Pointers and Memory Management in a C++

Pointer Variables and the Address of a Variable

- In C++, **you can access a variable (memory location) by using its address** instead of its name.
- In order to do this, you must first define a variable called **pointer variable** to hold that address.
- **You define a pointer variable** as follows:

<data-type> * *<pointer-variable>* ;

Example P1

a)

```
char  * cpt;      /* cpt is a pointer variable to hold the address of a character (char) variable */
int  * ipt;       /* ipt is a pointer variable to hold the address of an integer (int)  variable */
double  * dpt;    /* dpt is a pointer variable to hold the address of a double precision floating point
                     (double) variable */
```

b) You can also define pointer variables with other variables in the same declaration statement as in the following example:

```
int num = 10, * ipt1, result;
int  *ipt1,  num = 10,  *ipt2,  result;
```

Note

C++ distinguishes pointer variables defined to hold the addresses of variables (memory locations) of different data types. For example, a pointer variable that is defined to hold the address of an integer variable cannot hold the address of a double precision variable.

Storing an Address into a Pointer Variable

- You **access the address of a variable** by using the **address-of operator &**.
- For example, if *num* is a variable, then its address (the address of the corresponding memory location) is accessed by the expression: **&num**.

Example P2

Assume given the following definitions of variables with their memory representations:

```
int  *iptr;
int  ivar = 10, *iptrl;
double dvar = 7.5, *dptr;
char cvar = 'Z', *cptr;
```

Memory Locations

Address			Address		
103	iptr		135	ivar	10
206	iptr1		231	dvar	7.5
275	dptr		283	cvar	Z
298	cptr				

The following assignment statements are valid and have the specified effects on the memory locations:

```
iptr = &ivar;
dptr = &dvar;
cptr = &cvar;
iptr1 = iptr;
```

Memory Locations

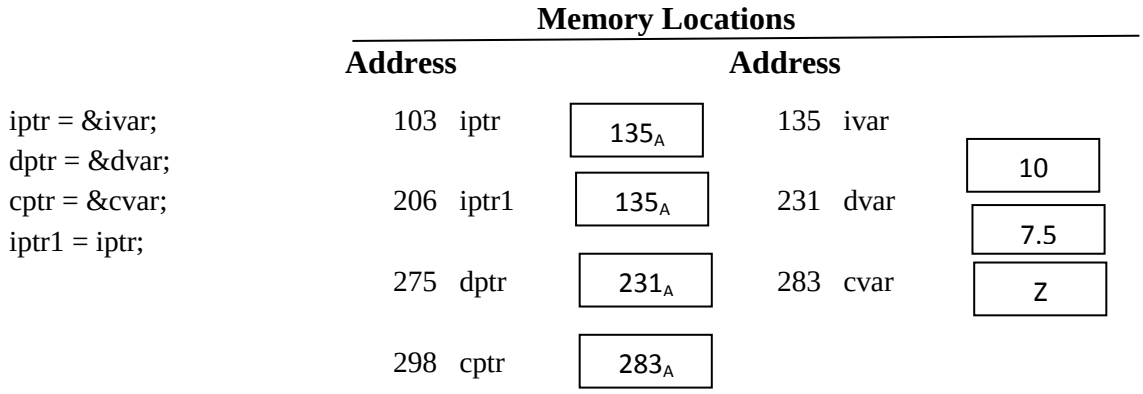
Address			Address		
103	iptr	135_A	135	ivar	10
206	iptr1	135_A	231	dvar	7.5
275	dptr	231_A	283	cvar	Z
298	cptr	283_A			

Using a Pointer Variable to Access a Variable (memory location)

- You use a pointer variable that holds the address of a variable (memory location) to access that variable (memory location) by preceding the pointer variable with the **indirection** or **deference operator** (*).

Example P3

Given the definitions of variables in example P2 and the following assignment statements:



The following statements will produce the specified output:

	Statements	**Output**
a)	cout << endl << "*iptr + 5 = \t" << (*iptr + 5);	*iptr + 5 = 15
b)	cout << endl << "*cptr = \t" << *cptr;	*cptr = Z
c)	cout << endl << "*dptr * 3 = \t" << (*dptr * 3);	*dptr * 3 =22.5

- The expression (*iptr + 5) is read: "get the value in the memory location (of the variable) at the address in pointer variable iptr, and add 5 to it."
- The expression *cptr is read: "get the value in the memory location (of the variable) at the address in pointer variable cptr."
- The expression (*dptr * 3) is read: "get the value in the memory location (of the variable) at the address in pointer variable dptr and multiply it by 3."

The following examples show how a value is stored into a memory location with its address in a pointer variable:

```
* cptr = 'X';
* dptr = dvar + 3;
* iptr = * iptr + 8;
cout << "\ncvar =\t" << cvar << "\nivar =\t" << ivar;
cout << endl << "dvar =\t" << dvar;
```

The corresponding output follows:

Output

```
cvar =    X
ivar =    18
dvar =    10.5
```

- The assignment statement * cptr = 'X'; says to store character 'X' into the memory location at the address in pointer variable *cptr*;
- the assignment statement * dptr = dvar + 3; says to add the current value of variable *dvar* to 3 and to store the result into the memory location at the address in pointer variable *dptr*;
- The assignment statement * iptr = * iptr + 8; says to get the current value into the memory location at the address in pointer variable *iptr*, add 8 to it, and to store the result into the memory location at the address in pointer variable *iptr*.

Exercise P1*

Show the output of the following code segment:

```
int *pt1, *pt2, num1= 10, num2 = 5, num3;
pt1 = &num1;
*pt1 = 20;
pt2 = &num2;
num3 = *pt2 + 7;
num2 += 4;
cout << endl << "num1=" << num1 << endl << "num2=" << num2 << endl << "num3="
     << num3 << endl << "*pt1=" << *pt1 << endl << "*pt2=" << *pt2;
```

Exercise P2

Show the output of each of the following code segments:

a.
```
int  num1 = 20, *numptr1, *numptr2, num2 = 0;
numptr1 = &num1;
numptr2 = &num2;
*numptr2 = 5;
cout << "\n num1 = " << num1 << "\n*numptr1 = " << *numptr1;
cout << "\n num2 = " << num2 << "\n*numptr2 = " << *numptr2;
cout << "\n*numptr1 + 8=\t" << (*numptr1 + 8);
```

b.
```
int  num1 = 20, *numptr1, *numptr2, num2 = 0;
numptr1 = &num1;
numptr2 = numptr1;
*numptr2 = 27;
cout << "\n num1 = " << num1 << "\n*numptr1 = " << *numptr1;
cout << "\n num2 = " << num2 << "\n*numptr2 = " << *numptr2;
```

c.
```
int  num1 = 20, *numptr1, *numptr2, num2 = 0;
numptr1 = &num1;
numptr2 = &num2;
num1 = 12;
*numptr2 = *numptr1 + 6;
cout << "\n num1 = " << num1 << "\n*numptr1 = " << *numptr1;
cout << "\n num2 = " << num2 << "\n*numptr2 = " << *numptr2;
```

Pointer Parameters

- A C++function can use a **pointer parameter** to access a local variable of the calling function in a way similar to the way it does it using a reference parameter.
- A function's **pointer parameter** is a value parameter that must be initialized with the address of a variable (memory location) when that function is called.
- The argument of a function that corresponds to a pointer parameter must be the address of a variable as follows:

Pointer Parameter	**Address of:**
Character pointer	character variables
Integer pointer	integer variables
Single precision floating point pointer	single precision floating point variables
Double precision floating point pointer	double precision floating point variables

Example P4

Assume given the following definitions of functions *tester( )* and *main( )*:

```
void tester ( int num, int * pt)
{
    * pt = * pt + num;
}

int main()
{
    int tvalue = 5;
    tester ( 7,  & tvalue);
    cout  <<  endl  <<  "tvalue = \t"  <<  tvalue;
    return 0;
}
```

After the call statement: **tester (7, & tvalue);**

The body of function tester is executed as if it was written as follows:

```
{
    num  =  7;
    pt  =  &tvalue;
    * pt = * pt + num;
}
```

So the output of the program is: **Output** tvalue = 12

Note

Using pointer parameters is similar to using reference parameters, except that with a pointer parameter, a value (which is an address) is passed to the function, whereas with reference parameters, no value is actually passed to the function.

