

## Unit 3

# C++ Language II

## Object Oriented Programming in C++

Takayuki Dan Kimura

- \* Unbounded Queue
- \* Extension of Queue Data Type
- \* Static Members
- \* Base and Derived Classes (Inheritance)
- \* Virtual Functions
- \* Polymorphism
- \* Dynamic Binding
- \* Pure Virtual Function and Abstract Class
- \* SortableObject (Example of Abstract Class)

## Unbounded (integer) Queue Dynamic Data Structure (Linked List)

---

```
class Cell
{
    friend class Queue;

    Cell( int v, Cell *p = 0, Cell *n = 0 )
        { value = v; prev = p; next = n; };

    int    value;
    Cell   *prev;
    Cell   *next;
};

class Queue
{
public:
    Queue( ) { head = tail = 0; };
    ~Queue( );

    void    enq( int x );
    int     deq();
    int     top();
    void    show( char *s );
private:
    Cell *head;
    Cell *tail;
};
```

### Program 1

#### *friend class*

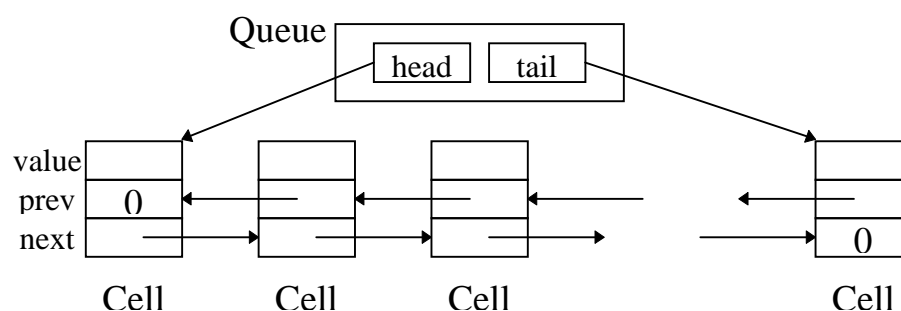
= the class that has direct access  
to the private data

ex. **Queue** (and no one else) has  
direct access to the private part of  
**Cell**

i.e., member functions of Queue has  
access to **value**, **prev**, and **next**  
of Cell class

Note:

- 1) **Cell** has no public part.
- 2) **Cell** is recursively defined.
- 3) Cell is bidirectional,  
doubly-linked list.



```
#include <iostream.h>
#include "queue2.h"

Queue::~Queue() {
    Cell *pt = head;
    Cell *tmp;

    while ( pt ) {
        tmp = pt->next;
        delete pt;
        pt = tmp; }

};

void Queue::enq( int x ) {
    Cell *pt = new Cell( x, tail );
    tail->next = pt;
    tail = pt;
    if ( head == 0 ) head = pt;
};

int Queue::deq() {
    if ( head == 0 ) {
        cout << "underflow" << endl;
        return( 0 ); }
    else {
        int v = head->value;
        if ( head->next == 0 ) {
            tail = 0;
            delete head;
            head = 0; }
        else {
            head = head->next;
            delete head->prev;
            head->prev = 0; };
        return( v );
    }
};
```

### Program 1 (2)

```
int Queue::top()
{
    if ( head == 0 ) {
        cout << "underflow" << endl;
        return( 0 ); }
    else { return( head->value ); };
};

void Queue::show( char *s ) {
    Cell *pt = head;
    Cell *tmp;

    cout << s;
    while ( pt ) {
        tmp = pt->next;
        cout << ' ' << pt->value;
        pt = tmp; };
    cout << endl;
};
```

### Program 1(3)

## New & Delete Operator

**New** = **malloc** with  
typed pointer +  
constructor call

**Delete** = **free** +  
destructor call

A \*a;    a = **new** A;  
         **delete** a;

B \*b;    b = new A; type violation!

