

## Object-Oriented Programming

*Though this be madness, yet there is method in't*

-- William Shakespeare



- Object orientation
  - ▲ encapsulation, polymorphism, inheritance
- Types of operations for classes
  - ▲ constructor, destructor, iterator, reads, writes
- Inheritance
- Polymorphism
  - ▲ overriding and abstract methods
  - ▲ operator overloading
  - ▲ runtime polymorphism
- Generics
- Miscellaneous

292

## Programming Paradigms

When we were much younger ( 🎵<sub>21-26</sub> ) we discussed four programming language paradigms. They were:

- *procedural/imperative* (the oldest)
  - ▲ programmer specifies the steps to be executed in order in the solution of a problem
  - ▲ the solution is the execution of these steps
- *functional* (almost the oldest)
  - ▲ programmer specifies a problem solution as mappings from elements of one set to elements of another set (functions!)
  - ▲ the solution is a mapping from the given to the desired
- *logic* (just old)
  - ▲ programmer specifies a problem as a set of facts and rules
  - ▲ the solution is the proof of a goal, using the stated rules and facts

293

## Paradigm Departures

- In the imperative paradigm programming is the use of a convenient notation for specifying sequences of machine-based instructions:
  - ▲ a program is a model of its execution on a lower-level virtual machine
- Functional and logic programming are complete departures from the imperative paradigm:
  - ▲ concentrate on a declarative description of the problem
  - ▲ ignore the underlying program execution architecture
- The *object-oriented* programming paradigm (being the youngest paradigm) takes advantage of many of the lessons learned from programming and language design in the other paradigms.

294

## What is a Program?

... which is not to say that the object-oriented programming paradigm is just an extension of some other paradigm. OO has its own view of what a program is.

- *imperative*
  - ▲ program is a model of a machine executing instructions one-by-one
- *functional*
  - ▲ program is a model of a mapping from data to results
- *logic*
  - ▲ program is a logical proof that results can be obtained from data
- *object-oriented*
  - ▲ program is a model of the entities and relationships involved in the real-world problem to be solved

295

## Object Orientation...

In the OO paradigm, programming language constructs are objects:

- the data are objects
- the control mechanisms are also objects

Beyond the object model, OO languages are expected to support:

- *encapsulation*
  - ▲ data type definitions have some visible elements (public, export) and some hidden elements (private, implementation details)
- *polymorphism*
  - ▲ to enhance orthogonality, *different* operations are allowed to have the same name, or the *same* operation can be made to behave appropriately for different types of operands
  - ▲ it is up to the compiler to resolve ambiguity through context

296

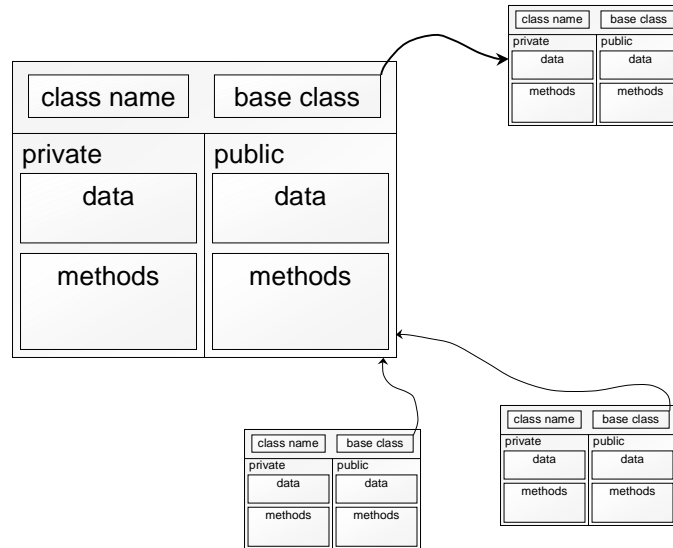
## More Object Orientation

But wait... that's not all! Classes are sort of like ADTs, encapsulation and polymorphism are not necessarily new. The OO paradigm adds another important twist of its own:

- *inheritance*
  - ▲ objects are *instances* of *classes*
  - ▲ classes are arranged in a *hierarchy*
  - ▲ if class *P* is a parent of class *C* in the hierarchy
    - *P* is the *base class* of *C*
    - *C* is a *derived class* of *P*
    - *C inherits* the data members and operations of *P*