Exercise P3*

Show the body of the function *tester* in the way it is executed after the function call. Then execute the program and show its output.

```
void tester ( int num, int * pt)
{
    * pt = * pt - num;
}
int main()
{
    int tvalue = 20;
    tester ( 4,  & tvalue);
    cout  <<  endl  <<  "tvalue = \t"  <<  tvalue;
    return 0;
}
```

Exercise P4

Show the body of the function *tester( )* in the way it is executed after the function call. Then execute the program and show its output.

```
void tester(int num, int *p1, int *p2)
{
    num  = num + 5;
    *p1 = *p1 + 5;
    *p2  = num ;
}
int  main()
{
    int i = 10,  j,  k, *ptr;
    j = 15;
    ptr = &j;
    tester(i, ptr, &k);
    cout  <<  "\ni =\t"  << i  <<  "\nj =\t"  <<  j
        <<  "\nk =\t"  <<  k;
    return(0);
}
```

Structures and Pointer Variables

- You can define a pointer variable to hold the address of a structure variable.
- A pointer variable that holds the address of a structure variable is used to access its member variable(s) by using the **arrow operator** (->).

Example P5

Given the following structure *Date*:

```
struct   Date
{
    int  month;
    int  day;
    int year;
};
```

And the following declaration of structure variables *birthDate* and *someDay*, and the pointer variable *dayPtr* :

```
Date  birthDate = {10, 21, 1989},
        * dayPrt,
        someDay;
```

The pointer variable *dayPtr* is used to access the member variables of the structure variable *birthDay* and *someDay* as follows:

```
dayPtr  = &birthDate;
cout  <<  endl  <<  "Your birthday is as follows:\t"    <<  "Month is:\t"  <<  dayPtr -> month
          <<  "Day is:\t"  <<  dayPtr -> day  <<  "Year is:\t"  <<  dayPtr -> year;

dayPtr = &someDay;
dayPtr -> year = 2011;
cin  >>  dayPtr -> day  >> dayPtr -> month;
   (dayPtr -> day) ++ ;
```

- **A function's parameter can be a pointer to a structure**.

Example P6

The following function *readDate( )* receives the address of a structure variable *Date* and reads values for its member variables *month*, *day*, and *year*.

```
void  readDate ( Date * datePtr )
{
    cin  >>  datePtr  ->  month   >>  datePtr -> day  >>  datePtr -> year;
}
```

This function is called to read the values for the member variables of structure variable *someDay* as follows:

```
Date  someDay;
readDate ( &someDay );
```

Note that a reference parameter instead of a pointer parameter could have been used in this function.

Classes and Pointer Variables

- **A pointer variable can be defined to hold the address of an object** in the same way that it is defined to hold the address of a structure variable.
- A pointer variable that holds the address of an object can be used to access its member variables and functions (by using the **arrow operator** (->)) in the same way that a pointer variable is used to access the members of a structure.

Example P7

Assume given the following class definition:

```
class  DemoP
{
    public:
        DemoP( int num1= 0, int num2 = 0);   // constructor with default arguments
        void readValues(void);               // to read the values of the member variables
        double  getValue1(void);             // to return the value of the first member variable
        double  getValue2(void);             // to return the value of the second member variable
        double  getAverage( );               // to compute the average of both values
    private:
        double computeSum( );                // to compute the sum of both values
        double val1;                         // the first member variable
        double val2;                         // the second member variable
};
```

1. Define the object *obj* and the pointer variable *oPtr* of the class *DemoP*.
2. Write the sequence of statements to do the following:
   - Assign the address of object *obj* to the pointer variable *oPtr*.
   - Use the pointer variable *oPtr* to read values into the member variables of object *obj*.
   - Use the pointer variable *oPtr* to compute the average of the values of the member variables of object *obj*.

Exercise P5*

Using the class *DemoP* provided in example P7, do the following:

1. Write the function *void add3P(DemoP *ptr)* that receives as argument the address of an object of the class *DemoP* and adds 3 to the value of each of its member variables.
2. Define the objects *obj1* and *obj2* and the pointer variable *oPtr* of the class *DemoP*. The member variables of object *obj1* are initialized with the values 10 and 15 respectively.
3. Using the pointer variable *oPtr*, write the statements to read values into the member variables of object *obj2* and then compute and print their average.
4. Write the statements to add 3 to the value of each member variable of object *obj1* (by calling function *add3P( )*) and then print their new values.

