If we have learned anything in three decades of building software systems,
it is that this design on the fly leads to disaster.
There is no substitute for thinking first and acting later,
acting only when all the essential thinking is completed.
Design is the thinking process that has to precede the action of implementation.
Tom Demarco, 1982

Those types are not "abstract"; they are as real as int *and* float*.*
Doug McIlroy

3.1 Introduction

The C++ class mechanism allows users to define their own data types. Classes are typically used to define abstractions that do not map naturally into the predefined or desired data types - for example a terminal display screen, an employee or bank account class. A C++ class has four associated attributes:

1. A collection of data members, the representation of the class. There may be zero or more data members of any type in a class.
2. A collection of member functions, the set operations that may be applied to objects of that class.
3. There may be zero or more member functions for a class. They are referred to as the class interface or levels of program access. Members of a class may be specified as private, protected or public. These levels control access to member from within the program. Typically the representation of a class is private while the operations that may be performed on the representation are public. This sort of public/private specification is referred to as information hiding. A private internal representation is said to be encapsulated. Anything declared private is accessible only to functions named in the class definition.
4. A class tag name serving as a type specifier for the user-defined class.

A class with a private representation and a public set of operations is referred to as an abstract data type.

3.1.1 Account class example

The C++ class definition has 2 parts, the class header, composed of the keyword class followed by the class tag name and the class body enclosed by a pair of curly braces which must be followed by either a semicolon (;) or a declaration list.

```
class account { };
```

3.1.1.1 Data Members

Declaration of class data members is the same as variable declarations.

```
class account
{
   char name[40], acctnum[20];
   double balance;
};
```

3.1.1.2 Member Functions

Users of the account class must perform a wide range of operations on an account class. Functions to initialize, deposit, withdraw and display the account must be provided. Member functions of a class are declared inside the class body. A declaration consists of the function prototype. Member functions have full access privilege to both the public and private members of the class while, in general, ordinary functions have access only to the public members of the class. However the member functions of one class, in general, have no access privileges to the members of another class. Member functions are defined within the scope of their class; ordinary functions are defined at file scope. Member function definition for account class follows:

3.1 Code listing

```
// prog31.h

#include <iostream>
using namespace std;

class account
{
   private:
      char name[40], acctnum[20];
      double balance;

   public:
      void init(void);
      void show(void);
      void deposit(double amt);
      void withdraw(double amt);
      double getBalance(void);
      void setBalance(double amt);
};

// prog31.cpp

#include <iostream>
using namespace std;
#include "prog31.h"

// Init function initializes the account data
void  account :: init(void)
{
   cout <<"Enter name " << endl ;
   cin >> name;
   cout << "Enter account number" << endl;
   cin >> acctnum;
   cout << "Enter initial balance" << endl;
   cin >> balance;
}

// Show function displays the account data
void account :: show(void)
{
   cout << "Name is " << name << endl;
   cout << "Account number is " << acctnum << endl;;
   cout << "Balance is " << balance << endl;;
}
```

```
// Deposit function deposits some amount into the customers account
void account :: deposit(double amt)
{
   balance += amt;
}

// Withdraw function withdraws some amount from the customers account
void account :: withdraw(double amt)
{
   balance -= amt;
}

// getBalance function returns the balance information of the customer
double account :: getBalance (void)
{
   return balance;
}

// setBalance function updates the balance of the customer
void account :: setBalance (double amt)
{
   balance = amt;
}

int main(void)
{
   /* Creating an instance of account class - john object */
   account john;
   john.init();
   john.show();
   john.deposit(500);
   cout << "Modified balance " << john.getBalance() << endl;
   john.setBalance(5000);
   john.withdraw(1000);
   cout << "Updated balance " << john.getBalance() << endl;

   // Why is the statement declared below an error?
   // cout << john.name << john.acctnum << john.balance;
   return 0;
}
```

3.1.1.3 Classification of member function

The member functions of the class are classified based on their functionality. They are

1. Manager functions - these functions are used to perform ‘Initialization’, ‘cleanup’, and other fundamental chores associated with the class.
2. Accessor functions - these functions return information about an object’s current state. As these functions are not supposed to change the state of the objects, they are declared as ‘const’ member functions of the class.
3. Implementor functions - these are functions that can change the state of the object. These are also called ‘mutators’

3.1.2 Information Hiding

It is a mechanism for restricting user access to the internal representation of a class type. The access specifiers are public, private and protected. Members defined within a public section become public members; those declared within private or protected section become private or protected members. A public member is accessible from anywhere within a program. A class enforcing information hiding limits its public members to the member functions meant to define the functional operations of the class. A protected member behaves as a public member to a derived class. It behaves as a private member to the rest of the program. Only the member functions and friends of its class can access a private member. A class that enforces information hiding declares its data members as private.

