Object-Oriented Programming

- Object-oriented programming - many definitions
  - exploitation of class objects, with private data members and associated access functions (cf. concept of an abstract data type)
  - However, Ellis and Stroustrup give a more limited meaning:

    ‘The use of derived classes and virtual functions is often called object-oriented programming’

    So, we need some more C++!

Object-Oriented Programming

Interface, Implementation, and Application Files

- Preferred practice for programs dealing with classes: 3 files
  - Interface
    - » between implementation and application
    - » Header File that declares the class type
    - » Functions are declared, not defined (except inline functions)
  - Implementation
    - » `#includes` the interface file
    - » contains the function definitions
  - Application ...

Object-Oriented Programming

Interface, Implementation, and Application Files

- Preferred practice for programs dealing with classes: 3 files
  - Interface
  - Implementation
  - Application
    - » `#includes` the interface file
    - » contains other (application) functions, including the `main` function

Object-Oriented Programming

Interface, Implementation, and Application Files

- When writing an application, we are class users
  - don’t want to know about the implementation of the class (c.f ADTs)
  - Thus, the interface must furnish all the necessary information to use the class
  - Also, the implementation should be quite general (cf. reusability)

Object-Oriented Programming
A Class for Sets

- Required data type: sets of integers
- Required functions:
  - declaration, e.g.
    `iset S, T=1000, U=T, V(1000);`
    - » S should be empty
    - » `T, U,` and V should contain just one element (1000)
  - adding an element, e.g.
    `S += x`
  - removing an element, e.g.
    `S -= x`
    - » must be valid even if x is not an element of S

Object-Oriented Programming
A Class for Sets

- Required data type: sets of integers
- Required functions:
  - Test if an element is included in the set, e.g.
    `if (S(x)) ... // is x in set X`
  - Display all elements in increasing order, e.g.
    `S.print();`
  - Set assignment, e.g.
    `T = S`

Object-Oriented Programming
A Class for Sets

- Required data type: sets of integers
- Required functions:
  - Inquiring about the number of elements by converting to type int, e.g.
    ```
    cout << “S contains “ << int(S) << “elements”;
    ```
  - Inquiring if the number of elements in the set is zero, e.g.
    ```
    if (S==0) ....
    if (!S) ....
    ```

Object-Oriented Programming

A Class for Sets

- Note that we have NOT specified how the set is to be represented or implemented
  (again, cf. ADTs)

Object-Oriented Programming

A Class for Sets - Application

```
// SETAPPL:  Demonstration program for set operations
//           (application; file name: setappl.cpp)

#include “iset.h”

int main()
{  iset S=1000, T, U=S;
   if (!T) cout << “T is empty.\n”;
   if int(U) cout << “U is not empty.\n”;
   S += 100;  S += 10000;
   (((s += 10) +=1) += 20) += 200;
   cout << “There are “ << int(S) << “elements in S\n”;
   T += 50;  T += 50;
   cout << “S: “; S.print();
   S -= 1000; cout <<“1000 removed from S\n”;
   if (S(1000))
      cout << “1000 belongs to S (error)\n”;
   else
      cout << “1000 is no longer in S\n”);
```

Object-Oriented Programming

A Class for Sets - Application

```
    if (S(100))
       cout << “100 still belongs to s\n”;
    cout << “S: “; S.print();
    cout << “T: “; T.print();
    cout << “U: “; U.print();
    T = S;
    cout << “After assigning S to T, we have T: “;
    T.print();
    return 0;
}
```

Object-Oriented Programming

A Class for Sets - Application

Expected output

```
T is empty
U is not empty
There are 7 elements in S
S: 1 10 20 100 200 1000 10000
1000 removed from S
1000 is no longer in S
100 still belongs to S
S: 1 10 20 100 200 10000
T: 50
U: 1000
After assigning S to T, we have T:1 10 20 100 200 10000
```

Object-Oriented Programming
A Class for Sets - Application

```
// ISET.H:  Header file for set operations
//          (interface; file name: iset.h)

#include “iostream.h”

class iset {
public:
   iset()                         // constructor to begin
   {  a = NULL;                   // with empty set
      n = 0;
   }
   iset(int x)                    // constructor to begin
   {  a = NULL;                   // with one element x
      *this += x;
      n = 1;
   }
   ~iset()                        // destructor
   {  delete[] a;
   }
```