## Problem: Queue assignment and equality?

```
#include "queue3.h"

void main()
{
    Queue q1, q2;
    Queue empty;

    q1.enq(9);
    q1.enq(8);
    q1.enq(7);

    q2 = q1;
    q1.show( "q1: " );    // q1: 9 8 7
    q2.show( "q2: " );    // q2: 9 8 7

    q1.enq( 6 );
    q2.deq();
    q1.show( "q1: " );    // q1: 9 8 7 6
    q2.show( "q2: " );    // q2: 8 7

    Queue q3 ( q1 );
    Queue q4 = q2;

    q3.show( "q3: " );    // q3: 9 8 7 6
    q4.show( "q4: " );    // q4: 8 7

    if ( q1 != q2 ) q1 = q2 = empty;

    q1.show( "q1: " );    // q1:
    q2.show( "q2: " );    // q2:

    if ( q1 == q2 ) q2.enq( 5 );

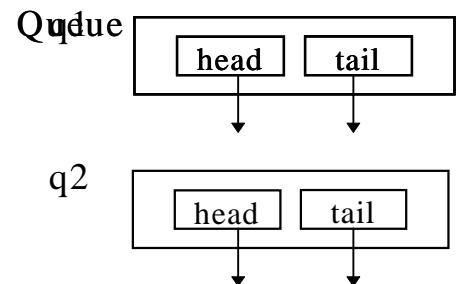
    q1.show( "q1: " );    // q1:
    q2.show( "q2: " );    // q2: 5
}
```

### Program 2(1)

### How to define **queue3.h** ?

#### 1) Assignment operation

**q2 = q1**



*Copying list structure is necessary!*

#### 2) Initialization operation

**Queue q4 = q2**

#### 3) Equality operation

**q1 == q2**

#### 4) Non-equality operation

**q1 != q2**

#### 5) Multiple assignment

**q1 = q2 = empty**

## Solution

---

```
class Cell { ... };

class Queue {
public:
    Queue() { head = tail = 0; };
    ~Queue();

    void   enq( int x );
    int    deq();
    int    top() const;
    void   show( char *s );

    Queue &operator=( const Queue &q );
    int    operator==( const Queue &q );
    int    operator!=( const Queue &q );

    Queue( const Queue &q );
        //copy constructor
private:
    void remove();

    Cell *head;
    Cell *tail;
};
```

### Program 2(2): Queue3.h

```
void Queue::remove() {
    Cell *pt = head;    Cell *tmp;
    while ( pt ) {
        tmp = pt->next;
        delete pt;
        pt = tmp; }; head = tail = 0; };

Queue::~Queue() { remove(); };
```

### Remove function

```
Queue &Queue::operator=( const Queue &q )
{
    if ( &q == this ) return( *this );
    remove();

    Cell *pt = q.head;
    while ( pt ) {
        enq( pt->value );    // copy
        pt = pt->next; };

    return( *this ); };
```

### Assignment operation

- Infix operators  
‘=’ (‘==’, ‘!=’) is a member function of q1 in ‘q1 = q2’, i.e., ‘q1.=( q2 )’.
- Operator overloading  
Differentiated by parameter types.
- The **this** pointer  
= a special pointer accessible to member functions. It points to the object for which the function is called.  
  
Used for multiple assignment:  
‘q1 = q2 = empty’ is same as  
‘q1.=( q2.=( empty ) )’

```
int Queue::operator==( const Queue &q ) {  
    // if ( this == &q ) return( 1 );  
  
    Cell *pt1 = head;  
    Cell *pt2 = q.head;  
  
    while ( pt1 ) {  
        if ( pt2 == 0 || pt1->value != pt2->value )  
            return( 0 );  
        pt1 = pt1->next;  
        pt2 = pt2->next; }  
  
    if ( pt2 == 0 ) return( 1 );  
    return ( 0 ); }  
  
int Queue::operator!=( const Queue &q ) {  
    if ( this == &q ) return ( 0 );  
    return( 1 ); };
```

