



Government Arts College(Autonomous)

Coimbatore – 641018

Re-Accredited with 'A' grade by NAAC

Object Oriented Programming with C++

Dr. S. Chitra

Associate Professor

Post Graduate & Research Department of Computer Science

Government Arts College(Autonomous)

Coimbatore – 641 018.

Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	OBJECT ORIENTED PROGRAMMING WITH C++	III	18BCS33C

Objective:

- Learn the fundamentals of input and output using the C++ library
- Design a class that serves as a program module or package.
- Understand and demonstrate the concepts of Functions, Constructor and inheritance.

UNIT – I

Principles of Object Oriented Programming: Software Crisis - Software Evolution - Procedure Oriented Programming - Object Oriented Programming Paradigm - Basic concepts and benefits of OOP - Object Oriented Languages - Structure of C++ Program - Tokens, Keywords, Identifiers, Constants, Basic data type, User-defined Data type, Derived Data type – Symbolic Constants – Declaration of Variables – Dynamic Initialization - Reference Variable – Operators in C++ - Scope resolution operator – Memory management Operators – Manipulators – Type Cast operators – Expressions and their types – Conversions – Operator Precedence - Control Structures

UNIT – II

Functions in C++: Function Prototyping - Call by reference - Return by reference - Inline functions - Default, const arguments - Function Overloading – Classes and Objects - Member functions - Nesting of member functions - Private member functions - Memory Allocation for Objects - Static Data Members - Static Member functions - Array of Objects - Objects as function arguments - Returning objects - friend functions – Const Member functions .

UNIT – III

Constructors: Parameterized Constructors - Multiple Constructors in a class - Constructors with default arguments - Dynamic initialization of objects - Copy and Dynamic Constructors - Destructors - Operator Overloading - Overloading unary and binary operators – Overloading Using Friend functions – manipulation of Strings using Operators.

UNIT – IV

Inheritance: Defining derived classes - Single Inheritance - Making a private member inheritable – Multilevel, Multiple inheritance - Hierarchical inheritance - Hybrid inheritance - Virtual base classes - Abstract classes - Constructors in derived classes - Member classes - Nesting of classes.

UNIT – V

Pointers, Virtual Functions and Polymorphism: Pointer to objects – this pointer- Pointer to derived Class - Virtual functions – Pure Virtual Functions – C++ Streams –Unformatted I/O- Formated Console I/O – Opening and Closing File – File modes - File pointers and their manipulations – Sequential I/O – updating a file :Random access –Error Handling during File operations – Command line Arguments.

TEXT BOOKS

1.E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.

Unit II – Functions

Modular Programming

“The process of splitting of a large program into small manageable tasks and designing them independently is known as Modular Programming or Divide-&-Conquer Technique.”

C++ Functions

- Self-contained program that performs a specific task.
- “**Set of program statements** that can be processed independently.”
- Like in other languages, called **subroutines** or **procedures**.

Advantages

- Elimination of redundant code
- Easier debugging
- Reduction in the Size of the code
- Leads to reusability of the code

Functions are broadly classified as

1. Built-in functions (C++ Library functions)
2. User-defined functions

1. Built-in functions or C++ Library functions

Library functions are shipped along with the compilers. They are predefined and pre-compiled into library files, and their prototypes can be found in the files with .h (called header files) as their extension in the include directory.

Some of the built-in library functions are

strlen(), strncpy(), strcmp() → available in string.h

pow(), sqrt(), sin(), tan() → available in math.h

getch(), clrscr() → available in conio.h

```
// namelen.cpp: use of string library functions
#include <iostream.h>
#include <string.h>           // string function header file
void main()
{
    char name[ 20 ];
    cout << "Enter your name: ";
    cin >> name;
    int len = strlen( name ); // strlen returns the length of name
    cout << "Length of your name = " << len;
}
```

Run

```
Enter your name: Raikumar
Length of your name = 8
```

2. User-defined Functions:

Function Components

- 1. Function Prototypes (or) Function Declaration
- 2. Function Definition(declarator & body)
- 3. Function call(actual parameters)
- 4. Function Parameters(formal parameters)
- 5. return statement

In C++, the main () returns a value of type int to the operating system. The functions that have a return value should use the return statement for terminating.

The main () function in C++ is therefore defined as follows.

```
int main( )  
{  
-----  
-----  
return(0)  
}
```

Since the return type of functions is int by default, the keyword int in the main() header is optional.

1. Function Declaration Syntax

return-type function-name(list of parameters with their type separated by comma);

eg. 1. `int add-function(int a,int b);`

eg. 2. `int largest(int a,int b,int c);`

eg. 3. `double power-function(float a, int b);`

2. Function Definition(declarator & body) Syntax

return-type function-name(list of parameters with their type separated by comma)

{....

statement block;

.....

return

}

eg. 1.

```
int add-function(int a,int b)
```

```
{
```

```
int c;
```

```
c=a+b;
```

```
return c;
```

```
}
```

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
    float v = a*b*c;
    .....
    .....
}
```

The function `volume()` can be invoked in a program as follows:

```
float cubel = volume(b1,w1,h1); // Function call
```

The variable `b1`, `w1`, and `h1` are known as the actual parameters which specify the dimensions of `cubel`. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

Sample function

```
int add_int(int a, int b)
{
    return (a+b) ;
}
```

Return type

Function name

Formal parameters

Function body

3. Function call(actual parameters) Syntax

function-name(actual parameters);

eg.

```
void main()
```

```
{
```

```
int k;
```

```
int add-function(int a,int b);    → function declaration
```

```
.....
```

```
.....
```

```
k=add-function(int a,int b);    → function call ; a & b are actual parameters
```

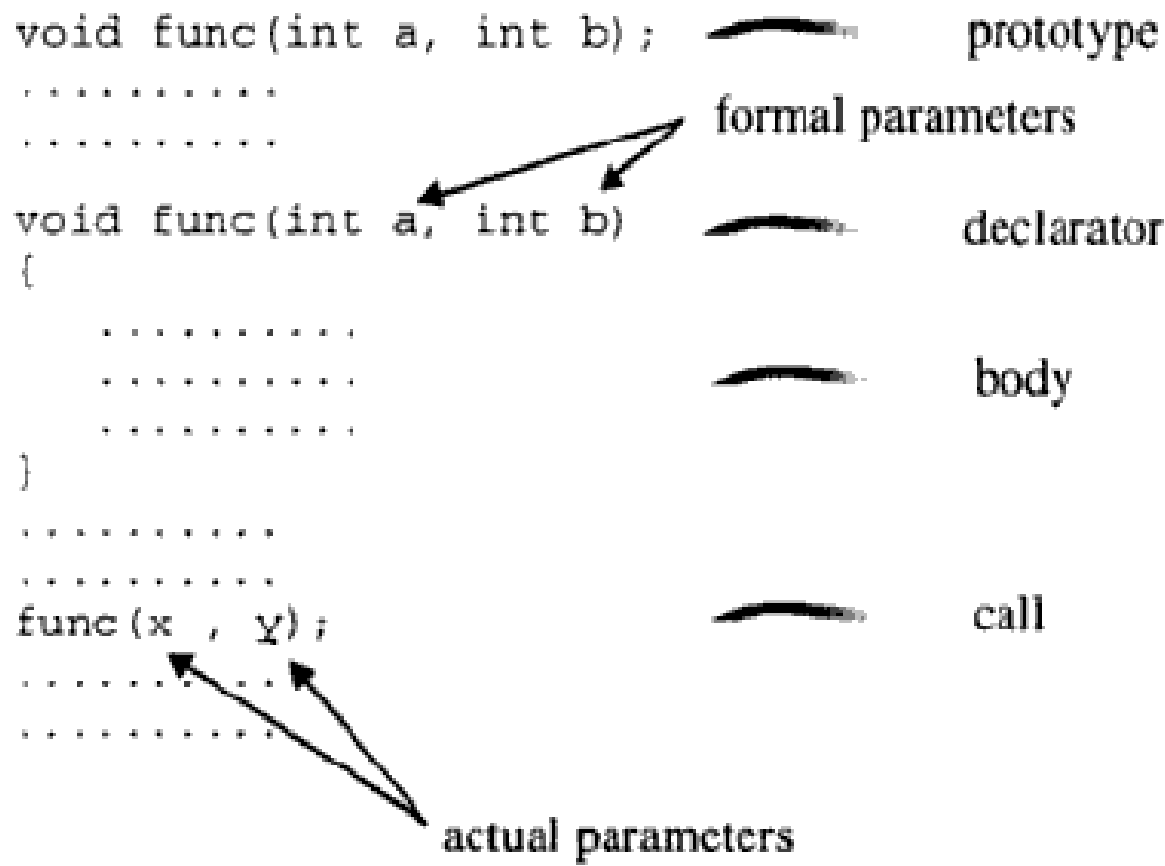
```
.....
```

```
}
```

```
int add-function(int x,int y)    → function definition ; x & y are formal parameters
```

```
{
```

```
void show();    /* Function declaration */
main()
{
    .....
    show();     /* Function call */
    .....
}
void show()    /* Function definition */
{
    .....
    .....     /* Function body */
    .....
}
}
```



The program `max1.cpp` illustrates the various components of a function. It computes the maximum of two integer numbers.


```
// max1.cpp: maximum of two integer numbers
#include <iostream.h>
int max( int x, int y );           // prototype
void main()                       // function caller
{
    int a, b, c;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    c = max( a, b );              // function call
    cout << "max( a, b ): " << c << endl;
}
int max( int x, int y )          // function definition
{
    // all the statements enclosed in braces forms body of the function
    if( x > y )
        return x;                // function return
    else
        return y;                // function return
}
```


Run


```
Enter two integers <a, b>: 20 10
max( a, b ): 20
```

```
int max( int x, int y ); // prototype
```

Function name
defines function

```
int max(int x, int y)  function declarator  
{  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

 **x** no semicolon

 function body

```
c = max( a, b ); // function call
```

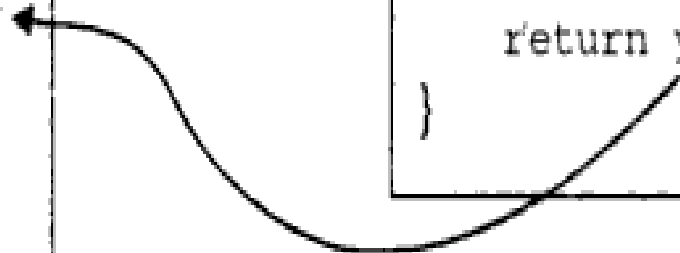
```
return x; // function return  
and  
return y; // function return
```

caller

```
void main()  
{  
    ~~~~~  
    ~~~~~  
    ~~~~~  
    c=max(a, b);  
    ~~~~~  
    ~~~~~  
    ~~~~~  
}
```

callee

```
int max(x, y)  
{  
    ~~~~~  
    ~~~~~  
    ~~~~~  
    else  
        return y;  
}
```



The value of y is,
returned to main()
and assigned to c.

Types of functions based on their **return type & parameters**

1. function that takes no parameters & doesn't return any value
2. function that takes parameters & doesn't return any value
3. function that takes parameters & returns a value
4. function that takes no parameters & returns a value – (rare type)

1. function that takes **no parameters & doesn't return any value**

```
void main()
{
void add();
add();
}
void add(void)
{
int a,b,c;
cin>>a>>b;
c=a+b;
cout<<c;
}
```

2. function that takes parameters & doesn't return any value

```
void main()
{
void add(int,int);
int a,b;
cin>>a>>b;
add(a,b);
}
void add(int x,int y)
{
int c;
c=x+y;
cout<<c;
}
```

3. function that takes parameters & returns a value

```
void main()
{
int add(int,int);
int a,b,c;
cin>>a>>b;
c=add(a,b);
cout<<c;
}
int add(int x,int y)
{
int c;
c=x+y;
return(c);
}
```

4. function that takes no parameters & returns a value

```
void main()
{
int add();
int c;
c=add();
cout<<c;
}
int add()
{
int x,y,z;
cin>>x>>y;
z=x+y;
return(z);
}
```


Parameter Passing in Functions

- * actual parameters – used in the function call
- * formal parameters – used in function declarator & definition

Parameter passing is a mechanism for communication of data and information between the calling function (caller) and the called function (callee). It can be achieved either by passing the value or address of the variable. C++ supports the following three types of parameter passing schemes:

- ◆ Pass by Value
- ◆ Pass by Address
- ◆ Pass by Reference (only in C++)

Passing Constant Values to Functions

The program `chart1.cpp` illustrates the passing of a numeric constant as an argument to a function. This constant argument is assigned to the formal parameter which is processed in the function body.