Object-Oriented Programming

A Class for Sets - Application

```
    iset &operator+=(int x); // adds x to the set
    iset &operator-=(int x); // removes x from the set
    int operator()(int x) const; // is x in the set?
    void print() const;      // prints all elements of
                             // the set on one line
    iset &operator=(iset S); // assignment operator
    iset (const iset &S);    // copy constructor NB
    operator int()           // convert iset to int
    {  return n;
    }
private:
    int n, *a;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
// ISET:  Implementation file for set operations;
//        (implmentation; file name: iset.cpp)
// Note:
// using a relatively inefficient array representation

#include “iset.h”

const int blocksize=5;  // may be replaced with larger
                        // value
```

Object-Oriented Programming

A Class for Sets - Application

```
static int *memoryspace(int *p0, int n0, int n1)

/* if p0 == NULL, allocate an area for n1 integers   */
/* if p0 != NULL, increase or decrease old sequence  */
/* p0[0], ... , p[n0-1]                              */
/* in either case, the resulting new sequence is     */
/* p1[0], ..., p1[n1-1], and p1 is returned          */

{  int *p1 = new int[n1];
   if (p0 != NULL) // copy from p0 to p1:
   {  for (int i=(n0<n1?n0:n1)-1; i>=0; i--)
         p1[i] = p0[i];
      delete p0;
   }
   return p1;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
int binsearch(int x, int *a, int n)

/* The array a[0], ... , a[n-1] is searched for x
   Return value:
      0 if n == 0 or x <= a[0]
      n if x > a[n-1]
      i if a[i-1] < x <= a[i]
*/

{  int m, l, r;
   if (n == 0 || x <= a[0]) return 0;
   if (x > a[n-1]) return n;
   l = 0; r = n-1;
   while (r - l > 1)
   {  m = (l + r)/2;
      (x <= a[m] ? r : l) = m;  // ouch! real C!
   }
   return r;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator+=(int x)
{  int i=binsearch(x, a, n), j;  // !!
   if (i >= n || x != a[i]) // x is not yet in set?
   {  if (n % blocksize == 0)
            a = memoryspace(a, n, n+ blocksize);
      for (j=n; j>i; j--)
            a[j] = a[j-1];
      n++;
      a[i] = x;
   }
   return *this;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator-=(int x)
{  int i=binsearch(x, a, n), j;  // !!
   if (i < n && x == a[i])
   {  n--;
      for (j=i; j<n; j++)
         a[j] = a[j-1];
      if (n % blocksize == 0)
         a = memoryspace(a, n+1, n);  // release one
   }                                  // block
   return *this;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
void iset::print() const
{  int i;
   for (i=0; i<n; i++)
      cout << a[i] << “ “;
}

void iset::operator()(int x) const
{  int i=binsearch(x, a, n);
   return i < n && x == a[i];
}

static int *newcopy(int n, int *a)
// copy a[0], ..., a[n-1] to a newly allocate area
// and return the new start address
{  int *p = new int[n];
   for (int i=0; i<n; i++)
      p[i] = a[i];
   return p;
}
```

Object-Oriented Programming

A Class for Sets - Application

```
iset &iset::operator=(iset S) // assignment operator
{  delete a;
   n = S.n;
   a = newcopy(n, S.a);
   return *this;
}

iset::iset(const iset &S) // copy constructor
{  n = S.n;
   a = newcopy(n, S.a);
}
```

Object-Oriented Programming

Exercises

13. Implement and test `iset` class as defined

Object-Oriented Programming
Derived Classes and Inheritance

- Given a class B, we can derive a new one D
  - comprising all the members of B
  - and some new ones beside
- we simply refer to B in the declaration of D
  - B is called the base class
  - D is called the derived class
- the derived class inherits the members of the base class

Object-Oriented Programming

A Class for Sets - Application

```
/* Consider the new class object geom_obj  */

#include  <iostream.h>

class geom_obj {
public:
   geom_obj(float x=0, float y=0): xC(x), yC(y){}
   void printcentre() const
   {  cout << xC << " " << yC << endl;
   }
protected:    // new keyword
   float xC, yC;
};


/* not a lot we can do with this class as it stands */
/* we wish to extend it to deal with circles and    */
/* squares                                          */
```

