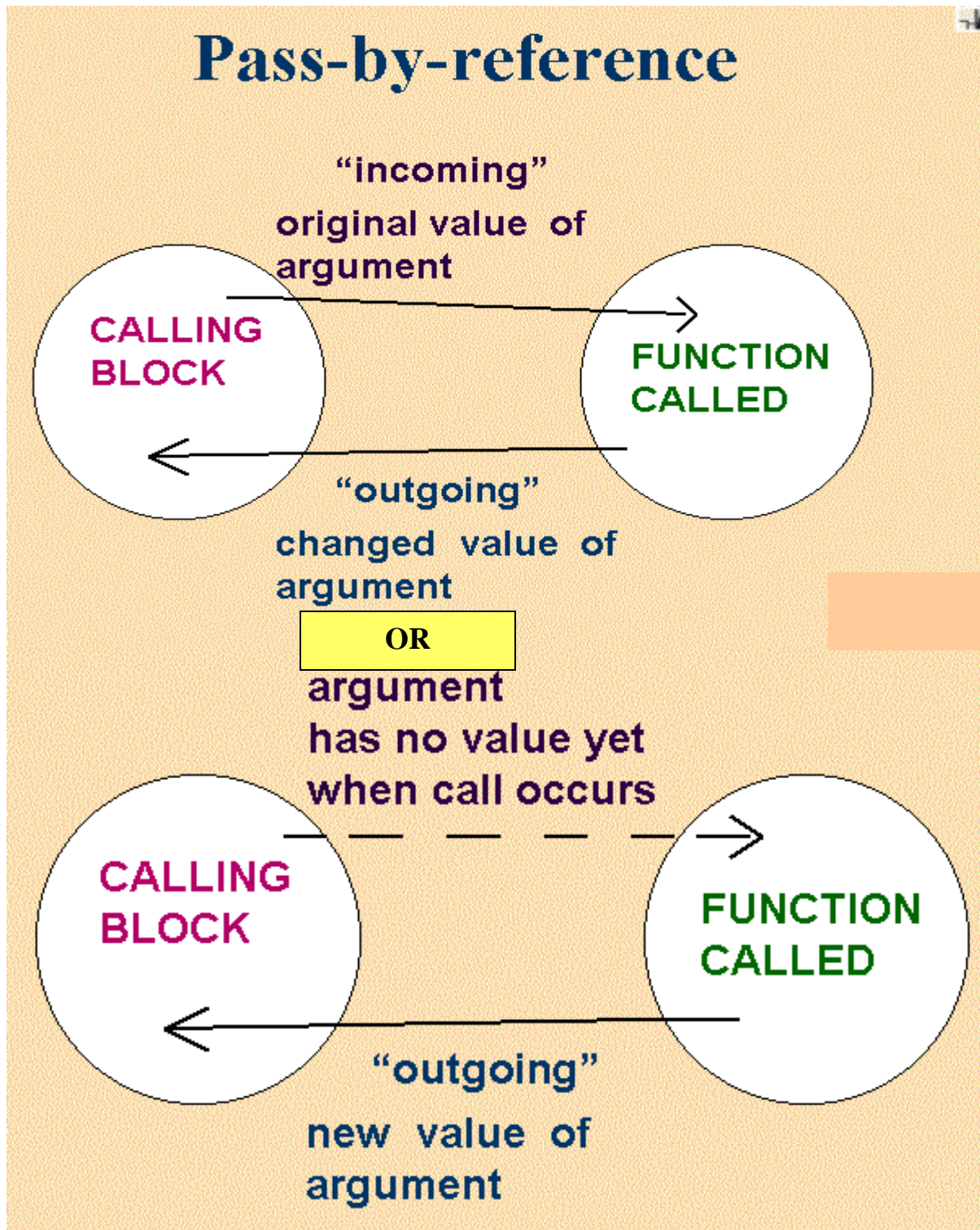
```
square_val = square ( (val + 5) % 17 );
```

```
string string1 = "Hello World";  
square_val = square ( string1.length( ) );
```

**Library string  
will be  
required.**

Any parameter, whose value C++ can evaluate before making a function call, may be used in the pass-by-value mechanism.

Let us now review the pass-by-reference mechanism. Figure 8.11 shows the schematics of its process.

**FIG. 8.11**

In pass-by-reference mechanism, only a variable, whose address can be determined by C++ at the time of function call, may be passed to by the calling block to the called function. There is a presumption in call-by-reference that the argument being passed could be altered or be treated as an L-value expression. Therefore, constants of any kind (literal or named)



cannot be passed-by-reference as they are not L-value expressions. Assume that we are making calls to the function `exchange ( )` in Listing 8.6. The legal calls will be as follows:

```
int val1 = 900;
int val2 = -400;

exchange (val1, val2);

int num1;
int num2;

exchange (num1, num2);
```

Function call is legal but useless in this case, unless `exchange` also has code to initialize `num1` and `num2`.

### More Examples of Pass by Reference

We find the roots of a quadratic Equation using functions, where we pass values from main to the function by reference. Figure 8.12 formulates the problem for us.

#### Example of Pass by Reference

- We wish to compute two real roots for a quadratic equation as below:  

$$ax^2 + bx + c = 0$$
- Write a prototype for a function `getRealRoots` that returns a bool value and takes five parameters as argument.
- The first three parameters are passed by value and are integer type (essentially  $a, b, c$ ). The last two parameters *root1* and *root2* are float types.

**FIG. 8.12**

We know now that a quadratic equation may be written as follows:

$$ax^2 + bx + c = 0$$

What we can do is that we can send the values of  $a, b, c$  to the function `getRealRoots`, by the pass-by-value mechanism. We also have two more argument in function `getRealRoots`, which are passed by reference, so that function can find the roots for us and fill the values to be communicated back to main function. The signatures of function `getRealRoots` and other detail are summarized in Figure 8.13.

**bool getRealRoots ( int , int , int , float& , float& );**

**Function  
proto-type!**

**Now write the function definition using this information.**

**This function uses 3 incoming values a, b, c from the calling block. It calculates 2 outgoing values root1 and root2 for the calling block. They are the 2 real roots of the quadratic equation with coefficients a, b, c.**

**FIG. 8.13**

The code for function getRoots is given by Listing 8.9.

**a, b, c are incoming  
or passed by value.**

**root1 and root2 are outgoing  
or passed by reference!**

```

bool getRoots(int a, int b, int c, float& root1, float& root2)
{
    int discr = b*b - 4.0 * a* c;

    if(discr <0 )
    {
        cout<<"Imaginary Roots\n";
        return false;
    }
    else
    {
        root1 = ( -b + sqrt(1.0* discr )) /(2.0*a);
        root2 = ( -b - sqrt(1.0* discr )) /(2.0*a);
        return true;
    }
}

```

**Listing 8.9**

In function `getRoots`, the values of quadratic equation coefficients  $a$ , and  $b$ , and constant  $c$  are passed by value as function `getRoots` simply needs read-only access to them. However, the parameters `root1` and `root2` are passed by reference, as the function would need both read and write access to them. The value of discriminant `discr` is calculated. If discriminant is negative then function simply prints out the message “*Imaginary Roots*”. In this case, the values of `root1` and `root2` remains unchanged and function `getRoots` returns a false boolean value. However, if the discriminant is not negative then variables `root1` and `root2` would be populated by the expressions in else clause and those values are passed back to main as outgoing values. The function in this case returns a true boolean value.