### Equality Operation

- Pair-wise comparison is necessary for equality

Without overloading,

q1 == q2

=

q1.head == q2.head &  
q1.tail == q2.tail

but, it should be

=

q1.head->value  
== q2.head->value  
&  
q1.head->next->value  
== q2.head->next->value  
& ... etc.

```
Queue::Queue( const Queue &q ){  
    head = tail =0;  
  
    Cell *pt = q.head;  
    while ( pt ) {  
        enq( pt->value );  
        pt = pt->next; } };
```

### Copy Constructor

- Copy Constructor  
= takes an object of the same type as an argument

No need of **remove()**.

Copy constructor is needed for parameter passing

ex. queue ff( queue xx ) { ...; queue yy; ... return yy; };  
... q2 = ff( q1 ); ...

queue xx( q1 ); ...; queue temp( yy ); ...; q2 = temp;

## How about arithmetic on Queues?

$$q3 = q1 + q2$$

---

```
#include "queue4.h"
void main() { Queue    q1, q2, q3, q4;
    q1.enq(9);
    q1.enq(8);
    q1.enq(7);
    q1.enq(6);
    q2.enq(5);
    q2.enq(4);
    q2.enq(3);

    q3 = q1 + q2;
    q1.show( "q1: " ); // q1: 9 8 7 6
    q2.show( "q2: " ); // q2: 5 4 3
    q3.show( "q3: " ); // q3: 9 8 7 6 5 4 3

    q4 = q2 + q1;
    q4.show( "q4: " ); // q4: 5 4 3 9 8 7 6
}
```

### Program 3

```
Queue operator+(
    const Queue &q1, const Queue &q2 )
{
    Queue t1;    Queue t2;
    t1 = q1;     t2 = q2;

    t1.tail->next = t2.head;
    t2.head->prev = t1.tail;
    t1.tail = t2.tail;
    t2.head = t2.tail = 0;
    return( t1 );
}
```

### Append Operation

```
class Cell {
    friend class Queue;
    friend Queue operator+(
        const Queue &q1,
        const Queue &q2 );
    ... };

class Queue
{
public:
    ...
    friend Queue operator+(
        const Queue &q1,
        const Queue &q2 );
private:
    ...
};

queue4.h
```

### *Friend function*

= the function that has direct access to the private data

The (overloaded) operator + is **not** a member function of Cell or Queue, but has access to there private parts.

Note: *In  $q1 + q2$ , + is not a property of either  $q1$  or  $q2$ .*

## Static Members

---

```
class A {
public:
    A() { count++; }; // after each creation
    ~A() { count--; }; // after each deletion
    static int howMany()
        { return count; };
private:
    static int count;
};

int A::count = 0;

void main() {
    cout << A::howMany() << endl;    // 0

    A *a[10];

    for ( int i = 0; i < 5; i++ )
        a[i] = new A;
    cout << a[0].howMany() << endl;    // 5

    delete a[1];    delete a[3];
    cout << a[2].howMany() << endl;    // 3

    for ( i = 0; i < 5; i++ )
        a[i] = new A;
    cout << A::howMany() << endl;    // 8
};
```

### Program 4: Counting

- **Static Data Member**
  - = only one copy of it is allocated, no matter how many instances of the class is declared
  - \* Ex. count
  - \* Must be initialized at file scope
  - \* Class variables vs instance variables
- **Static Member Function**
  - = have access to only static data members of the class
  - \* Ex. howMany
  - \* No **this** pointer
  - \* Can be access either
    - globally A::howMany()
    - or
    - locally a[0].howMany()