Object-Oriented Programming

A Class for Sets - Application

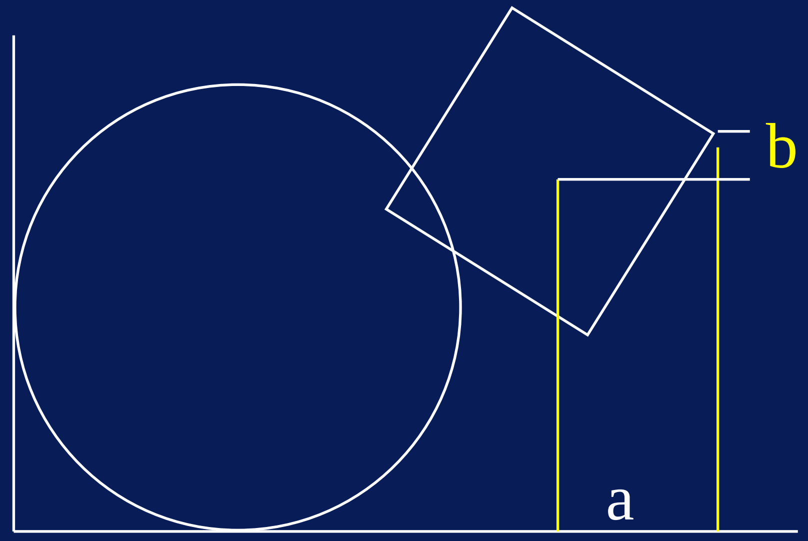
```
/* define derived class objects circle and square */

class circle: public geom_obj {  // base class
public:                          // inherit all public
   circle(float x_C, float y_C, float r)
      : geom_obj(x_C, y_C)
   { radius = r;
   }
   float area() const
   {  return PI * radius * radius;
   }
private:
   float radius;
};
```

Object-Oriented Programming

A Class for Sets - Application

```
/* define derived class objects circle and square */
/* square defined by its centre and vertex        */
class square: public geom_obj {  // base class
public:                          // inherit all public
   square(float x_C, float y_C, float x, float y)
      : geom_obj(x_C, y_C)
   {  x1 = x;
      y1 = y;
   }
   float area() const
   {  float a, b;
      a = x1 - xC; b = y1 - yC;
      return 2 * (a * a + b * b);
   }
private:
   float x1, y1;
};
```

Object-Oriented Programming
Derived Classes and Inheritance



Object-Oriented Programming
Derived Classes and Inheritance

- `circle` and `square` are extensions of their base class `geom_obj`
- the keyword public used in the declaration specifies that all public members of the base class `geom_obj` are also regarded as public members of the derived class
  - class `square` is publicly derived from `geom_obj`

    ```
    square S(3, 3.5, 4.37, 3.85);
    S.printcentre();
    ```

    » printcentre() is a public class member function of geom_obj

Object-Oriented Programming
Derived Classes and Inheritance

- `xc` and `yc` are protected members of the base class `geom_obj`
- but they are used in the area member function of square
- protected members are similar to private ones
  - except that a derived class has access to protected members of its base class
  - a derived class does not have access to private members of its base class

Object-Oriented Programming
Derived Classes and Inheritance

- User-defined assignment operators
  - if an assignment operator is defined as a member function of a base class, it is not inherited by any derived class

Object-Oriented Programming
Derived Classes and Inheritance

- Constructors and destructors of derived and base classes
  - When an object of (derived) class is created
    - » the constructor of the base class is called first
    - » the constructor of the derived class is called next
  - When an object of a (derived) class is destroyed
    - » the destructor of the derived class is called first
    - » the destructor of the base class is called next

Object-Oriented Programming
Derived Classes and Inheritance

- Constructors and destructors of derived and base classes
  - We can pass arguments from the constructor in a derived class to the constructor of its base class
    - » normally do this to initialize the data members of the base class
    - » write a constructor initializer

```
class square: public geom_obj {
public:
   square(float x_C, float y_C, float x,float y)
      : geom_obj(x_C, y_C)  // con. init.
```

Object-Oriented Programming
Derived Classes and Inheritance

» since the geom_obj constructor has default arguments, this initializer is not obligatory here

» if we omit it, the constructor of geom_obj is called with its default argument values of 0

» however, the initializer would really have been required if there had been no default arguments, i.e. if we had omitted the `=0` from the constructor:

```
class geom_obj {
public:
   geom_obj(float x=0, float y=0): xC(x), yC(y){}
```