### Practice Questions

Answer the questions below for the code following the questions.

A] For what value of variable `data` in main, the loop in function `fun` will never execute.

B] For what value of `data` in main the function `mystery` will be in the infinite loop.

```

//*****

```

```

#include <iostream>
using namespace std;

```

```

void fun (int data);
void mystery(int data);
void main( )
{
    int data;
    cout<<"Please enter an integer value : ";
    cin>>data;
    fun(data);
    mystery(data);
}

```

```

void fun(int count)
{
    char index = count;
    while(index > 0)
    {
        index--;
    }
}

```

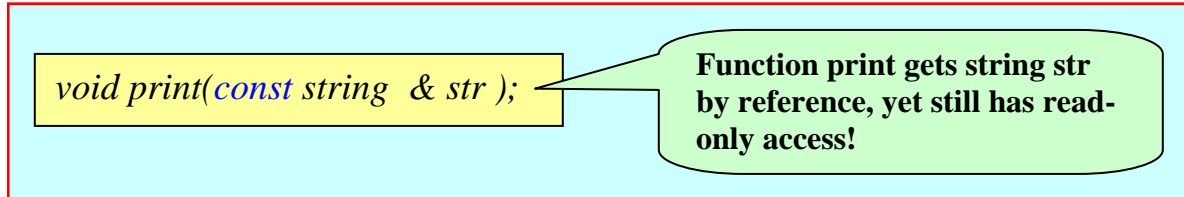
```

void mystery(int count)
{
    char index = count;
    while(index<count)
    {
        index++;
    }
}
//*****

```

### Passing a const Reference

When function takes parameters by reference, it gets read and write access to the parameters being passed to it. This feature is useful when parameter passed “must” be changed by the called function. Function exchange discussed earlier is an excellent example of usefulness of parameter passing by reference. However there are situation where we wish to pass a parameter by reference, yet provide read-only access to it by the called function. In such cases the keyword `const` is appended before the data type in the formal parameter list.



For example the function `print` above gets string `str` by reference. However keyword `const` before the data type `string` ascertains that `print` function would have a read-only access to `str` and cannot change its contents accidentally<sup>3</sup>. Thus by using the word `const` the parameters passed to the function are treated (inside the function) as R-value only expressions, preventing write access.

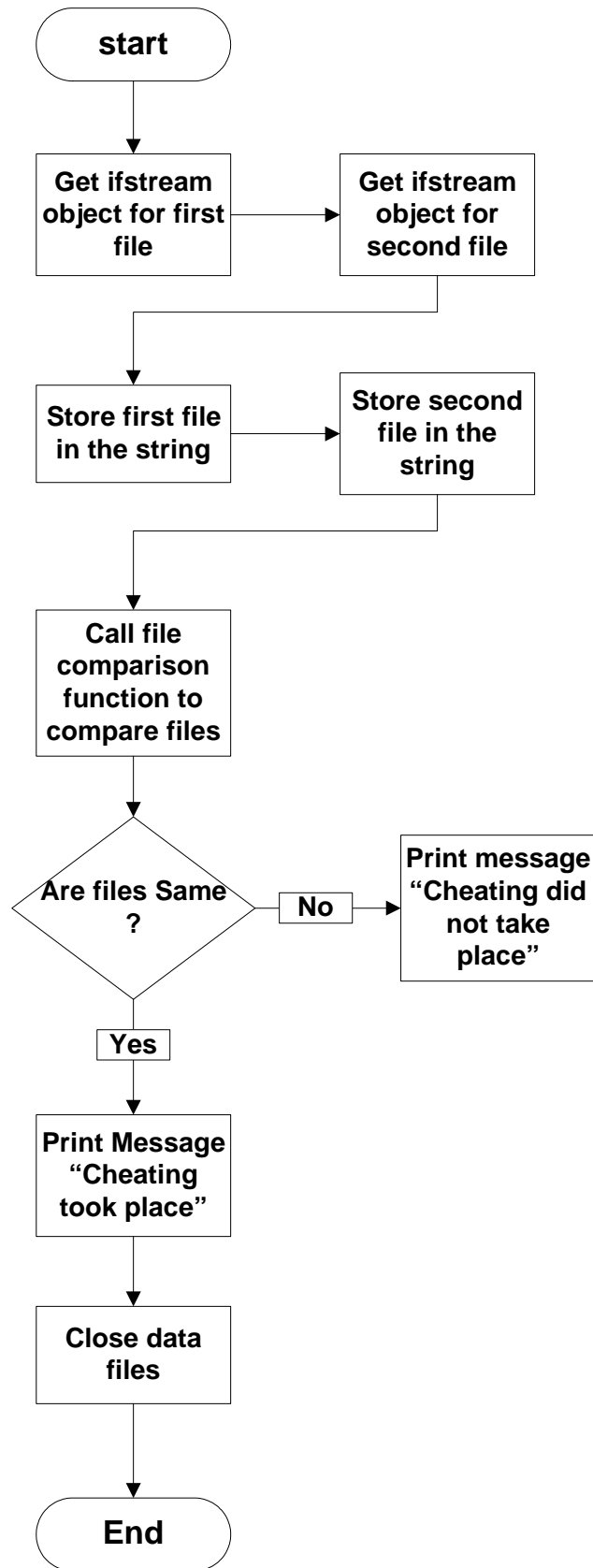
### Example of Use of Const Reference Passing

Assume that for summer internship you get a research project with a university professor in English department. After knowing your background in software engineering professor asks you to write a program so that software can accept input for electronic files of student term papers and decide that which files are identical – thus in that case cheating took place. For the sake of simplicity, we assume all files are text files<sup>4</sup>. The flow chart in Figure 8.14 shows us the approximate algorithm for development of our program.

<sup>3</sup> A deliberate write-access to `str` using `const_cast` is still possible.

<sup>4</sup> It is easy to convert a Microsoft Word file into a text file and then read it by a computer program.



**FIG. 8.14**

In words, the stages in the above algorithms are followings:

1. Get valid ifstream objects for both files so that files exist and they are not empty.
2. Store each file in a string.
3. Send the string versions of both files to the file comparison function, which returns a true value if files are identical. If files are not identical then function returns a false value.
4. If files are identical then print, message “Cheating took place”. Otherwise, print the message “Cheating may not have taken place”.

Listing 8.10 gives most of the functions for this software. Missing code for function `getInFile` is the one for which we would like you to write the code yourself as practice.