### Another Example:

```
class SavingsAccount {
public:
    SavingsAccount( const char *nm, float init ) { strncpy( name, nm, 30 ); balance = init; };
    void deposit( float amount ) { balance += amount; };
    void withdraw( float amount ) { balance -= amount; };
    float currentBalance() { return balance; };
    void earnInterest() { balance += currentRate * balance; };

    static float currentRate;
    // cannot be initialized here
private:
    char name[30];
    float balance; };

float SavingsAccount::currentRate = 0.001;
    // static data members must be initialized at file scope. This is not an assignment

void main() {
    SavingsAccount::currentRate = 0.00525;

    SavingsAccount a1( "Home", 1000.0 );
    SavingsAccount a2( "Business", 1000.0 );

    a1.earnInterest(); a2.earnInterest();
    cout << "Home: \t" << a1.currentBalance() << endl; // Home: 1005.25
    cout << "Business:\t" << a2.currentBalance() << endl << endl; // Business: 1005.25

    a1.currentRate = 0.006; // misleading syntax

    a1.earnInterest(); a2.earnInterest();
    cout << "Home: \t" << a1.currentBalance() << endl; // Home: 1011.28
    cout << "Business:\t" << a2.currentBalance() << endl << endl; // Business: 1011.28

    SavingsAccount::currentRate = 0.0065; // better syntax

    a1.earnInterest(); a2.earnInterest();
    cout << "Home: \t" << a1.currentBalance() << endl; // Home: 1017.85
    cout << "Business:\t" << a2.currentBalance() << endl << endl; }; // Business: 1017.85
```

## Program 5

## Base and Derived Classes

---

```
class employee {  
    char name[30];  
public:  
    employee(){ *name = 0; };  
    employee( char *nm ) { strncpy( name, nm, 30 ); };  
    ~employee(){};  
  
    char *getName() const { return name; };  
    void setName( char *nm ) { strncpy( name, nm, 30 ); };  
  
    virtual void show() { cout << name << endl; };  
    virtual float pay() { return 0; };  
};
```

**Base: Employee (employee.h)**

```
#include "employee.h"  
  
class partTimer : public employee {  
    float wage;  
    float hours;  
public:  
    partTimer( char *nm,  
               const float wg = 5.25,  
               const float hrs = 0 )  
        : employee( nm )  
        { wage = wg; hours = hrs; };  
  
    partTimer() : employee() { };  
    ~partTimer() { };  
  
    void setWage( float wg ) { wage = wg; };  
    void setHours( float hrs ) { hours = hrs; };  
  
    void show();  
    float pay();  
};
```

**Derived: partTimer (ptime.h)**

```
#include "employee.h"  
  
class fullTimer : public employee {  
    float salary;  
public:  
    fullTimer( char *nm,  
               const float s = 0.0 )  
        : employee( nm )  
        { salary = s; };  
  
    fullTimer() : employee() { };  
    ~fullTimer() { };  
  
    void setSalary( float s ) { salary = s; };  
  
    void show();  
    float pay();  
};
```

**Derived: fullTimer**

*Base Initializer* (calling the constructor of the base class)

```
#include "ptime.h"

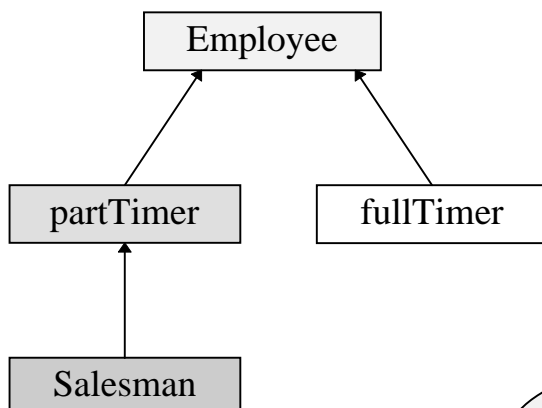
class salesman : public partTimer
{
    float    commission;
    float    sales;
public:
    salesman( char *nm,
              const float wg = 5.25,
              const float hrs = 0.0,
              const float cm = 0.15,
              const float s = 0.0 )
    : partTimer( nm, wg, hrs )
    { commission = cm; sales = s; };

    salesman() : partTimer() { };
    ~salesman() { };

    void setCommission( float cm )
        { commission = cm; };
    void setSales( float s ) { sales = s; };

    void show();
    float pay();
}
```

