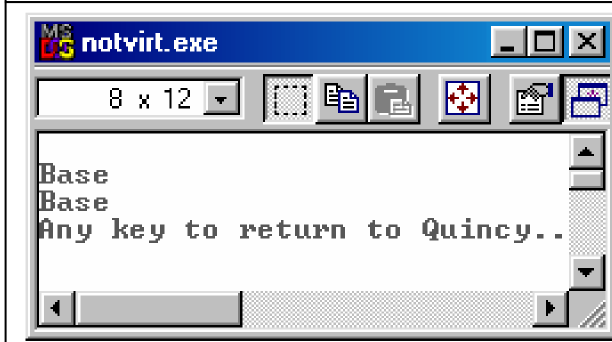
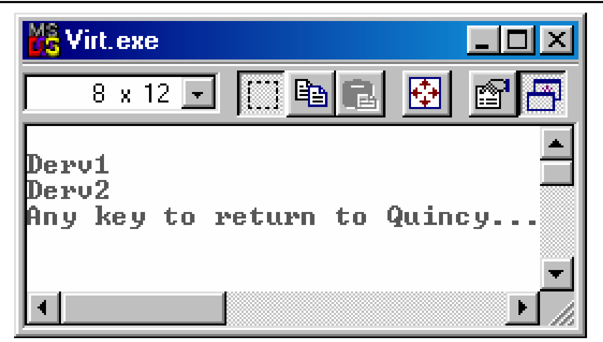
Virtual functions : means existing in effect but not in reality.
Before going to all in details, please read Base /derived class in the chapter Class Inheritance. Virtual member functions are created with a key word virtual in preceding prototype. C++ uses Late or dynamic binding for virtual methods. Static or early binding for nonvirtual methods. Virutal functions are the building block of polymorphism.
Example Not virtual :

```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream.h>
class Base          // base class
  {
  public:
    void show()        // normal function
   { cout << "\nBase"; }
  };
class Derv1 : public Base      // derived class 1
  {
  public:
    void show()
          { cout << "\nDerv1"; }
  };
class Derv2 : public Base     // derived class 2
  {
  public:
    void show()
   { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;          // object of derived class 1
  Derv2 dv2;          // object of derived class 2
  Base* ptr;          // pointer to base class
  ptr = &dv1;   // put address of dv1 in pointer
  ptr->show();        // execute show()
  ptr = &dv2;   // put address of dv2 in pointer
  ptr->show();        // execute show()
  }
```