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

bool fileCompare(const string & , const string& );
void storeInString(ifstream & , string& );
void getInFile(ifstream &, string & );

int main()
{
    string Msg = "first";
    ifstream in;
    string content1="", content2="";
    getInFile(in, Msg);
    storeInString(in , content1);
    in.close();
    Msg = "second";
    getInFile(in, Msg);
    storeInString(in , content2);
    in.close();

    if(fileCompare(content1,content2))
        cout<<"Cheating took place.\n";
    else
        cout<<"Cheating may not have taken place.\n";

    return 0;
}

bool fileCompare(const string & str1, const string& str2)
{
    bool result = true;

    if(str1 == str2)
        result = true;
    else
        result = false;

    return result;
}
```

```

}
void storeInString(ifstream & input, string & str)
{
    str = "";
    char ch = ' ';

    while((ch = input.peek()) != EOF)
    {
        ch = input.get();
        str = str + ch;
    }
}
//For function getInFile write the code to get some practice.
void getInFile(ifstream & input, string & str)
{
}
//Listing 8.10

```

The functions *storeInString* and *fileCompare* are deceptively simple. *storeInString* takes *ifstream* object and a string by reference. Stream objects such as *ifstream* and *ofstream* are always passed by reference. This has to do for the fact that same object can be continually used for further input or output after call to the function using them is completed. The file reading loop in function *storeInString* must exit when EOF character is read. The *peek* function is ideal for this. *peek* function returns the copy of next character to be read. Therefore the loop termination condition is:

$$\text{input.peek}() == \text{EOF}$$

Using De Morgan's rule the loop test condition would be negation of termination condition, or:

$$\text{!(input.peek}() == \text{EOF)}$$

Carrying out the necessary conversion using De Morgan's rule the loop test condition is:

$$\text{input.peek}() \neq \text{EOF}$$

Coding the loop as below is logical because due to precedence rules the expression inside the innermost parenthesis is evaluated first which populates the variable *ch*.

```
while((ch = input.peek()) != EOF)
```

After *input.peek()* function call populates the variable *ch*, only after that the relational expression *ch != EOF* is evaluated. Loop is only entered only if *ch* is not EOF or end of file is not reached. Loop task first gets a character from the file using *get* function by statement *ch = input.get()*; . Then this character is simply concatenated to the existing string *str*. Since *get* function is used, all characters,

including white spaces are read from the input file and string *str* becomes exact replica of the input file.

Function *storeInString* is used by main function twice, to get string versions of two files being compared. Then the function *fileCompare* that takes two aforementioned strings as const reference as arguments is called. The proto-type for function *fileCompare* is:

```
bool fileCompare(const string & str1 , const
string& str2 );
```

Why shall we pass two string objects to *fileCompare* as const references? The requirement is that function *fileCompare* not change contents of the two string objects even accidentally. Appending keyword *const* before the formal parameter data type takes away the accidental write-access. For example, the code fragment such as below inside *fileCompare* would then cause compile error!

```
str1 = str1 + 'x';
```

Compile error! No write access to *str1* as it is being passed as a const reference.

The simplicity of code for *fileCompare* function shows the beauty of C++. Function simply uses the relational == operator to compare the contents of two strings passed to it. If contents are identical then the expression *str1 == str2* evaluates to true and function returns true, otherwise it returns false.

Since *fileCompare* returns a boolean value, it can be called directly inside the if expression as:

```
if(fileCompare(content1,content2))
```

In evaluating the boolean expression inside the parenthesis for if structure, the function call to *fileCompare* is completed first and then depending on its value either the message “cheating took place” or the message “cheating may not have taken place” are printed.

You may ask why not pass *str1* and *str2* to *fileCompare* by value. Certainly one can do that. However, its expensive in terms of memory usage as a string containing an entire file would be a large object in terms of number of bytes. Making copy of large objects in pass by value mechanism is memory inefficient. Therefore, general rules of thumb are:

1. For primitives (such as int or float) use pass by value if parameter should only be incoming to the called function (see Figure 8.10).

2. For all data types, use pass-by-reference if the parameter value is to be altered by the called function. In other words, the parameters passed have an outgoing value they must possess after the function call is completed.
3. For objects use const pass-by-reference when called function is not to alter the object. Objects are almost never passed using pass-by-value mechanism.

There are other ways to code function *fileCompare* where the function would take two ifstream objects bonded to files to be compared as parameters. We leave that for practice as the following practice question.

### Practice Question

Write the code for function *fileCompare* when it has the following proto-type:

```
bool fileCompare (ifstream & file1, ifstream & file2);
```

### Function Name Overloading

By now, you are aware that when using *pow* function in *cmath* library the following two function calls can be made:

```
double base = 2.0;  
double exp = 3.0;  
double result = pow(base , exp);
```

C++ calls two different versions of *pow* for these two calls.

```
int exponent = 5;  
result = pow(base , exponent);
```

Notice that signatures of function *pow* are different in these two calls. The first call is of type *pow(double, double)*; whereas second call of type *pow(double, int)*. Program actually calls two different versions of *pow*. When different functions have same name, this C++ feature is called “function name overloading” or function name “polymorphism”. Function name overloading allows two or more user defined functions to have same names as long as the signatures of each function are different. The signature of a function constitutes the following components:

- *Function name*
- *Number of parameters passed to the function*
- *Order and data types of formal parameters*

No two functions with same name can have all of above three components to be identical. For example in C++, it would be a compile error to declare two functions like below:

```
int myFunction (int val1, string name, double score)  
{ //function body }
```

```
float myFunction (int val1, string name, double score)  
{ //function body }
```

Compile error.  
Return type is not part of signature.

The above is compile error because function's return type is not part of its signatures. However, the four functions having name myFunction as shown below have different signatures and no compile error is issued.

```
int myFunction (int val1, string name, double score)
{ //function body}
```

```
int myFunction (string name, int val1, double score)
{ //function body}
```

```
int myFunction (int val1, double score, string name)
{ //function body}
```

```
int myFunction (int val1, string name)
{ //function body}
```

No compile error as all functions have different signatures.

The first three functions have same number of parameters, and types, but the order of parameters is different, making them identifiable by their signatures. The last function has fewer numbers of arguments than other three – thus having different signatures from the all others. The ability to overload function names allows software engineer to use the same name for functions whose algorithms or actions are similar. This facilitates the “economy” of inventing the function names in C++. cmath library is an excellent example of overloaded functions. Notice that changing one passing mechanism of one of the parameters to pass-by-reference does not change the function signatures! For example, the following two functions do not have proper overloading, as program will fail to resolve as which function to call when actual call is made and a compile error is issued.