3.1.3 C++ Structures and Unions

In ‘C’ it is illegal to declare functions within structures and unions. But ‘C++’ allows functions as structure elements. The set of functions that are allowed also include constructors and destructors.

When it is possible to bind together data members and member functions using structures, why have classes in C++? The default specifier in a C++ structure is public, while in a class it is private (hence, we use classes for the purpose of data encapsulation).

3.2 Complex class definition

This class introduces the concept of constructors, destructors, operator overloading, friend and member functions, inline functions, default functions (assignment, copy constructor etc.) provided by the C++ compiler. The class definition consists of two parts. The private paragraph consists of the complex data (the real part and the imaginary part). The public section consists of the functions required by the complex object to access the private part. This class definition allows the user to perform a set of operations like complex addition, subtraction, multiplication etc.

3.2 Complex class

```
#include <math.h>
#include <iostream>
using namespace std;

class complex
{
   float rpart, ipart;      // Data members

   public:
      // binary operators
      complex operator + (complex);
      complex operator - (complex);
      complex operator * (complex);
      complex operator / (complex);

      complex operator - ();                // unary minus operator

      complex operator += (const complex &);// concatenation operator

      // comparison operator
      friend int operator == (const complex &, const complex &);

      friend double magnitude(const complex &);      // magnitude
      //output operator
      friend ostream & operator << (ostream &, complex &);

      // default arguments constructor
      complex(float = 0.0, float = 0.0);
      complex operator ++ ();          // pre increment operator

      complex operator ++ (int);       // post increment operator

      // Inline functions
      float getRpart()
      {
         return rpart;
      }
      float getIpart()
      {
         return ipart;
      }
      ~complex();           // destructor
};
```

Member functions have an implied first argument. If a function body appears in the class definition, it is automatically made an inline function.

3.2.1 Class Constructors

A function with the same name as the class is a constructor. A constructor does not have a return type (not even void). There can be many constructors with different argument lists.

```
complex();
complex(int);
complex(float, float);
```

Constructors give us a place for instance initialization.

```
complex :: complex(float r, float i)
{
   rpart = r;
   ipart = i;
}
```

A constructor is called anytime a new instance is created

```
void main()
{
   complex a = 5;              // Same as complex a(5);
   complex *bp = new complex;
}
```

Constructors also serve as casting operators

```
void main()
{
   float a = 2.3;
   complex b = complex(a, a); // Explicit cast
   complex c = b - 3;       // Implicit cast
}
```

Try: Define the constructor with explicit keyword and test the following statement? Justify?

```
   complex c = b - 3;
```

Note

A constructor
- Cannot be declared const, but a constructor can be invoked for a const object.
- Cannot be declared static, virtual.
- Should have public access within the class.
- ‘inline' is the only legal storage class for constructors

3.2.2 Class destructors

A function with the same name as a class preceded by a tilde (~) character is a destructor. A destructor has neither arguments nor return type (~complex()). There can be only one destructor for a class. A destructor is called anytime an object leaves scope or is deleted. The destructor does all the cleanup operations.
Destructor definition

```
complex :: ~complex()
{
   cout << "\n Destructor of complex class called";
}

void main()
{
   complex a = 5;                // Same as complex a(5)
   complex *bp = new complex;
   {
      complex c(1, 3);         // Constructor called
      *bp = a + c;
   }                           // Destructor called on c
   a = *bp;
   delete bp;                  // Destructor called on *bp;
}                              // Destructor called on a
```

Note

A destructor
- Cannot be declared static, const.
- Should have public access in the class declaration.

3.2.3 Member functions

Member functions can be defined outside the class definition. We prefix the function name with the class name to associate this function with the one declared in the class definition. Member functions that are defined inside the class are by default inlined. The member functions defined outside the class can also be inlined by preceding the function header with the keyword ‘inline’.