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream.h>
class Base                        // base class
  {
  public:
    virtual void show()        // virtual function
          { cout << "\nBase"; }
  };
class Derv1 : public Base        // derived class 1
  {
  public:
    void show()
          { cout << "\nDerv1"; }
  };
class Derv2 : public Base        // derived class 2
  {
  public:
    void show()
          { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;          // object of derived class 1
  Derv2 dv2;          // object of derived class 2
  Base* ptr;          // pointer to base class
  ptr = &dv1;         // put address of dv1 in pointer
  ptr->show();        // execute show()
  ptr = &dv2;         // put address of dv2 in pointer
  ptr->show();        // execute show()
  }
```





Note the differences, in the first we did not get Base replaced with Derv1/Derv2.
In the first example The Derv1 and Derv2 classes are derived from class Base. Each of these three classes has a member function show().

In main() we create two objects from corresponding derived classes.

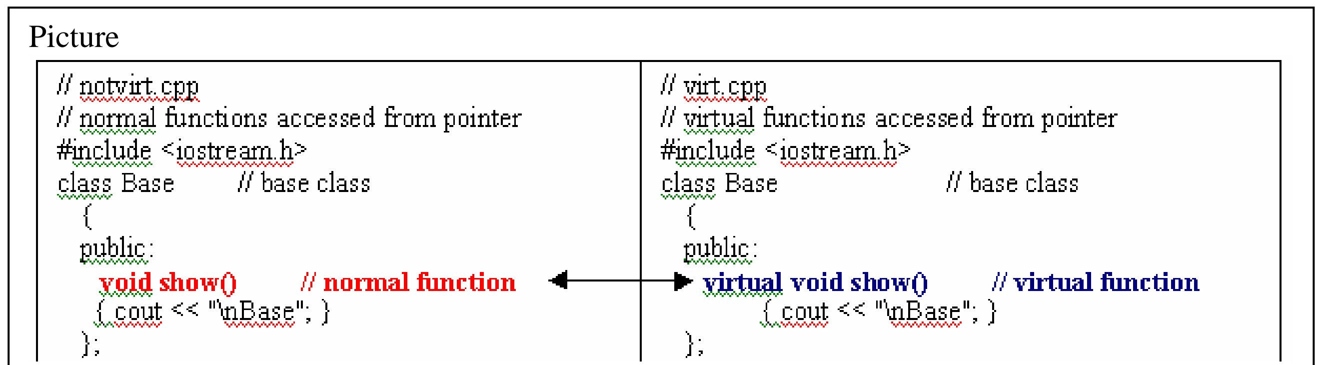
```
Derv1 dv1;        // object of derived class 1
Derv2 dv2;        // object of derived class 2
```

Then we put the address of a derived calass pointer in the line
ptr = &dv1

But which function to call in
Ptr->show();

Then the base class answer this question as
Base
Base.

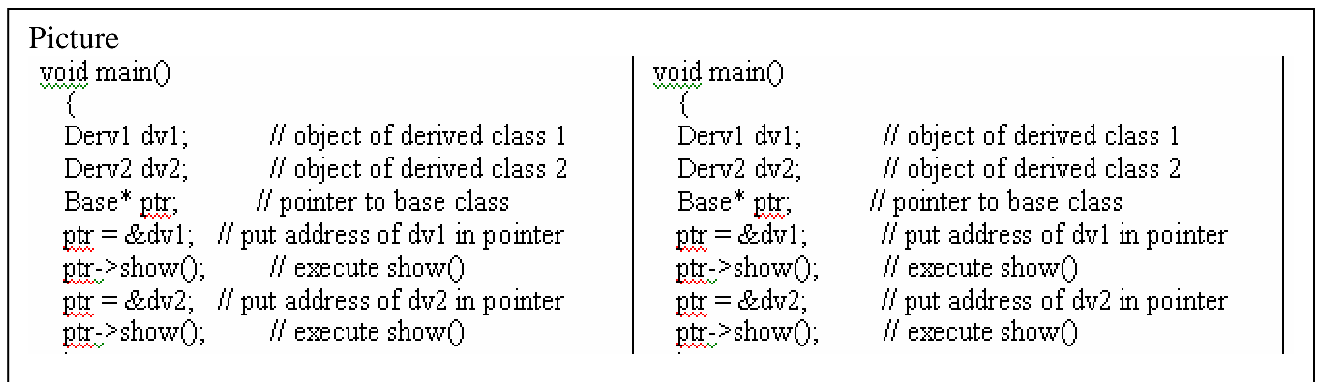
In the second example Virtual key word takes care of the derivatives in the derived classes. Although Virtual pointer access does not know how and when it would be derived , and would defer output till a pointer access derived function show(). In the first example cursor wont go through derived classes.


Picture

```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream.h>
class Base       // base class
  {
  public:
    void show()       // normal function
    { cout << "\nBase"; }
  };
```

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream.h>
class Base                  // base class
  {
  public:
    virtual void show()       // virtual function
         { cout << "\nBase"; }
  };
```

The pointer to the derive classes &dr1, &dr2


Picture

```
void main()
  {
   Derv1 dv1;          // object of derived class 1
   Derv2 dv2;          // object of derived class 2
   Base* ptr;          // pointer to base class
   ptr = &dv1;   // put address of dv1 in pointer
   ptr->show();        // execute show()
   ptr = &dv2;   // put address of dv2 in pointer
   ptr->show();        // execute show()
```

```
void main()
  {
   Derv1 dv1;          // object of derived class 1
   Derv2 dv2;          // object of derived class 2
   Base* ptr;          // pointer to base class
   ptr = &dv1;         // put address of dv1 in pointer
   ptr->show();        // execute show()
   ptr = &dv2;         // put address of dv2 in pointer
   ptr->show();        // execute show()
```

Picture

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
    };
class Derv2 : public Base           // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                  // object of derived class 1
    Derv2 dv2;                  // object of derived class 2
    Base* ptr;                  // pointer to base class

    ptr = &dv1;                 // put address of dv1 in pointer
    ptr->show();                // execute show()

    ptr = &dv2;                 // put address of dv2 in pointer
    ptr->show();                // execute show()
    }
```

Press F1 for help | Ln 25, Col 1 | NUM

derived obect 1 is being initialized

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                 // object of derived class 1
    Derv2 dv2;                 // object of derived class 2
    Base* ptr;                 // pointer to base class
```

Press F1 for help | Ln 16, Col 1 | NUM

Cursor moves back to the object

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                 // object of derived class 1
    Derv2 dv2;                 // object of derived class 2
    Base* ptr;                 // pointer to base class
```

Press F1 for help | Ln 25, Col 1 | NUM

Late or derived binding by virtual

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
#include <iostream.h>

class Base                         // base class
    {
    public:
       virtual void show()         // virtual function
     { cout << "\nBase"; }
    };
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
```

Picture shows late binding with virtual , and backs to derived object.