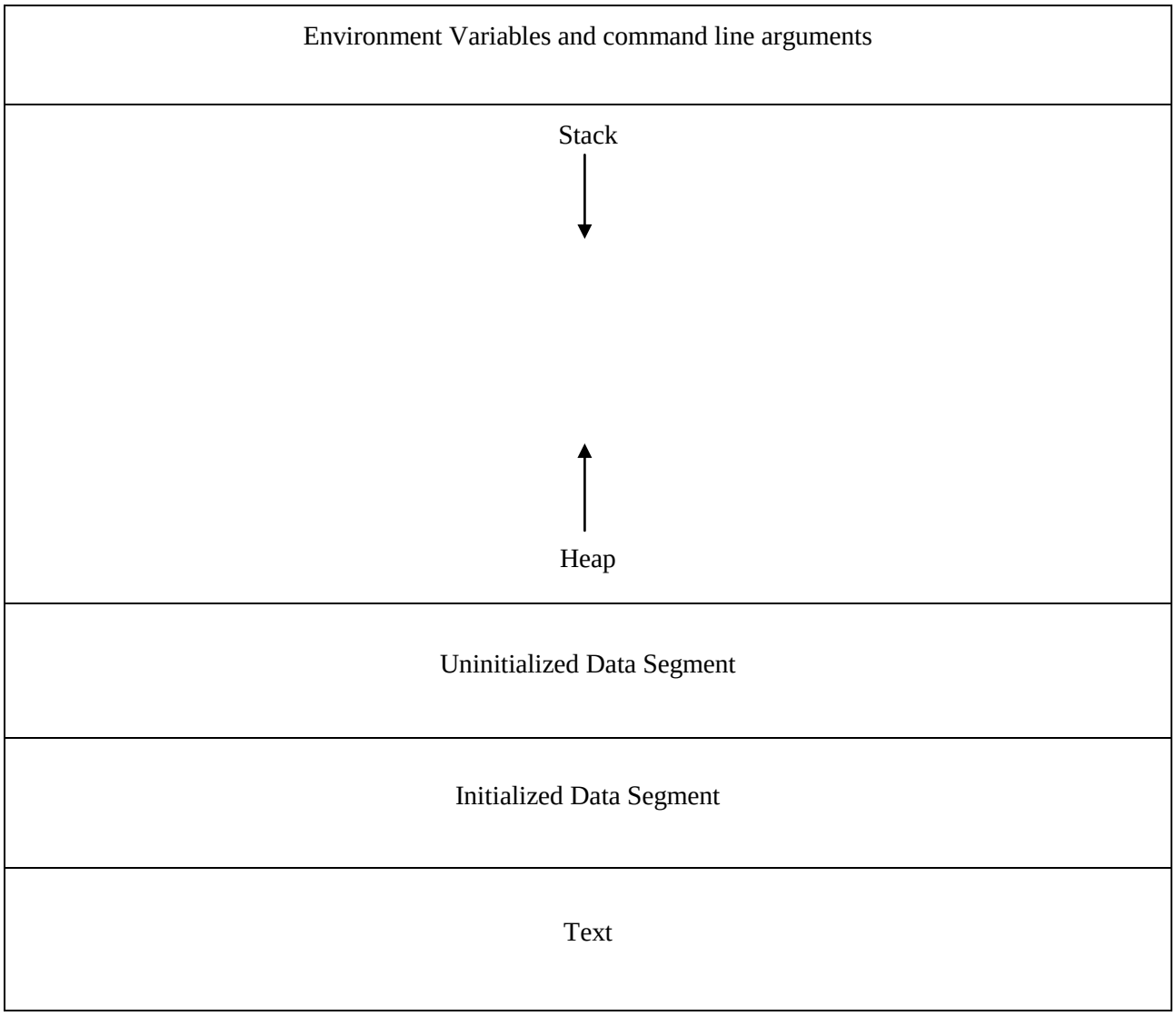
Exercise P6

Using the class *DemoP* provided in example P7, do the following:

1. Write the function *void sub5P(DemoP *ptr)* that receives as argument the address of an object of the class *DemoP* and subtract 5 from the value of each of its member variables.
2. Define the objects *obj1* and *obj2* and the pointer variable *oPtr* of the class *DemoP*. The member variables of object *obj1* are initialized with the values 20 and 9 respectively.
3. Using the pointer variable *oPtr*, write the statements to compute and print the average of the values of the member variables of object *obj1*.
4. Using the pointer variable *oPtr*, write the statements to read values into the member variables of object *obj2*, subtract 5 to the value of each of its member variables (by calling function sub5P( ), and then print the new values.

Memory Management in a C++ Program

- The memory allocated to a C++ program consists of five sections:
  a. The first section holds the **environment variables** and the **command line arguments** of the program.
  b. The second section consists of the **stack** and the **heap**: the stack grows downward whereas the heap grows upward.
  c. The third section is the **uninitialized data segment**.
  d. The fourth section is the **initialized data segment**.
  e. And the fifth section is **text segment**.



The Text Segment

- The text segment is also called instruction segment.
- It contains the following things:
  a. The machine language instructions of the program.
  b. The constant data of the program.

  Examples of constant data of a program:

  - *char *address = "300 Pomton road, Wayne, NJ 07470";*

    The string constant *"300 Pomton road, Wayne, NJ 07470"* will be stored in the text segment with the address of its first character stored in the pointer variable *address*.

  - The name of a function is a pointer constant that holds the address of the first instruction of the function. Given the following definition of function *procedure( )*:

```
void  procedure( int num, int &res1, int &res2 )
{
    int tum = 12;
    res1  /=  num + tnum;
    res2 /= num - tnum;
}
```

    A memory location named *procedure* will be created in the text segment with the address of the first instruction of the function stored into it.

- Memory locations are **created** (**allocated**) in the text segment before the execution of the program (when the program is loaded in memory), and they are **de-allocated** at the end of the execution of the program.

The Stack Segment

- A memory location is created in the stack segment for each of the following variables/parameters:
  a. The local (non-static) variables of functions.
  b. The value parameters of functions.
- Memory locations are **created** (**allocated**) in the stack every time a function is called and they are **de-allocated** at the end of the execution of the function.

  For example, after the following call of function *procedure( )* defined above:

```
int   value1 = 4,  snum  = 25,  fnum = 33;
procedure ( value1, snum, fnum );
```

We have the following creation of memory locations in the stack:

- A memory location is created for the value parameter *num* and is initialized with 4.
- A memory location is created for the local variable *tnum*.

- These memory locations are de-allocated at the end of the execution of the function *process( )*.
- Note that memory locations are not created for the reference parameters *res1* and *res2*.

The Initialized Data Segment

- This segment holds the following memory locations:
  a. Memory locations allocated for global variables with an initial value.
  b. Memory locations allocated to *static variables*: a static variable always have initial value.
- These memory locations are **created** (**allocated**) before the execution of the program (when the program is loaded in memory) and they are **de-allocated** at the end of the execution of the program.

The uninitialized data segment

- A memory location is created for each global variable without initial value in the uninitialized data segment.
- This memory location is created (allocated) before the execution of the program (when the program is loaded in memory) and is de-allocated at the end of the execution of the program.

The heap

- The heap is also called the **free store**.
- Memory locations are allocated in the heap during program execution by using the **new** operator.
- A memory location allocated in the heap is referred to as a **heap dynamic memory location**.
- A heap dynamic memory location is de-allocated (or freed) by using the **delete** operator.

Operators new and delete

- You use the **new** operator to allocate a memory location in the head as follows:

  <*pointer-variable*> = **new** <*Data-type*>;

  Where:
  <*Data-type*> is any valid data type.
  <*pointer-variable*> is a pointer variable to hold the address of a variable with data type *Data-type*.

- The **new** operator returns the address of the memory location allocated.
- The address returned by the **new** operator is used to have access to the allocated memory location.
- When you are done using a memory location allocated in the heap by using the **new** operator, use the **delete** operator to return it to the heap (**de-allocate**) as follows:

    delete *<pointer-variable>*;

  Where
  <pointer-variable> is a pointer variable that holds the address of the memory location.

Example M1 new/delete Operator with Basic Data Types

```
int  *ipt1, *ipt2;
char *cpt;
ipt1  =  new int;          // allocate a memory location to hold an integer value
ipt2  =  new int;          // allocate a memory location to hold an integer value
cpt  =  new char;          // allocate a memory location to hold a character value

/*---------------------- accessing  the allocated memory locations ----------------------------------*/
*ipt1  =  15;        //  assign 15 to the first (integer) allocated memory location
cin  >>  *ipt2     // read an integer value for the second (integer) allocated memory location
cin  >>  *cpt;     // read a character for the (character) allocated memory location
cout  <<  endl  <<  *ipt1  <<  " + "  <<  *ipt2  <<  " = "  <<  (*ipt1 + *ipt2);

/*-------------------------------de-allocate the memory locations---------------------------------------*/
delete  ipt1;
delete  ipt2;
delete  cpt;
```

Note

The **new** operator can also be used in a declaration statement as in the following example:

```
int * ipt1  =  new int;
```

Example M2 new/delete Operator with Structures/Classes

Given the following class *DayOfYear* defined in example O9:

```
class  DayOfYear
   {
      public:
         DayOfYear(int newMonth = 1, int  newDay = 1);    // constructor with default arguments
         void  input( );                                  // to input the month and the day
         void  output( );                                 // to output the month and the day
         int getMonth( );                                 // to return the month
         int getDay( );                                   // to return the day
      private:
         void  checkDate( );                    // to validate the month and the day
         int month;                             // to hold the month (1 – 12)
         int day;                               // to hold the day (1 – 31)
   };
```

And the following allocations of dynamic memory locations:

```
DayOfYear  *dayPtr1, *dayPtr2 ;
dayPtr1  = new DayOfYear;       // allocate memory locations for the object's member variables
dayPtr2  = new DayOfYear;       // allocate memory locations for the object's member variables
```

These dynamic memory locations may be accessed as followed:

```
dayPtr1 ->input(   );
cout  <<  endl   <<   "Month=\t"   <<   dayPtr1 ->  getMonth(   )
        <<   endl   <<   "Day=\t"   <<   dayPtr1 -> getDay(  );
*dayPtr2  =  DayOfYear( 12, 25 );
dayPtr2 -> output(   );
```