```
int getSquare(int val);
void getSquare(int & val);
```

Improper overloading as merely changing parameter passing mechanism does not make function signature unique.

Adding a const modifier for a formal parameter also does not change function signatures. For example below is also a compile error.

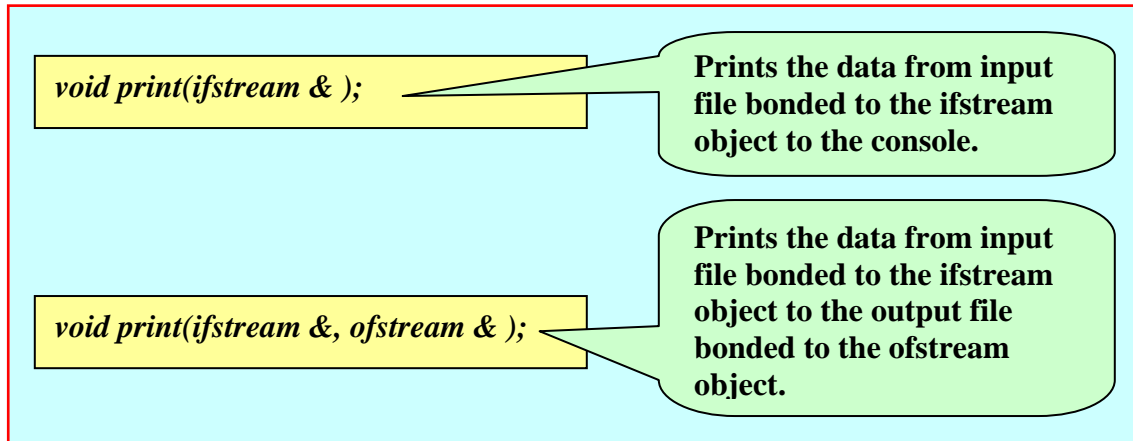
```
int getSquare(int val);
void getSquare(const int & val);
```

Improper overloading, as appending the const modifier and passing by reference does not make function signatures unique.

### Example of Function Name Overloading

In chapter five, we did the miles per gallon program. At that time we did learn looping structure, thus we had to read all lines from file one by one and display the results one by one. Now we remove that limitation. Additionally we would like to provide two print functions, one that prints the miles per gallon data to the console

and other to an ASCII file. Thus, we provide overloaded/polymorphic print function that has two proto-types given below (Figure 8.15):



**FIG. 8.15**

The first print function takes the input file with gas mileage data bonded to the ifstream object and prints the gas mileage and other processed results to the standard output (usually a console). The overloaded (second) version takes ifstream and ofstream objects as arguments. The data are read from file bonded to ifstream object and written into file bonded to ofstream object. Listing 8.11 gives the source code for upgraded miles per gallon program. Once again, we do not provide code for functions *getInFile* and *getOutFile*, as we would like you to write them as practice.

////////////////////////////////////

```

1  #include <iostream>
2  #include <string>
3  #include <fstream>
4  #include <iomanip>
5  using namespace std;
6
7  void getInFile(ifstream &, string&);
8  void getOutfile(ofstream&);
9  void print(ifstream & );
10 void print(ifstream &, ofstream & );
11
12 int main()
13 {
14     string Msg = "First";
15     ifstream in;
16     getInFile(in, Msg);
17     print(in);
18     ofstream out;
19     getOutfile(out);
20     in.clear(); //repairs the failed input stream
21     in.seekg(0,ios::beg); //Sets the reading marker to the beginning of file
22     cout<<"Printing to output file.\n";
23     print(in,out);
24     in.close();
25     out.close();
26

```



```

27     return 0;
28 }
29
30 void print(ifstream & input,  ofstream & out)
31 {
32     double miles = double();
33     double gallons = double();
34     int counter = 1;
35
36     out<<"\n";
37     out<<fixed<<showpoint;
38     out<<setprecision(2);
39     out<<endl;
40     out<<"=====\\n";
41     out<<"          GAS MILEAGE DATA FOR YOUR CAR          "<<endl;
42     out<<"-----\\n";
43     out<<left<<setw(15)<<"Trip Number"<<setw(15)
44         <<"Miles"<<setw(15)<<"Gallons\\t"<<setw(15)
45         <<" MPG"<<endl;
46     out<<"=====\\n\\n";
47     input>>miles;
48     double sum = double();
49
50     while(input)
51     {
52         input>>gallons;
53         out<<setw(15)<<counter<<setw(15)<<miles;
54
55         if(int(gallons)<10)
56             out<<" ";
57
58         out<<setw(15)<<gallons<<"\\t"<<setw(15)<<miles/gallons<<"\\n\\n";
59         sum = sum + miles/gallons;
60         counter++;
61         input>>miles;
62     }
63     out<<"=====\\n";
64     out<<"The Average Miles Per Gallon for all Trips:      "
65         <<sum/(counter-1)<<endl;
66     out<<"=====\\n\\n";
67 }
68 void print(ifstream & input)
69 {
70     double miles = double();
71     double gallons = double();
72     int counter = 1;
73
74     cout<<"\n";
75     cout<<fixed<<showpoint;
76     cout<<setprecision(2);
77     cout<<endl;
78     cout<<"=====\\n";
79     cout<<"          GAS MILEAGE DATA FOR YOUR CAR          "<<endl;
80     cout<<"-----\\n";
81     cout<<left<<setw(15)<<"Trip Number"<<setw(15)
82         <<"Miles"<<setw(15)<<"Gallons\\t"<<setw(15)
83         <<" MPG"<<endl;
84     cout<<"=====\\n\\n";
85
86     input>>miles;
87
88     double sum = double();
89

```

Loop initialization.

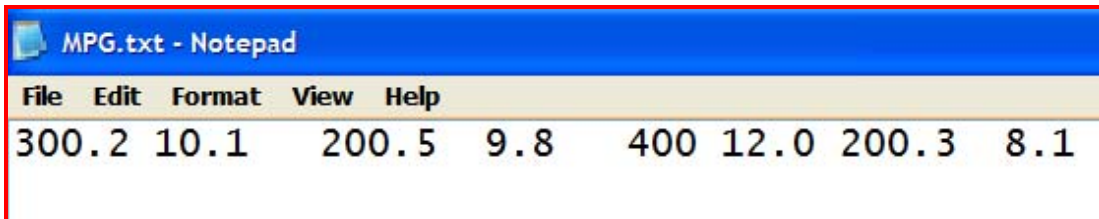
```

90     while(input)
91     {
92         input>>gallons;
93         cout<<setw(15)<<counter<<setw(15)<<miles;
94
95         if(int(gallons)<10)
96             cout<<' ';
97
98         cout<<setw(15)<<gallons<<"\t"<<setw(15)<<miles/gallons<<"\n\n";
99
100
101         sum = sum + miles/gallons;
102         counter++;
103         input>>miles;
104     }
105     cout<<"=====\n";
106     cout<<"The Average Miles Per Gallon for all Trips:  "
107         <<sum/(counter-1)<<endl;
108     cout<<"=====\n\n";
109 }
110
111 void getInFile(ifstream & input, string & str)
112 {
113     //Please use the code that you wrote as practice in Listing 8.10
114 }
115
116 void getOutfile(ofstream & oust)
117 {
118     //Write the code of this function as practice.
119 }
120
121 //Listing 8.11

```

### Overloading of print Function

Two overloaded functions in Listing 8.1 only differ in their signatures. The source code for the functions is almost identical. Let us discuss the function *print* (*ifstream* &) that prints the processed results to the console. The function must use the end of file control loop so that loop exits when EOF is read. Therefore, we simply check for the failed state of the input stream in the loop test. The data structure of file is given by Figure 8.16 below.



File	Edit	Format	View	Help			
300.2	10.1	200.5	9.8	400	12.0	200.3	8.1

**FIG. 8.16**

For each set of data pairs the first datum is miles driven and second the number of gallons used. For example for the first set of data *300.2 10.1*, 300.2 is the miles driven and 10.1 is the number of gallons of gasoline consumed. Loop initialization is done by making a reading for miles outside the loop (Line #86). Understand that for EOF controlled loops a read must be done before loop pre-test, otherwise EOF does

not have a valid value. Since the function, *getInFile* already ascertains that input file is valid and has data in it. The loop is entered (Line #90). Then inside the loop input for number of gallons is taken (Line #92). Then trip number given by variable *counter*, *miles*, *gallons*, and *miles per gallon* are outputted and a table format is applied (Lines # 93 to 98). One purpose of the program is to calculate average miles per gallon for all the trips whose data are recorded in the input file. Thus, sum of miles per gallon values is done (Line #101). Loop iteration counter is then incremented by one (Line #102). Finally, the update of loop condition is done by reading miles (next pair of data) (Line #103). Finally, the data for average miles per gallon for all trips is displayed (Line #107).

### main Function

main function calls *getInFile* to get a valid input file. Then it calls first *print* function that displays the output to the console (Line #17). The output for the data file of Figure 8.16 is shown by Figure 8.17.