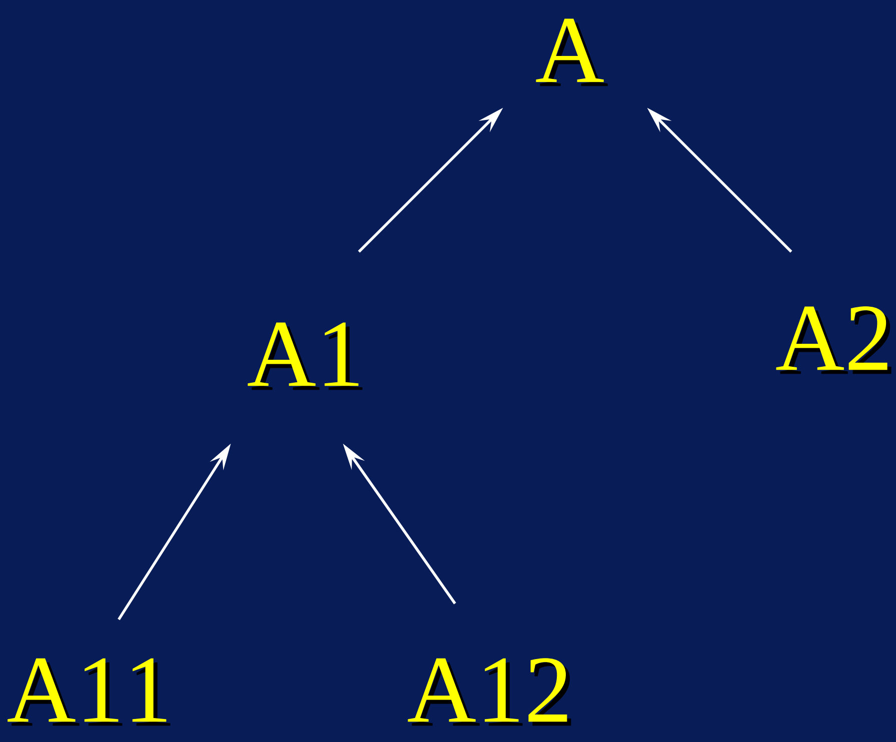
Object-Oriented Programming

Derived Classes and Inheritance

```
int main()
{  circle C(2, 2.5, 2);
   square S(3, 3.5, 4.37, 3.85);
   cout << “Centre of circle: “; C.printcentre();
   cout << “Centre of square: “; S.printcentre();
   cout << “Area of circle:   “ << C.area() << endl;
   cout << “Area of square:   “ << S.area() << endl;
   return 0;
}

/* output */

Centre of circle: 2 2.5
Centre of square: 3 3.5
Area of circle:   12.5664
Area of square:   3.9988
```

Object-Oriented Programming

Derived Classes and Inheritance



A tree of (derived) classes

Object-Oriented Programming
Derived Classes and Inheritance

- Conversion from derived to base class
  - allowed: conversion from derived to base class
  - NOT allowed: conversion from base to derived
  - Same applies to corresponding pointer types
  - why? derived class objects may contain members that do not belong to the base class

Object-Oriented Programming

Derived Classes and Inheritance

```
/* code fragments to illustrate legal and illegal */
/* class conversions                               */

class B {...};           // base class B
class D: public B{ ...} // derived class D
...
B b, *pb;
D d, *pd;
...
b = d;       // from derived to base: OK
pb = pd;     // corresponding pointer types: OK
d = b;       // from base to derived: error
d = (D)b;    // even with a cast: error
pd = pb;     // corresponding pointer types: error
pd = (D*)b; // with cast: technically OK but suspicious
```