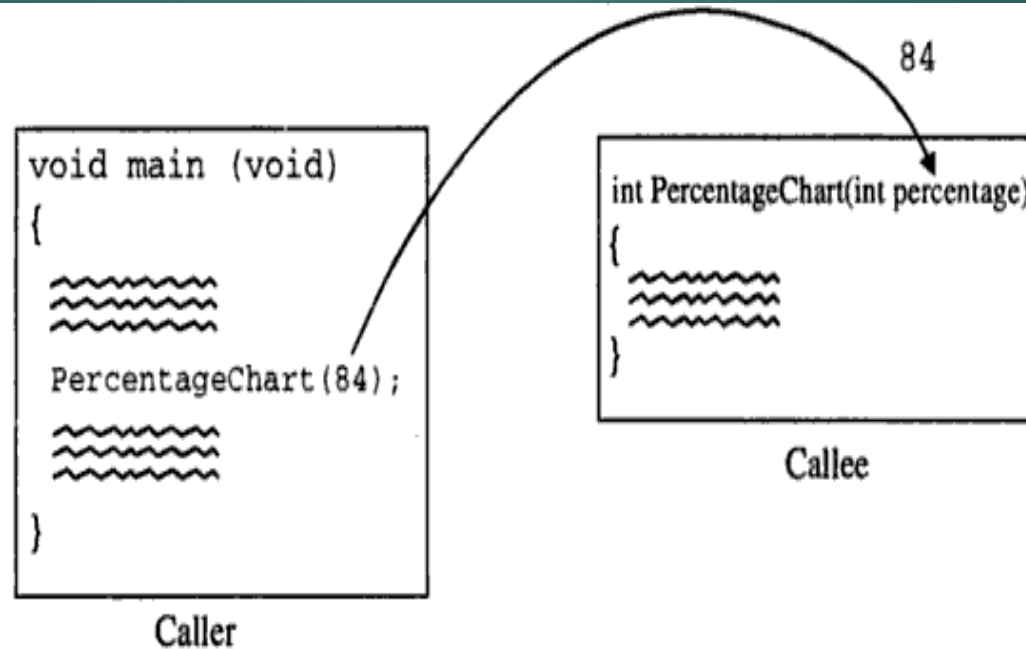
```
// chart1.cpp: Percentage chart by passing numeric value
#include <iostream.h>
void PercentageChart( int percentage );
void main()
{
    cout << "Sridevi : ";
    PercentageChart( 50 );
    cout << "Rajkumar: ";
    PercentageChart( 84 );
    cout << "Savithri: ";
    PercentageChart( 79 );
    cout << "Anand   : ";
    PercentageChart( 74 );
}
void PercentageChart( int percentage )
{
    for( int i = 0; i < percentage/2; i++ )
        cout << '\xCD';      // double line character (see ASCII table)
    cout << endl;
}
```

Run

```
Sridevi : =====
Rajkumar: =====
Savithri: =====
Anand   : =====
```

In `main()`, the statement

```
PercentageChart( 84 );
```



Passing Variable Values to Functions

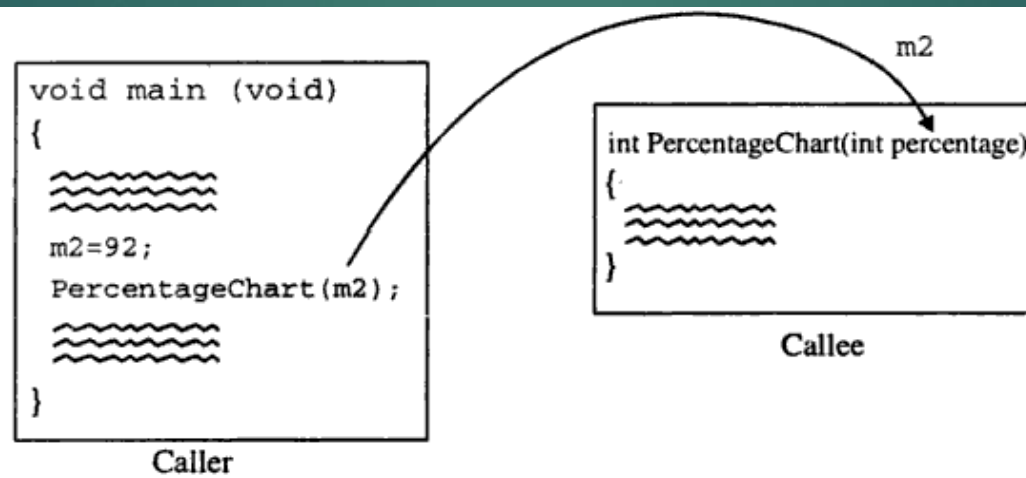
```
// chart2.cpp: Percentage chart by passing variables
#include <iostream.h>
void PercentageChart( int percentage );
void main()
{
    int m1, m2, m3, m4;
    cout << "Enter percentage score of Sri, Raj, Savi, An: ";
    cin >> m1 >> m2 >> m3 >> m4;
    cout << "Sridevi : ";
    PercentageChart( m1 );
    cout << "Rajkumar: ";
    PercentageChart( m2 );
    cout << "Savithri: ";
    PercentageChart( m3 );
    cout << "Anand   : ";
    PercentageChart( m4 );
}
void PercentageChart( int percentage )
{
    for( int i = 0; i < percentage/2; i++ )
        cout << '\xCD';      // double line character (see ASCII table)
    cout << endl;
}
```

Run

```
Enter percentage score of Sri, Raj, Savi, An: 55 92 83 67
Sridevi : =====
Rajkumar: =====
Savithri: =====
Anand   : =====
```

In `main()`, the statement

```
PercentageChart( m2 );
```



Functions with Multiple Arguments

```
// chart3.cpp: Percentage chart by passing multiple variables
#include <iostream.h>
void PercentageChart( int percentage, char style );
void main()
(
    int m1, m2, m3, m4;
    cout << "Enter percentage score of Sri, Raj, Savi, An: ";
    cin >> m1 >> m2 >> m3 >> m4;
    cout << "Sridevi : ";
    PercentageChart( m1, '*' );
    cout << "Rajkumar: ";

    PercentageChart( m2, '\xCD' );
    cout << "Savithri: ";
    PercentageChart( m3, '~' );
    cout << "Anand : ";
    PercentageChart( m4, '!' );
}
void PercentageChart( int percentage, char style )
(
    for( int i = 0; i < percentage/2; i++ )
        cout << style;
    cout << endl;
}
```

Run

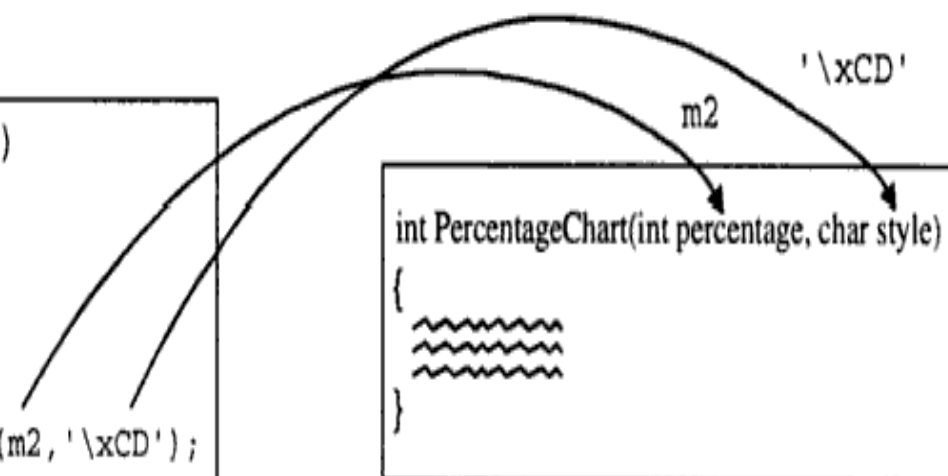
```
Enter percentage score of Sri, Raj, Savi, An: 55 92 83 67
Sridevi : .*****
Rajkumar: =====
Savithri: ~~~~~
Anand : !!!!!!!!!!!!!
```

```
void main (void)
{
  ~~~~~
  m2=92;
  PercentageChart (m2, '\xCD');
  ~~~~~
}
```

Caller

```
int PercentageChart(int percentage, char style)
{
  ~~~~~
}
```

Callee



```
// ifact.cpp: factorial computation Returns a long integer value
#include <iostream.h>
long fact( int n )
{
    long result;
    if( n == 0 )
        result = 1;    // factorial of zero is one
    else
    {
        result = 1;
        for( int i = 2; i <= n; i++ )
            result = result * i;
    }
    return result;
}
void main( void )
{
    int n;
    cout << "Enter the number whose factorial is to be found: ";
    cin >> n;
    cout << "The factorial of " << n << " is " << fact(n) << endl;
}
```

Run

```
Enter the number whose factorial is to be found: 5
The factorial of 5 is 120
```

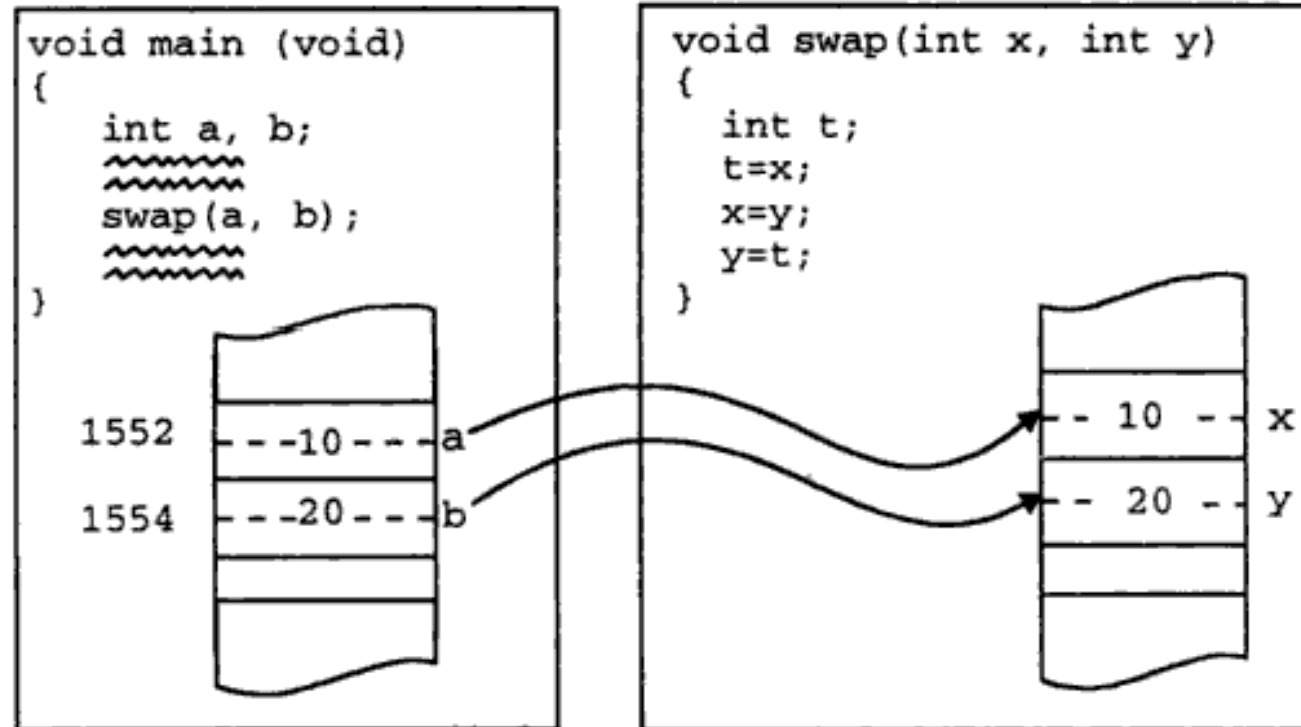


```
// swap1.cpp: swap integer values by value
#include <iostream.h>
void swap( int x, int y )
{
    int t;    // temporary used in swapping
    cout<<"Value of x and y in swap before exchange: "<< x <<" " << y << endl;
    t = x;
    x = y;
    y = t;
    cout<<"Value of x and y in swap after exchange: "<< x <<" " << y << endl;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b );
    cout << "Value of a and b on swap( a, b ) in main(): " << a << " " << b;
}
```

Run

```
Enter two integers <a, b>: 10 20
Value of x and y in swap before exchange: 10 20
Value of x and y in swap after exchange: 20 10
Value of a and b on swap( a, b ) in main(): 10 20
```

Memory Allocation for Functions



Pass by Address

C++ provides another means of passing values to a function known as pass-by-address. Instead of passing the value, the address of the variable is passed. In the function, the address of the argument is copied into a memory location instead of the value. The de-referencing operator is used to access the variable in the called function.

```
// swap2.cpp: swap integer values by pointers
#include <iostream.h>
void swap( int * x, int * y )
{
    int t;    // temporary used in swapping
    t = *x;
    *x = *y;
    *y = t;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( &a, &b );
    cout << "Value of a and b on swap( a, b ): " << a << " " << b;
}
```

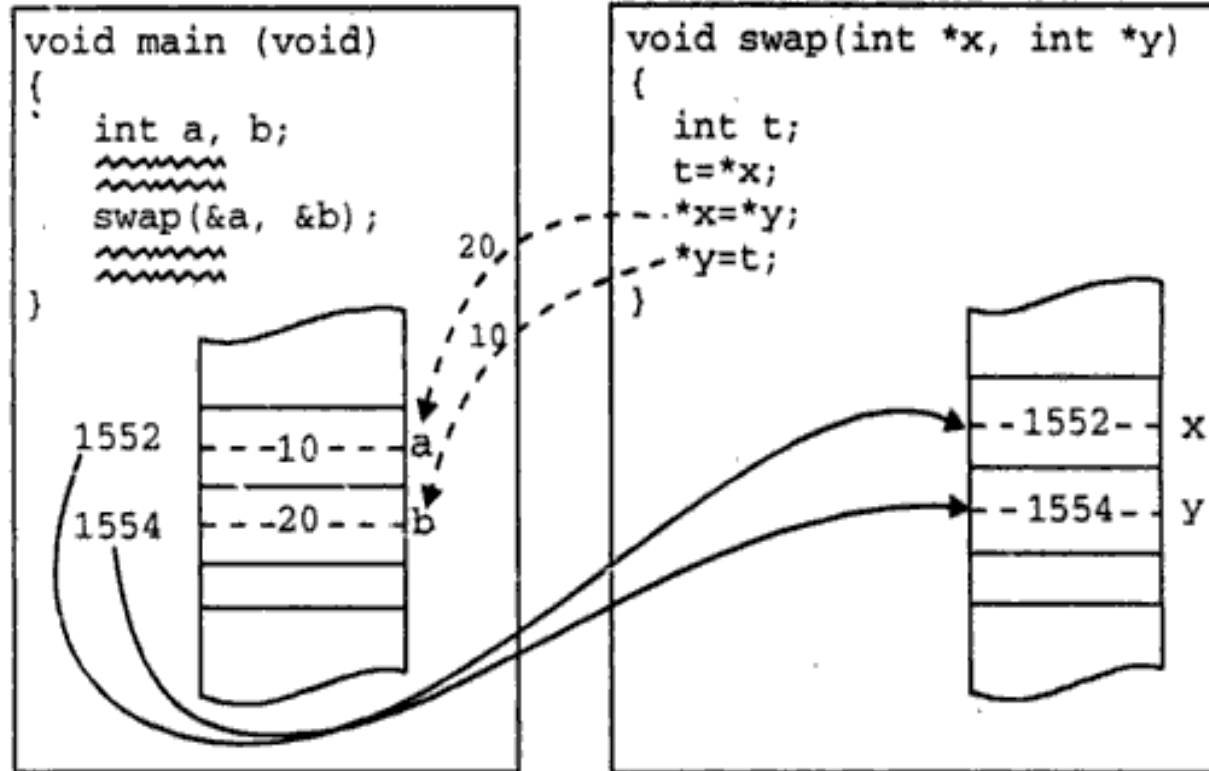
Run

```
Enter two integers <a, b>: 10 20
Value of a and b on swap( a, b ): 20 10
```

In main(), the statement