```
Please type the name of the file for mileage data for First car:
mpg.txt
The file mpg.txt opened successfully.
File mpg.txt has data in it.

=====
                GAS MILEAGE DATA FOR YOUR CAR
=====
Trip Number      Miles           Gallons           MPG
=====
1                 300.20           10.10            29.72
2                 200.50           9.80             20.46
3                 400.00           12.00            33.33
4                 200.30           8.10             24.73
=====
The Average Miles Per Gallon for all Trips:      27.06
=====

Please type the name of the file to which output is to be written : result.txt
The file name entered is : result.txt
The file result.txt opened successfully.
Printing to output file.
```

**FIG. 8.17**

The table of gas mileage data outputted to data file (in this case *result.txt*) is identical to the display in Figure 8.17. After displaying the data to console, the main function takes the control from print function (Line #18). Since next goal is to output data to an output file, first an ofstream object is created (Line #18). Then the function *getOutFile* is called so that a valid output file is created. Here is an important point. You must recall that loop exit in the print function on line #90 took place because EOF was read and input stream went into failed state. The same input file stream cannot be used in failed state to read the data input file, unless it is repaired and the file reading marker is rewound to the beginning of the file. Code lines #20 and 21 in

main function do these tasks. The statement *in.clear ( )* repairs the failed stream. To set the file reading marker to the beginning of file the stream function *seekg* is used. The function *seekg* used here has the following form:

*basic\_istream & seekg( offset , From\_Where);*

Let us just focus on the two arguments for *seekg* function. Second argument, *From\_Where* marks the location from where the marker movement would begin. The first argument *offset* gives number of character moved from the location *From\_Where*. Since in this case we wish to set the location to be beginning of the file, the parameter *From\_Where* is set *ios::beg*. This tells *seekg* function that start counting the offset value from the beginning of the file. Since we wish marker to stay at beginning of file, the *offset* value passed to *seekg* is zero. If reading marker is not at the beginning of file then it is possible to specify a negative value for the offset and set is back by required number of characters.

main function calls the second overloaded version of print function (*print(istream &, ostream &)*) to print the data to output file (Line #23). Finally, both input and output files are closed (Line # 24, 25).

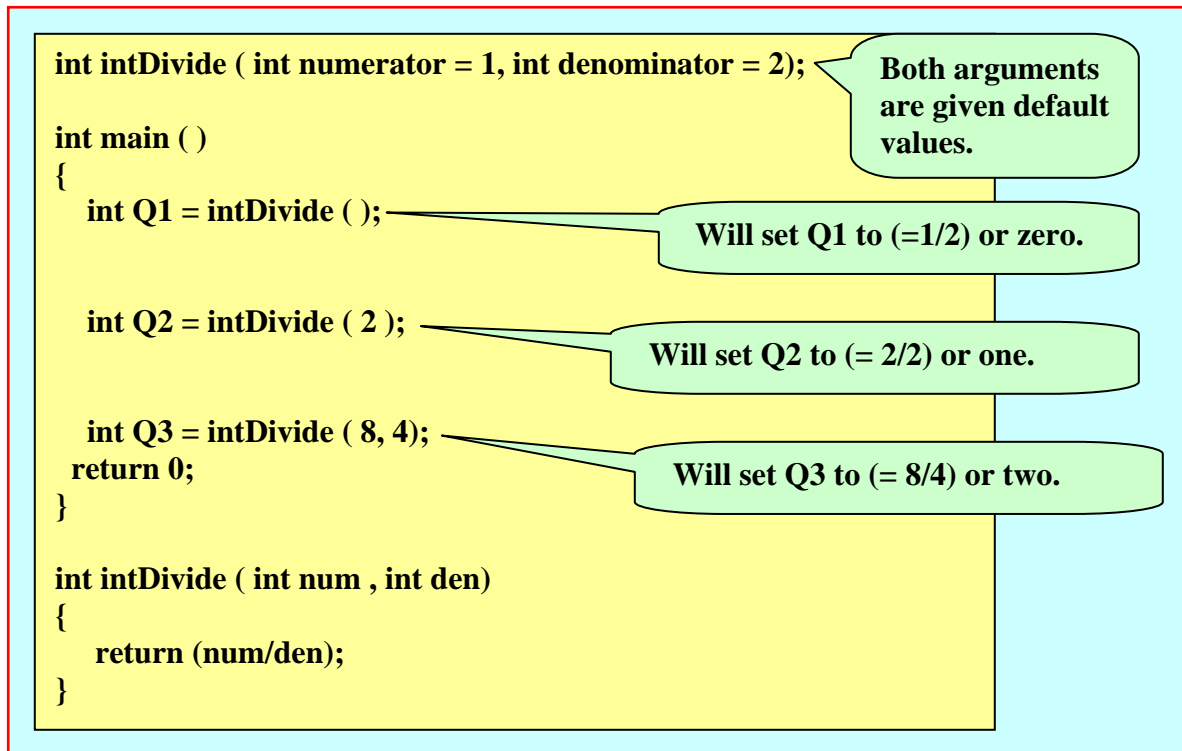
### Functions with Default Arguments

So far, we have discussed functions where the number of arguments in function call must match the number in the formal parameter list for the function. For example if we have a function *int intDivide(int numerator, int denominator)* that takes two integers as arguments and returns the result *numerator/denominator* then the call below to this function would be illegal:

*int quotient = intDivide(5);*

Compile error because *intDivide* takes two parameters.

C++ allows default arguments to be given to formal parameters, so that starting from argument at extreme right, one, more, or all arguments can be left out during function call. For example, the toy function *intDivide* can be written as below with the calls possible to invoke it (Figure 8.18).

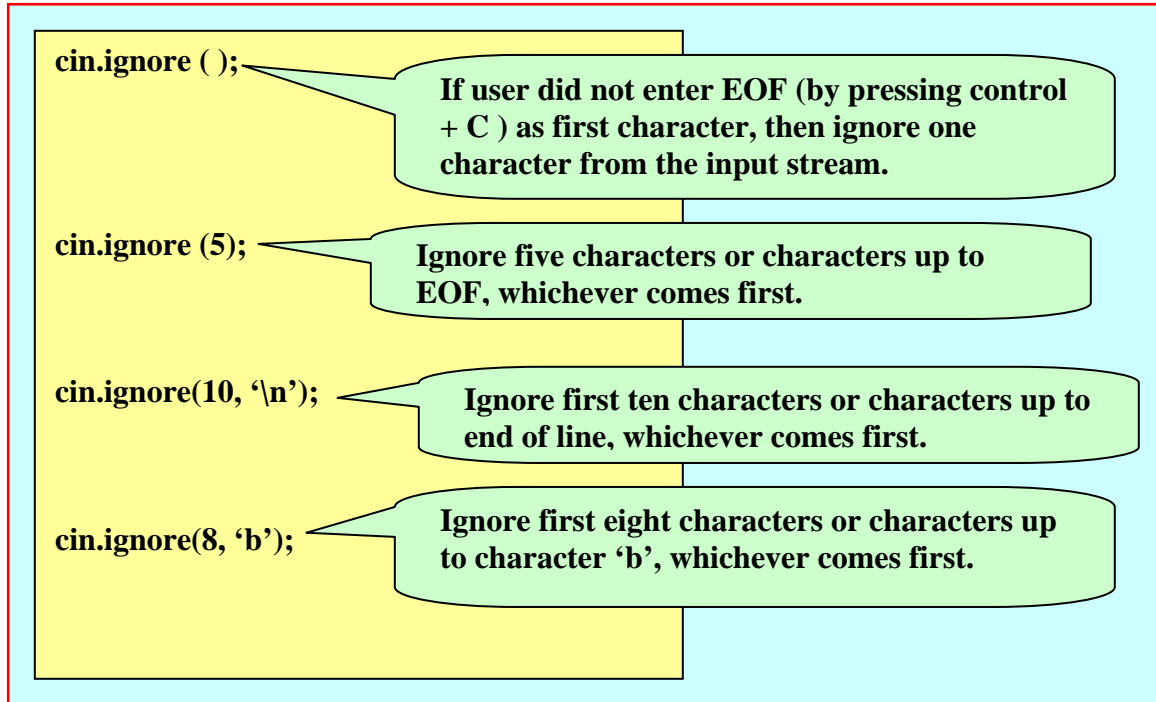
**FIG. 8.18**

Notice that default arguments are only specified in the proto-type. Their values are not repeated when full function definition is written after main function. When function with the default arguments is invoked, the actual arguments are mapped with the formal arguments from left to right. For example in Figure 8.18, in call *intDivide* (2) the value 2 is mapped with the left argument, numerator. Since right argument *denominator* has a default value of 2 also, the return value from function call *intDivide* (2) is one. It is important to choose default values that are sensible and are used most often. For example in case of function *intDivide*, it would be nonsensical to use a default value of zero for right argument *denominator*, as it would lead to divide by zero error. You have already used functions that take default arguments. Stream function *ignore* is one such function. For example the proto-type for *ignore* is as follows:

*istream & ignore (streamsize n = 1, int delim = EOF);*

Here, *n* is the maximum number of characters to ignore. *delim* is the delimiter provided. Function *ignore* is coded to ignore *n* characters, or up to character *delim*<sup>5</sup>, whichever comes first! Since both arguments have default values, the four calls to *ignore*, shown below have following affects (Figure 8.19).

<sup>5</sup> The characters for argument *delim* are actually represented by their ASCII value. That is why *delim* has an int data type.

**FIG. 8.19**

The presence of default arguments for *ignore* simplifies its invocation significantly. Very often, the program needs to ignore either certain number of characters or all characters up to end of file. In such cases, the call such as `cin.ignore(n)` is adequate.

#### print Function with Default cout argument

The facility of providing default argument can be used to upgrade the print function in Listing 8.11. This surprisingly leads to one print function doing the duty of two versions of print function in Listing 8.11 and reducing the code volume accordingly. Listing 8.12 shows the revised print and main functions.