Complex class member function definitions follow

```
// Binary + operator definition
complex complex :: operator + (complex b)
{
   return complex (rpart + b.rpart, ipart + b.ipart);
}

// Binary - operator definition
complex complex :: operator - (complex b)
{
   return complex (rpart - b.getRpart(), ipart - b.getIpart());
}

// Binary * operator definition
complex complex :: operator * (complex b)
{
   return complex ((rpart * b.rpart) - (ipart * b.ipart),
      (ipart * b.rpart) + (rpart * b.ipart));
}

// Binary / operator definition
complex complex :: operator / (complex b)
{
   complex result;

   result.rpart = ((rpart * b.rpart) + (ipart * b.ipart)) /
      ((b.rpart * b.rpart) + (b.ipart * b.ipart));
   result.ipart = ((ipart * b.rpart) - (rpart * b.ipart)) /
      ((b.rpart * b.rpart) + (b.ipart * b.ipart));

   return result;
}

// Unary - operator defintion
complex complex :: operator - ()
{
   return complex(-rpart, -ipart);
}
```

3.2.3.1 Magic pointer - this

We can refer to the current object inside a member function by using the implied '**this**' argument. In a nonstatic member function, the keyword this is a pointer to the object for which the function was invoked. Most uses of this are implicit. Every instance of the class gets a new copy of each data member, whereas the member functions are not part of each object. In other words member functions are maintained as a single copy irrespective of the number of instances. While implementing member functions we are not qualifying the data member names with any object name (direct access) or pointer name (indirect access).

Each non-static member function is implicitly passed a pointer that holds the address of the object that is calling the member function. And each data member name used in the member function code is qualified by this special pointer name. For example, when getX() is called using cobj1, the statement

```
   return x;
```

is evaluated as

```
   return this->x;
```

where 'this' is the magic pointer that is implicitly passed to getX(). And the value of 'this' pointer is the address of cobj1.

Always use "**this**" explicitly !!!

Member functions continued

```
// Operator += definition
complex complex :: operator += (const complex & b)
{
   return complex (this->rpart += b.rpart, this->ipart += b.ipart);
}
```

Pre and post increment operators

The int argument is used to indicate that the function is to be invoked for postfix application of ++. This int is never used; the argument is simply a dummy used to distinguish between prefix and postfix application. The way to remember which version of operator++ is prefix is to note that the version without the dummy is prefix, exactly like the other unary arithmetic and logical operators.

```
complex complex :: operator ++()
{
   return complex(++rpart, ++ipart);
}

complex complex :: operator ++(int)
{
   return complex(rpart++, ipart++);
}
```

It can also be written as follows, but the prototypes in the class would look like

```
complex & operator++();             // prefix
const complex operator++(int);      // postfix

complex & complex :: operator ++()
{
   this->rpart += 1;                // increment
   this->ipart += 1;                // increment
   return *this;                    // fetch
}

const complex  complex :: operator ++(int)
{
   complex oldcomplex = *this;   // fetch
   ++(*this);                    // increment
   return oldcomplex;            // return what was fetched
}
```

Note

In the prefix, why are we returning a reference?
In the postfix, why are we returning a const?

3.2.4 Friend Functions

An ordinary member function declarations specifies three things:

1. The function can access the private part of the class declaration, and

2. The function is in the scope of the class, and
3. The function must be invoked on an object (has this pointer).

By declaring a member function static, we can give it the first two properties only. By declaring a function as a friend, we can give it the first property only.

Sometimes we would like to have a non-member function that needs access to the private part of a class. To do this, we add the function to the class declaration as a friend of the class. In the complex class declaration you have seen such declarations (Equality operator, Magnitude function and Output operator). Friend functions of a class must have an instance of that class in the argument list. It can be placed in either the private or public part of a class declaration. Like a member function, a friend function is explicitly declared in the declaration of the class of which it is a friend.

```
// Equality operator definition
int operator == (const complex & a, const complex & b)
{
   return((a.rpart == b.rpart) && (a.ipart == b.ipart));
}

// Magnitude function definition
double magnitude(const complex &a)
{
   double result = sqrt(a.rpart * a.rpart + a.ipart * a.ipart);
   return result;
}

// Output operator definition
ostream & operator << (ostream & o, complex & a)
{
   return ( o << a.rpart << " + " << a.ipart << "i" );
}

// Operator + as friend function

// prototype in class
friend complex operator +(complex &, complex &);

// Implementation
complex operator + (complex & a, complex & b)
{
   return complex(a.rpart + b.rpart, a.ipart + b.ipart);
}
```

Note

We use member functions when we want to take advantage of runtime operator identification. We use friend function when the first argument (operand) is not an instance of that class.