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                 // object of derived class 1
    Derv2 dv2;                 // object of derived class 2
    Base* ptr;                 // pointer to base class
```

Press F1 for help | Ln 25, Col 1 | NUM

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                 // object of derived class 1
    Derv2 dv2;                 // object of derived class 2
    Base* ptr;                 // pointer to base class
```

Press F1 for help | Ln 26, Col 1 | NUM

derived object2 is being initialized, in derived class 2.

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
    {
    public:
       void show()
     { cout << "\nDerv2"; }
    };
void main()
    {
    Derv1 dv1;                 // object of derived class 1
    Derv2 dv2;                 // object of derived class 2
    Base* ptr;                 // pointer to base class
```

Press F1 for help | Ln 22, Col 1 | NUM

Late binding for derived object 2

Quincy 99 - Virt.cpp
File Edit View Project Debug Tools Window Help

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream.h>

class Base                         // base class
    {
    public:
       virtual void show()         // virtual function
     { cout << "\nBase"; }
    };
class Derv1 : public Base          // derived class 1
    {
    public:
       void show()
     { cout << "\nDerv1"; }
    };
class Derv2 : public Base          // derived class 2
```

Press F1 for help | Ln 10, Col 1 | NUM

Picture

```
  };
class Derv2 : public Base          // derived class 2
  {
  public:
    void show()
    { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;              // object of derived class 1
  Derv2 dv2;              // object of derived class 2
  Base* ptr;              // pointer to base class

  ptr = &dv1;             // put address of dv1 in pointe
  ptr->show();            // execute show()

  ptr = &dv2;             // put address of dv2 in pointe
```

After cursor calls show derived class is accessed

```
  public:
    virtual void show()            // virtual function
    { cout << "\nBase"; }
  };
class Derv1 : public Base          // derived class 1
  {
  public:
    void show()
    { cout << "\nDerv1"; }
  };
class Derv2 : public Base          // derived class 2
  {
  public:
    void show()
    { cout << "\nDerv2"; }
  };
void main()
```

Same way cursor fetches pointer to the derived object .

```
  public:
    void show()
    { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;              // object of derived class 1
  Derv2 dv2;              // object of derived class 2
  Base* ptr;              // pointer to base class

  ptr = &dv1;             // put address of dv1 in pointe
  ptr->show();            // execute show()

  ptr = &dv2;             // put address of dv2 in pointe
```

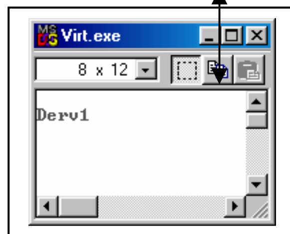
show function is called

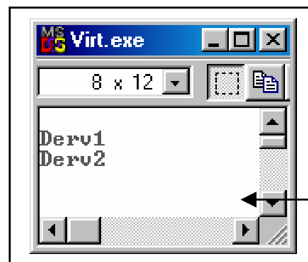
```
    void show()
    { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;              // object of derived class 1
  Derv2 dv2;              // object of derived class 2
  Base* ptr;              // pointer to base class

  ptr = &dv1;             // put address of dv1 in pointe
  ptr->show();            // execute show()

  ptr = &dv2;             // put address of dv2 in pointe
  ptr->show();            // execute show()
```

value in derived class2 is accessed

```
  {
  public:
    void show()
    { cout << "\nDerv1"; }
  };
class Derv2 : public Base          // derived class 2
  {
  public:
    void show()
    { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;              // object of derived class 1
```

```
    { cout << "\nDerv2"; }
  };
void main()
  {
  Derv1 dv1;              // object of derived class 1
  Derv2 dv2;              // object of derived class 2
  Base* ptr;              // pointer to base class

  ptr = &dv1;             // put address of dv1 in pointe
  ptr->show();            // execute show()

  ptr = &dv2;             // put address of dv2 in pointe
  ptr->show();            // execute show()
  }
```





Pure and alternate way to call virtual function

```
// virtpure.cpp
// pure virtual function
#include <iostream.h>
class Base                    // base class
  {
  public:
    virtual void show() = 0;    // pure virtual function
  };

class Derv1 : public Base      // derived class 1
  {
  public:
    void show()
         { cout << "\nDerv1"; }
  };
class Derv2 : public Base    // derived class 2
  {
  public:
    void show()
         { cout << "\nDerv2"; }
  };
void main()
  {
  Base* list[2];       // list of pointers to base class
  Derv1 dv1;           // object of derived class 1
  Derv2 dv2;           // object of derived class 2
  list[0] = &dv1;      // put address of dv1 in list
  list[1] = &dv2;      // put address of dv2 in list
  list[0]->show();     // execute show() in both objects
  list[1]->show();
  }
```