They are de-allocated as follows:

```
delete  dayPtr1;
delete  dayPtr2;
```

- A **dynamic memory location** can be initialized when it is allocated as follows:

  <type-pointer> = **new** *<type>(<initial-value>)*;

Example M3

*double *dptr = **new** double (72.15);*

*DayOfYear *dayPtr = **new** DayOfYear (12, 25); // dynamic DayOfYear object*

Exercise M1

1. Fill in the following table with the variables/parameters of the following program.

Variable/Parameter	Where is Memory Location Allocated	When is Memory Location Allocated	When is Memory Location De-allocated

2. Trace the execution of this program and show its output.

```
int gnum;
int procedure (int  n1, int &n2)
{
    static  int pnum  = 10;
    n1 ++ ;
    n2 ++ ;
    gnum = n1 + pnum;
    pnum ++;
    return ( n1 + n2);
}

    int main(  )
    {
       int result1, result2, val1 = 5, val2= 0, *pt;
       pt = new int ( 5 );
      *pt ++;
        result1 = procedure (*pt, val1);
        result2 = procedure (6, val2);
        cout << endl << "result1=" << result1 << endl << "result2=" << result2;
        return 0;
    }
```

Exercise M2*

What is the output produced by the following code:

```
int   *pt1 =  new int( 10 ),
      *pt2 =  new int( 15 );
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
pt1  =  pt2;
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
*pt1  = 20;
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
```

Exercise M3

What is the output produced by the following code:

```
int   *pt1 =  new int( 5 ),
      *pt2 =  new int( 20 );
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
*pt1  =  *pt2;
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
*pt1  = 20;
cout   <<  endl  <<  *pt1  <<  '\t'  <<  *pt2;
```

Exercise M4

Using the class *DayOfYear* defined in example O9 and used in example M2, do the following:

a. Allocate a dynamic object of the class *DayOfYear*, read values for its member variables, and output the values of its member variables.
b. Allocate a dynamic object of the class *DayOfYear*, initialize its member variables with 10 and 15 respectively, and then output the values of its member variables.

Call by Value Vs Call by Reference and the *const* Modifier

- The following things take place each time a function is called:
  - A memory location is created on the stack for each value parameter and is initialized with the corresponding argument.
  - This memory location is de-allocated (destroyed) at the end of the execution of the function.
  - A reference to the variable (argument) that corresponds to a reference parameter is passed to the function without the overhead of pushing and popping values from the stack.
- A **call by reference is therefore more efficient than a call by value,** especially when a large amount of data needs to be passed to a function as it is the case for most objects.

- It is therefore a good programming practice to use a reference parameter instead of a value parameter when a function's parameter is an object.
- When this is the case, the ***const*** modifier can be used to mark the parameter so that the compiler knows that the parameter should not be changed (since the intension is not to modify the value of the argument): you do this by placing the ***const*** modifier before the parameter type in the function prototype and the function header.
- The ***const*** modifier can also be used to mark a class member function if that member function should not modify the data members of the class: you do this by placing the ***const*** modifier after the right parenthesis in the function prototype and the function header.

Example M4

We define class *Demo10* that follows with two friend functions:

- *Demo10 addDemo10( )* has two reference parameters that are not modified inside the function.
- *void updateDemo10( )* has a reference parameter that is modified inside the function.

```
class  Demo10
{
public:
    Demo10(int n1 = 0, int n2 = 0);  // constructor
    int getVal1( ) const;            // returns the value of the first member variable
    int getVal2( ) const;            // returns the value of the second member variable
    friend  Demo10  addDemo10( const Demo10 &ob1,  const Demo10 &obj2);
    friend  void  updateDemo10( Demo10 &obj);
private:
    int val1;
    int val2;
};


/*-------------------------------------function addDemo10( ) ----------------------------------------------*/
/*  receives two objects of the class Demo10 and returns another object of the same class with each
   member variable the sum of the corresponding member variables of both objects
*/
Demo10  addDemo10( const Demo10 &obj1, const Demo10 &obj2 )
{
    Demo10 objResult;
    objResult.val1 = obj1.val1 + obj2.val1;
    objResult.val2 = obj1val2 + obj2.val2;
    return ( objResult);
}
```

```
/*---------------------------------------function updateDemo10( ) ----------------------------------------*/
/*  receives an object of the class Demo10 and adds 1 to the value of each of its member variables
*/
void  updateDemo10( Demo10 &obj )
{
     obj.val1++;
     obj.val2 ++;
}


/*-----------------------------------member function getVal1( ) ----------------------------------------*/
/*  returns the value of the first member variable
*/
int Demo10 :: getVal1 (  ) const
{
     return  val1 ;
}


/*-----------------------------------member function getVal2( ) ----------------------------------------*/
/*  returns the value of the second member variable
*/
int Demo10 :: getVal2 (  ) const
{
     return val2 ;
}

/*---------------------- calling the functions addDemo10( ) and updateDemo10( ) ------------------------*/

     Demo10  tobj(14, 25), sobj(5, 9), robj;
     robj = addDemo10( tobj, sobj );
     cout  <<   endl   <<   "first value is:\t"   <<   robj.getVal1( )
               <<   endl   <<   "second value is:\t"   <<   robj.getVal2(  );
     updateDemo10 ( tobj );
```

Note

A function with a const parameter cannot call a non const function.

Exercise M5

The following class *Demo11* has two friend functions *Demo11 subDemo11A ( )* and *void decrementDemo11A( )*.

s*ubDemo11A ( )* receives as arguments the reference of two objects of the class *Demo11* and then builds and returns another object of the class *Demo11* such that the value of each of its member variable is the value of the corresponding member variable of the first object argument minus the value of the corresponding member variable of the second object argument.

decrementDemo11A( ) receives as argument a reference to an object of the class Demo*11* and then subtracts one from the value of each of its member variables.

```
class  Demo11
{
public:
    Demo11 (int n1 = 0, int n2 = 0);  // constructor
    int getVal1( )  const;            // returns the value of the first member variable
    int getVal2( )  const;            // returns the value of the second member variable
    friend  Demo11  subDemo11 ( const Demo11 &ob1,  const Demo11 &obj2);
    friend  void  decrementDemo11 ( Demo11 &obj);
private:
    int val1;
    int val2;
};
```

1. Write the definitions of the member functions *getVal1( )* and *getVal2( )* and the functions *subDemo11A ( )* and *decrementDemo11A( )*.
2. Write the definitions of the ordinary function s*ubDemo11B ( )* that receives as arguments the reference of two objects of the class *Demo11* and then builds and returns another object of the class *Demo11* such that the value of each of its member variable is the value of the corresponding member variable of the first object argument minus the value of the corresponding member variable of the second object argument.
3. Write the definition of the ordinary function *decrementDemo11B( )* that receives as argument a reference to an object of the class Demo*11* and then subtracts one from the value of each of its member variables.
4. Given the following objects:
   Demo11 obj1(5, 8), obj2(12, 21), obj3(9, 6), obj4;

   A. Write the statements to assign values the member variables of object *obj4* such that the value of each member variable is the difference of the value of the corresponding member variable of object *obj1* minus the value of the corresponding member variable of object *obj2* (by calling function *subDemo11A( )*).
   B. Write the statements to assign values the member variables of object *obj4* such that the value of each member variable is the difference of the value of the corresponding member variable of object *obj1* minus the value of the corresponding member variable of object *obj2* (by calling function *subDemo11B( )*).