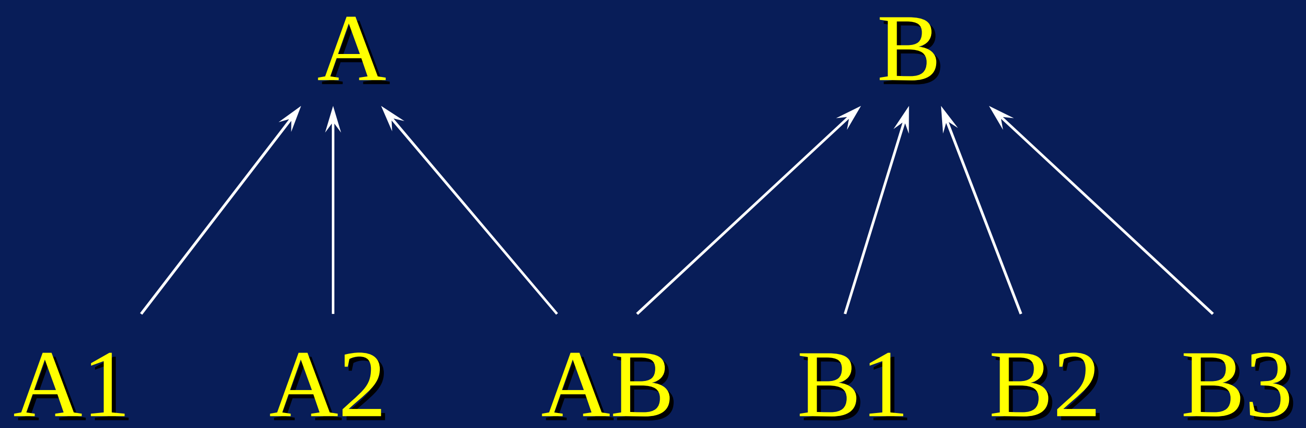
Object-Oriented Programming
Derived Classes and Inheritance

- Multiple inheritance
  - a class can be derived from more than one base class
    - » C++ Release 2

```
class A {...}; // base class A
class B {...}; // base class B
class AB: public A, public B {
...
}
```

Object-Oriented Programming

Derived Classes and Inheritance



Multiple Inheritance

Object-Oriented Programming
Derived Classes and Inheritance

- Multiple inheritance
  - If AB has a constructor with parameters, we can pass these to each base class:

```
class AB {
public:
   AB(int n=0, float x=0, char ch='A')
      :A(n, ch), B(n, x)
{ ...
}
```

  - The creation of an object of class AB causes the three constructors for A, B, and AB, in that order, to be called.

Object-Oriented Programming

Virtual Functions and Late Binding

- Suppose we have declared class `ctype` as follows:

```
class ctype {
public:
   virtual void f()
   { ...
   }
   ...
};
```

  - the keyword `virtual` is important if
    - » ctype has derived classes, e.g. ctype1 and ctype2, (i.e. it is a base class for derived classes ctype1 and ctype2)
    - » and we are using pointers to class objects

Object-Oriented Programming
Virtual Functions and Late Binding

- If we define only a pointer to the class and create the class object dynamically

```
ctype *p;
...
p = new ctype;
```

  - the `ctype` object `*p` created in this way is only loosely related to `p`
    - » p can also point to the other `ctype` objects

Object-Oriented Programming
Virtual Functions and Late Binding

- Let’s declare two derived classes

```
class ctype1: public ctype {
public:
   void f() {...}
   ...
};

class ctype2: public ctype {
public:
   void f() {...};
};

ctype *p
```

Object-Oriented Programming

Virtual Functions and Late Binding

- since conversion of pointer to derived class to pointer to base class is allowed we can have:

```
p = new ctype1

/* or */

p = new ctype2
```

- the three class types `ctype`, `ctype1`, and `ctype2` have member functions with the same name (`f`)
  - `f` is a virtual function

Object-Oriented Programming

Virtual Functions and Late Binding

- `f()` is a virtual function
  - given the function call

    `p->f()`

  - the decision as to which if the three possible functions

    `ctype::f`

    `ctype1::f`

    `ctype2::f`

  - is made at run-time on the basis of type of object pointed to by `p`

Object-Oriented Programming

Virtual Functions and Late Binding

- This run-time establishment of the link between the function `f` and the pointer `p` is called
  - late binding
  - dynamic binding

Object-Oriented Programming

Virtual Functions and Late Binding

- If the keyword `virtual` had been omitted from the definition of `f` in declaration of `ctype`
- only the type of `p` would have been used to decide which function to call, *i.e.*, `ctype::f` would have been called
- This establishment of the link between the function `f` and the pointer `p` is made at compile time and is called
  - early binding
  - static binding