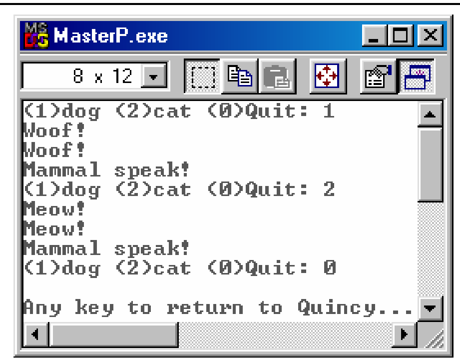
Data slicing when passing by value

```
//cmaster.cpp
#include <iostream>
#include "cdog.cpp"
#include "ccat.cpp"
#include "cmammals.h"
void ValueFunction (Mammal);
void PtrFunction   (Mammal*);
void RefFunction (Mammal&);
int main()
{
Mammal* ptr=0;
int choice;
while (1)
{
bool fQuit = false;
 cout << "(1)dog (2)cat (0)Quit: ";
        cin >> choice;
        switch (choice)
        {
        case 0: fQuit = true;
                        break;
        case 1: ptr = new Dog;
                        break;
        case 2: ptr = new Cat;
                break;
        default: ptr = new Mammal;
                        break;
        }
        if (fQuit)
                break;
        PtrFunction(ptr);
        RefFunction(*ptr);
        ValueFunction(*ptr);
        }
        return 0;
}
void ValueFunction (Mammal MammalValue)
{
        MammalValue.Speak();
}
void PtrFunction (Mammal * pMammal)
{
        pMammal->Speak();
}
void RefFunction (Mammal &
rMammal)
{
        rMammal.Speak();
}
```

```
//cmammals.h
#ifndef _CMAMMALS_H_
#define _CMAMMALS_H_
class Mammal
{
public:
Mammal():itsAge(1) {  }
virtual ~Mammal() { }
virtual void Speak() const { cout << "Mammal
speak!\n"; }
protected:
        int itsAge;
};
#endif
```

```
//cdog.cpp
#include <iostream>
#include "cmammals.h"
class Dog : public Mammal
{
public:
void Speak()const { cout << "Woof!\n"; }
};
```

```
//ccat.cpp
#include <iostream>
#include "cmammals.h"
class Cat : public Mammal
{
public:
void Speak()const { cout << "Meow!\n"; }
};
```



More example of virtual function
Before going through this please refer ios_base in class inheritance document. With Qunicy use ios than using ios_base.

Base class bankacct.h

// bankacct.h -- a simple BankAccount class with virtual functions
#ifndef _BANKACCT_H_
#define _BANKACCT_H_
class BankAccount
{
private:
enum {MAX = 35};
char fullName[MAX];
long acctNum;
double balance;
public:
BankAccount(const char *s = "Nullbody", long an = -1,
double bal = 0.0);
void Deposit(double amt);
virtual void Withdraw(double amt); // virtual method
double Balance() const;
virtual void ViewAcct() const; // virtual method
};
#endif
	// overdrft.h -- Overdraft class declaration
	#ifndef _OVERDRFT_H_
	#define _OVERDRFT_H_
	#include "bankacct.h" // modified to use virtual
	class Overdraft : public BankAccount
	{
	private:
	double maxLoan;
	double rate;
	double owesBank;
	public:
	Overdraft(const char *s = "Nullbody", long an = -1,
	double bal = 0.0, double ml = 500,
	double r = 0.10);
	Overdraft(const BankAccount & ba,
	double ml = 500, double r = 0.1);
	void ViewAcct()const;
	void Withdraw(double amt);
	void ResetMax(double m) { maxLoan = m; }
	void ResetRate(double r) { rate = r; };
	void ResetOwes() { owesBank = 0; }
	};
	#endif

Derived class from bankacct.h

```
// bankacct.cpp -- methods for BankAccount class
#include <iostream>
using namespace std;
#include "bankacct.h"  // virtual version
#include <cstring>
//defining and creating derived class constructor
BankAccount::BankAccount(const char *s, long an, double bal)
{
   strncpy(fullName, s, MAX - 1);
   fullName[MAX - 1] = '\0';
   acctNum = an;
   balance = bal;
}

void BankAccount::Deposit(double amt)
{
   balance += amt;
```

```
}

void BankAccount::Withdraw(double amt)
{
   if (amt <= balance)
      balance -= amt;
   else
 cout << "Withdrawal amount of $" << amt
   << " exceeds your balance.\n"
  << "Withdrawal canceled.\n";
}
double BankAccount::Balance() const
{
   return balance;
}
void BankAccount::ViewAcct() const
{
   // set up ###.## format
   ios::fmtflags initialState =
      cout.setf(ios::fixed, ios::floatfield);
   cout.setf(ios::showpoint);
   cout.precision(2);
 cout << "Client: " << fullName << endl;
cout << "Account Number:" << acctNum << endl;
 cout << "Balance: $" << balance << endl;
 cout.setf(initialState); // restore original format
}
```

- The derived class constructors are responsible for initializing any data members to those inherited form the base class.
- The base class constructors are responsible for initializing the inherited data members.
- A constructor for derived class can use the initializer list mechanism to pass values along to a base class constructor.