### Derived: Salesman



Class Hierarchy

```
class salesman
{
    char    name[30];

    float    wage;
    float    hours;

    float    commission;
    float    sales;
public:
    salesman( char *nm,
              const float wg = 5.25,
              const float hrs = 0.0,
              const float cm = 0.15,
              const float s = 0.0)

        { strncpy( name, nm, 30 ); }

        { wage = wg; hours = hrs; }

        { commission = cm; sales = s; }

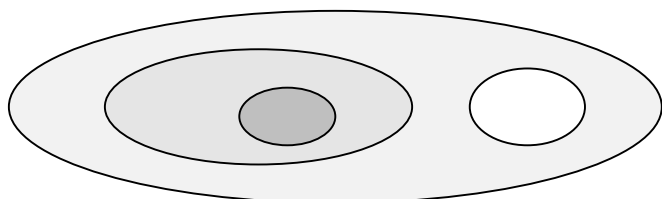
    char *getName() const { ... };
    void setName( char *nm ) { ... };

    void setWage( float wg ) { ... };
    void setHours( float hrs ) { ... };

    void setCommission( float cm ) { ... };
    void setSales( float s ) { ... };

    void show();
    float pay();
}
```

### Concatenation of three classes



## Definitions

---

- Superclass (Base) and Subclass (Derived)
  - \* **employee** *is the superclass of partTimer*
  - \* **partTimer** *is a subclass of employee*
  - \* **partTimer** *is the superclass of Salesman*
- The member functions of a derived class do not have access to the private members of its base class.

ex. Salesman::setCommission() does not have access to partTimer::wage.

Otherwise, the privacy of the base class can be compromised by any derived class.
- *Public Base Class*
  - \* **employee** is the *public base class* of **partTimer**

ex. class **partTimer** : **public** **employee**

= the **public** members of the base class (employee) are **public** members of the derived class (partTimer)

[the **protected** members of the base class are **protected** members of the derived class]
  - \* **employee** could be the *private base class* of **partTimer**

with class **partTimer** : **private** **employee**

= the **public** members of the base class are **private** members of the derived class

[the **protected** members of the base class are also **private** members of the derived class]

## Redefinition of Base Class Members

---

```
class employee {  
    ...  
    virtual void    show() { cout << name << endl; };  
    virtual float   pay() { return 0; };  
    ... };
```

```
void partTimer::show() {  
    cout << getName()  
    << "   wage: " << wage  
    << "   hours: " << hours  
    << endl; };
```

```
float partTimer::pay() {  
    return( wage * hours ); };
```

```
void fullTimer::show() {  
    cout << getName()  
    << "   salary: " << salary  
    << endl; };
```

```
float fullTimer::pay() {  
    return( salary ); };
```

```
void salesman::show() {  
    partTimer::show();  
    cout << "   commission: " << commission  
    << "   sales: " << sales  
    << endl; }
```

```
float salesman::pay() {  
    return( partTimer::pay()  
           + commission * sales ); }
```

## *Virtual Function*

= a member function that is expected to be redefined in derived classes

- When it is called **through a pointer** to a base class, the derived class's version of the function is executed.
- All subsequent versions of a virtual function are implicitly declared as **virtual**.

```
void main() {
    employee payroll[5] = {
        employee( "John Smith" ),
        employee( "Amy Jones" ) };
}
```

```
int i = 0;
while ( *payroll[i].getName() != 0 && i < 5 ) {
    payroll[i].show();
    cout << "paid: " << payroll[i].pay() << endl;
    i++; }; }
```