Object-Oriented Programming

Virtual Functions and Late Binding

```
// VIRTUAL: A virtual function in action

#include <iostream.h>

class animal {
public:
   virtual void print() const
   {  cout << “Unknown animal type\n”;
   }
protected:
   int nlegs;
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class fish: public animal {
public:
   fish(int n)
   { nlegs = n;
   }
   void print() const
   {  cout << “A fish has “ << nlegs << “ legs\n”;
   }
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class bird: public animal {
public:
   bird(int n)
   { nlegs = n;
   }
   void print() const
   {  cout << “A bird has “ << nlegs << “ legs\n”;
   }
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
class mammal: public animal {
public:
   mammal(int n)
   { nlegs = n;
   }
   void print() const
   {  cout << “A mammal has “ << nlegs << “ legs\n”;
   }
};
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
int main()
{  animal *p[4];
   p[0] = new fish(0);
   p[1] = new bird(2);
   p[2] = new mammal(4);
   p[3] = new animal;
   for (int i=0; i<4; i++)  // key statement
      p[i]->print();        // which print is called?
   return 0;                //   fish::print
}                           //   bird::print
                            //   mammal::print
                            //   animal::print
                            // the choice is made at
                            // run-time
```

Object-Oriented Programming

Virtual Functions and Late Binding

```
/* output */

A fish has 0 legs
A bird has 2 legs
A mammal has 4 legs
Unknown animal type
```

- If print had not been defined as virtual, the binding would have been early (static ... at compile time) and the fourth output line would have been printed four times

Object-Oriented Programming
Virtual Functions and Late Binding

- Object-Oriented Programming
  - the use of virtual functions and derived classes
- The style of programming is also called Polymorphism
  - objects of different (derived) types
  - are accessed in the same way
- Member functions are sometimes called methods
  - calling an object's member function is referred to as sending a message to an object

Object-Oriented Programming

Virtual Functions and Late Binding

- Suppressing the virtual mechanism
  - in the following function call

    ```
    p[1]->print();
    ```

  - the function calls the print function for the derived class bird
  - we can over-ride this with the scope resolution operator

    ```
    p[1]->animal::print();
    ```

  - in which case the function calls the print function for the derived class animal

Object-Oriented Programming
Virtual Functions and Late Binding

- Pure virtual functions and abstract classes
  - We can declare the function `print` in the base class as a pure virtual function by writing
    `virtual void print() const = 0;`
  - This has some important consequences
    - » the base class now becomes an abstract class
    - » an abstract class cannot be used to create objects (or this type)
    - » they can only be used for the declaration of derived classes

Object-Oriented Programming
Virtual Functions and Late Binding

- Pure virtual functions and abstract classes
  » So, for example, the following are illegal:

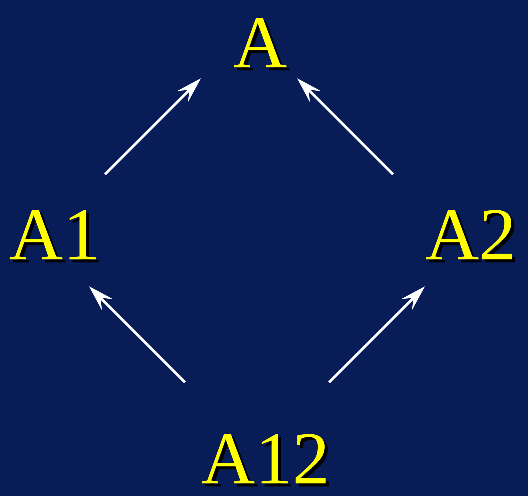
```
p[3] = new animal;// error
animal A;         // error
```

  – Making the base class animal abstact makes it impossible to omit any of the print functions in the derived classes

Object-Oriented Programming

Virtual Functions and Late Binding

- Virtual base classes
  - assume we have a base class A
  - and derived classes A1 and A2
  - and a further derived class A12 with multiple inheritance



Object-Oriented Programming

Virtual Functions and Late Binding

- Virtual base classes
  - If base class A had a member a, then
  - Derived class A12 will inherit two members called a (one through A1 and one through A2)
  - Such duplication of indirectly inherited members can be suppressed by using the keyword virtual in the declarations of A1 and A2

```
class A { ... };
class A1: virtual public A { ... };
class A2: virtual public A { ... };
class A12: public A1, public A2 { ... };
```

Object-Oriented Programming
Virtual Functions and Late Binding

- Virtual base classes
  - A derived class cannot directly inherit the members of a base class more than once

```
class A12: public A, public A2 { ... };
// error
```