```
swap( &x, &y )
```

invokes the function swap and assigns the address of the actual parameters a and b to the formal parameters x and y respectively.



Parameter passing by Pointer

Pass by Reference

Passing parameters by reference has the functionality of pass-by-pointer and the syntax of call-by-value. Any modifications made through the formal pointer parameter is also reflected in the actual parameter. Therefore, the function body and the call to it is identical to that of call-by-value, but has the effect of call-by-pointer.

To pass an argument by reference, the function call is similar to that of call by value. In the function declarator, those parameters, which are to be received by reference must be preceded by the & operator. The reference type formal parameters are accessed in the same way as normal value parameters. However, any modification to them will also be reflected in the actual parameters. The program `swap3.cpp` illustrates the mechanism of passing parameters by reference.

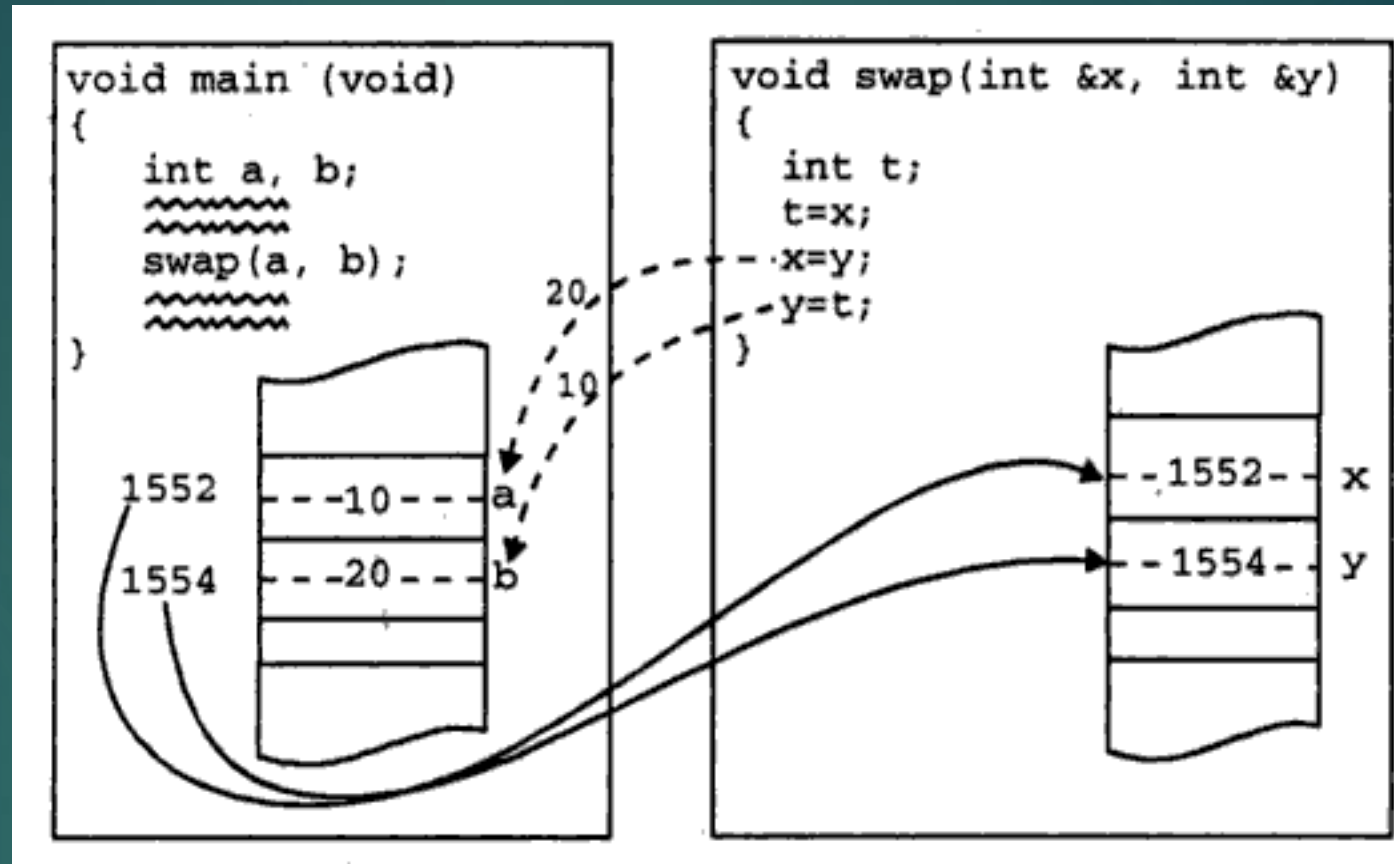
```
// swap3.cpp: swap integer values by reference
#include <iostream.h>
void swap( int & x, int & y )
{
    int t;    // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b );
    cout << "Value of a and b on swap( a, b ): " << a << " " << b;
}

```

Run

```
Enter two integers <a, b>: 10 20
Value of a and b on swap( a, b ): 20 10
```

Parameter passing by reference



The following points can be noted about reference parameters:

- A reference can never be null, it must always refer to a legitimate object (variable).
- Once established, a reference can never be changed so as to make it point to a different object.
- A reference does not require any explicit mechanism to dereference the memory address and access the actual data value.

Return by Reference

```
// ref.cpp: return variable by reference
#include <iostream.h>
int & max( int & x, int & y );      // prototype
void main()
{
    int a, b, c;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    max( a, b ) = 425;
    cout<<"The value of a and b on execution of max(a,b) = 425; ..." << endl;
    cout << "a = " << a << " b = " << b;
}
int & max( int & x, int & y )      // function definition
{
    // all the statements enclosed in braces form body of the function
    if( x > y )
        return x;                  // function return
    else
        return y;                  // function return
}
```

Run1

Enter two integers <a, b>: 1 2

The value of a and b on execution of max(a, b) = 425; ...
a = 1 b = 425

Run2

Enter two integers <a, b>: 2 1

The value of a and b on execution of max(a, b) = 425; ...
a = 425 b = 1

In main(), the statement

```
max( a, b ) = 425;
```


Functions with default arguments

- * Usually functions should be passed values during function call.
- * C++ allows function calls with fewer argument values if the remaining arguments are assigned default values

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j=5, int k=10);    // legal
int mul(int i=5, int j);              // illegal
int mul(int i=0, int j, int k=10);   // illegal
int mul(int i=2, int j=5, int k=10); // legal
```

Example:

```
#include<iostream.h>
#include<stdio.h>
main()
{
float amount;
float value(float p,int n,float r=15);
void printline(char ch='*',int len=40);
printline( );
amount=value(5000.00,5);
cout<<"\n final value="<<amount<<endl;
printline('=');
//function definitions
float value (float p,int n, float r)
{
float si;
si=p+(p*n*r)/100;
return(si);
}
void printline (char ch,int len)
{
for(inti=1;i<=len;i++)
cout<<ch<<endl;
}
```

output:-

final value = 8750.00

=====

const arguments

In C++, an argument to a function can be declared as `const` as shown below.

```
int strlen(const char *p);  
int length(const string &s);
```

The qualifier `const` tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

INLINE FUNCTION:

An inline function is a function that is expanded inline when it is invoked. That is the compiler replaces the function call with the corresponding function code.

The inline functions are defined as follows:-

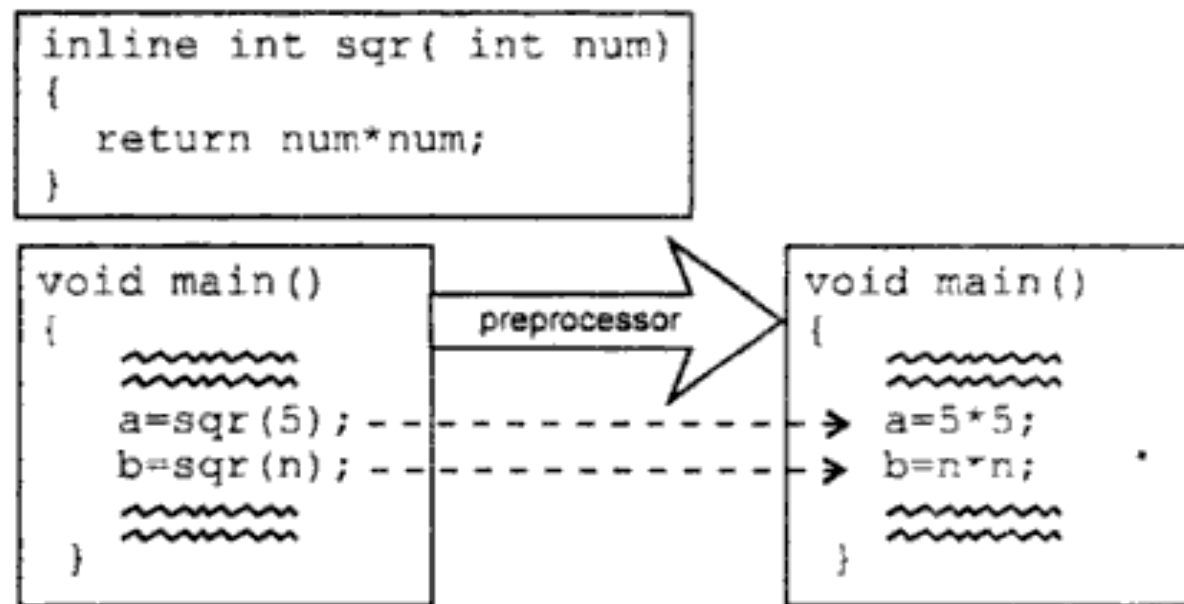
```
inline function-header  
{  
function body;  
}
```

Example:

```
inline int sqr(int num)  
{  
return(num*num);  
}
```

C++ provides an alternative to normal function calls in the form of inline functions. Inline functions are those whose function body is inserted in place of the function call statement during the compilation process. With the inline code, the program will not incur any context switching overhead. The concept of inline functions is similar to *macro functions* of C. Hence, inline functions enjoy both the flexibility and power offered by normal functions and macro functions respectively.

An inline function definition is similar to an ordinary function except that the keyword `inline` precedes the function definition. The syntax for defining an inline function is shown in Figure 7.12..



```
inline double cube(double a) {return(a*a*a);}
```

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

Function Overloading

Function polymorphism, or function overloading is a concept that allows multiple functions to share the same name with different argument types. Function polymorphism implies that the function definition can have multiple forms. Assigning one or more function body to the same name is known as *function overloading* or *function name overloading*.

```
// Declarations
int add(int a, int b);           // prototype 1
int add(int a, int b, int c);    // prototype 2
double add(double x, double y); // prototype 3
double add(int p, double q);     // prototype 4
double add(double p, int q);     // prototype 5

// Function calls
cout << add(5, 10);              // uses prototype 1
cout << add(15, 10.0);           // uses prototype 4
cout << add(12.5, 7.5);         // uses prototype 3
cout << add(5, 10, 15);         // uses prototype 2
cout << add(0.75, 5);           // uses prototype 5
```

```
// swap5.cpp: multiple swap functions, function overloading
#include <iostream.h>
void swap( char & x, char & y )
{
    char t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void swap( int & x, int & y )
{
    int t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void swap( float & x, float & y )
{
    float t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
    char ch1, ch2;
    cout << "Enter two Characters <ch1, ch2>: ";
    cin >> ch1 >> ch2;
    swap( ch1, ch2 ); // compiler calls swap( char &a, char &b );
    cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b ); // compiler calls swap( int &a, int &b );
    cout << "On swapping <a, b>: " << a << " " << b << endl;
}
```



```

float c, d;
cout << "Enter two floats <c, d>: ";
cin >> c >> d;
swap( c, d ); // compiler calls swap( float &a, float &b );
cout << "On swapping <c, d>: " << c << " " << d;
:

```

Run

```

Enter two Characters <ch1, ch2>: R_K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5_10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5_99.5
On swapping <c, d>: 99.5 20.5

```

In the above program, three functions named `swap()` are defined, which only differ in their argument data types: `char`, `int`, or `float`. In `main()`, when the statement

```
swap( ch1, ch2 );
```

is encountered, the compiler invokes the `swap()` function which takes character type arguments. This decision is based on the data type of the arguments. (see Figure 7.13).

```

void swap(float &x, float &y);
void swap(int &x, int &y);
void swap(char &x, char &y);

void main()
{
    char ch1, ch2;
    int a, b;
    float x, y;
    swap(ch1, ch2);
    swap(a, b);
    swap(x, y);
}

```

Scope & Extent of Variables

*The region of source code in which the identifier is visible is called the scope of the identifier.

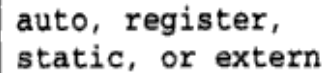
* The period of time during which the memory is associated with a variable is called the extent of the variable.

Storage Classes

The period of time during which memory is associated with a variable is called the extent of the variable. It is characterized by storage classes. The storage class of a variable indicates the allocation of storage space to the variable by the compiler. Storage classes define the extent of a variable. C++ supports the following four types of storage classes:

- ◆ auto
- ◆ register
- ◆ extern
- ◆ static

Syntax of declaring variables with storage class



auto, register,
static, or extern

StorageClass DataType Variable1,....;

Declaration Versus Definition

A declaration informs the compiler about the existence of the data or a function some where in the program. A definition allocates the storage location. In C++, a piece of data or function can be declared in several different places, but there must only be one definition.

Auto Variables

By default, all the variables are defined as auto variables. They are created when the function/block is entered and destroyed when the function/block is terminated. The memory space for local auto variables is allocated on the stack. The global auto variables are visible to all the modules of a program, and hence, they cannot be defined many times unlike the declarations.

Register Variables

The allocation of CPU (processor) registers to variables, speeds up the execution of a program; memory is not referred when such variables are accessed. The number of variables, which can be declared as `register` are limited (typically two or three), within any function or as global variables (else they are treated as auto variables).

Static Variables

The `static` storage class allows to define a variable whose scope is restricted to either a block, a function, or a file (but not all files in multimodule program) and extent is the life-span of a program. The memory space for local static and global variables is allocated from the *global heap*. Static variables that are defined within a function remember their values from the previous call (i.e., the values to which they are initialized or changed before returning from the function). The static variables defined outside all functions in a file are called *file static variables*. They are accessible only in the file in which they are defined. The program `count.cpp` illustrates the use of function static local variables.

```
// count.cpp: use of static variables defined inside functions
#include <iostream.h>
void PrintCount( void )
{
    static int Count = 1; // Count is initialized only on the first call
    cout << "Count = " << Count << endl;
    Count = Count + 1;    // The incremented value of Count is retained
}
void main( void )
{
    PrintCount();
    PrintCount();
    PrintCount();
}
```

Run

```
Count = 1
Count = 2
Count = 3
```

Extern Variables

When a program spans across different files, they can share information using global variables. Global variables must be defined only once in any of the program module and they can be accessed by all others. It is achieved by declaring such variables as `extern` variables. It informs the compiler that such variables are defined in some other file. Consider a program having the following files:

```
// file1.cpp: module one defining global variable
int done; // global variable definition
void func1()
{
    ....
    ....
}
void disp()
{
    ....
    ....
}
// file2.cpp: module two of the project
extern int done; // global variable declaration
void func3
{
    ....
    ....
}
```

In `file1.cpp`, the statement

```
int done;
```

defines the variable `done` as a global variable. In `file2.cpp`, the statement

```
extern int done;
```

declares the variable `done` and indicates that it is defined in some other file. Note that the definition of the variable `done` must appear in any one of the modules, whereas `extern` declaration can appear in any or all modules of a program. When the linker encounters such variables, it binds all references to the same memory location. Thus, any modification to the variable `done` is visible to all the modules accessing it.