```

1 ///////////////////////////////////////////////////////////////////
2 #include <iostream>
3 #include <string>
4 #include <fstream>
5 #include <iomanip>
6 using namespace std;
7
8 void getInFile(ifstream &, string&);
9 void getOutfile(ofstream&);
10 void print(istream & inst, ostream & oust = cout);
11
12 int main()
13 {
14     string Msg = "First";
15     ifstream in;
16     getInFile(in, Msg);
17     print(in);
18     ofstream out;
19     getOutfile(out);
20     in.clear(); //repairs the failed input stream
21     in.seekg(0,ios::beg); //Sets the reading marker to the beginning of file
22     cout<< "Printing to output file.\n";

```

```

23     print(in,out);
24     in.close();
25     out.close();
26
27     return 0;
28 }
29
30 void print(istream & input,  ostream & out)
31 {
32     double miles = double();
33     double gallons = double();
34     int counter = 1;
35
36     out<<"\n";
37     out<<fixed<<showpoint;
38     out<<setprecision(2);
39     out<<endl;
40     out<<"=====\\n";
41     out<<"          GAS MILEAGE DATA FOR YOUR CAR          "<<endl;
42     out<<"-----\\n";
43     out<<left<<setw(15)<<"Trip Number"<<setw(15)
44         <<"Miles"<<setw(15)<<"Gallons\\t"<<setw(15)
45         <<"MPG"<<endl;
46     out<<"=====\\n\\n";
47     input>>miles;
48     double sum = double();
49
50     while(input)
51     {
52         input>>gallons;
53         out<<setw(15)<<counter<<setw(15)<<miles;
54
55         if(int(gallons)<10)
56             out<<" ";
57
58         out<<setw(15)<<gallons<<"\\t"<<setw(15)<<miles/gallons<<"\\n\\n";
59
60
61         sum = sum + miles/gallons;
62         counter++;
63         input>>miles;
64     }
65     out<<"=====\\n";
66     out<<"The Average Miles Per Gallon for all Trips:      "
67         <<sum/(counter-1)<<endl;
68     out<<"=====\\n\\n";
69 }

```

## 70 Listing 8.12

In Listing 8.12 the functions *getInFile* and *getOutFile* are not shown as they remain unaltered. Let us analyze the new proto-type of function *print* given below.

```
void print(istream & inst,  ostream & oust = cout);
```

Notice that instead of *ifstream* and *ofstream* we are now using *istream* and *ostream* as the formal parameters. The reason for this change is that *ostream* as formal parameter would accept either *cout* or *ofstream* object as actual argument during the function call. This has to do with the property of objects called “polymorphism”.



*ostream* formal parameter can map with its two polymorphs, namely *cout*, and *ofstream* object. (By now it must be clear that polymorphism is a ubiquitous concept in software engineering as we have already seen polymorphic operators, function names and objects). Similarly *istream* formal parameter can accept either *cin* or *ifstream* object as actual arguments.

In modified print function we give default value, *cout* for the *ostream* object *out*. Affect of doing that is that if we make following call in main (Line #17):

```
print(in);
```

the default value *cout* is taken for right parameter and call is resolved as:

```
print(in, cout);
```

Then during invocation of print function the parameter *out* actually morphs into *cout*, and function outputs the data to the console. The call to print data to the output file `print(in, out);` identical to Listing 8.11. Outputs from Listings 8.11 and 8.12 are identical. Writing print function in form shown in Listing 8.12 is an excellent example of making use of default arguments. This is because in standard output user has only two choices – output to a file or to console. By providing *ostream* as formal parameter and *cout* as default parameter the function does work of two functions.

### Congruency of Function Prototype, Function Header and Function Call

In our discussion, so far one may have noticed that three properties of a function, namely prototype, header and function call must adhere to a pattern. If one of the three falls out of pattern, it becomes incongruent with the others. This may lead either to compile error or link errors. Rules for maintaining congruency are as follows:

1. Every function that is invoked and has a prototype must also have the full definition of that function. Consider the Listing 8.13 A below.