Derived::derived (type x, type y) :base(x, y) // initializer list

Name of the base class in derived class.

//You define and create a constructor using :: operator.

NameBaseClass:: NameBaseclass(parameters, parameters) : call BaseClass Constructor (parameter, paramater,----)
//passing arguments fromt the derived –class constructor to the the base-class constructor

Overdraft::Overdraft(const char *s, long an, double bal, double ml, double r) : BankAccount(s, an, bal)

Base-class constructor

Create Overdraft object

BankAccount::BankAccount**(const char *s, long an, double bal)**

Codes for derived class from overdft.h

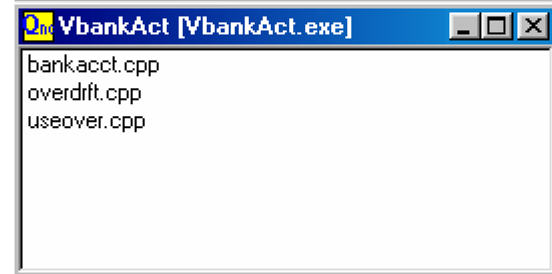
```
// overdrft.cpp -- Overdraft class methods
//use ios instead of ios_base
#include <iostream>
using namespace std;
#include "overdrft.h"
//defining derived class constructor
Overdraft::Overdraft(const char *s, long an, double bal,
        double ml, double r) : BankAccount(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}

Overdraft::Overdraft(const BankAccount & ba, double ml, double r)
        : BankAccount(ba)   // uses default copy constructor
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
```

```
}

// redefine how ViewAcct() works
void Overdraft::ViewAcct() const
{
   // set up ###.## format
   ios::fmtflags initialState =
      cout.setf(ios::fixed, ios::floatfield);
   cout.setf(ios::showpoint);
   cout.precision(2);

   BankAccount::ViewAcct();   // display base portion
   cout << "Maximum loan: $" << maxLoan << endl;
   cout << "Owed to bank: $" << owesBank << endl;
   cout.setf(initialState);
}

// redefine how Withdraw() works
void Overdraft::Withdraw(double amt)
{
   // set up ###.## format
   ios::fmtflags initialState =
      cout.setf(ios::fixed, ios::floatfield);
   cout.setf(ios::showpoint);
   cout.precision(2);

   double bal = Balance();
   if (amt <= bal)
  BankAccount::Withdraw(amt);
 else if ( amt <= bal + maxLoan - owesBank)
   {
      double advance = amt - bal;
      owesBank += advance * (1.0 + rate);
      cout << "Bank advance: $" << advance << endl;
      cout << "Finance charge: $" << advance * rate << endl;
      Deposit(advance);
      BankAccount::Withdraw(amt);
   }
   else
      cout << "Credit limit exceeded. Transaction cancelled.\n";
   cout.setf(initialState);
}
```

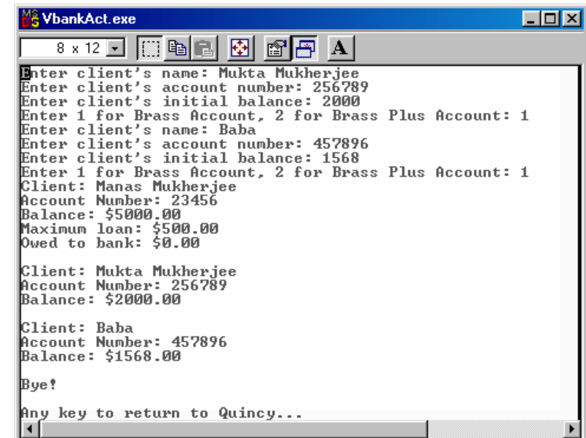
Testing Overdraft class.

```
// useover.cpp -- test Overdraft class
// compile with bankacct.cpp and overdrft.cpp
// virtual version
#include <iostream>
using namespace std;
#include "overdrft.h"
const int SIZE = 3;
const int MAX = 35;
inline void EatLine() {while (cin.get() != '\n') continue;}

int main()
{
  BankAccount * baps[SIZE];
  char name[MAX];
  long acctNum;
  double balance;
  int acctType;
  int i;
  for (i = 0; i < SIZE; i++)
  {
     cout << "Enter client's name: ";
     cin.get(name,MAX);
     EatLine();
     cout << "Enter client's account number: ";
     cin >> acctNum;
     cout << "Enter client's initial balance: ";
     cin >> balance;
     cout << "Enter 1 for Brass Account, 2 for Brass Plus "
         << "Account: ";
     cin >> acctType;
     EatLine();
     if (acctType == 2)
        baps[i] = new Overdraft(name, acctNum, balance);
     else
     {
        baps[i] = new BankAccount(name, acctNum, balance);
        if (acctType != 1)
           cout << "I'll interpret that as a 1.\n";
     }
   }
   for (i = 0; i < SIZE; i++)
   {
      baps[i]->ViewAcct();
      cout << endl;
   }
   cout << "Bye!\n";
   return 0;

}
```