Operators Overloading

- In C++, an operator such as +, *, -, /, %, =, = =, <, > and many others is a function that is defined and called in a special way:
  - When you call a binary operator, the first argument is listed before the operator and the second argument is listed after it.
  - When you call a unary operator, the argument is listed after the operator (for a prefix operator) or before the operator (for a postfix operator).
- You can overload any of these operators so that it can accept objects of some class as argument(s).
- When you write the definition of an operator function, write the keyword **operator** before the name of the operator in the function header.
- An overloaded operator function may be implemented as an ordinary function, a friend function, or a member function of a class.
- However, the operators functions for the operators ( ), [ ], and -> and the assignment operators must be implemented as class member functions.
- Regardless of how an operator function is implemented, that operator is used in the same way in an expression.

Implementing an Operator Function as a friend of a Class

Example M5

The following class *Demo12* has two binary operators:

+ returns an object of the class *Demo12* with the value of each member variable the sum of the values of the corresponding member variables of its operands.

= = returns **true** if the values of the corresponding member variables of its operands are equal to each other and **false** otherwise.

It also has a unary prefix operator:

++ increments the value of each member variable of its object operand and then returns it.

All these operators are implemented as friend functions of the class.

```
class  Demo12
{
    public:
        Demo12 (int n1 = 0, int n2 = 0);  // constructor
        int getFirst( );                  // returns the value of the first member variable
        int getSecond( );                 // returns the value of the second member variable
        friend  Demo12  operator +(const Demo12 &ob1, const Demo12 &obj2);
        friend  bool  operator = =(const Demo12 &ob1, const Demo12 &obj2);
        friend  Demo12  operator ++( Demo12 &obj);
    private:
        int val1;
        int val2;
};


/*---------------------------------------------- Function operator + ----------------------------------------------*/
/*  receives two Demo12 objects and returns another Demo12 object with the values of the
   member variables the sum of the values of the corresponding member variables of both objects
*/
Demo12  operator +( const Demo12 &obj1, const Demo12 &obj2)
{
    Demo12 objResult;
    objResult.val1 = obj1.val1 + obj2.val1;
    objResult.val2 = obj1val2 + obj2.val2;
    return ( objResult);
}


/*-----------------------------------------Function operator = = ----------------------------------------------*/
/*  receives two Demo12 objects and returns true if the value of each member variable of
   the first  object is equal to the value of the corresponding member variable of  the second object.
   Otherwise it returns false
*/
bool  operator = =( const Demo12 &obj1, const Demo12 &obj2)
{
    bool answer;
    answer = ( (obj1.val1 = = obj2.val1) && (obj1val2  = = obj2.val2) );
    return ( answer);
}
```

```
/*---------------------------- Function operator (prefix) ++ ----------------------------------------------------*/
/*  receives a Demo12 object, increments each of its member variables by 1 and returns it
*/
Demo12 operator ++ ( Demo12 &obj )
{
    obj.val1 ++ ;
    obj.val2 ++ ;
    return ( obj );
}


/*-------------------------------------- Using the overloaded operators ----------------------------------------*/
Demo12  tobj(14, 25), sobj(5, 9), robj, uobj;

robj = tobj + sobj;
cout  <<  endl  <<  "first value is:\t"  <<  robj.getFirst( )
        <<  endl  <<  "second value is:\t"  <<  robj.getSecond( );
++ tobj;                    // function is called as a void function: the value returned is lost
uobj  =  ++ sobj  + Demo12 ( 13, 24 );                  // calling the constructor
if (uobj  == robj )
    cout  <<  endl  <<  "This is my lucky day";
else
    cout  <<  endl  <<  "I will do better next time";
```

Exercise M6

1. Rewrite the definition of the class *Demo12* of example M5 above such that the operators: - (minus), > (greater than), and pre-decrement - - are overloaded.

   - returns an object of the class *Demo12* with the value of each member variable the difference of the values of the corresponding member variables of the first operand minus the value of the corresponding member variable of the second operand.

   > returns true if the values of each member variable of its first operand is greater than the value of the corresponding member variable of the second operand, and false otherwise.

   - - decrements the value of each member variable of its object operand and then returns that object.

   All these operators are implemented as friend functions of the class.
2. Given the following definitions of objects: *Demo12 tobj(14, 25), sobj(5, 9), robj, uobj;*
   - Write a statement to subtract object *sobj* from object *tobj* and to assign the result to object *robj*.
   - Write a statement to decrement the value of each member variable of object *robj* by 1.
   - Write the statements to read values into the member variables of object *uobj* and to do the following: if object *uobj* is greater than object *robj*, output the message "I am happy" otherwise, output the message "I am sad".

Returning a Reference to an Object or a Variable

- A function that receives as argument an object or a variable (it has a reference parameter that corresponds to that object or variable) can return a reference to that object or variable.
- For a function to return a reference to an object or variable that it has received as argument, its return type must be the reference type of that object or variable.
- A reference to an object or a variable can be used in a program as that object or variable.

Example M6

We modify the class *Demo12* above such that the operator function ++ returns a reference to its object argument.

```
class  Demo12
{
    public:
        Demo12 (int n1 = 0, int n2 = 0);  // constructor
        int getFirst( );                  // returns the value of the first member variable
        int getSecond( );                 // returns the value of the second member variable
        friend  Demo12  operator +(const Demo12 &ob1, const Demo12 &obj2);
        friend  bool  operator = =(const Demo12 &ob1, const Demo12 &obj2);
        friend  Demo12  & operator ++( Demo12 &obj);
    private:
        int val1;
        int val2;
};
```

The definitions of the operator function operator (prefix) ++ follows:

```
/*----------------------------------Function operator (prefix) ++ ----------------------------------------------*/
/*  receives an object of the class Demo12, increments each of its member variables by 1 and
    returns its reference
*/
Demo12 & operator ++ ( Demo12 &obj )
{
    obj.val1 ++ ;
    obj.val2 ++ ;
    return ( obj );
}
```

Exercise M7

A class named *Triplet1* has three integer private data members named *first*, *second*, and *third*. Write the definition of this class as follows:

a. Given that the default initial value of each of these data members is 1, write the class constructor(s) that will be used to initialize the objects of this class with zero, one, two, or three arguments.
b. Write the member functions *getFirst( )*, *getSecond( )*, and *getThird( )* that returns the values of member variables *first*, *second*, and *third* respectively.
c. Overload the operator pre-increment ++ for the class *Triplet1*.
   pre-increment ++ adds 1 to the value of each member variable of its operand and returns its reference.

Implementing an Operator Function as a Member Function

Pointer Variable *this*

1. C++ associates with each object a constant pointer variable named ***this*** that holds its address.
2. This pointer variable can be used in any member function of a class to refer to the calling object.

Example M7

Given the following definition of class *Sample*:

```
class Sample
{
    Public:
        Sample ( int num = 0 );
        int getValue ( void );
    private:
        int value;
};
```

The following are two possible implementations of member function *getValue( )*:

```
int  Sample ::  getValue( void )
{
    return ( value );
}
```

```
int  Sample ::  getValue( void )
{
    return ( this -> value );      // a little bit silly!
}
```

- A **binary operator function** can be implemented as a member function of a class only if its left-most operand is an object (or a reference to an object) of that class.
- A **binary operator function** implemented as a member function of a class has the following characteristics:
  - The first operand is the object calling the operator function: It can be referenced in the body of the operator function using ***this*** pointer variable.
  - The second operand is the argument of the function. However, its private members can be accessed in the function if it is an object of that class.
- A **unary operator function** can be implemented as a member function of a class only if its unique operand is an object (or a reference to an object) of that class.
- A **unary operator function** implemented as a member function of a class has the following characteristics:
  - It has no parameter and its unique operand is the object calling the operator function: It can be referenced in the body of the operator function using ***this*** pointer variable.