Recursive Functions

*A function calling itself repeatedly until a condition is satisfied is called a recursive function

Two important conditions which must be satisfied by any recursive function are:

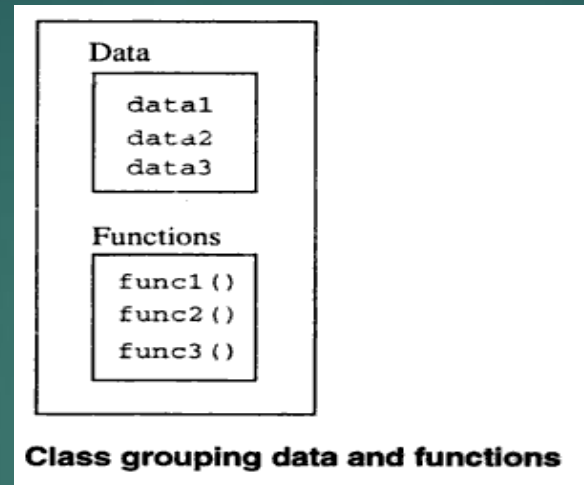
1. Each time a function calls itself it must be nearer, in some sense, to a solution.
2. There must be a decision criterion for stopping the process or computation.

```
// rfact.cpp: factorial of a number using recursion
#include <iostream.h>
void main( void )
{
    int n;
    long int fact( int );    // prototype
    cout << "Enter the number whose factorial is to be found: ";
    cin >> n;
    cout << "The factorial of " << n << " is " << fact(n) << endl;
}
long fact( int num )
{
    if( num == 0 )
        return 1;
    else
        return num * fact( num - 1 );
}
```

Run

```
Enter the number whose factorial is to be found: 5
The factorial of 5 is 120
```

Classes & Objects



Object-oriented programming constructs modeled out of data types called *classes*. Defining variables of a class data type is known as a *class instantiation* and such variables are called *objects*. (Object is an instance of a class.) A class encloses both the *data* and *functions* that operate on the data, into a *single unit*

The variables and functions enclosed in a class are called *data members* and *member functions* respectively. Member functions define the permissible operations on the data members of a class.

Using Class in C++ needs 3 steps to be followed

1. Specify the class

- i. Declaration of class
- ii. Defintion of member functions

2. Create objects for the class

3. Access the public members of the class using objects

Specifying a class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

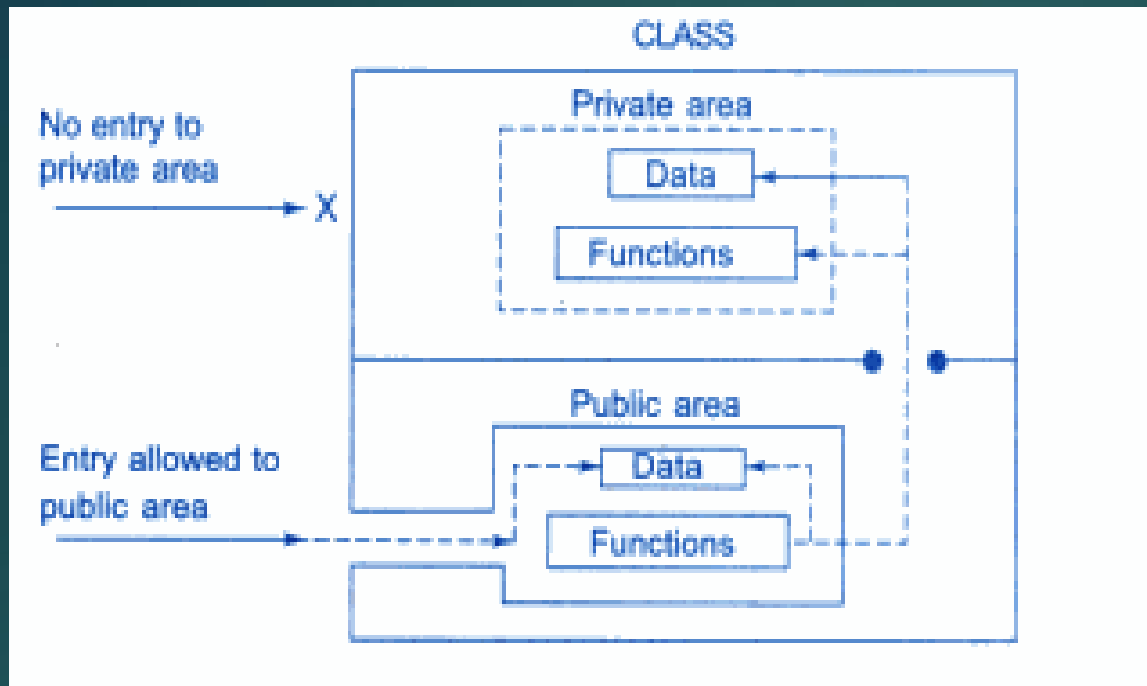
```
class class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declaration;
};
```

```
class ClassName
{
    // body of a class
};
```

Semicolon required here

Syntax of class specification

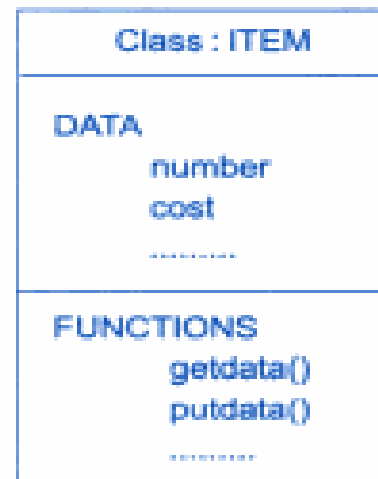
The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.



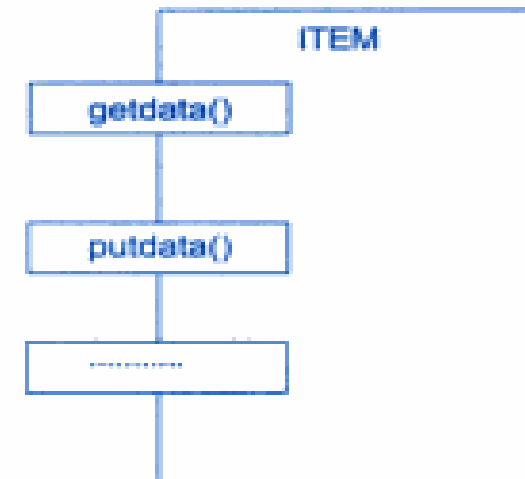
A Simple Class Example

A typical class declaration would look like:

```
class item
{
    int number;           // variables declaration
    float cost;          // private by default
public:
    void getdata(int a, float b); // functions declaration
    void putdata(void);          // using prototype
}; // ends with semicolon
```

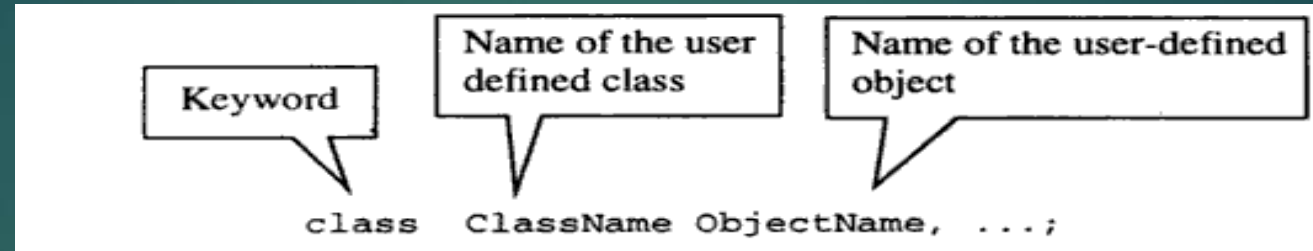


(a)



(b)

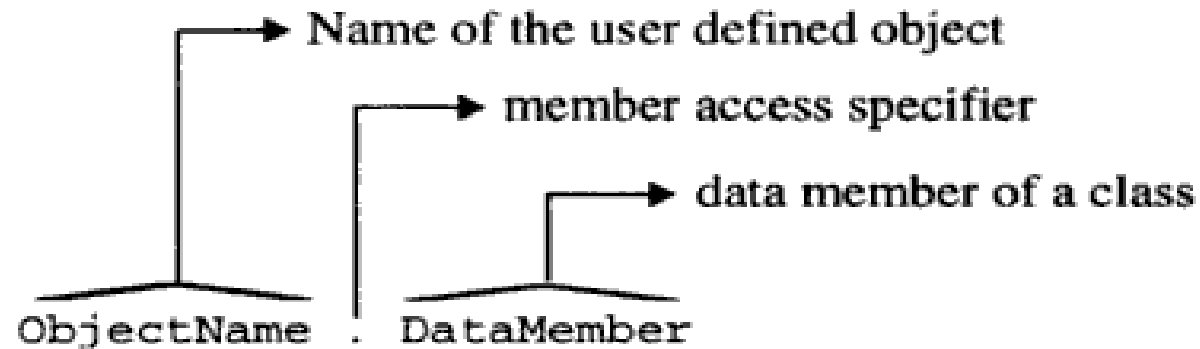
Creating Objects



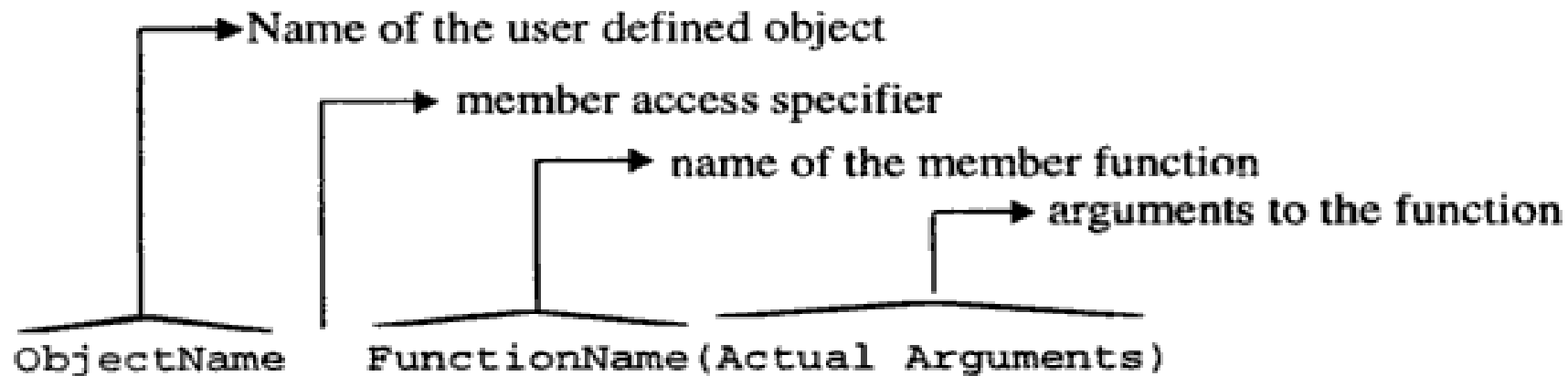
```
item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Accessing the Members of the Class



(a) Syntax for accessing data member of a class



(b) Syntax for accessing member function of a class

Accessing the Members

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the **number** (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```


sends a message to the object **x** requesting it to display its contents.


Defining Member Functions


Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

Member function definition outside a class declaration

```
class ClassName
{
    ....
    ReturnType MemberFunction(arguments);  function prototype
    ....
};
```

 user defined class name

 Scope resolution operator

```
ReturnType ClassName :: MemberFunction ( arguments )
{
    // body of the function
}
```

Member function definition Outside the class specification

```
return-type class-name :: function-name (argument declaration)  
{  
    Function body  
}
```

```
void item :: getdata(int a, float b)  
{  
    number = a;  
    cost = b;  
}
```

```
void item :: putdata(void)  
{  
    cout << "Number :" << number << "\n";  
    cout << "Cost   :" << cost   << "\n";  
}
```

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
    int number;
    float cost;
public:
    void getdata(int a, float b);    // declaration
        // inline function
    void putdata(void)              // definition inside the class
    {
        cout << number << "\n";
        cout << cost << "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.





Access Specifiers/Visibility Modes:

C++ provides 3 types of Visibility Modes

1. private
2. public
3. protected

Private Members

The private members of a class have strict access control. Only the member functions of the same class can access these members. The private members of a class are inaccessible outside the class, thus, providing a mechanism for preventing accidental modifications of the data members.

```
class Person
{
    private :  Note: colon here
               access specifier
    // private members
    .....
    int age;  private data
    int getage();  private function
    .....
};

Person p1;
a=p1.age; x cannot access private data
p1.getage(); x cannot access private function
```

Private members accessibility

```
class Inaccessible
{
    int x;
    void Display()
    {
        cout << "\nData = " << x;
    }
};
void main()
{
    Inaccessible obj1;           // Creating an object.
    obj1.x = 5;                 // Error: Invalid access.
    obj1.Display();            // Error: Invalid access.
}
```

The class having all the members with private access control is of no use; there is no means available to communicate with the external world. Therefore, classes of the above type will not contribute anything to the program.

Protected Members

The access control of the protected members is similar to that of private members and has more significance in inheritance.

Access control of protected members is

```
class Person
{
    protected:           access specifier
    // protected members
    .....
    int age;             protected data
    int getage();        protected function
    ....
};
Person p1;
a=p1.age;
p1.getage(); }          cannot access protected member
                       ( same as private )
```

Protected members accessibility

Public Members

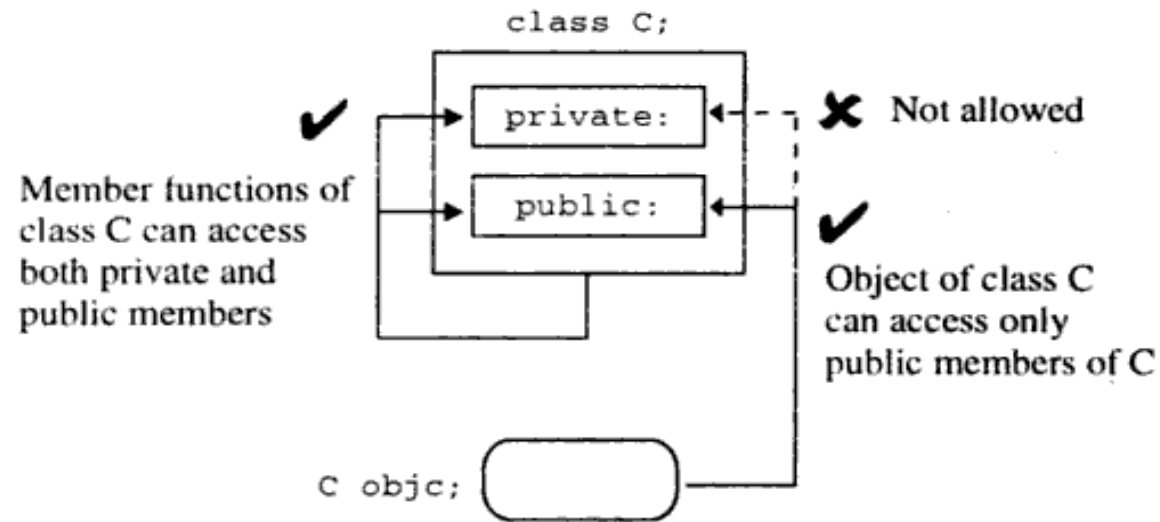
The members of a class, which are to be visible (accessible) outside the class, should be declared in *public* section. All data members and functions declared in the public section of the class can be accessed without any restriction from anywhere in the program, either by functions that belong to the class or by those external to the class. Accessibility control of public members is