Can we define operator + as both friend and member functions? Justify?

prog32main.cpp

```
#include "prog32.h"
#include <iostream>
```

```
using namespace std;

void main()
{
   complex a(1, 2), b(3), c;
   c = a + b;
   cout << c<< endl;
   c = a - b;
   cout << c<< endl;
   c = a + 5;
   cout << c<< endl;
   c = 5 + a;
   cout << c << endl;
   cout << c++ << endl;
   cout << ++c << endl;
}
```

3.2.5 Friend Classes

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

For example, consider two classes A and B. If class A grants its friendship to class B, then the private data members of the class A are permitted to be accessed by the public members of the class B. But on the other hand the public member functions of the class A cannot access the private members of the class B. Friendship is not commutative.

3.3 Code listing

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

class A {
  int a;
  public:
    A() {
      a = 10;
    }
    friend class B;     // Friend Class
  };

class B {
  int b;
  public:
    void showA(A& x) {
      // Since B is friend of A, it can access private members of A
      cout << "A::a=" << x.a;
    }
};

int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

3.2.6 Default member functions

`Copy constructor: complex(complex &)`

Copy constructor creates a new object from an existing one. It takes only one argument, a reference to an object of the same class.

`Why is the object always passed by reference?`

Because if it were passed by value, a local copy of it would be created for which you would invoke the copy constructor once again, and so it would result in a non-terminating recursive function. The default copy constructor copies the source bit by bit to the new object. In many classes that do not have pointers or references as data members, the default copy constructor is adequate.

3.4 Code listing

```
complex :: complex (const complex & c)
{
   rpart = c.rpart;
   ipart = c.ipart;
}

void main()
{
   complex a(0, 1);
   complex b(a);         // Calls complex (complex &)
   complex c = a;           // Calls complex (complex &)
}
```

Note

Copy constructor is called automatically

- Whenever an object is copied by means of a declaration initialization.
- An object is passed by value to a function.
- An object is returned by value from a function.

`Assignment Operator: complex & operator = (const complex &)`

The default assignment operator copies the source bit by bit to the destination. The assignment operator does not create a new object but changes the data of an existing object.

3.5 Code listing

```
complex & complex :: operator =  (const complex & c)
{
   rpart = c.rpart;
   ipart = c.ipart;
   return (*this);
}

void main()
{
   complex a(0, 1);
   complex b, c;
   b = a;                 // Calls operator = (complex &)
   c = a + b;             // Calls operator = (complex &)
}
```

For a class that contains pointer data members, the default copy and the assignment operators give problems. They need to be overridden.

3.3 Static members

A variable that is part of a class, yet is not part of an object of that class, is called a static member. It is a class variable. There is exactly one copy of a static member irrespective of the number of objects created, unlike a non-static data member, where there is a separate copy maintained for each object created. Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a static member function.

Let us suppose, we need to get a count of how many account holders we have. We can define count as a static data member in the class. In addition we could define a function called getCount, which returns the count of number of account holders. Our account class would look as follows:

3.6 Code listing

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

class account
{
   static int count;
   int val;

   public:
      account() {
         count ++;
         val = 5;
      }

      ~account() {
         count --;
      }

      static int getCount() {
         return count;
      }
};

int account:: count = 0;

void main()
{
   account cust;
   cout << "Initial count " << account::getCount() << endl;
   {
      account cust2, cust3;
      cout << "Inside count " << cust2.getCount() << endl;
   }
   cout << "Outside count " << account::getCount() << endl;
}
```

Static members must be initialized somewhere outside the class, before calling any member functions.

```
int account :: count = 0;
```

In the init (constructor) function of the account class, we increment the count (count++). In the destructor we decrement the count (count--). This way, whenever an account object is created, the count increases, and the count decreases when the account object is destroyed. A static member can be referred to like any other member. In addition, a static member can be referred to without mentioning an object. Instead, its name is qualified by the name of its class in any function.