Example M8

The following class *Demo13* is similar to class *Demo12* except that operators + and ++ are implemented as member functions.

```
class  Demo13
{
    public:
        Demo13 (int n1 = 0, int n2 = 0);  // constructor
        int getFirst( );                  // returns the value of the first member variable
        int getSecond( );                 // returns the value of the second member variable
        Demo13  operator  +( const Demo13 & rightOp );
        Demo13 & operator  ++( );
    private:
        int val1;
        int val2;
};
```

```
/*---------------------------------------Member function operator + ----------------------------------------*/
/*  receives a Demo13 object and returns another Demo13 object with the values of its  member
variables the sum of the values of the corresponding member variables of the current object and the
object received as argument
*/
Demo13  Demo13 ::  operator +( const Demo13 & rightOp )
{
    Demo13 objResult;
    objResult.val1 = val1 + rightOp.val1;
    objResult.val2 = val2 + rightOp.val2;
    return ( objResult);
}


/*-----------------------------------Member function operator (prefix) ++ -------------------------------------*/
/*  increments each member variables of an object by 1 and returns its reference
*/
Demo13 & Demo13 :: operator ++ (  )
{
    val1 ++ ;
    val2 ++ ;
    return ( * this );        // return a reference to the current object
}
```

Exercise M8

Write the definition of the class *Triplet2* which is similar to the class *Triplet1* of exercise M7, except that its the operators - (minus) , <= (less than or equal to) , and pre-increment ++ are implemented as member functions.

Class Destructor

- A **class destructor** is a special class member function with the following characteristics:
  1. Its name consists of the **tilde ~** symbol followed by the name of the class.
  2. Like constructors, it has no return type (not even **void**).
  3. It is called automatically whenever an object of that class goes out of scope.

Example M9

The following class *SampleD0* has an integer data member and a class destructor:

```
class   SampleD0
{
    public:
        SampleD0( int num = 0 );      // default constructor
        ~SampleD0(   );               // destructor
        void  setValue( int val );    //  sets the value of the data member
        int  getValue( void );        //  returns the value of the data member
    private:
        int  value;
};
```

The member functions of the class are defined as follows:

```
/*----------------------------------------------constructor ----------------------------------------------------------*/
SampleD0 :: SampleD0( int num )
{
    value   =   num;
}


/*----------------------------------------------destructor ----------------------------------------------------------*/
SampleD0 :: ~SampleD0(  )
{
    cout   <<  endl   <<  "Thank you for using an object of the class SampleD0";
}

/*-------------------------------------Member function setValue( )-------------------------------------------*/
void SampleD0 ::  setValue( int val )
{
    value  = val;
}


/*-------------------------------------Member function getValue( )-------------------------------------------*/
int SampleD0 ::  getValue( void )
{
    return ( value );
}
```

The output of the following program will be:

OUTPUT

15
Thank you for using an object of the class SampleD0
Thank you for using an object of the class SampleD0

```
/*----------------------------------------Program---------------------------------------------------------*/
int  Main(   )
{
    SampleD0  obj1( 5 ), obj2( 10 );
    cout  <<  obj1.getValue(  )  +  obj2.getValue(  );
    return 0;
}
```

Exercise M9

Given the class SampleD0 defined in example M9, what is the output of the following program?

```
/*----------------------------------------Program---------------------------------------------------------*/
int  Main(   )
{
    SampleD0  obj1( 2 ), obj2( 3 ), obj3( 5 );
    cout  <<  obj1.getValue(  )  +  obj2.getValue(  )  +  obj3.getValue(  );
    return 0;
}
```

Pointer Variables as Data Members of a Class

- A class data member can be a pointer variable to hold the address of a dynamic memory location or an array.
- When a class data member is a pointer variable to hold the address of a dynamic memory location or an array, you must provide a way for this memory location or array to be freed when an object of this class is no longer needed or is out of scope. This is done by using a **destructor**.
- The **Destructor of a class with pointer variables** (to the hold the addresses of dynamic memory locations) as data members must also have the **delete** statements to release those memory locations whenever an object of that class goes out of scope.

Example M10

The following class *SampleD1* has an integer pointer variable (to hold the address of a dynamic memory location) as data member:

```
class   SampleD1
{
    public:
        SampleD1( int num = 0 );      // default constructor
        ~SampleD1(  );                // destructor
        void  setValue( int val );    //  sets the value of the dynamic memory location
        int  getValue( void );        //  returns the value of the dynamic memory location
    private:
        int  * vptr;
};
```

The member functions of the class are defined as follows:

```
/*-----------------------------------------------constructor ----------------------------------------------------------*/
/*  Allocates a dynamic memory location to hold an integer value;
    sets that memory location to the initial value if one is provided; otherwise sets it to 0 (default value)
   The pointer variable data member of the class is set to the address of the dynamic memory location.
*/
SampleD1 :: SampleD1( int num )
{
    vptr   =  new  int;
    *vptr  =  num;
}


/*-----------------------------------------------destructor ----------------------------------------------------------*/
/*  Returns the object's allocated dynamic memory location back to the heap
*/
SampleD1 :: ~SampleD1(  )
{
    delete vptr;
}


/*-------------------------------------Member function setValue( )-------------------------------------------*/
/*   Sets the value of the dynamic memory location to the given value
*/
void SampleD1 ::  setValue( int val )
{
    *vptr  = val;
}
```

```
/*-------------------------------------Member function getValue( )-----------------------------------------*/
/*  Returns the value of the dynamic memory location
*/
int SampleD1 ::  getValue( void )
{
    return ( *vptr );
}
```

Problem with Default Copy Constructor and Assignment Operator

- When a class has a pointer variable (to the hold the address of a dynamic memory location) as a data member, the default copy constructor and assignment operator do no longer behave as expected: They copy the addresses of the dynamic memory locations instead of their contents.

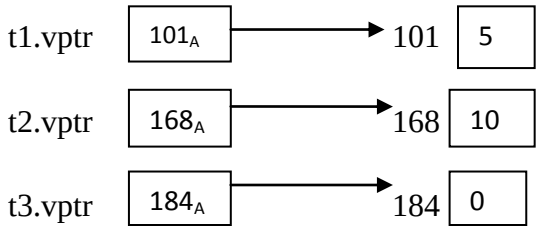
Example M11

Using the definition of the class *SampleD1* provided in example M10, we have the following:

Declaration of Object

SampleD1 t1(5),
t2 (10),
t3;

Allocations of Memory Locations

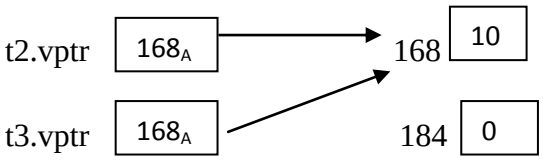


Problem with the Default Assignment Operator:

The statement

t3 = t2;

has the following effect:



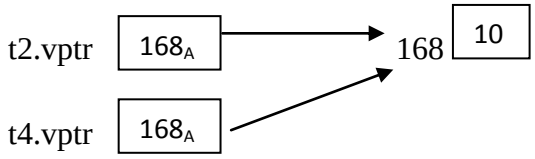
And the dynamic memory location with address 184 is no longer accessible to the program. It is called a **lost heap-dynamic variable**. This problem is sometime referred to as **memory leakage**.

Problem with the Default Copy Constructor:

The statement

SampleD1 t4(t2);

has the following effect:



- **These problems are resolved by redefining the assignment operator and the copy constructor so that the values of the dynamic memory locations are copied instead of their addresses**.