```
class Person
{
    public:           access specifier
    // public members
    .....
    int age;         public data
    int getage();   public function
    .....
};
Person p1;
a=p1.age; ✓ can access public data
p1.getage(); ✓ can access public function
```

Public members accessibility

Access Specifier	Accessible to	
	Own class Members	Objects of a Class
private:	Yes	No
protected:	Yes	No
public:	Yes	Yes

Table 10.1: Visibility of class members



Class member accessibility

```

#include <iostream>

using namespace std;

class item
{
    int number;    // private by default
    float cost;   // private by default
public:
    void getdata(int a, float b);    // prototype declaration,
                                     // to be defined
    // Function defined inside class
    void putdata(void)
    {
        cout << "number :" << number << "\n";
        cout << "cost   :" << cost   << "\n";
    }
};

//..... Member Function Definition .....
void item :: getdata(int a, float b)    // use membership label
{
    number = a;    // private variables
    cost = b;     // directly used
}

//..... Main Program .....

int main()
{
    item x; // create object x

    cout << "\nobject x " << "\n";

    x.getdata(100, 299.95);    // call member function
    x.putdata();              // call member function

    item y;                    // create another object

    cout << "\nobject y" << "\n";

    y.getdata(200, 175.50);
    y.putdata();

    return 0;
}

```

```
// student.cpp: member functions defined inside the body of the student class
#include <iostream.h>
#include <string.h>
class student
{
private:
    int roll_no;           // roll number
    char name[ 20 ];      // name of a student
public:
    // initializing data members
    void setdata( int roll_no_in, char *name_in )
    {
        roll_no = roll_no_in;
        strcpy( name, name_in );
    }
    // display data members on the console screen
    void outdata()
    {
        cout << "Roll No = " << roll_no << endl;
        cout << "Name = " << name << endl;
    }
};
void main()
{
    student s1;           // first object/variable of class student
    student s2;           // second object/variable of class student
    s1.setdata( 1, "Tejaswi" ); // object s1 calls member setdata()
    s2.setdata( 10, "Rajkumar" ); //object s2 calls member setdata()
    cout << "Student details..." << endl;
    s1.outdata();         // object s1 calls member function outdata()
    s2.outdata();         // object s2 calls member function outdata()
}
```

Run

```
Student details...
Roll No = 1
Name = Tejaswi
Roll No = 10
Name = Rajkumar
```


In `main()`, the statements

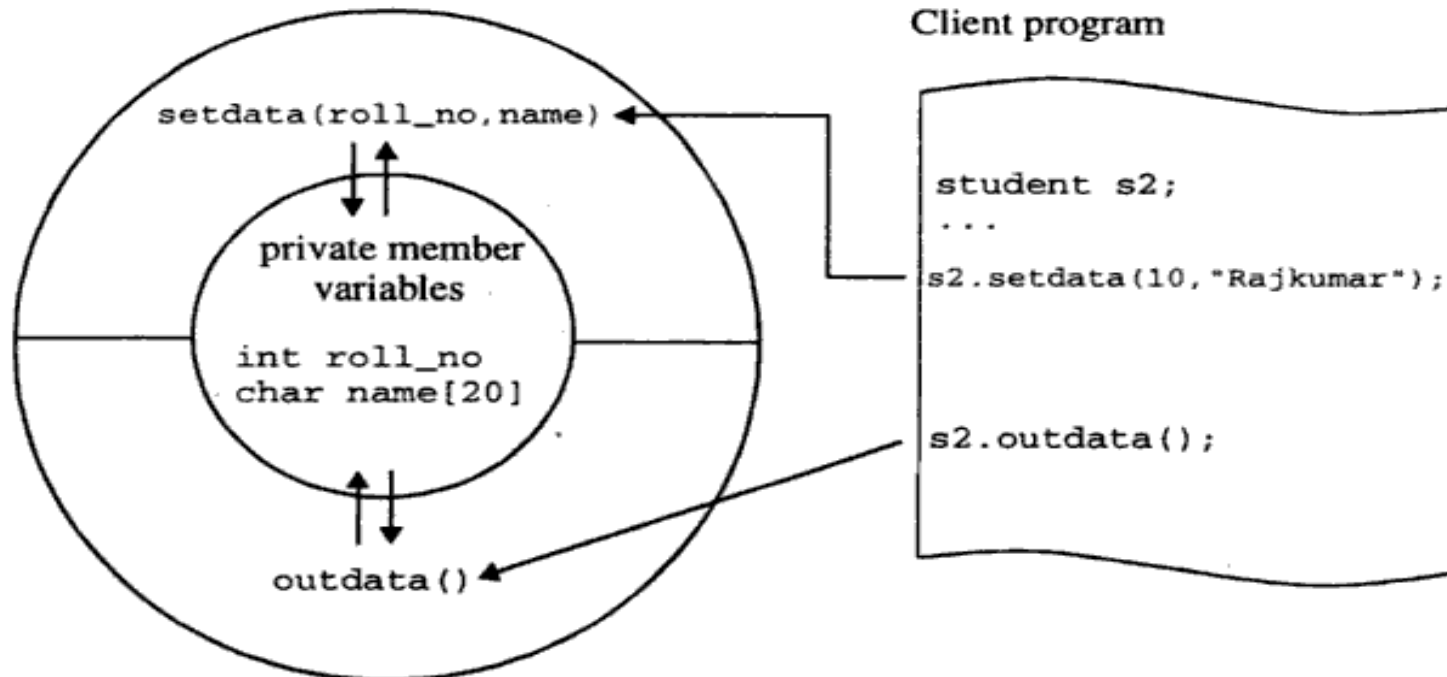
```
student s1;    // first object/variable of class student
student s2;    // second object/variable of class student
```

create two objects called `s1` and `s2` of the `student` class. The statements

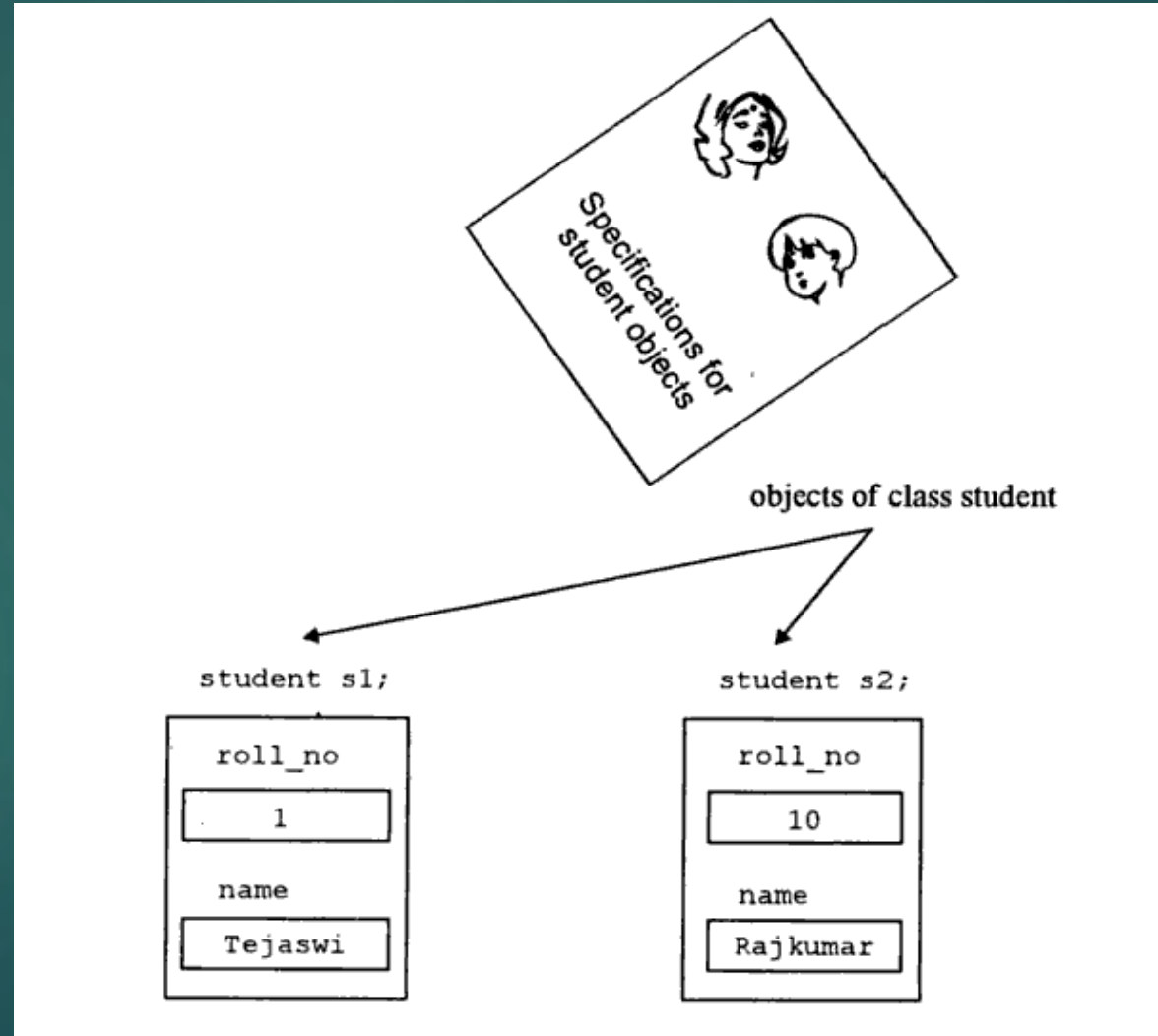
```
s1.setdata( 1, "Tejaswi" ); //object s1 calls member function setdata
s2.setdata( 10, "Rajkumar" ); //object s2 calls member function setdata
```

initialize the data members of the objects `s1` and `s2`. The object `s1`'s data member `roll_no` is assigned 1 and name is assigned Tejaswi. Similarly, the object `s2`'s data member `roll_no` is assigned 10 and name is assigned Rajkumar.

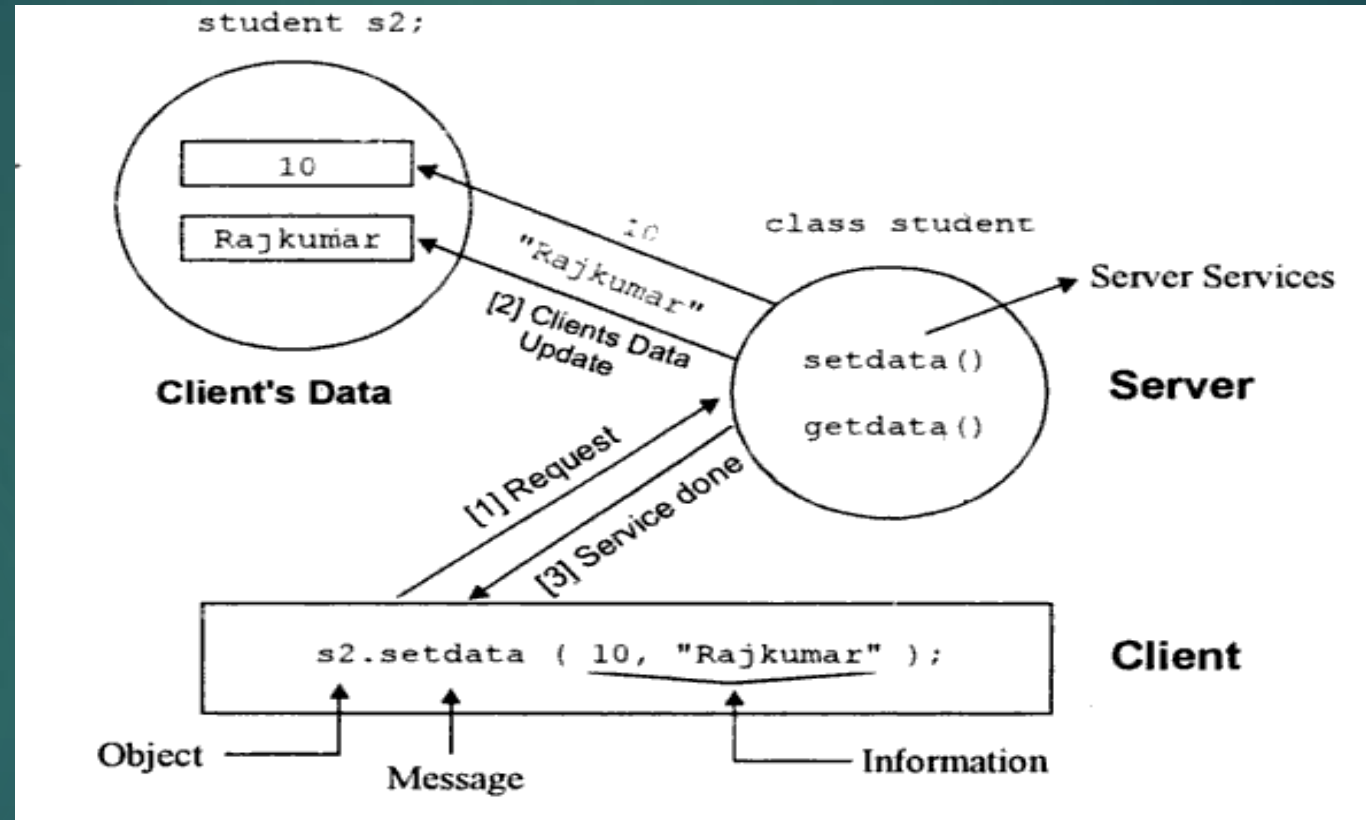
Instance of the class `student`



Two objects of the class student



Client-Server model for message communication



Characteristics of Member Functions:

- A program can have several classes and they can have member functions with the same name. The ambiguity of the compiler in deciding *which function belongs to which class* can be resolved by the use of membership label (`ClassName::`), the scope resolution operator.
- Private members of a class, can be accessed by all the members of the class, whereas non-member functions are not allowed to access. However, friend functions can access them.
- Member functions of the same class can access all other members of their own class without the use of dot operator.
- Member functions defined as `public` act as an interface between the service provider (server) and the service seeker (client).
- A class can *have* multiple member functions with the same name as long as they differ in terms of argument specification (data type or number of arguments).

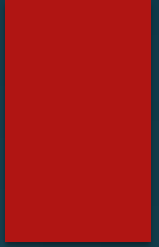
In OOPs, the process of programming involves the following steps:

- **Creation of classes for defining objects and their behaviors.**
- **Creation of class objects; class declaration acts like a blueprint for which physical resources are not allocated.**
- **Establishment of communication among objects through message passing**

Write a simple program using class in C++ to input subject mark and prints it.