Compiling and linking



Runtime consol

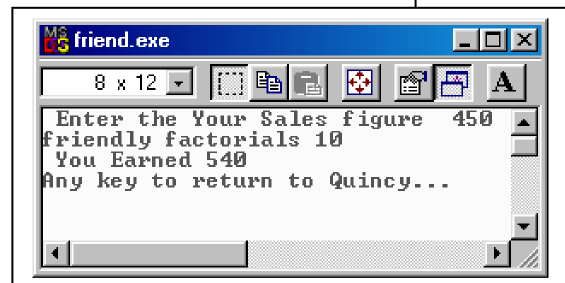


Friend Class

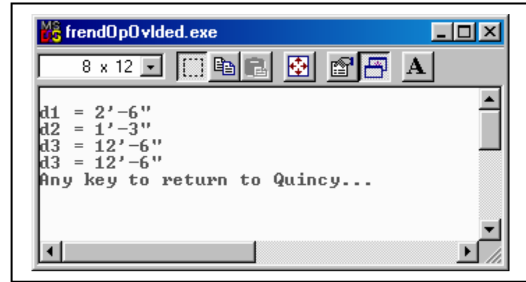
If you want to expose your private member data or function to another class, you must declare that class to be a friend. This extends the interface of your class to include the friend class.

Friendships are not transferable, means your friend can't be my friend, just because you are my friend. And friendship could be one way. Class one is ready to share but class two may not share to class one.

```
// friend.cpp
// friend functions
#include <iostream.h>

class beta;             // needed for frifunc declaration

class alpha
   {
   private:
      int data;
   public:
      alpha()  { data = 3; }           // no-arg constructor
      friend int frifunc(alpha, beta);  // friend function
   };
class beta
   {
   private:
      int data;
   public:
      beta()  { data = 7; }            // no-arg constructor
      friend int frifunc(alpha, beta);  // friend function
   };

int frifunc(alpha a, beta b)           // function definition
   {
   return( a.data + b.data );
   }
void main()
   {
double incentive;
cout << " Enter the Your Sales figure  "  ;
cin >> incentive;
 alpha aa;
 beta bb;
   cout <<"friendly factorials " << frifunc(aa, bb) <<"\n";            // call the function
             incentive = incentive  + ( (incentive * 0.20) * (frifunc(aa,bb))/10 ) ;
             cout << " You Earned " <<  incentive ;
   }
```



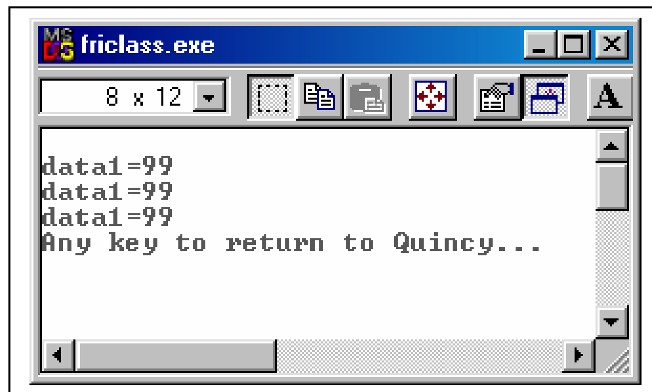
Friend overloaded operator.




```
// frendOpOvlded.cpp
// friend overloaded + operator
#include <iostream.h>
class Distance                    // English Distance class
  {
  private:
    int feet;
    float inches;
  public:
    Distance()                // constructor (no args)
                            { feet = 0; inches = 0.0; }
    Distance( float fltfeet )  // constructor (one arg)
                      {                 // convert float to Distance
                      feet = int(fltfeet);          // feet is integer part
                    inches = 12*(fltfeet-feet);    // inches is what's left
                                        }
    Distance(int ft, float in)  // constructor (two args)
                            { feet = ft; inches = in; }
    void showdist()            // display distance
                      { cout << feet << "\'-" << inches << '\"'; }
    friend Distance operator + (Distance, Distance);  // friend
  };
 // add D1 to d2
Distance operator + (Distance d1, Distance d2)
  {
  int f = d1.feet + d2.feet;        // add the feet
  float i = d1.inches + d2.inches;  // add the inches
  if(i >= 12.0)                // if inches exceeds 12.0,
    { i -= 12.0; f++;  }        // less 12 inches, plus 1 foot
  return Distance(f,i);         // return new Distance with sum
  }
void main()
  {
  Distance d1 = 2.5;              // constructor converts
  Distance d2 = 1.25;              // float-feet to Distance
  Distance d3;
  cout << "\nd1 = "; d1.showdist();
  cout << "\nd2 = "; d2.showdist();
  d3 = d1 + 10.0;               // distance + float: ok
  cout << "\nd3 = "; d3.showdist();
  d3 = 10.0 + d1;               // float + Distance: ok
  cout << "\nd3 = "; d3.showdist();
  }
```