```
cout << account :: getCount();
```

Note

A static member function

- Cannot make use of this pointer. Why?
- Cannot access any non-static data. Why?
- Cannot be declared const. Why?

3.7 Code listing (Singleton design pattern)

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

class singleton
{
   int value;
   singleton();
   static singleton* obj;
   public:
      int getValue();
      void setValue(int val);
      static singleton* getInstance();
};

singleton* singleton :: obj = NULL;

singleton :: singleton() {
   value = 20;
}

int singleton :: getValue() {
   return value;
}

void singleton :: setValue(int val) {
   value = val;
}

singleton * singleton :: getInstance() {
   if (obj == NULL)  {
      cout << "if block" << endl;
      obj = new singleton();
   }
   else {
      cout << "else block" << endl;
   }
   return obj;
}

void main() {
   singleton *s1 = singleton::getInstance();
   s1->setValue(40);
   cout << "Value 1 " << s1->getValue() << endl;

   singleton *s2 = singleton::getInstance();
   cout << "Value 2 " << s2->getValue() << endl;

   s2->setValue(50);
   cout << "Value 1 " << s1->getValue() << endl;
   cout << "Value 2 " << s2->getValue() << endl;
   singleton *s3 = singleton::getInstance();
   cout << "Value 3 " << s3->getValue() << endl;
}
```

3.4 Constant member functions

Consider a class rectangle.

3.8 Code Listing

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

class rectangle
{
   int left, top, right, bottom;
   public:
      rectangle(int l, int t, int r, int b)
      {
         left = l;
         top = t;
         right = r;
         bottom = b;
      }
      int getLeft() const;
      int getTop()  const        { return top; }
      int getRight()  const      { return right; }
      int getBottom()  const     { return bottom; }
};

inline int rectangle :: getLeft() const
{
   return left;
}

void main()
{
   rectangle obj(10, 20, 30, 40);
   cout << obj.getLeft();
}
```

Note the const after the (empty) argument list in the function declarations. It indicates that these functions do not modify the state of a rectangle. The compiler will catch accidental attempts if you violate this promise.

```
inline int rectangle :: getLeft() const
{
   // Error: attempt to change member value in const function
   return left++;
}
```

When a const member function is defined outside its class, the const suffix is required

```
inline int rectangle :: getLeft() const     // correct
{
   return left;
}

// Error: const missing in member function type
inline int rectangle :: getLeft()
{
   return left;
}
```

3.4.1 Const objects

3.9 Code Listing

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

class square
{
   int side;
   public:
      square(int s = 0)
      {
         side = s;
      }

      void print(void) const
      {
         cout << side;
      }

      void setside(int s)
      {
         side = s;
      }
};

void main()
{
   const square obj;
   obj.setside(20);
   obj.print();
}
```

It is good programming practice to make an object constant if it should not be changed. This is done with the const keyword. Like variables and function parameters, objects can also be declared as constant.

```
   const square sqr(4);
```

However, when this is done, the C++ compiler restricts access to the object member functions. For example, with the square class defined previously, the setside(4) function could not be called for this object.

```
   // illegal (compiler dependent - some give warning, some give error)
   sqr.setside(4);
   sqr.print();     // legal, since print() is a const member function
```

The only member functions that could be called for const objects would be the constructors, destructor and const member functions

Note

When the program declares an object to be const, the program is in effect declaring that this pointer is a pointer to a const object. A const pointer can be used only with const function members.

We can also declare a pointer to a const object

```
   const square *sp = new square;
   sp->print()      // legal
   sp->setside(3)   // illegal
```

We can also declare a const pointer to an object

```
   square *const spq = new square;
```

It is a const pointer to square. This can call non-const member functions because the pointer is const and it cannot point to anything else.

```
   spq->print()       // legal
   spq->setside(4)    // legal
```

3.4.2 The'mutable' keyword

If the data members of the class are to be modified even when it belongs to a constant object, then its declaration should be preceded by a C++ keyword 'mutable'.