```
class marks
{
private :
int roll;
int m1,m2;
public:
void getdata();
void displaydata();
};
void marks: :getdata()
{ cout<<"enter the roll-no:"<<cin>>roll;
cout<<"enter 1st subject mark:"<<
cin>>m1;
cout<<"enter 2nd subject mark:"<<
cin>>m2;
}
void marks: :displaydata()
{ cout<<"Roll No."<<roll;
cout<<"1st subject mark:"<<m1<<endl ;
cout<<"2nd subject mark:"<<m2;
}
```

```
void main()
{
clrscr();
marks x;
x.getdata();
x.displaydata();
}
```



Nesting of Member Functions

```
// nesting.cpp: A member function accessing another member function
#include <iostream.h>
class NumberPairs
{
    int num1, num2;        // private by default
public:
    void read()
    {
        cout << "Enter First Number: ";
        cin >> num1;
        cout << "Enter Second Number: ";
        cin >> num2;
    }
    int max()              // member function
    {
        if( num1 > num2 )
            return num1;
        else
            return num2;
    }
    // Nesting of member function
    void ShowMax()
    {
        // calls member function max()
        cout << "Maximum = " << max();
    }
};
void main()
{
    NumberPairs n1;
    n1.read();
    n1.ShowMax();
}
```

Run

```
Enter First Number: 5
Enter Second Number: 10
Maximum = 10
```

Private Member Functions

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
    int m;
    void read(void);          // private member function
public:
    void update(void);
    void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();          // won't work; objects cannot access
                   // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
    read();          // simple call; no object used
}
```


Arrays within a Class

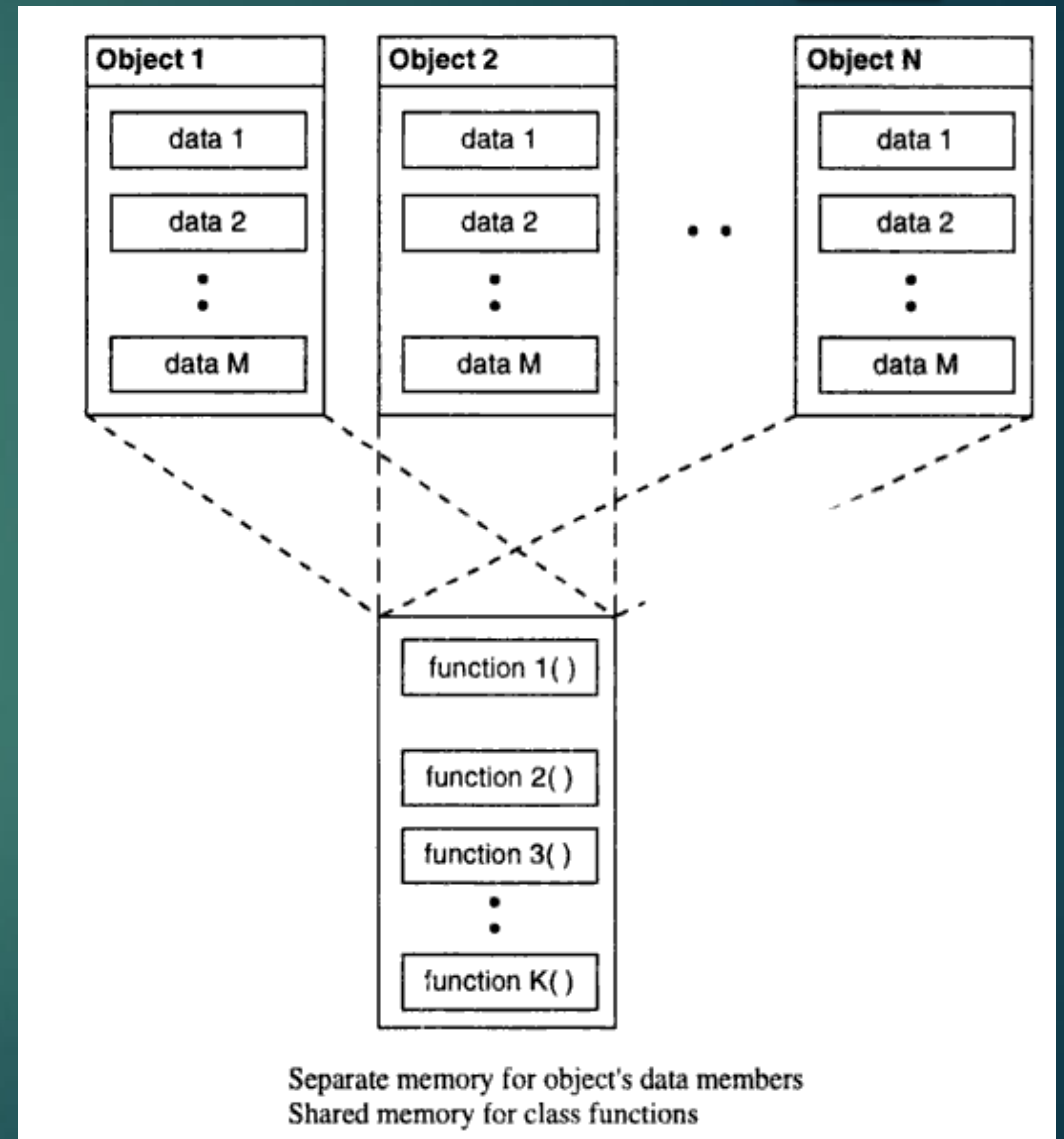
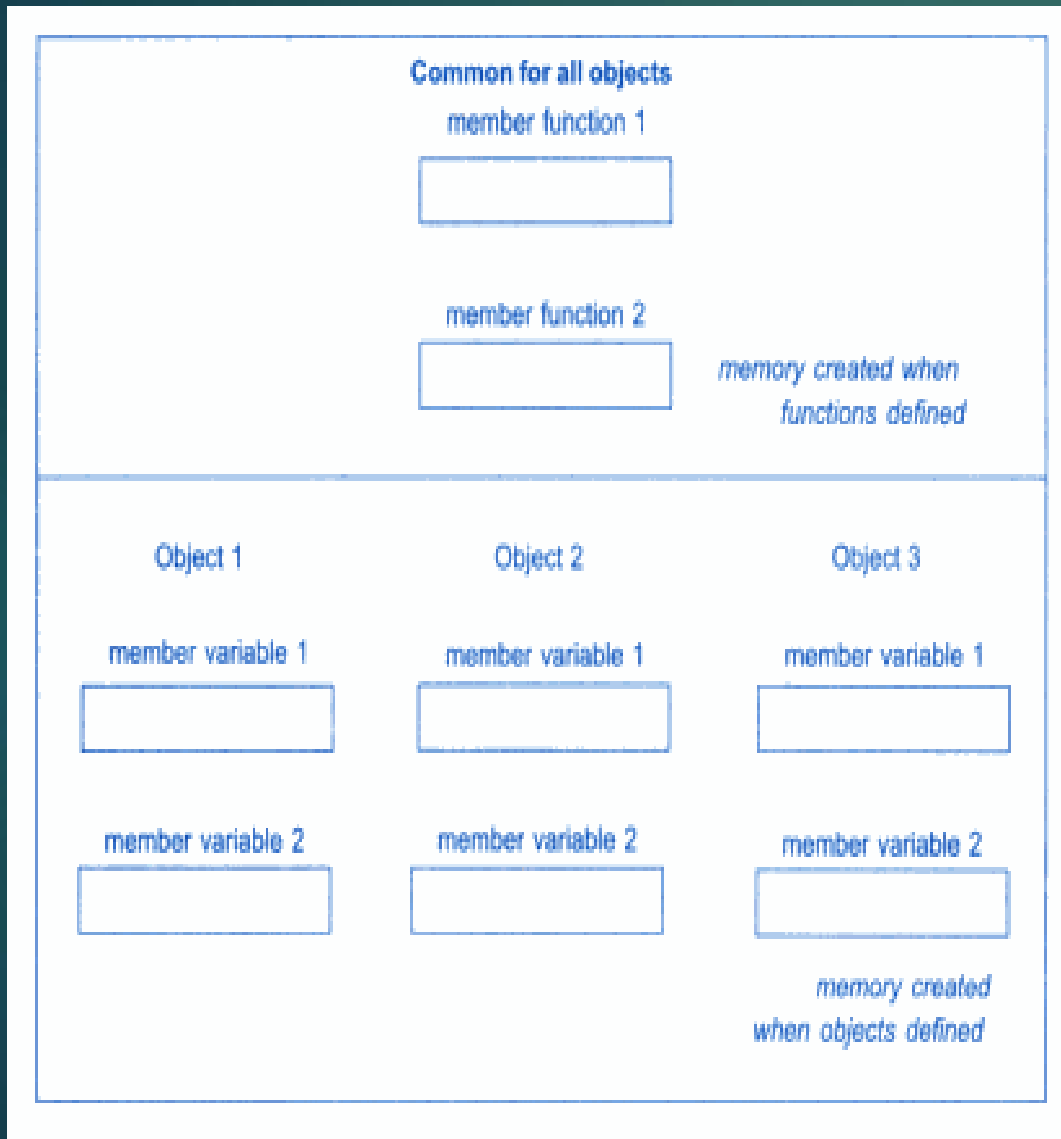
The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;    // provides value for array size

class array
{
    int a[size];      // 'a' is int type array
public:
    void setval(void);
    void display(void);
};
```

The array variable **a[]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Memory Allocation for Objects



Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 5.4 illustrates the use of a static data member.

STATIC CLASS MEMBER

```
#include <iostream>
using namespace std;
class item
{
    static int count;
    int number;
public:
    void getdata(int a)
    {
        number = a;
        count ++;
    }
    void getcount(void)
    {
        cout << "count: ";
        cout << count << "\n";
    }
};
int item :: count;

int main()
{
    item a, b, c;           // count is initialized to zero
    a.getcount();          // display count
    b.getcount();
    c.getcount();

    a.getdata(100);        // getting data into object a
    b.getdata(200);        // getting data into object b
    c.getdata(300);        // getting data into object c

    cout << "After reading data" << "\n";

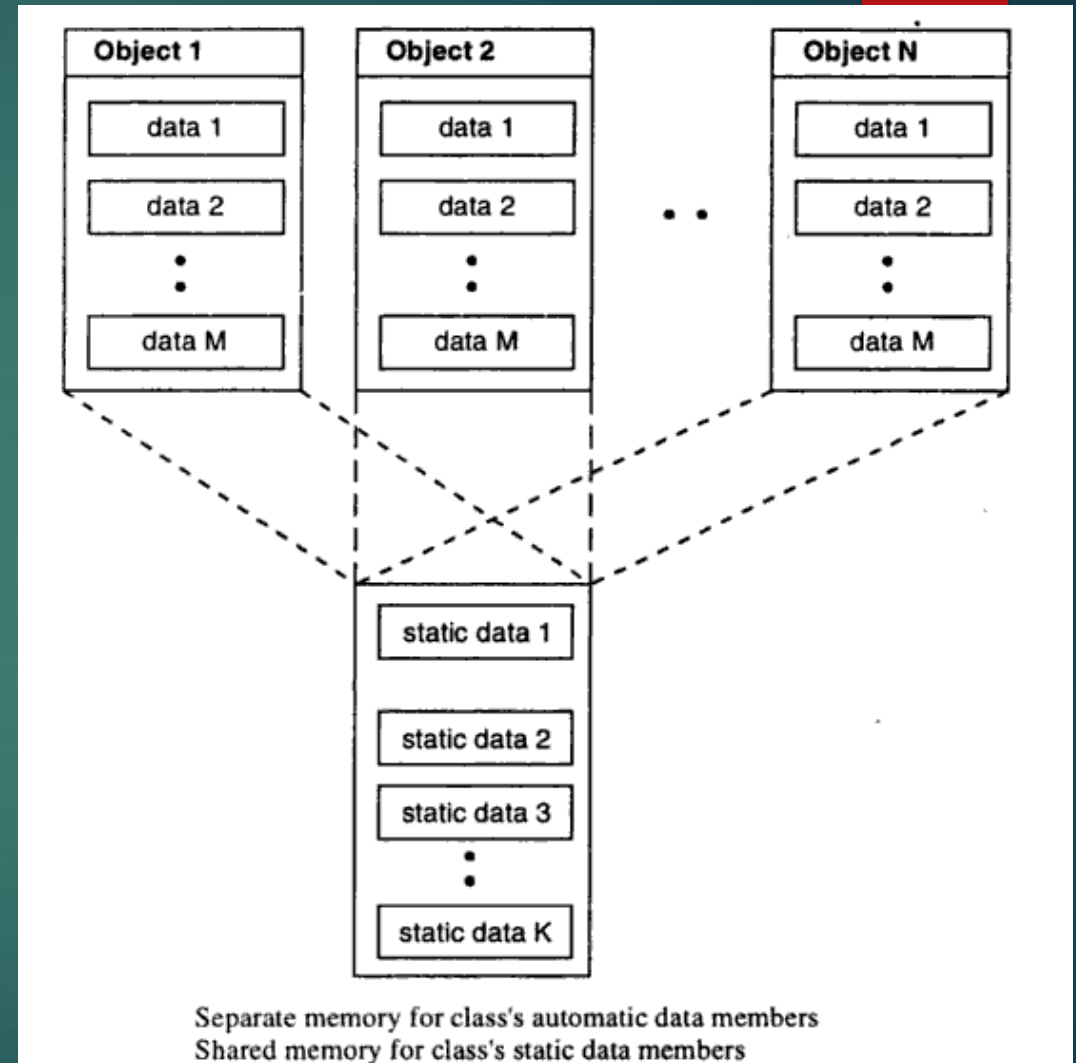
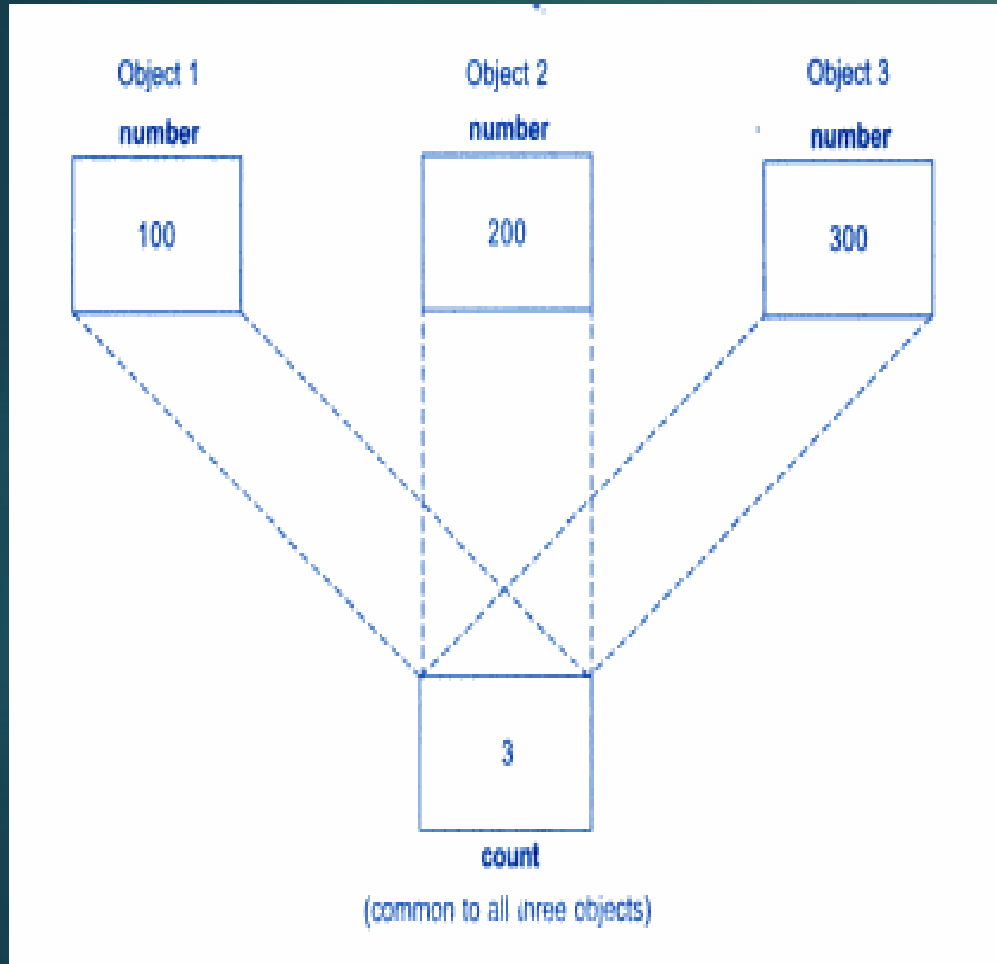
    a.getcount();          // display count
    b.getcount();
    c.getcount();
    return 0;
}
```

Output of Program

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