Example The copy constructor of the class *SampleD1* follows:

```
SampleD1 :: SampleD1( const SampleD1 &obj )
{
    vptr = new int;          // allocate a dynamic memory location for the object
    *vptr = *obj.vptr ;      // copy the value of the argument into that memory location
}
```

Example The assignment operator for the class *SampleD1* is overloaded as follows:

```
void SampleD1 :: operator =( const SampleD1 &obj )
{
    *vptr = *obj.vptr ;      // copy the value of the argument into that memory location
}
```

Notes

1. In C++, the function *operator =* that overloads the assignment operator can only be implemented as a class member function.
2. If you want to be able to use the overloaded assignment *operator =* more than once in an expression (such as A = B = C), then the function *operator =* must return a reference to the object that it receives as argument. This reference will be used again as an operand for the next assignment operator.
3. In order to make the assignment operator right-associative (That means that A = B = C is processed as A = (B = C)), you have to specify its return type as a *const*.

Example The assignment operator for the class *SampleD1* is overloaded as follows:

```
const SampleD1 & SampleD1 :: operator =( const SampleD1 &obj )
{
    *vptr = *obj.vptr ;      // copy the value of the argument into that memory location
    return *this;            // return a reference to the current object
}
```

The class *SampleD1* is therefore defined as follows:

```
class   SampleD1
{
    public:
      SampleD1( int num = 0 );                          // default constructor
     SampleD1( const  SampleD1 &obj );                  // copy constructor
      ~SampleD1(   );                                   // destructor
      const SampleD1 & operator  =( const  SampleD1 &obj );   // overloaded assignment operator
      void  setValue( int val );                        //  sets the value of the dynamic memory location
      int  getValue( void );                            //  returns the value of the dynamic memory location
    private:
      int  * vptr;
};
```

Exercise M10*

Define a class named *DynamicPair* with private data members the integer pointer variables *fpt* and *spt* as follows:

- The default constructor initializes the first dynamic memory location with 10 and the second with 25.
- The constructor with parameters *DynamicPair(int n1, int n2)* initializes the first dynamic memory location with *n1* and the second with *n2*.
- Also include in the definition of this class the set member function that sets the values of the allocated memory locations, and the get member functions that return their values.

Exercise M11

A class named *DynamicP* has the following private data members:

a. *symb* (char)
b. *iptr* (integer pointer) to hold the address of an integer dynamic memory location.
c. *dptr* (double pointer) to hold the address of a double precision dynamic memory location.

In addition to its constructors, destructor and the overloaded assignment operator, it has the following public member functions:

a. *void read( )* to read values into its data members.
b. *void write( )* to output the values of its data members.
c. A set member function for all the data members.
d. A get member function for each data member.

The default values for the data members are 'A' for the character data member, 50 for the integer data member, and 12.50 for the double precision data member.

a. Write the definition of the class *DynamicP*.
b. Write the definitions of its member functions including the constructors and the destructor.

Inline Functions and Inline Member Functions of a Class

- A function call involves the following execution-time overhead:
  - Before the control is passed to the called function, a memory location is created in the stack for each of its values parameters and local variables, and also for the *reentry point* in the calling function.
  - After the execution of the called function, all these memory locations are de-allocated (destroyed).
- **Inline functions** are provided in C++ to help reduce a function call execution overhead.
- An **inline function** is defined as follows:

```
inline   <return-type>  <function-name>( <parameter-list> )
{
        <body-of-the-function>
}
```

- By placing the keyword **inline** before a function's return type in the function definition, you are advising the compiler to generate a copy of the function's code in the place where the function is called.
- The compiler can ignore the *inline* qualifier and generates a function call instead of the function's code, especially when the function is not small.
- Note that an inline function cannot be declared and must appear in the program before the statement in which it is called.
- A reusable inline function is placed in a header file, so that its definition can be included in each source module in which it is called.

Example M12

The following program illustrates the use of an inline function:

```
/*---------------------------------- function computeArea(    ) ----------------------------------------*/
/*----------------------------- computer the area of a rectangle ----------------------------------------*/
inline  double  computeArea(  double len, double wid )
{
    return ( len  * wid );
}
```

```
int  main(   )
{
    double width, length;

    /*-------------compute and print the area of a rectangle with length 78.0 and width 49.0 -----------*/
    cout   <<   endl   <<   "The area of the rectangle is:\t"   <<   computeArea( 78.0, 49.0 );

    /*-------read the length and the width of a rectangle and compute and print its area ----------------*/
    cin   >>   width   >>   length;
    cout   <<   endl   <<   "The area of the rectangle is:\t"   <<   computeArea( length, width );
    return 0;
}
```

- **You can also make a class member function an inline function** by providing its definition instead of its prototype in the class definition.

Example M13

The class *SampleD1* of example M9 is redefined with all its member functions as inline functions as follows:

```
/*-------------------------------------------------sampleD1.h-------------------------------------------------------*/
#ifndef    SAMPLED1_H
#define    SAMPLED1_H

    class   SampleD1
    {
        public:
          SampleD1( int num = 0 )            //  constructor
          {
            vptr   =   new   int;
            *vptr   =   num;
          }

          SampleD1( const  SampleD1 &obj )          // copy constructor
          {
            vptr  = new int;              // allocate a dynamic memory location for the object
            *vptr   =   *obj.vptr ;         // copy the value of the argument into that memory location
          }

          ~SampleD1(   )                // destructor
          {
            delete vptr;
          }
```

```
        SampleD1 & operator  =( const  SampleD1 &obj )      // overloaded assignment operator
        {
          *vptr  =  *obj.vptr ;      // copy the value of the argument into that memory location
          return  *this;             // return a reference to the current object
        }

        void  setValue( int val )        //  sets the value of the dynamic memory location
        {
          *vptr  = val;
        }

        int  getValue( void )            //  returns the value of the dynamic memory location
        {
          return ( *vptr );
        }

      private:
        int  * vptr;
  };
#endif
```

Exercise M12

Rewrite the definition of the class *Triplet2* of exercise M8 by making the member functions inline functions.

Virtual Functions and Dynamic Objects

Assume given the class *Tile* and its derived classes *RectangleTile* and *TriangleTile* of example O12 with the following modification: member function *print( )* is now a virtual function.

```
class  Tile
{
  public:
    Tile( );
    Tile( double uprice );
    double  getUprice( );
    virtual  double computeArea( );        //default definition of  member function computeArea( )
    double  computePrice( );
    virtual void  print( );
  private:
    double unitPrice;
};
```

```
/*------------------------------ Definition of constructors and member functions -------------------------*/
Tile ::  Tile(  ) : unitPrice ( 2.0 )
{
}

Tile ::  Tile ( double uprice ) : unitPrice( uprice )
{
}

double  Tile ::  getUprice( void )
{
    return unitPrice ;
}

double  Tile ::  computeArea( void )
{
    return ( 1 );          // return a dummy area of 1sqrt inch
}
double  Tile :: computePrice(  )
{
    double  area;
    area  =  computeArea(  );          // call to the virtual function compute Area is delayed
    return( unitPrice  *  area );
}

void  Tile ::  print( void )
{
    cout  <<  endl  <<  "The price of the tile is:\t"  <<  computePrice (  );
}
```