John Smith  
Amy Jones

```
void main() {
    fullTimer payroll[5] = {
        fullTimer( "Mary Wilson", 650.0 ),
        fullTimer( "Dave Johnson", 530.0 ),
        fullTimer( "Mike Bennett", 700.0 ) };
}
```

```
int i = 0;
while ( *payroll[i].getName() != 0 && i < 5 ) {
    payroll[i].show();
    cout << "paid: " << payroll[i].pay() << endl;
    i++; }; }
```

Mary Wilson salary: 650  
paid: 650  
Dave Johnson salary: 530  
paid: 530  
Mike Bennett salary: 700  
paid: 700

```
void main() {
    partTimer payroll[5] = {
        partTimer( "John Smith", 6.0, 35.0 ),
        partTimer( "Amy Jones", 7.5, 28.5 ) };
}
```

```
int i = 0;
while ( *payroll[i].getName() != 0 && i < 5 ) {
    payroll[i].show();
    cout << "paid: " << payroll[i].pay() << endl;
    i++; }; }
```

John Smith wage: 6 hours: 35  
paid: 210  
Amy Jones wage: 7.5 hours: 28.5  
paid: 213.75

```
void main(){
    salesman payroll[5] = {
        salesman( "Susan Fuller", 5.5, 36.0, 0.2, 245.0 ),
        salesman( "Edward Levy", 6.2, 27.0, 0.25, 120.0 ) };
}
```

```
int i = 0;
while ( *payroll[i].getName() != 0 && i < 5 ) {
    payroll[i].show();
    cout << "paid: " << payroll[i].pay() << endl;
    i++; }; }
```

Susan Fuller wage: 5.5 hours: 36  
commission: 0.2 sales: 245  
paid: 247  
Edward Levy wage: 6.2 hours: 27  
commission: 0.25 sales: 120  
paid: 197.4

# Polymorphism

- = The capability to call member functions for an object without specifying the object's exact type
- = The ability to assume many forms

```
void main() {
    partTimer smith( "John Smith", 6.0, 35.0 );
    salesman levy( "Edward Levy", 6.2, 27.0, 0.25, 120.0 );
    partTimer jones( "Amy Jones", 7.5, 28.5 );
    fullTimer wilson( "Mary Wilson", 650.0 );
    salesman fuller( "Susan Fuller", 5.5, 36.0, 0.2, 245.0 );
```

```
employee *payroll[5] = {
    &smith, &levy, &jones, &wilson, &fuller };

```

```
int i = 0;
while ( *payroll[i]->getName() != 0 && i < 5 ){
    payroll[i] ->show();
    cout << "paid: " << payroll[i]->pay() << endl << endl;
    i++; }; }
```

## Program 6

John Smith wage: 6 hours: 35  
paid: 210

Edward Levy wage: 6.2 hours: 27  
commission: 0.25 sales: 120  
paid: 197.4

Amy Jones wage: 7.5 hours: 28.5  
paid: 213.75

Mary Wilson salary: 650  
paid: 650

Susan Fuller wage: 5.5 hours: 36  
commission: 0.2 sales: 245  
paid: 247

```
void main() {
    ...
    employee payroll[5] = {
        smith, levy, jones, wilson, fuller };

```

```
int i = 0;
while ( *payroll[i].getName() != 0 && i < 5 ){
    payroll[i].show();
    cout << "paid: " << payroll[i].pay() << endl << endl;
    i++; }; }
```

payroll[i].show() = emploryee::show()

John Smith  
paid: 0

Edward Levy  
paid: 0

Amy Jones  
paid: 0

Mary Wilson  
paid: 0

Susan Fuller  
paid: 0

## *Dynamic Binding*

---

= Run-time evaluation of a function call statement to decide what type of object the function belongs to.  
(*late binding*)

ex.    payroll[0].show() ⇒ partTimer::show()  
       payroll[1].show() ⇒ Salesman::show()