```
#include <iostream>
using namespace std;

void Foo();

int main()
{
    Foo();

    return 0;
}
```

#### Listing 8.13 A

**The function Foo invoked by the main has a prototype. However, the full definition of the function is missing. Compiler will flag this error with a cryptic error message as:**

```
error LNK2019: unresolved external symbol "void __cdecl
Foo(void)" (?Foo@YAXXZ) referenced in function _main
fatal error LNK1120: 1 unresolved externals
```

**What compiler is intimating here is that it cannot find the full definition of Foo, thus it cannot link to it. It is inconceivable that in a program as small as this one a Software Engineer will forget to write the full function definition. However, the beginners do make precisely this kind of error when number of functions increase and they simply forget to write the body some of them. This error can be avoided when software development is done incrementally – *one function at a time*. This approach requires that write a function and immediately test that it works properly. This may lead to lot of test code in main, which finished software will not have. However, it would lead to significant reduction in development time.**

2. Second kind of incongruence is introduced when there is a mismatch between the type, number or order of arguments between function call and the function header/prototype. Consider example in Listings 8.13B.

```
#include <iostream>
using namespace std;

void Foo();

int main()
{
    int val = 5;
    Foo(val);

    return 0;
}

void Foo()
{
    cout<<"Hello from Foo.\n";
}
```

### **Listing 8.13B**

**In Listing 8.13B the function Foo is defined correctly. However an incorrect call to it causes the following compile error:**

```
error C2660: 'Foo' : function does not take 1 arguments
```

**We already discussed examples of compile errors when wrong data type is passed as arguments and no automatic conversion is possible.**

3. If full function definition is not provided before the function is called then at least its prototype must be declared. This rule is a subtext of C++ rule that every identifier must be declared before its use. In Listing 8.13 C, the function Foo is called before either its full definition or its prototpe were declared.

```

#include <iostream>
using namespace std;

int main()
{
    Foo();

    return 0;
}
void Foo()
{
    cout<<"Hello from Foo.\n";
}

```

### **Listing 8.13C**

Compiler will give the following error message for Listing 8.13C.

error C3861: 'Foo': identifier not found

The message is very simple here. The function Foo is being called in main with out ever having being defined or declared, as prototype for Foo is missing.

## **Data Validation Functions**

Often the software needs to validate the user-inputted data. When entering data from keyboard, user may press a wrong key and alphabets could be entered for numeric data. Here we first write a function called *getValidInt*, where user is asked to enter an integer. If inputted data are not integer then user is informed and looped through until an integer is entered. One example of requiring an integer input would be where in a census we are recording number of people, which would always be an integer value. Another example would be as recording person's age on their last birthday. Previous examples are of situations, where integer values are positive. In mathematics, when finding average of numbers, the negative integers would also be encountered. First, we write the algorithm for function *getValidInt*. Then we convert it into a source code.

### **Algorithm for Function *getValidInt***

1. Prompt user to enter an integer.
2. Store the input.
3. If input stream has failed then inform user.
  - 3.1 Repair the input stream.
  - 3.2 Ignore extra characters in the stream.
  - 3.3 Continue with the fresh iteration of data input.
4. Else Ignore the extra characters in the input stream and end the data input loop.

Listing 8.14A shows the program with the function *getValidInt*.

```

1
2  #include <iostream>
3  using namespace std;
4
5  int getValidInt();
6  int main()
7  {
8      cout<<"The value of first integer entered is: "
9          <<getValidInt()<<endl;
10     cout<<"The value of second integer entered is: "
11         <<getValidInt()<<endl;
12     return 0;
13 }
14 int getValidInt()
15 {
16     int num = 0;
17     bool done = false;
18
19     while(!done)
20     {
21         cout<<"Enter an integer : ";
22         cin>>num;
23
24         if(!cin)
25         {
26             cout<<"Illegal value entered." <<endl;
27             cin.clear();
28             cin.ignore(128, '\n');
29             done = false;
30         }
31         else
32         {
33             cin.ignore(128, '\n');
34             done = true;
35         }
36     }
37
38     return num;
39 }

```

#### Listing 8.14A

Listing 8.14, main calls the function *getValidInt* on lines 9 and 11. When the control is transferred to the function, the algorithm for the function is executed. As noted in chapter five the input stream will fail if non-numeric data are entered, in which case the user is informed and prompted to re-input valid data (Figure 8.20A).

```

Enter an integer : T
Illegal value entered.
Enter an integer : nwjrr 4rfnrj5rf5
Illegal value entered.
Enter an integer : 5
The value of first integer entered is: 5
Enter an integer : -77
The value of second integer entered is: -77

```

**FIG. 8.20A**

Input stream would fail when either a single character or a string is entered when an integer is required. Program correctly loops through until user enters the integers.

Above function fulfills most of our needs. It however, is less robust when user enters a floating-point number in place of integer. If base is provided then the program stores the base as input integer and rejects the mantissa value (Figure 8.20B).

```

Enter an integer : -8.22305
The value of first integer entered is: -8
Enter an integer : .48
Illegal value entered.
Enter an integer : 0.48
The value of second integer entered is: 0

```

**FIG. 8.20B**

The current design of program assumes that user entered the mantissa mistakenly, and the base values of (as in this case) -8 or zero are the intended value. Well that is hard to confirm. For all we know, user may have been looking at the wrong data sheet and entering data from there. Therefore, a more robust version of function `getValidInt` would require that input fail for everything other than an integer entry. This can be done by using so-called string streams.

### string streams

String streams use requires inclusion of header file *sstream*. The two main classes in it are *istringstream* and *ostringstream*. In using *istringstream*, first entire user input is stored into a string using `getline`. Then this string is used to create an *istringstream* object. After that, the *istringstream* object so created is used as a source just like *cin* to populate the program variables. The use of *istringstream* will become clear when we see function `getValidIntVersion2` and its functionality (Listing 8.14B). In this program, we are guided by the failure of *istringstream* object, which presents a more accurate picture of data input.

```

1  #include <iostream>
2  #include <sstream>
3  #include <string>
4  using namespace std;
5
6  int getValidIntVersion2();
7
8  int main()
9  {
10     cout<<"The value of first integer entered is: "
11         <<getValidIntVersion2()<<endl;
12     cout<<"The value of second integer entered is: "
13         <<getValidIntVersion2()<<endl;
14
15     return 0;
16 }
17
18 int getValidIntVersion2()
19 {
20     int num = 0;
21     bool done = false;
22
23     while(!done)
24     {
25         cout<<"Enter an integer : ";
26         string input = "";
27         getline(cin,input);
28         istringstream is;
29         is.str(input);
30         is>> num;
31
32         if(is)
33         {
34             string remaining = "";
35             is>>remaining;
36
37             if(remaining == "")
38                 done = true;
39
40             else
41             {
42                 cout<<"Illegal or floating point value entered.\n";
43                 is.clear();
44                 done = false;
45             }
46         }
47         else
48         {
49             cout<<"Illegal value entered.\n";
50             done = false;
51         }
52     }

```

```

53
54         return num;
55     }

```

#### 56 **Listing 8.14B**

In upgraded version *getValidIntVersion2* we store entire user input into a *string* using *getline* function (line 27). In this case *cin* can never fail as every input can be stored into a *string*. Then we declare an object *is* of type *istringstream* (line 28). The object *is* can be populated with a string to be parsed by calling its *str* function and passing the string to it as an argument. Thus in line 29 the action *is.str(input)* simply populates the *is* with the string that was entered by the user. If user only entered an integer then next line *is >> num* stores that integer into variable *num*. However, if user had entered an alphabetic or non-numeric character (excluding the – or + sign) the *is* will fail and *else* block from lines 47 to 51 will execute, informing user that they have entered invalid data causing the loop to execute again. If *is* did not fail then there are only two options: user has either entered the required integer or a floating point number, or a numeric followed by non-numeric. In that case then we first test for remaining characters in the *is*. We store the remaining characters in the string *remaining* (line 35). If *remaining* is a zero length string then user correctly entered a required integer. Otherwise either a floating point number or some illegal value followed by the integer data were entered. In that case the *else* block executes (lines 40 to 45). User is informed accordingly, the *is* is repaired by calling its *clear* function and boolean flag *done* is set to *false* and data entry loop goes through another iteration. Figure 8.20C shows a test run from Listing 8.14C.