Derived Class *RectangleTile*:

```
class RectangleTile   :   public   Tile       // inherits class Tile
{
    public:
        RectangleTile (  );                                          // default constructor
        RectangleTile ( double uprice , double len , double wth);    // constructor
        virtual  double computeArea(  );              // function to be called for Rectangle objects
        virtual void  print(  );
    private:
        double length;
        double  width;
};
```

```
/*-------------------------------- Definition of constructors and member functions -------------------------*/
RectangleTile ::  RectangleTile(  ) : Tile(  ) , length( 0.5 ) , width( 1.0 )
{
}

RectangleTile :: RectangleTile(double len , double wth, double uprice ):Tile( uprice ) , length( len ) ,
width( wth )
{
}

double  RetangleTile ::  computeArea( void )
{
    return ( length * width );                    // return the area of a rectangular tile
}

void  RectangleTile ::  print( void )
{
    cout  <<  endl  <<  "The length is:\t"  <<  length
          <<  endl  <<  "The width is:\t"  <<  width;
    cout  <<  endl  <<  "The price of the tile is:\t"  <<  computePrice (  );
}
```

Derived Class *TriangleTile*:

```
class TriangleTile   :   public   Tile        // inherits class Tile
{
    public:
        TriangleTile (  );                                          // default constructor
        TriangleTile (double ht , double bse, double uprice );        // constructor
        virtual  double computeArea(  );          // function to be called for triangle objects
        virtual void  print(  );
    private:
        double height;
        double  base;
};

/*-------------------------------- Definition of constructors and member functions -------------------------*/
TriangleTile ::  TriangleTile (  ) : Tile(  ) , height( 0.5 ) , base( 1.0 )
{
}
```

```
TriangleTile::TriangleTile (double ht , double bse, double uprice ):Tile( uprice) , height( ht) , base(bse )
{
}

double  TriangleTile ::  computeArea( void )
{
    return ( height * base / 2.0 );              // return the area of a rectangular tile
}

void  TriangleTile ::  print( void )
{
    cout  <<  endl  <<  "The height is:\t"  <<  height
          <<  endl  <<  "The base is:\t"  <<  base;
    cout  <<  endl  <<  "The price of the tile is:\t"  <<  computePrice ( );
}
```

Assume given the following definitions of dynamic objects:

```
Tile * pt1, * pt2, * pt3;
pt1 = new Tile( 1.5 );
pt2 = new RectangleTile( 5.0, 3.0, 2.0 );
pt3 = new TriangleTile( 4.0, 6.0, 1.0 );
```

The execution of the following statements will produce the following output:

```
pt1-> print( );
cout  <<  endl;
pt2-> print( );
cout  <<  endl;
pt3-> print( );
cout  <<  endl;
```

OUTPUT

```
The price of the tile is:     1.5

The length is:  5.0
The width is:   3.0
The price of the tile is:     30.0

The height is:  4.0
The base is:    6.0
The price of the tile is:     12.0
```

Note

- A **handle for an object** is a pointer variable that holds the address of that object.
- A pointer variable defined to be a **handle** for a base class object can also be a handle for an object of a derived class of that base class.
- A call to virtual member function on a handle for an object is bound to the definition of the function provided in the class of that object.

Exercise M13

Given the following two classes:

```
class Symbol
{
  public:
    Symbol( char ch = 'A');
    virtual  void   print(   )
    {
        cout   <<   endl   <<   symb;
    }
   private:
     char  symb;
};
```

```
class TwoSymbol  : public  Symbol
{
  public:
    TwoSymbol( char ch1, char  ch2  = 'Z');
    virtual  void   print(   )
    {
        Symbol  :: print(   );
        cout   <<   endl   <<   extraSymb;
    }
   private:
     char  extraSymb;
};
```

1. Show the output of the following code segment:

```
Symbol  *pt1,  *pt2;
Pt1  =  new  Symbol( 'K' );
Pt2  =  new  TwoSymbol( 'C' , 'P' );
Pt1  ->  print(   );
Pt2  ->  print(   );
```

2. What would the output be if the member function *print( )* was not a virtual function?

Overloaded Operators and Inheritance

- An overloaded operator defined on objects of the base class can also be used on objects of derived classes.
- A call to a virtual member function in the definition of an overloaded operator is bound to the definition of that function provided in the class of the object on which the function is called.

Example

Assume that the operator > is defined on objects of the class Tile as follows:

```
bool  operator > ( Tile obj1, Tile obj2 )
{
    return(  obj1.computerArea(  )  > obj2.computeArea(   ) );
}
```

That means that a tile *obj1* is greater than a tile *obj2* if its area is greater than that of *obj2*.

Given the following definitions of objects:

```
Tile tile1;
RectangleTile tile2( 5.0, 3.0, 2.0 );
TriangleTile tile3( 4.0, 6.0, 1.0 );
```

We have the following:

tile1 > tile2	is false
tile2 > tile3	is true

Solutions of Exercises

Exercise P1

num1 = 20
num2 = 9
num3 = 12
*pt1= 20
*pt2= 9

Exercise P3

Body of function tester in the way it is executed after the function call:

```
{
    num = 4;
    pt = &tvalue;
    *pt = *pt - num;
}
```

output:

tvalue = 16

Exercise P5

1.

```
void add3P ( DemoP  * ptr )
{
    double  num1, num2;
    num1 = ptr ->  getvalue1(   );
    num2 = ptr ->  getvalue2(   );
    *ptr  =  DemoP( num1 + 3, num2 + 3);
}
```

2.

```
DemoP  obj1( 10, 15), obj2, *oPtr;
```

3.

```
oPtr = & obj2;
oPtr  -> readValues(   );
cout   <<   "\n the average of the values of the member variables of object obj2 is:\t"
          <<   oPtr ->  getAverage(  );
```

4.

```
add3P( & obj1 );
cout  <<  endl  <<  "val1="  <<  obj1. getvalue1(  )  <<  endl  <<  "val2="  <<  obj1.getvalue2(  );
```

Exercise M2

10 15
15 15
20 20

Exercise M10

```
 class DynamicPair
 {
       public:
            DynamicPair( int  nl = l0 ,  int  n2 = 25 );                    // constructor
            DynamicPair( const DynamicPair & obj );                         // copy constructor
            ~ DyrnamicPair(   ) ;                                           // destructor
            DynamicPair  & operator  = ( const  DynamicPair & rightObj );   //  overloaded assignment
            void setValues (int n1, int n2 );
            int  getValuel( )   const;
            int  getValue2( )  const;

       private:
           int * fpt;
           int  *spt;
};

/*-----------------------------------constructor ------------------------------------------------*/

DynamicPair  : :  DynamicPair( int n1,  int n2)
{
     fpt   =  new  int;
     *fpt  =  n1;
     spt  =  new  int;
     *spt  =  n2;
}

/*-------------------------------- copy constructor ------------------------------------------------------*/
DynamicPair ::  DynamicPair(  const DynamicPair & obj )
{
     fpt =  new  int;
     *fpt = *obj.fpt;
     spt = new  int
     *spt  =  *obj.spt;
}

/*--------------------------------destructor ------------------------------------------------------------------------*/
DynamicPair  ::  ~DynamicPair(  )
{
     delete  fpt;
     delete  spt;
}
```

```
/*--------------------------------overloaded assignment-----------------------------------------------------------*/
DynamicPair & DynamicPair ::  operator  = ( const  DynamicPair & rightObj )
{
     *fpt   =  *rightObj.ptl;
     *spt  =  *rightObj.spt;
     return( * this );
}

/*-----------------------------function setValues ----------------------------------------------------------------*/
void DynamicPair ::  setValues( int n1, int n2)
{
     *ptl  = n1;
     *spt  = n2;
}

/*-----------------------------function getValue1 ----------------------------------------------------------------*/
  int DynamicPair ::  getValuel( ) const
 {
        return  ( *ptl );
}

/*-----------------------------function getValue2 ----------------------------------------------------------------*/
  int DynamicPair ::  getValue2( ) const
 {
        return  (*spt );
}
```