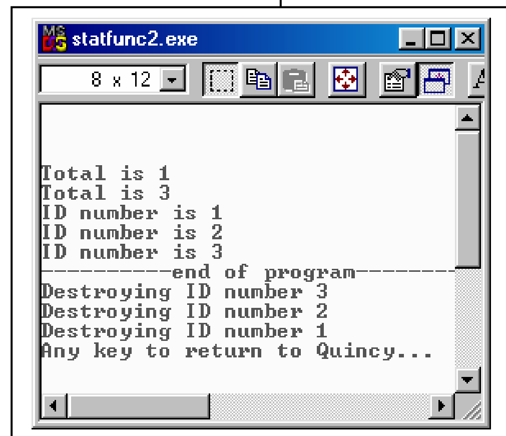
Friend Class

```
// friclass.cpp
// friend classes
#include <iostream.h>

class alpha
  {
  private:
    int data1;
  public:
    alpha()  { data1 = 99; }
    friend class beta;         // beta is a friend class
  };

class beta
  {                            // all member functions can
  public:                      // access private alpha data
    void func1(alpha a)  { cout << "\ndata1=" << a.data1; }
    void func2(alpha a)  { cout << "\ndata1=" << a.data1; }
    void func3(alpha a)  { cout << "\ndata1=" << a.data1; }
  };

void main()
  {
  alpha a;
  beta b;

  b.func1(a);
  b.func2(a);
  b.func3(a);
  }
```

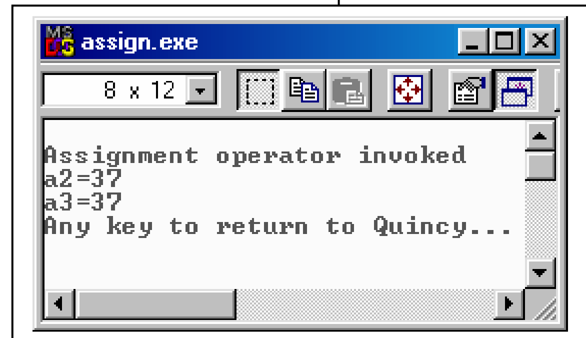


Static Functions

```
// statfunc2.cpp
// static functions and ID numbers for objects
#include <iostream.h>
class gamma
  {
  private:
    static int total;       // total objects of this class
                            //    (declaration only)
    int id;                 // ID number of this object
  public:
    gamma()                 // no-argument constructor
          {
          total++;          // add another object
          id = total;       // id equals current total
          }
    ~gamma()                // destructor
          {
          total--;
          cout << "\nDestroying ID number " << id;
          }
    static void showtotal()  // static function
          {
          cout << "\nTotal is " << total;
          }
    void showid()           // non-static function
          {
          cout << "\nID number is " << id;
          }
  };

int gamma::total = 0;              // definition of total
void main()
  {
  cout << endl << endl;
  gamma g1;
  gamma::showtotal();
  gamma g2, g3;
  gamma::showtotal();
  g1.showid();
  g2.showid();
  g3.showid();
  cout << "\n----------end of program----------";
  }
```



Overloading assignment Operator.




```
// assign.cpp
// overloads assignment operator (=)
#include <iostream.h>

class alpha
  {
  private:
    int data;
  public:
    alpha()                    // no-arg constructor
        { }
    alpha(int d)                // one-arg constructor
        { data = d; }
    void display()             // display data
        { cout << data; }
    alpha operator = (alpha& a)  // overloaded = operator
        {
        data = a.data;           // not done automatically
        cout << "\nAssignment operator invoked";
   return alpha(data);       // return copy of this alpha
        }
  };
void main()
  {
  alpha a1(37);
  alpha a2;
  a2 = a1;                       // invoke overloaded =
  cout << "\na2="; a2.display();  // display a2
  alpha a3 = a2;                 // does NOT invoke =
  cout << "\na3="; a3.display();  // display a3
  }
```

- Passing by reference the argument operator=() is passed by reference. When program is loarge it may take large memory.

alpha operator = (alpha& a) // overloaded = operator

- Returning a value

return alpha(data);   // return copy of this alpha

- Not inherited: assignment operator is unique operatos in that it is not inherited. If you overload the assignment ooperator in a base class, you can't use this same function in any derived class.