```
int item :: count; // definition of static data member
```

Memory allocation for static member



Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

...

STATIC MEMBER FUNCTION

```
#include <iostream>

using namespace std;

class test
{
    int code;
    static int count;           // static member variable
public:
    void setcode(void)
    {
        code = ++count;
    }
    void showcode(void)
    {
        cout << "object number: " << code << "\n";
    }
    static void showcount(void) // static member function
    {
        cout << "count: " << count << "\n";
    }
};

int test :: count;
int main()
{
    test t1, t2;

    t1.setcode();
    t2.setcode();

    test :: showcount();      // accessing static function

    test t3;
    t3.setcode();

    test :: showcount();

    t1.showcode();
    t2.showcode();
    t3.showcode();

    return 0;
}
```

Output of Program

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

Remember, the following function definition will not work:

```
static void showcount()
{
    cout << code;    // code is not static
}
```

Array of objects

```
#include <iostream>

using namespace std;

class employee
{
    char name[30];    // string as class member
    float age;
public:
    void getdata(void);
    void putdata(void);
};

void employee :: getdata(void)
{
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter age: ";
    cin >> age;
}

void employee :: putdata(void)
{
    cout << "Name: " << name << "\n";
    cout << "Age: " << age << "\n";
}

const int size=3;
int main()
{
    employee manager[size];
    for(int i=0; i<size; i++)
    {
        cout << "\nDetails of manager" << i+1 << "\n";
        manager[i].getdata();
    }
    cout << "\n";
    for(i=0; i<size; i++)
    {
        cout << "\nManager" << i+1 << "\n";
        manager[i].putdata();
    }
    return 0;
}
```

Output of Program

Interactive input

Details of manager1

Enter name: xxx

Enter age: 45

Details of manager2

Enter name: yyy

Enter age: 37

Details of manager3

Enter name: zzz

Enter age: 50

Program output

Manager1

Name: xxx

Age: 45

Manager2

Name: yyy

Age: 37

Manager3

Name: zzz

Age: 50

Objects as Function Arguments

It is possible to have functions which accept objects of a class as arguments, just as there are functions which accept other variables as arguments. Like any other data type, an object can be passed as an argument to a function by the following ways:

- pass-by-value, a copy of the entire object is passed to the function
- pass-by-reference, only the address of the object is passed implicitly to the function
- pass-by-pointer, the address of the object is passed explicitly to the function

In the case of *pass-by-value*, a copy of the object is passed to the function and any modifications made to the object inside the function is not reflected in the object used to call the function. Whereas, in *pass-by-reference* or *pointer*, an address of the object is passed to the function and any changes made to the object inside the function is reflected in the actual object. The parameter passing by reference or pointer is more efficient since, only the address of the object is passed and not a copy of the entire object.

Objects as Function Arguments

Output of Program

T1 = 2 hours and 45 minutes

T2 = 3 hours and 30 minutes

T3 = 6 hours and 15 minutes

```
#include <iostream>

using namespace std;

class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    { hours = h; minutes = m; }
    void puttime(void)
    {
        cout << hours << " hours and ";
        cout << minutes << " minutes " << "\n";
    }
    void sum(time, time); // declaration with objects as arguments
};

void time :: sum(time t1, time t2) // t1, t2 are objects
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = minutes%60;
    hours = hours + t1.hours + t2.hours;
}

int main()
{
    time T1, T2, T3;

    T1.gettime(2,45); // get T1
    T2.gettime(3,30); // get T2

    T3.sum(T1,T2); // T3=T1+T2

    cout << "T1 = "; T1.puttime(); // display T1
    cout << "T2 = "; T2.puttime(); // display T2
    cout << "T3 = "; T3.puttime(); // display T3

    return 0;
}
```

note

Since the member function `sum()` is invoked by the object `T3`, with the objects `T1` and `T2` as arguments, it can directly access the `hours` and `minutes` variables of `T3`. But, the members of `T1` and `T2` can be accessed only by using the dot operator (like `T1.hours` and `T1.minutes`). Therefore, inside the function `sum()`, the variables `hours` and `minutes` refer to `T3`, `T1.hours` and `T1.minutes` refer to `T1`, and `T2.hours` and `T2.minutes` refer to `T2`.

Figure 5.6 illustrates how the members are accessed inside the function `sum()`.

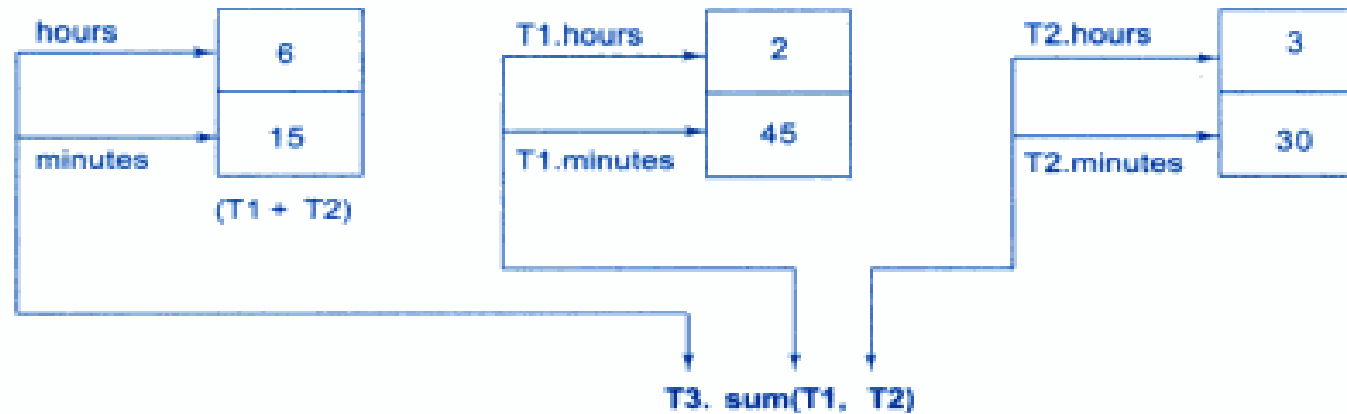


Fig. 5.6 ⇔ *Accessing members of objects within a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

Passing Objects by Value

The program `distance.cpp` illustrates the use of objects as function arguments in *pass-by-value* mechanism. It performs the addition of distance in feet and inches format.

```
// distance.cpp: distance manipulation in feet and inches
#include <iostream.h>
class distance
{
    private:
        float feet;
        float inches;
    public:
        void init( float ft, float in )
        {
            feet = ft;
            inches = in;
        }
        void read()
        {
            cout << "Enter feet: ";    cin >> feet;
            cout << "Enter inches: ";  cin >> inches;
        }
}
```

```

void show()
{
    cout << feet << "-" << inches << "\n";
}
void add( distance d1, distance d2 )
{
    feet = d1.feet + d2.feet;
    inches = d1.inches + d2.inches;
    if( inches >= 12.0 )
    {
        // 1 foot = 12.0 inches
        feet = feet + 1.0;
        inches = inches - 12.0;
    }
}
};
void main()
{
    distance d1, d2, d3;
    d2.init( 11.0, 6.25 );
    d1.read();
    cout << "d1 = ";    d1.show();
    cout << "\nd2 = ";    d2.show();
    d3.add( d1, d2 ); // d3 = d1 + d2
    cout << "\nd3 = d1+d2 = ";    d3.show();
}

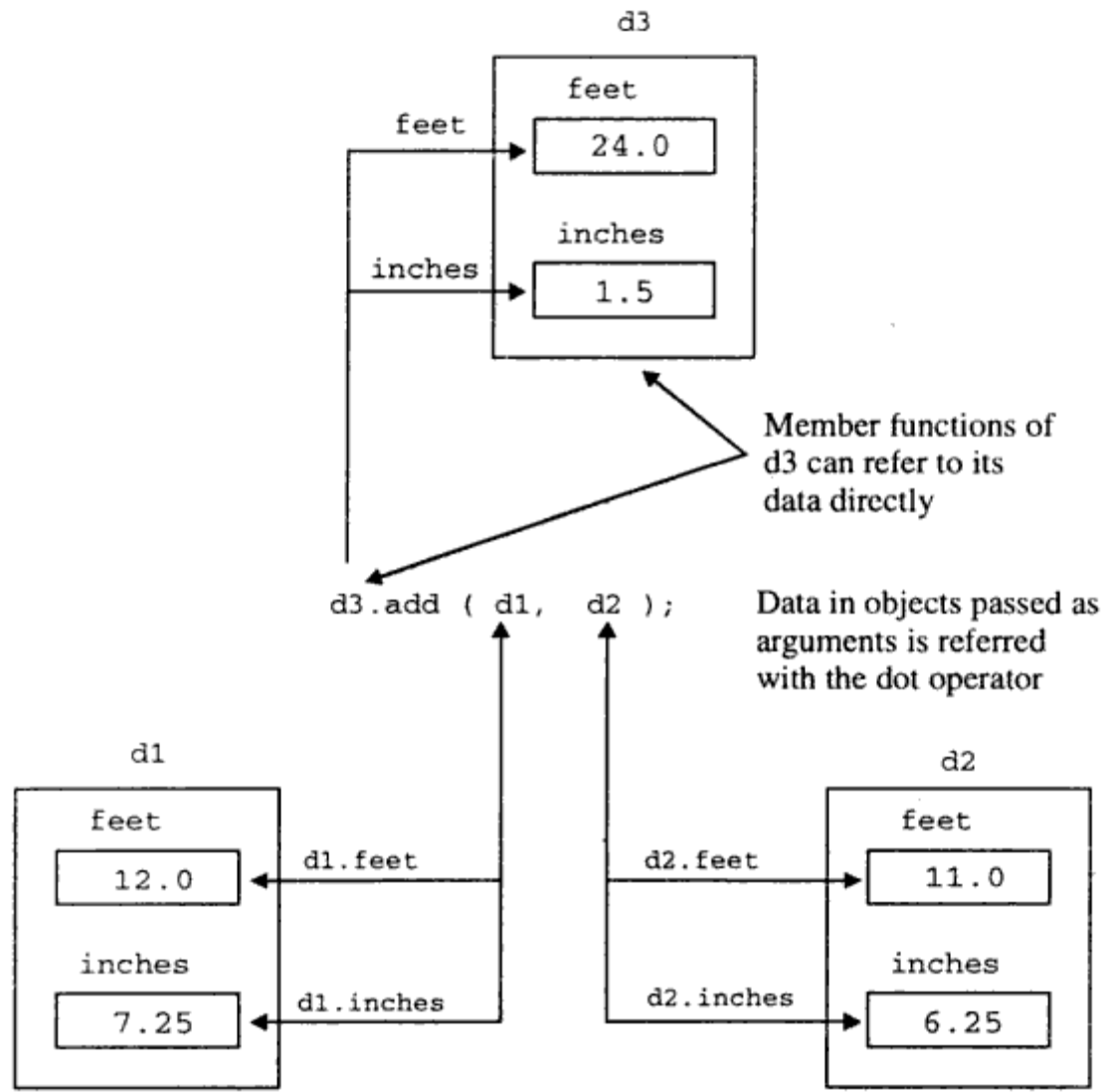
```

Run

```

Enter feet: 12.0
Enter inches: 7.25
d1 = 12'-7.25"
d2 = 11'-6.25"
d3 = d1+ d2 = 24'-1.5"

```



Objects of the distance class as parameters

Passing Objects by Reference

Accessibility of the objects passed by reference is similar to those passed by value. Modifications carried out on such objects in the called function will also be reflected in the calling function. The method of passing objects as reference parameters to a function is illustrated in the program `account.cpp`. Given the account numbers and the balance of two accounts, this program transfers a specified sum from one of these accounts to the other and then, updates the balance in both the accounts.