3.10 Code Listing

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

class Test
{
   int fixedvar;
   mutable int ordvar;
   public:
      Test (int arg1, int arg2)
      {
         fixedvar = arg1;
         ordvar   = arg2;
      }
      void show() const
      {
         // cout << "\n fixed var = " << fixedvar++ << endl; // illegal
         cout << "\n ord var = " << ++ordvar << endl;           // legal
      }
};

void main()
{
   const Test const_obj(10, 20);
   const_obj.show();
}
```

3.5 What functions C++ silently writes and calls

When a class is defined without anything the C++ compiler will write its own version of a copy constructor, an assignment operator, and a pair of address-of operators. If we don't declare any constructors, it will write a default constructor for you. All these functions will be public. In other words, if you write this

```
   class empty{};
```

it will end up like this

```
class empty {
   public:
      empty();                                // default constructor
      empty(const empty & rhs);                  // copy constructor
      empty & operator = (const empty & rhs);    // assignment operator
      empty * operator & ();                     // address of operator
      const empty * operator & () const;
};
```

Given these functions, what do they really do? The default constructor does nothing; it just enables you to define objects of the class. The default address of operators just return the address of the object. These are defined like this

```
inline empty :: empty() { }

inline empty * empty :: operator & ()
{
   return this;
}

inline const empty * empty :: operator & () const
{
   return this;
}
```

The default copy constructor and the assignment operator perform member wise copy construction of the non-static data members of the class.

What is the size of empty class? Why?

3.11 Code Listing

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

class empty{};

void main()
{
   empty e;
   cout << "Size of a empty class: "<<sizeof(e) <<endl;

   empty e1[5];
   for (int i = 0;i < 5 ;i++)
      cout << "Address of: "<<&e1[i] <<endl;
}
```

3.6 Arrays of object

3.12 Code Listing

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

class test_array
{
  int x, y;
   public:
      test_array(int xval = 0, int yval = 0)
      {
         x = xval;
         y = yval;
      }

      void show( )
      {
         cout << " x = " << x << " y = " << y << endl ;
      }
};

void main()
{
   int i;
   test_array X[2] = {1,5};
   for(i = 0; i < 2; i++)
      X[i].show();

   test_array Y[3];
   for(i = 0; i < 3; i++)
      Y[i].show();
}
```

For a class to permit creation of arrays of instances, it must have a matching constructor. A constructor with default arguments is also sufficient.

Objects ‘X’, and 'Y' are arrays of instances of class test_array. Object 'X' is created using the constructor to which one argument is passed and the second argument is the default one. Object 'Y' is initialized by the constructor (both are default arguments).

3.7 Different meanings of new and delete

What is the difference between new operator and operator new?

When you write code like this,

```
   string *ps = new string ("memory management");
```

the new you are using is the **new operator**. This operator is built into the language and, like sizeof, you can't change its meaning; it always does the same thing. What it does is twofold. First, it allocates enough memory to hold an object of the type requested. In the example above, it allocates enough memory to hold a string object. Second, it calls a constructor to initialize an object in the memory that was allocated. The new operator always does these two things; you can't change its behavior in any way.

What you can change is how the memory for an object is allocated. The new operator calls a function to perform the requisite memory allocation, and you can write or overload that function to change its behavior. The name of the function the new operator calls to allocate memory is **operator new**. It is declared as follows

```
   void * operator new (size_t size);
```

The return type is void *, because this function returns a pointer to raw, uninitialized memory. The size_t parameter specifies how much memory to allocate. You can overload operator new by adding additional parameters, but the first parameter must always be of type size_t.

Like malloc, operator new's only responsibility is to allocate memory. It knows nothing about constructors. What operator new understands is memory allocation. It is the job of the new operator to take the raw memory that operator new returns and transform it onto an object.

When the compiler sees a statement like

```
   string *ps = new string ("memory management");
```

they must generate code that more or less corresponds to this:

```
// get raw memory for a string object
void *memory = operator new(sizeof(string));

// initialize the object in the memory
call string :: string("memory management") on *memory;

// make ps point to the new object
string *ps = static_cast<string *>(memory);
```

If you want to create an object on the heap, use the new operator. It both allocates memory and calls a constructor for the object. If you only want to allocate memory, call operator new; no constructor will be called. If you want to customize the memory allocation that takes place when heap objects are created, write your own version of operator new and use the new operator; it will automatically invoke your custom version of operator new.

Deletion and memory deallocation

To avoid resource leaks, every dynamic allocation must be matched by an equal and opposite deallocation. The function operator delete is to the built in delete operator as operator new is to the new operator. When you write something like this,