Object-Oriented Programming
Static Class Members

- Static data members
  - Normally, a data member of a class type is stored in every object of that type
  - If we use the keyword static for a class data member, however, there will be only one such member for the class, regardless of how many objects there are
    - » a static class member belongs to its class type rather than to the individual ojects of that type

Object-Oriented Programming

Static Class Members

- Static member functions
  - cannot use any data members that are specific for objects
  - the this pointer is not available in static member functions

Object-Oriented Programming

Static Class Members

```
/* STATMEM: Using a static class member to count
            how many times the constructor person()
            is called                                   */

#include <iostream.h>
#include <string.h>

class person {
public:
   person (char *str)
   {  strcpy(name, str);
      count++;   // increment the static counter
   }
   void print() const
   {  cout << name << endl;
   }
```

Object-Oriented Programming

Static Class Members

```
    static void printcount()  // static function member
    {  cout << “There are “ << count
            << “ persons.” << endl;
    }
private:
    char name[20];
    static int count;  // static data member
};

int person::count=0; // must define (instantiate) the
                     // count member
int main()
{  person A(“Mary”), B(“Ciana”), C(“Georgina”), *p;
   p = new person(“Brighid”);
   A.print(); B.print(); C.print(); p->print();
   person::printcount();
   return 0;
}
```

Object-Oriented Programming
Static Class Members

```
the output is as follows:

Mary
Ciana
Georgina
Brighid
There are 4 persons.
```

Object-Oriented Programming

Static Class Members

- Static member functions - key points
  - must define a static class member outside the class definition (to instantiate the member)

    ```
    int person::count=0
    ```

  - since the static class member is associated with the class and not the object, we must reference the member with the class name and not an object name

    ```
    person::count
    person::printcount()
    ```

Object-Oriented Programming
Pointers to Members

- To use pointers to class member functions
  - use the class name
  - followed by ::
  - as a prefix to the declaration of the pointer
  - which will point to the required class member function

Object-Oriented Programming

Static Class Members

```
class example {
public:
   example (int ii, int jj):i(ii), j(jj) {}
   int ivalue(){return i;}
   int jvalue(){return j;}
private:
   int i, j;
};

int (example::*p)(); // pointer to a member function in
                     // class example (no parameters
                     // and returning int
example u(1,2);
...

p = example::ivalue;
cout << (u.*p)();  // call *p for u ... output is 1
```

Object-Oriented Programming
Polymorphism and Reusability

- Example to demonstrate the power and efficiency of object-oriented programming
  - Heterogeneous linked-list of geometric shapes
  - Begin with circles and lines
  - Extend to triangles
  - Use virtual function `print`
  - in an abstract class `element` from which we will derive the appropriate class for each geometry
    - Lines: defined by two end points
    - Circles: defined by centre and radius

Object-Oriented Programming

Polymorphism and Reusability

```
/* FIGURES.H: interface file to build linked-list   */
/*            for lines and circles                 */

#include <iostream.h>
#include <stdio.h>

class point {
public:
   float x, y;
   point(float xx=0, float yy=0):x(xx), y(yy) {}
}

class element {                  // abstract class
public:
   element *next;
   virtual void print() const=0; // pure virtual fn.
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
class line: public element {
public:
   line (point &P, point &Q, element *ptr);
   void print() const;
private:
   point A, B;
};

class circle: public element {
public:
   circle (point &center, float radius, element *ptr);
   void print() const;
private:
   point C;
   float r;
};
void pr(const point &P, const char *str=", ");
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FIGURES: Implementation file (figures.cpp) for
//          linked lists of circles and lines

#include “figures.h”

line::line(point &P, point &Q, element *ptr)
{  A = P;
   B = Q;
   next = ptr;
}

void line::print() const
{  cout << “Line: “;
   pr(A); pr(B, “\n”);
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
circle::circle(point &center, float radius,
               element *ptr);
{  C = center;
   r = radius;
   next = ptr;
}

void circle::print() const
{  cout << “Circle: “;
   pr(C);
   cout << r << endl;
}

void pr(const point &P, const char *str)
{  cout << “(“<< P.x << “, “<< P.y << “)”<< str;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FIGURES: sample application file

#include “figures.h”

int main()
{  element *start=NULL;
   start = new line(point(3,2), point(5,5)), start);
   start = new circle(point(4,4), 2, start);
}
```