- To modify the behavior of program that has already been compiled.
- To enhance code **reusability** and **maintainability**
- To add new types without having to modify the source and recompile it.



```
int i = 0;
while ( *payroll[i]->getName() != 0 && i < 5 ){
    payroll[i] ->show();
    cout << "paid: " << payroll[i]->pay() << endl << endl;
    i++; }; }
```

Precompiled

**process()**

```
#include "ftime.h"
class manager : public fullTimer {
    float    bonus;
public:
    manager( char *nm,
             const float s = 0.0,
             const float b = 0.0)
        : fullTimer( nm, s )
        { bonus = b; };
    manager() : fullTimer() { };
    ~manager() {};

    void setBonus( float b )
        { bonus = b; };
    void show();
    float pay(); }
```

**manager.h**

```
#include "manager.h"

void manager::show() {
    fullTimer::show();
    cout << "    bonus: " << bonus
    << endl; }

float manager::pay() {
    return( fullTimer::pay()
           + bonus ); }
```

**manager.cpp**

Newly Compiled into the library

```
void main() {
    partTimer  smith( "John Smith", 6.0, 35.0 );
    salesman   levy( "Edward Levy", 6.2, 27.0, 0.25, 120.0 );
    partTimer  jones( "Amy Jones", 7.5, 28.5 );
    fullTimer  wilson( "Mary Wilson", 650.0 );
    salesman   fuller( "Susan Fuller", 5.5, 36.0, 0.2, 245.0 );
    manager    johnson( "Dave Johnson", 800.0, 150.0 );

    employee   *payroll[6] = {
        &smith, &levy, &jones, &wilson, &fuller, &johnson };
    process(); }
```

**New Application Program**

## *Pure Virtual Function*

---

= A virtual function without definition and to be defined in all derived classes

To provide a polymorphic interface for the derived classes. ( = Procedure variables )

## *Abstract Class*

= A class containing a pure virtual function as a member function.

ex.

```
class employee {  
    ...  
    virtual void show() { cout << name << endl; };  
    virtual float pay() = 0;  
    ... };
```

**employee** is an abstract class, and  
employee.pay() is a pure virtual function.

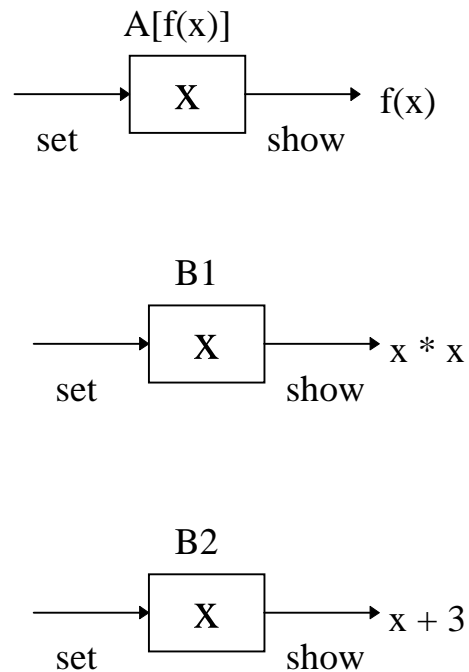
- No instantiation (no declaration) is allowed.
- Declaration of a pointer type is allowed
- If a derived class does not define a pure virtual function, the function is inherited as pure virtual, and the derived class becomes an abstract class.

## Function Variable

---

```
class A {  
public:  
    virtual int f( int x ) = 0;  
  
    void show() { cout << f( x ); }  
    void set( int a ) { x = a; }  
private:  
    int    x; };  
  
class B1 : public A {  
public:  
    int f( int x ) { return( x * x ); }; };  
  
class B2 : public A {  
public:  
    int f( int x ) { return( x + 3 ); }; };  
  
void main() {  
    B1    * b1 = new B1;  
    B2    * b2 = new B2;  
  
    b1->set( 3 );  
    b2->set( 5 );  
  
    b1->show(); // 9 16  
    b2->show(); // 8 9  
};
```