```
   string *ps;
   …
   delete ps;
```

your compiler must generate code both to destruct the object ps points to and to deallocate the memory occupied by that object.
The memory deallocation is performed by the **operator delete** function, which is usually declared like this:

```
   void operator delete(void *memory_to_be_deallocated);
```

Hence,

```
   delete ps;
```

causes compilers to generate code that approximately corresponds to this:

```
   ps->~string()         // call the object’s destructor
   operator delete(ps);  // deallocate the memory the object occupied
```

3.13 Code Listing

```
#include <stdlib.h>
#include <iostream>
using namespace std;

class point
{
  int x, y;
   public:
      point() {}
     point(int a, int b)
      {
       x = a;
       y = b;
      }

      void show()
      {
         cout << x << " ";
         cout << y << endl;
   }
  void* operator new(size_t size);
  void operator delete(void *p);
};

void* point :: operator new(size_t size)
{
   cout << "Overloaded operator new called" << endl;
   return malloc(size);
}

void point :: operator delete(void *p)
{
   cout << "Overloaded operator delete called" << endl;
   free(p);
}
```

```
int main()
{
   point *p1, *p2;
   p1 = new point(10, 20);
   if(!p1)
   {
      cout << "Allocation error" << endl;
      exit(1);
   }

   p2 = new point(-10, -20);
   if(!p2)
   {
      cout << "Allocation error" << endl;
      exit(1);
   }
   p1->show();
   p2->show();
   delete p1;
   delete p2;
   return 0;
}
```

The global 'operator new()' must be prepared to allocate blocks of any size. Similarly, the default version of delete must be prepared to deallocate blocks of whatever size new allocated. In order for delete to know how much memory to deallocate, it must have some way of knowing how much memory new allocated in the first place. Generally this is done by sending an extra chunk of memory along with the size of the actual memory allocated for storing objects. This consumes extra blocks of memory space. Some times the size of the extra blocks is more than that of the actual blocks, and it is to be avoided. By writing the class specific 'operator new()' one can take advantage of the fact that all objects of the same class are of the same size, so there is no need for maintaining the size information at the cost of few extra blocks of memory.

3.9 Type conversions

In C and C++, there is automatic type conversion to the operands when constants and variables of different types are mixed in an expression. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
   int x;
   float y = 2.345;
   x = y;
```

converts y to an integer before its value is assigned to x. The type conversion functions are automatic as long as the data types involved are built-in types. What happens when they are user-defined data types? In the complex example, we had added two complex numbers and assigned the result to another complex number. There was no problem, as the objects are of the same type and the assignment operator works fine. What happens, if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes? Since the user defined data types are designed by us to suit our requirement, the compiler does not support automatic type conversions for such data types. The possible situations are:

1. Conversion from built-in type to class type (refer to section 3.2.1).
2. Conversion from class type to built-in type (refer to section 4.2 and 4.3).

3. Conversion from one class type to another class type. We will discuss the same here, where we convert an object of account to complex type (the examples which we discussed earlier).

3.14 Code Listing

```
#include <string.h>
#include <iostream>
using namespace std;

class complex
{
   float rpart,ipart;
   public:
      float& getRpart()
      {
         return rpart;
      }

      float& getIpart()
      {
         return ipart;
      }

      complex(float r = 0, float i = 0)
      {
         rpart = r;
         ipart = i;
      }

      friend ostream& operator<<(ostream &o, complex &c)
      {
         o << "rpart = "<< c.rpart << "\tipart = " << c.ipart << endl;
         return o;
      }
};

class account
{
  char acctnum[20], name[20];
  double balance;
   public:
      double getbal()
      {
         return balance;
      }

      account(char anum[], double bal, char name[])
      {
         strcpy(acctnum, anum);
         balance = bal;
         strcpy(this->name, name);
      }

      // type cast operator to convert account object to complex object
      operator complex()
      {
         complex a;
         a.getRpart() = balance;
         a.getIpart() = 0;
```

```
        return a;
      }
};

void main()
{
   account a("AB1234",12.2,"john");
   complex b;
   b = a;             // Same as b=complex(a);
   cout << endl << b;
}
```