A string Counter class : also an example of friend class

```
// strimem.cpp
// memory-saving String class
// overloaded assignment and copy constructor
#include <iostream.h>
#include <string.h>              // for strcpy(), etc.
#include <conio.h>               // for getch()

class strCount                   // keep track of number
  {                              // of unique strings
  private:
    int count;                   // number of instances
    char* str;                   // pointer to string
    friend class String;         // make ourselves available
  public:
    strCount(char* s)            // one-arg constructor
 {
 int length = strlen(s);  // length of string argument
 str = new char[length+1];  // get memory for string
 strcpy(str, s);          // copy argument to it
 count=1;                 // start count at 1
 }
   ~strCount()                   // destructor
 {
 delete[] str;            // delete the string
 }
  };

class String                     // String class
  {
  private:
    strCount* psc;               // pointer to strCount
  public:
    String()                     // no-arg constructor
 {
 psc = new strCount("NULL");
 }
    String(char* s)              // 1-arg constructor
 {
 psc = new strCount(s);
 }
    String(String& S)            // copy constructor
```

```
{
psc = S.psc;
        (psc->count)++;
}
    ~String()                   // destructor
{
if(psc->count==1)    // if we are its last user,
 delete psc;     //delete our strCount
else  //otherwise,
  (psc->count)--;     //decrement its count
}
void display()                                    // display the String
{
cout << psc->str;                 // print string
cout << " (addr=" << psc << ")";  // print address
                                                  }
    void operator = (String& S) // assign the string
{
if(psc->count==1)                        // if we are its last user,
   delete psc;   //                            delete our strCount
else               //otherwise,
(psc->count)--;       //decrement its count
psc = S.psc;            // use argument's strCount
(psc->count)++;    // increment its count
}
  };

void main()
  {
  String s3 = "When the fox preaches, look to your geese.";
  cout << "\ns3="; s3.display();   // display s3

  String s1;                      // define String
  s1 = s3;                        // assign it another String
  cout << "\ns1="; s1.display();   // display it

  String s2(s3);                  // initialize with String
  cout << "\ns2="; s2.display();   // display it
  getch();
  }
```

Reference point

Shape

```
//Listing 13.7. Shape classes.

#include <iostream.h>

class Shape
{
public:
Shape(){}
virtual ~Shape(){}
virtual long GetArea() { return -1; } // error
virtual long GetPerim() { return -1; }
virtual void Draw() {}
private:
};

class Circle : public Shape
{
public:
Circle(int radius):itsRadius(radius){}
~Circle(){}
long GetArea() { return 3 * itsRadius * itsRadius; }
long GetPerim() { return 6 * itsRadius; }
void Draw();
private:
int itsRadius;
int itsCircumference;
};

void Circle::Draw()
{
cout << "Circle drawing routine here!\n";
}
class Rectangle : public Shape
{
public:
Rectangle(int len, int width):
itsLength(len), itsWidth(width){}
virtual ~Rectangle(){}
virtual long GetArea() { return itsLength * itsWidth; }
virtual long GetPerim() {return 2*itsLength + 2*itsWidth; }
virtual int GetLength() { return itsLength; }
virtual int GetWidth() { return itsWidth; }
virtual void Draw();
```

```
private:
int itsWidth;
int itsLength;
};
void Rectangle::Draw()
{
for (int i = 0; i<itsLength; i++)
{
for (int j = 0; j<itsWidth; j++)
cout << "x ";
cout << "\n";
}
}

class Square : public Rectangle
{
public:
Square(int len);
Square(int len, int width);
~Square(){}
long GetPerim() {return 4 * GetLength();}
};

Square::Square(int len):
Rectangle(len,len)
{}

Square::Square(int len, int width):
Rectangle(len,width)
{
if (GetLength() != GetWidth())
cout << "Error, not a square... a Rectangle??\n";
}

int main()
{
int choice;
bool fQuit = false;
Shape * sp;
while ( !fQuit )
{
cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
cin >> choice;
switch (choice)
{
case 0: fQuit = true;
```

```
break;
case 1: sp = new Circle(5);
break;
case 2: sp = new Rectangle(4,6);
break;
case 3: sp = new Square(5);
break;
default: cout << "Please enter a number between 0 and 3" << endl;
continue;
break;
}
if( !fQuit )
sp->Draw();
delete sp;
sp = 0;
cout << "\n";
}
return 0;
}
```