### Program 7



## SortableObject (Example of Abstract Class)

---

```
class Object{
public:
    Object(){};
    virtual void show() = 0; };

class SortableObject : public Object {
public:
    virtual int less( SortableObject *other ) = 0; };

class Cell {
    friend class List;
    friend class SortableList;

    Cell( Object *v, Cell *n = 0 )
        { value = v; next = n; };

    Object *value;
    Cell *next; };

class List {
    friend class SortableList;
public:
    List() { head = 0; size = 0; };
    ~List();
    void add( Object *item ) {
        Cell *pt = new Cell( item, head );
        head = pt; size++; };
    void reset();
    void show();
    int length() { return size; };
private:
    void remove();
    Cell *head;
    int size; };

class SortableList : public List {
public:
    SortableList() { };
    void sort();
};
```

```
void List::remove() {
    Cell *pt = head;
    Cell *tmp;

    while ( pt ) {
        tmp = pt->next;
        delete pt;
        pt = tmp;  };
    head = 0;  };

List::~~List() { remove(); };

void List::reset() { remove(); };

void List::show() {
    Cell *pt = head;

    while ( pt ) {
        (pt->value)->show();
        pt = pt->next;  };  };

void SortableList::sort() {
    Cell *pt = head;
    Cell *last = 0;
    while ( last != head ) {
        pt = head;
        while ( pt->next != last ) {
            Cell *tmp = pt->next;
            SortableObject
                *a = (SortableObject *) pt->value;
            SortableObject
                *b = (SortableObject *) tmp->value;
            if ( a->less( b ) ) {
                Object *t = pt->value;
                pt->value = tmp->value;
                tmp->value = t;  };
            pt = tmp;  };
        last = pt;  };  };
```

```
#include "sort.h"
#include "ftime.h"
#include "sales.h"

class PaidEmployee : public SortableObject {
public:
    PaidEmployee( employee *e ) { worker = e; };

    int less( SortableObject *other ) {
        PaidEmployee *p = (PaidEmployee *) other;
        return( worker->pay() < p->worker->pay() ); };

    void show() { worker->show();
        cout << "pay: " << worker->pay() << endl << endl; };
private:
    employee      *worker; };

void main() {
    SortableList  Y;

    partTimer      smith( "John Smith", 6.0, 35.0 );
    salesman        levy( "Edward Levy", 6.2, 27.0, 0.25, 120.0 );
    partTimer       jones( "Amy Jones", 7.5, 28.5 );
    fullTimer       wilson( "Mary Wilson", 650.0 );
    salesman        fuller( "Susan Fuller", 5.5, 36.0, 0.2, 245.0 );

    PaidEmployee payroll[5] = {
        PaidEmployee( &smith ),
        PaidEmployee( &levy ),
        PaidEmployee( &jones ),
        PaidEmployee( &wilson ),
        PaidEmployee( &fuller ) };

    for ( int i = 0; i < 5; i++ ) Y.add( &payroll[i] );

    Y.show();
    Y.sort(); cout << "----- after sorting " << endl << endl;
    Y.show();
};
```

### Program 8

Susan Fuller wage: 5.5 hours: 36  
commission: 0.2 sales: 245  
pay: 247

Mary Wilson salary: 650  
pay: 650

Amy Jones wage: 7.5 hours: 28.5  
pay: 213.75

Edward Levy wage: 6.2 hours: 27  
commission: 0.25 sales: 120  
pay: 197.4

John Smith wage: 6 hours: 35  
pay: 210

----- after sorting

Mary Wilson salary: 650  
pay: 650

Susan Fuller wage: 5.5 hours: 36  
commission: 0.2 sales: 245  
pay: 247

Amy Jones wage: 7.5 hours: 28.5  
pay: 213.75

John Smith wage: 6 hours: 35  
pay: 210

Edward Levy wage: 6.2 hours: 27  
commission: 0.25 sales: 120  
pay: 197.4

## Outputs