```

Enter an integer : -8.12
Illegal or floating point value entered.
Enter an integer : 8 bb a      a      qi
Illegal or floating point value entered.
Enter an integer : 0.223
Illegal or floating point value entered.
Enter an integer : .4
Illegal value entered.
Enter an integer : -44
The value of first integer entered is: -44
Enter an integer : anw ccfbvhre
Illegal value entered.
Enter an integer : c
Illegal value entered.
Enter an integer : +8.22
Illegal or floating point value entered.
Enter an integer : +7
The value of second integer entered is: 7

```

**FIG. 8.20C**



## ostringstream

To complete the topic of string streams, we now discuss the use of *ostringstream*. The *ostringstream* uses insertion operator << to push and store data into it. (In that sense, its functioning is similar to *cout*). Then *str* function of *ostringstream* can be used to return the string version of whatever has been stored inside it. In addition to insertion operator << the *put* member function can be used to input characters into an *ostringstream* object. Net result is that one can design complex output string into an *ostringstream* object and then conveniently output its string form. In Listing 8.15 where we use both *istringstream* and *ostringstream*, we first read entire ASCII file into a string. Then we parse this string file for any errors in the age recorded for various people and edit the errant data. Finally, the edited string file is written to a new output file.

Listing 8.15 has three functions:

1. **copyIntoString** – This function copies entire input file into one string. The input file contains records of age of people – one record per line.
2. **editFileData** – This function parses the string created by **copyIntoString** function for invalid data for age and corrects the errant age data.
3. **getPositiveInt** – This function is a modified version of function **getValidIntVersion2** of Listing 8.14B. The function only gets a positive integer from the user as inputted from the keyboard.

The main function opens the input file, and then calls **copyIntoString** function to copy the input file into it. Then main calls **editFileData** function, which corrects the errant file and returns, by reference, to main the corrected file, stored into the string in a format ready to be written into output file. Then finally, the edited error corrected output file is created.

```

1  #include <iostream>
2  #include <string>
3  #include <sstream>
4  #include <fstream>
5  using namespace std;
6  void copyIntoString(istream & in, string & data);
7
8  void editFileData(string & data);
9
10
11
12
13
14 int getPositiveInt();
15
16 int main()
17 {
18     ifstream in;
19     in.open("Age.txt");
20     string data = "";
21     copyIntoString(in, data);
22     cout<<"Copy of Age.txt file.\n"
23          <<data<<endl;
24     in.close();

```

1. Function **copyIntoString** copies the entire file into a string.

2. Function **editFileData** gets user input to replace errant data in the file.

3. Function **getPositiveInt** ascertains that user only enters a positive non-zero integer for age.

```

25     editFileData(data);
26     cout<<"The edited file:\n"<<data;
27     ofstream out;
28     out.open("AgeEdited.txt");
29     out<<data;
30     out.close();
31     return 0;
32 }
33 void copyIntoString(istream & in, string & copy)
34 {
35     copy = "";
36     if(in != cin)
37     {
38         char ch;
39         ostringstream os;
40
41         while((ch = in.peek()) != EOF)
42         {
43             os.put(in.get());
44         }
45         copy = os.str();
46     }
47 }
48
49
50
51 void editFileData(string & data)
52 {
53     istream is;
54     ostringstream os;
55
56     is.str(data);
57     string name = "";
58     int age = 0;
59
60     while(is)
61     {
62         is>>name>>age;
63
64
65
66
67         if(age <=0 && is)
68         {
69             int num = 0;
70             cout<<"For " <<name<<" age is " <<age<<endl;
71             num = getPositiveInt();
72             os<<name<<" " <<num<<endl;
73         }
74         else if(is)
75             os<<name<<" " <<age<<endl;
76     }
77     data.clear();
78     data = os.str();
79 }
80
81

```

4. Ascertain that *in* is actually an *ifstream* object.

5. EOF controlled loop ascertains file bonded to *in* is copied to *ostringstream* object *os*.

6. Function returns by reference the string copy that is copy of input file.

7. Function gets the *string* version of the file to be edited by reference.

8. *istream* object is populated by the unedited string data using its member function *str*.

9. *istream* object is used to populate local variables *name* and *age*. Action is same as using *cin*.

10. If *age* is invalid then if structure gets the valid *age* using *getPositiveInt* function.

11. *ostringstream* object is populated with valid *name/age*.

12. the *str* function of object *os* returns the *string* stored inside it and populates the *data* with edited values.

```

82 int getPositiveInt()
83 {
84     int num = 0;
85     bool done = false;
86
87     while(!done)
88     {
89         cout<<"Enter valid age : ";
90         string input = "";
91         getline(cin,input);
92         istringstream is;
93         is.str(input);
94         is>>num;
95
96         if(is)
97         {
98             string remaining = "";
99             is>>remaining;
100
101             if(remaining == "" && num>0)
102                 done = true;
103             else
104             {
105                 cout<<"Illegal or floating point value entered.\n";
106                 is.clear();
107                 done = false;
108             }
109         }
110         else
111         {
112             cout<<"Illegal value entered.\n";
113             done = false;
114         }
115     }
116     return num;
117 }

```

### 118 Listing 8.15

First, we discuss all the functions used in Listing 8.15.

#### copyIntoString

The function takes an *istream* and *string* objects as formal parameters. Thus, caller function can pass either a *cin* or an *ifstream* object as actual argument for first parameter. The if structure (*bubble #4*) ascertains that object *in* passed to function is actually not *cin*, in which case it can only be an *ifstream* object. Then we use the *EOF* controlled loop to read the file and populate the *ostringstream* object *os* (*bubble #5*). The design of this type of *EOF* controlled loop has already been discussed in chapter seven. The *get* function gets a character from the input stream and *put* function of *os* adds that character to it. Therefore, the loops exits when entire input file has been read into the *ostringstream* object *os*. Finally the *str* function of *ostringstream* object returns the string version and stores in *string* copy which is returned to caller of function by reference (*bubble #6*).

**editFileData**

Function takes string to be edited as a reference (*bubble #7*). This function shows a more robust use of object *ostream*. Function then prepares an *istream* object *is* that is populated by the unedited *string data* (*bubble #8*).

**Incremental Development**

Most sensible thing to do while using functions is, to break down software development into functions, making each function a developmental unit. Process of decomposing the software system in functions is called decomposition. The DFD diagram developed in chapter two is an example of decomposition.

**Designing with Invariants****Software Engineering Related Practices**

1.

**User Notes**