3.10 Operator Overloading Issues

3.15 Code Listing

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

class myComplex
{
  float rpart, ipart;
  public:
    myComplex(float = 0.0, float = 0.0);
    myComplex operator ++();
    void operator --(int);
    friend ostream & operator << (ostream &, myComplex &);
    void operator= (myComplex);
};

myComplex::myComplex(float r, float i)
{
  rpart = r;
  ipart = i;
}

ostream & operator << (ostream & o, myComplex & a)
{
  return ( o << a.rpart << " + " << a.ipart << "i" << endl );
}

myComplex myComplex :: operator ++()
{
   return myComplex(++rpart, ++ipart);
}

void myComplex :: operator --(int)
{
   myComplex(rpart--, ipart--);
}

void myComplex::operator=(myComplex ob)
{
   rpart = ob.rpart;
   ipart = ob.ipart;
};

void main()
{
   myComplex a(1, 2), b (2, 3), c;

   // What is the warning in the following statement?
   cout << a++;

   // What is the error in the following statement? How to fix it?
   c = b = a;
   cout << c;

   // What is the error in the following statement? How to fix it?
   c = b--;
   cout << c;
}
```

3.11 Exercise

Theory Questions

1. What is the difference between C and C++ structure?
2. How does a C++ structure differ from a C++ class?
3. Can we use the same name for a class member function and a non-member function in the same program file? If yes, how do we distinguish them? If no, Why?
4. What is a friend function? When do we use one? What are the merits and demerits of using friend functions?
5. What is a constructor and a destructor? When are they called?
6. What is the difference between this and *this operator?
7. Friend functions always have one more argument than the corresponding member function. Comment. Justify?
8. What are the default member functions provided by the C++ compiler? When do we need to override them?
9. What is a default constructor and what is the advantage of having one?
10. How does a class accomplish data hiding and encapsulation?
11. What is the relationship between an object and a class?
12. List some of the special properties of the constructor functions?
13. Can we have more than one constructor in a class? If yes, explain the need for such a situation?
14. How is dynamic initialization of objects achieved?
15. Distinguish between the following two statements: time T2 (T1) and time T2 = T1, where T1 and T2 are objects of time class.
16. What is operator overloading? Why is it necessary to overload an operator?
17. What is an operator function? Describe the syntax of operator function?
18. Why do we pass an argument by reference to the copy constructor?
19. What is the use of static data member and static member function?
20. What is the significance of constant member function and const object?
21. Static member function cannot make use of this pointer. Justify?
22. How are arrays of objects created?
23. What is the difference between interface and implementation of class?
24. Why should the stream operators << and >> be overloaded as friend functions?

25. What does an empty class include?

Problems

1. Implement the input (>>) and output (<<) operator class point. Write a main function to test the same.

   ```
   class point
   {
      int x, y, z;
   };
   ```

2. Create a base class "date" that has separate int members for day, month & year. Write a constructor to initialize date. Define functions to update the date & output the same (perform all validations).

3. Create a class myint with one private data i. Overload arithmetic operators (+, -, *, /, %) so that they operate on objects of type myint. Output all the results using 2 instances. Define the increment and decrement operators (both pre and post). Without overloading the output operator, output the data using the statement as shown below

   ```
   cout << a;      // a is an object of myint class
   ```

4. Enhance the complex class example to implement the following operators (!=, <, >, >=, <=, -=, *=, /=)

5. Create a class called "time" that has separate int members for hours, minutes and seconds. One constructor should initialize this data to 0 and another should initialize it to fixed values. A member function or overloaded operator should display it in 11:59:59 format. Overload the + operator to add two objects of type "time" passed as arguments. Write a main program to test it.

6. Given the program

   ```
   # include <iostream.h>

   void main()
   {
      cout << "Hello World" << endl;
   }
   ```

   modify it to produce the output

   ```
   Initialize
   Hello World
   Cleanup
   ```

   Do not change main() in any way.

7. Modify the account example to implement constructors, the input operator (>>) and the output operator (<<).

8. Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks: Create a vector, modify the value of a given element, multiply

the vector object by a scalar value and to display the vector in the form (10, 20, 30, …..). Write a program to test the same.

9. Modify the above program such that the program would be able to add two vectors and display the resultant vector. (We can pass objects as function arguments).

10. Create a class Matrix of size M * N. Define all possible matrix operations like addition, transpose, multiplication etc.