Object-Oriented Programming

Polymorphism and Reusability

- Some time later, we may wish to add a triangle type to our systems
- We then write a new interface file
- and a new implementation

Object-Oriented Programming

Polymorphism and Reusability

```
/* TRIANGLE.H: adding a triangle (interface file)   */
/*      Class triangle is derived from class element */

class triangle: public element {
public:
   triangle(point &P1, point &P2, point &P3,
            element *ptr);
   void print() const;
private:
   point A, B, C;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// TRIANGLE: adding a triangle class
//           (implementation file for triangle.cpp)

#include “figures.h”
#include “triangle.h”

triangle::triangle(point &P1, point &P2, point &P3,
                   element *ptr)
{  A = P1;
   B = P2;
   C = P3;
   next = ptr;
}

void triangle::print() const
{  cout << “Triangle: “;
   pr(A); pr(B); pr(C, “\n”);
}
```

Object-Oriented Programming

Polymorphism and Reusability

- Later again, we may wish to add the ability to distinguish between lines of different thickness
- Instead of deriving a new class from the base class `element`, we can derive one from the class `line` for example `fatline`
- We then write a new interface file
- and a new implementation

Object-Oriented Programming

Polymorphism and Reusability

```
/* FATLINE.H: additional interface file for fat lines*/

class fatline: public line {
public:
   fatline(point &P, point &Q, float thickness,
            element *ptr);
   void print() const;
private:
   float w;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// FATLINE: Implementation file (fatline.cpp) for
//          for thick lines

#include “figures.h”
#include “fatline.h”

fatline::fatline(point &P, point &Q, float thickness,
                 element *ptr): line(P, Q, ptr)
                    // note the constructor initializer
{  w = thickness;
}

void fatline::print() const
{  this->line::print();
   cout << “   Thickness: “ << w << endl;
}
```

Object-Oriented Programming

Polymorphism and Reusability

```
// DEMO: This program builds a heterogeneous linked
//       list in which data about a line, a circle,
// a triangle, and a 'fat'line are stored
// To be linked with FIGURES, TRIANGLE, and FATLINE

#include "figures.h"
#include "triangle.h"
#include "fatline.h"

int main()
{  element *start=NULL, *p;
// Build a heterogeneous linked list
start = new line(point(3, 2), point(5, 5), start);
start = new circle(point(4, 4), 2, start);
start = new triangle(point(1, 1), point(6, 1),
                     point(3, 6), start);
start = new fatline(point(2, 2), point(3, 3), 0.2, start);
```

Object-Oriented Programming

Polymorphism and Reusability

```
// DEMO: This program builds a heterogeneous linked
//       list in which data about a line, a circle,
// a triangle, and a ‘fat’line are stored
// To be linked with FIGURES, TRIANGLE, and FATLINE

#include “figures.h”
#include “triangle.h”
#include “fatline.h”

int main()
{  element *start=NULL, *p;
// Build a heterogeneous linked list
start = new line(point(3, 2), point(5, 5), start);
start = new circle(point(4, 4), 2, start);
start = new triangle(point(1, 1), point(6, 1),
                     point(3, 6), start);
start = new fatline(point(2, 2), point(3, 3), 0.2, start);
```

Object-Oriented Programming

Polymorphism and Reusability

```
    for (p=start; p!=NULL; p = p->next)
       p->print(); // polymorphic data handling and
                   // late binding!
}
```

Object-Oriented Programming
Exercises

14. Modify (and test) the `iset` class as follows
    - replace the element addition `+=` with the addition operator `+`
      `S = S + x;`
    - replace the element removal `-=` with the assignment operator `-`
      `S = S - x;`
    - replace the null set check with the function
      `isempty();`
    - add a set union operator `+`
    - add a set intersection operation `*`

Object-Oriented Programming
Exercises

15. Modify (and test) the `iset` class as follows
    - add a function null which removes all elements from the set and returns an empty set
    - replace the inclusion operator `s(x)` with the function `contains()` returning TRUE or FALSE

Object-Oriented Programming
Exercises

16. Modify (and test) the `iset` class
    - represent the set with a linear linked list and do insertions with an insertion sort
    - represent the set with a binary tree

Object-Oriented Programming

Exercises

17. Create a new set class for character strings `string_set`