```
// account.cpp: passing objects as parameters to functions
#include<iostream.h>
class AccClass
{
private:           // class data members
    int accno;
    float balance;
public:           // class function members
    void getdata()
    {
        cout << "Enter the account number for accl object: ";
        cin >> accno;
        cout << "Enter the balance: ";
        cin >> balance;
    }
}
```

```
void setdata( int accIn )
{
    accno = accIn;
    balance = 0;
}
void setdata( int accIn, float balanceIn )
{
    accno = accIn;
    balance = balanceIn;
}
void display()
{
    cout << "Account number is: " << accno << endl;
    cout << "Balance is: " << balance << endl;
}
void MoneyTransfer( AccClass & acc, float amount );
};
// acc1.MoneyTransfer( acc2, 100 ), transfers 100 rupees from acc1 to acc2
void AccClass::MoneyTransfer( AccClass & acc, float amount )
{
    balance = balance - amount; // deduct money from source
    acc.balance = acc.balance + amount; // add money to destination
};
```



```
void main()
{
    int trans_money;
    AccClass acc1, acc2, acc3;
    acc1.getdata();
    acc2.setdata( 10 );
    acc3.setdata( 20, 750.5 );
    cout << "Account Information..." << endl;
    acc1.display();
    acc2.display();
    acc3.display();
    cout << "How much money is to be transferred from acc3 to acc1: ";
    cin >> trans_money;
    acc3.MoneyTransfer(acc1,trans_money); //transfers money from acc3 to acc1
    cout << "Updated Information about accounts..." << endl;
    acc1.display();
    acc2.display();
    acc3.display();
}
```

Run

Enter the account number for acc1 object: 1

Enter the balance: 100

Account Information...

Account number is: 1

Balance is: 100

Account number is: 10

Balance is: 0

```
Account number is: 20
Balance is: 750.5
How much money is to be transferred from acc3 to acc1: 200
Updated Information about accounts...
Account number is: 1
Balance is: 300
Account number is: 10
Balance is: 0
Account number is: 20
Balance is: 550.5
```

In `main()`, the statement

```
acc3.MoneyTransfer( acc1, trans_money );
```

transfers the object `acc1` by reference to the member function `MoneyTransfer()`. It is to be noted that when the `MoneyTransfer()` is invoked with `acc1` as the object parameter, the data members of `acc3` are accessed without the use of the class member access operator, while the data members of `acc1` are accessed by using their names in association with the name of the object to which they belong. An object can also be passed to a non-member function of the class and that can have access to the public members only through the objects passed as arguments to it.

Passing Objects by Pointer

The members of objects passed by pointer are accessed by using the `->` operator, and they have similar effect as those passed by value. The above program requires the following changes if parameters are to be passed by pointer:

1. The prototype of the member function `MoneyTransfer()` has to be changed to:

```
void MoneyTransfer( AccClass * acc, float amount );
```

2. The definition of the member function `MoneyTransfer()` has to be changed to:

```
void AccClass::MoneyTransfer( AccClass & acc, float amount )  
{  
    balance = balance - amount;    // deduct money from source  
    acc->balance = acc->balance + amount; // add money to destination  
}
```

3. The statement invoking the member function `MoneyTransfer()` has to be changed to:

```
acc3.MoneyTransfer( &acc1, trans_money );
```

10.13 Returning Objects from Functions

Similar to sending objects as parameters to functions, it is also possible to return objects from functions. The syntax used is similar to that of returning variables from functions. The return type of the function is declared as the return object type. It is illustrated in the program `complex.cpp`.

```
// complex.cpp: Addition of Complex Numbers, class complex as data type
#include <iostream.h>
#include <math.h>

class complex
{
private:
    float real;    // real part of complex number
    float imag;   // imaginary part of complex number
```

```

public:
    void getdata()
    {
        cout << "Real Part ? ";
        cin >> real;
        cout << "Imag Part ? ";
        cin >> imag;
    }
    void outdata( char *msg )    // display number in x+iy form
    {
        cout << msg << real;
        if( imag < 0 )
            cout << "-i";
        else
            cout << "+i";
        cout << fabs(imag) << endl;
    }
    complex add( complex c2 );    // addition of complex numbers
};
complex complex::add( complex c2 ) // add default and c2 objects
{
    complex temp;                // object temp of complex class
    temp.real = real + c2.real;  // add real parts
    temp.imag = imag + c2.imag;  // add imaginary parts
    return( temp );             // return complex object
}

```

```

void main()
{
    complex c1, c2, c3;      // c1, c2, and c2 are objects of complex
    cout << "Enter Complex Number c1 .." << endl;
    c1.getdata();
    cout << "Enter Complex Number c2 .." << endl;
    c2.getdata();
    c3 = c1.add( c2 );      // add c1 and c2 assign to c3
    c3.outdata( "c3 = c1.add( c2 ): ");
}

```

Run

```

Enter Complex Number c1 ..
Real Part ? 1.5
Imag Part ? 2
Enter Complex Number c2 ..
Real Part ? 3
Imag Part ? -4.3
c3 = c1.add( c2 ): 4.5-i2.3

```

In main(), the statement

```

    c3 = c1.add( c2 );      // add c1 and c2 assign to c3

```

invokes the function add() of the class complex by passing the object c2 as a parameter. The statement in this function,

```

    return( temp );      // return complex object

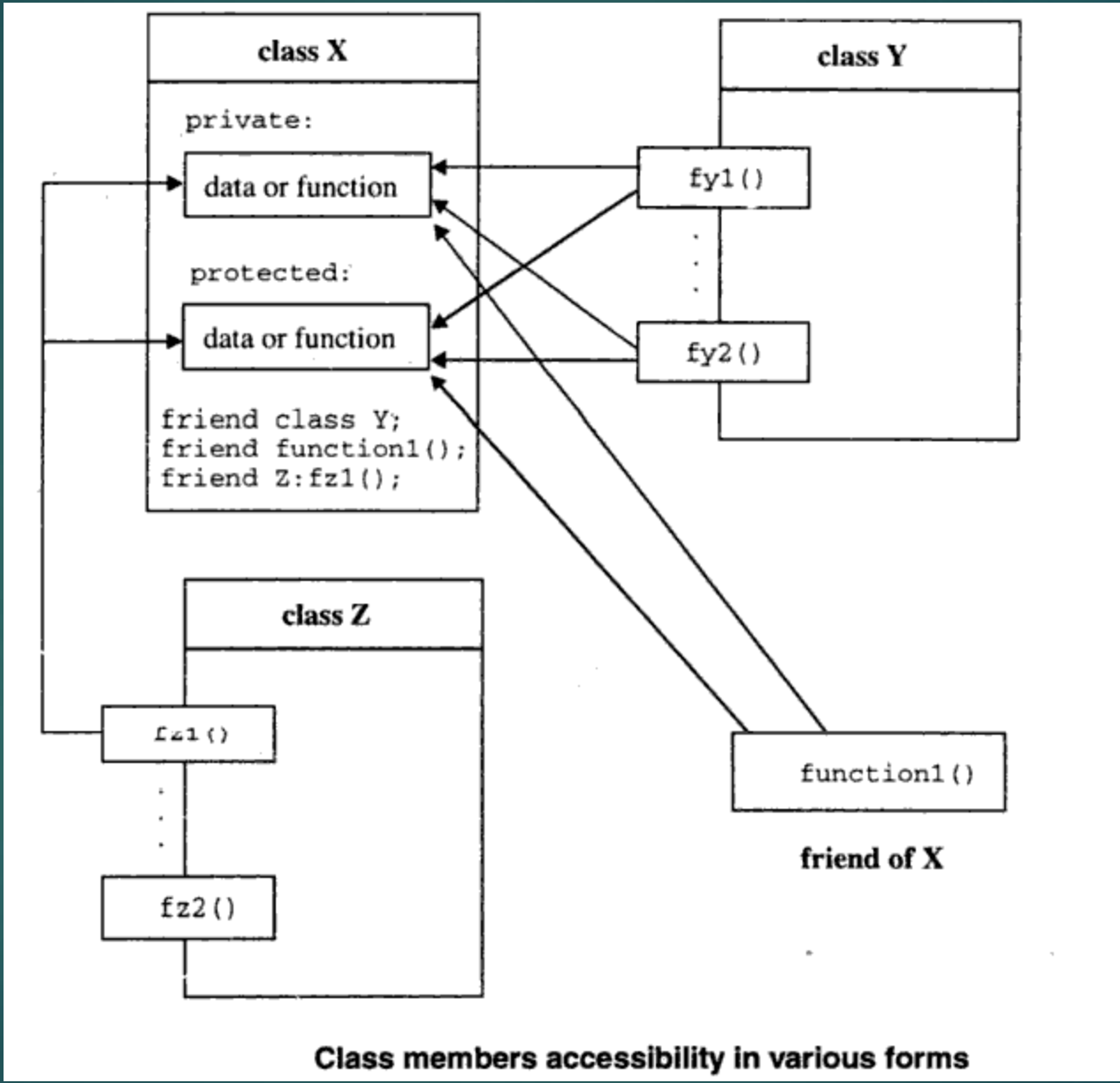
```

returns the object temp as a return object.

Friendly Functions

The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members. The policy is, *if you are not a member you cannot get it*. Sometimes this feature leads to considerable inconvenience in programming. Imagine that the user wants a function to operate on objects of two different classes. At such times, it is required to allow functions outside a class to access and manipulate the private members of the class. In C++, this is achieved by using the concept of *friends*.

One of the convenient and a controversial feature of C++ is allowing non-member functions to access even the private members of a class using friend functions or friend classes. It permits a function or all the functions of another class to access a different class's private members.



Class members accessibility in various forms

To make an outside function “friendly” to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
    .....
    .....
    public:
        .....
        .....
        friend void xyz(void); // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator `::`. The functions that are declared with the keyword **friend** are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
#include <iostream>

using namespace std;

class sample
{
    int a;
    int b;
public:
    void setvalue() {a=25; b=40; }
    friend float mean(sample s);
};

float mean(sample s)
{
    return float(s.a + s.b)/2.0;
}

int main()
{
    sample X; // object X
    X.setvalue();
    cout << "Mean value = " << mean(X) << "\n";

    return 0;
}
```

note

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(X)** passes the object **X** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
    .....
    .....
    int fun1();      // member function of X
    .....
};

class Y
{
    .....
    .....
    friend int X :: fun1();      // fun1() of X
    *                          // is friend of Y
    .....
};
```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
    .....
    friend class X;      // all member functions of X are
                       // friends to Z
};
```

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>

using namespace std;

class ABC;    // Forward declaration
//-----//
class XYZ
{
    int x;
public:
    void setvalue(int i) {x = i;}
    friend void max(XYZ, ABC);
};
//-----//
class ABC
{
    int a;
};
//-----//
void max(XYZ m, ABC n)    // Definition of friend
{
    if(m.x >= n.a)
        cout << m.x;
    else
        cout << n.a;
}
//-----//
int main()
{
    ABC abc;
    abc.setvalue(10);
    XYZ xyz;
    xyz.setvalue(20);
    max(xyz, abc);

    return 0;
}
```

note

The function `max()` has arguments from both **XYZ** and **ABC**. When the function `max()` is declared as a friend in **XYZ** for the first time, the compiler will not acknowledge the presence of **ABC** unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

Returning Objects

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

class complex // x + iy form
{
    float x; // real part
    float y; // imaginary part
public:
    void input(float real, float imag)
    { x = real; y = imag; }

    friend complex sum(complex, complex);
    void show(complex);
};

complex sum(complex c1, complex c2)
{
    complex c3; // objects c3 is created
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3); // returns object c3
}

void complex :: show(complex c)
{
    cout << c.x << " + j" << c.y << "\n";
}

int main()
{
    complex A, B, C;

    A.input(3.1, 5.65);
    B.input(2.75, 1.2);

    C = sum(A, B); // C = A + B

    cout << "A = "; A.show(A);
    cout << "B = "; B.show(B);
    cout << "C = "; C.show(C);

    return 0;
}
```

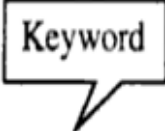
Output of Program

```
A = 3.1 + j5.65
B = 2.75 + j1.2
C = 5.85 + j6.85
```

const Member Functions

Certain member functions of a class, access the class data members without modifying them. It is advisable to declare such functions as `const` (constant) functions. The syntax for declaring `const` member functions is shown :

A `const` member function is used to indicate that it does not alter the data fields of the object, but only inspects them.



Keyword

```
ReturnType FunctionName(arguments) const
```

Syntax of declaring a constant member function

If a member function does not alter any data in the class, then we may declare it as a `const` member function as follows:

```
void mul(int, int) const;  
double get_balance() const;
```

The qualifier `const` is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator `&` to a “fully qualified” class member name. A class member pointer can be declared using the operator `::*` with the class name. For example, given the class

```
class A
{
    private:
        int m;
    public:
        void show();
};
```

We can define a pointer to the member `m` as follows:

```
int A::* ip = &A :: m;
```

The `ip` pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase `A::*` means “pointer-to-member of `A` class”. The phrase `&A::m` means the “address of the `m` member of `A` class”.

Remember, the following statement is not valid:

```
int *ip = &m;    // won't work
```

This is because `m` is not simply an `int` type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;    // display
cout << a.m;      // same as above
```

Now, look at the following code:

```
ap = &a;          // ap is pointer to object a
cout << ap -> *ip; // display m
cout << ap -> m;  // same as above
```

The *dereferencing operator* `->*` is used to access a member when we use pointers to both the object and the member. The *dereferencing operator* `*` is used when the object itself is used with the member pointer. Note that `*ip` is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of `()` is higher than that of `.*` and `->*`, so the parentheses are necessary.

```
#include <iostream>

using namespace std;

class M
{
    int x;
    int y;
public:
    void set_xy(int a, int b)
    {
        x = a;
        y = b;
    }
    friend int sum(M m);
};

int sum(M m)
{
    int M::* px = &M::x;
    int M::* py = &M::y;
    M *pm = &m;
    int S = m.*px + pm->*py;
    return S;
}

int main()
{
    M n;
    void (M::* pf)(int,int) = &M::set_xy;
    (n.*pf)(10,20);
    cout << "SUM = " << sum(n) << "\n";

    M *op = &n;
    (op->*pf)(30,40);
    cout << "SUM = " << sum(n) << "\n";

    return 0;
}
```

Output of Program

```
sum = 30
sum = 70
```

Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test(int a)      // function
{
    .....
    .....
    class student    // local class
    {
        .....
    };
    .....
    .....
    student s1(a);   // create student object
    .....           // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

REFERENCES:

- 1.E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.
2. LECTURE NOTES ON Object Oriented Programming Using C++ by Dr. Subasish Mohapatra, Department of Computer Science and Application College of Engineering and Technology, Bhubaneswar Biju Patnaik University of Technology, Odisha
3. K.R. Venugopal, Rajkumar, T. Ravishankar, “Mastering C++”, Tata McGraw-Hill Publishing Company Limited
4. Object Oriented Programming With C++ - PowerPoint Presentation by Alok Kumar
5. OOPs Programming Paradigm – PowerPoint Presentation by an Anonymous Author