297

## Abstract View of a Class

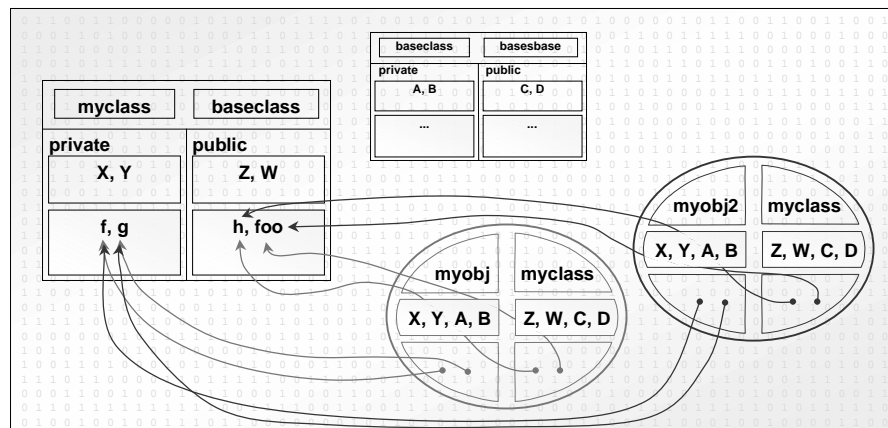


298

## Objects and Classes

An *object* is an instance of a class

- normally, every object gets its own variables: *instance variables*
- normally, all objects share the same set of methods: *instance methods*



299

## Talking to Objects

It is a useful part of the abstraction to think of objects as “taking care of themselves”.

- With a regular user-defined data type, the program must still know the structure of the data type in order to assign it values, check its components, etc.
- With an object, the program can only request that data from the object
  - ▲ a program does not assign a value to an object, it requests that the object assign itself a value
  - ▲ a program does not check the value of a component of an object, it requests that the object report such values

300

## Types of Operations on Objects

The operations to be performed on objects fall into a small number of general categories:

- Constructors
  - ▲ initializations, allocations, etc.
- Destructors
  - ▲ deallocations, etc.
- Reads
  - ▲ requests for values, state information, etc.
- Writes
  - ▲ requests to modify values, state information, etc.
- Iterators
  - ▲ requests to access individuals within collections of objects

301

## Constructors

A *constructor* is a method that is invoked when an object is created.

- an object can be created *statically* or *dynamically*
  - ▲ *statically*: in an object declaration (like a variable declaration)
  - ▲ *dynamically*: requested by a program statement (like `new`)
- an object can be created *explicitly* or *implicitly*
  - ▲ *explicitly*: by declaration or requested by program statement
  - ▲ *implicitly*: as a formal pass-by-value parameter (or return value)
- a constructor may :
  - ▲ initialize instance variables for the object
  - ▲ dynamically allocate memory for instance variables
  - ▲ *do whatever the programmer of the constructor wants it to do!*



*You can define zero, one or many constructors in a class!*

302

## Destructors

A *destructor* is a method invoked when an object is destroyed.

- an object is destroyed at the end of its lifetime
  - ▲ *e.g.* when control reaches the end of the block defining the object
- a destructor may:
  - ▲ deallocate memory for instance variables
  - ▲ *do whatever the programmer of the destructor wants it to do!*
- in some languages destructors are essential for all objects containing dynamically allocated memory
- in some languages dynamically allocated memory is deallocated automatically (automatic garbage collection)
  - ▲ such languages either don't have destructors or don't require them



*Destructors are invoked automatically at the end of an object's life; but some languages allow destructors to be called explicitly!*

303

## Reads and Writes

A *read* is a method that allows an object's clients to request values of instance variables, computed values, etc.

A *write* is a method that allows an object's clients to request a modification of instance variables

- it is good encapsulation to keep instance variables private and report values to clients or change values on request only
- the implementation of a variable or computation is kept separate from its interface with clients

304

## Putting It All Together...

```
#include <iostream.h>
class balloon {
    private: int psi, max;
    public:
        balloon() {
            max = 10;
            psi = 0;
            cout << "taking the balloon out of the bag!\n";
        }
        ~balloon() {
            cout << "bang!\n";
        }
        int get_psi() { return psi; }
        void inflate(int addpsi) {
            if(psi + addpsi < max)
                psi += addpsi;
            else {
                cout << "pricking the balloon... ";
                balloon::~~balloon();
            }
        }
};
```

constructor

destructor

read

write

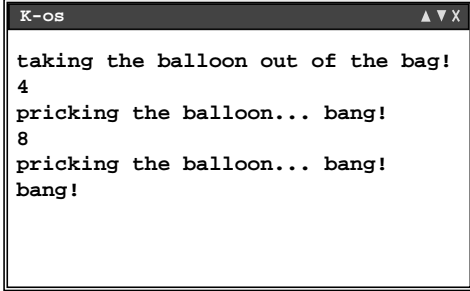
305

## Still Putting It All Together

```
int main() {  
    balloon b;  
  
    b.inflate(1);  
    b.inflate(3);  
  
    cout << b.get_psi() << endl;  
  
    b.inflate(4);  
    b.inflate(3);  
  
    cout << b.get_psi() << endl;  
  
    b.inflate(2);  
    return 0;  
}
```

→ class

→ object



```
K-os  
taking the balloon out of the bag!  
4  
pricking the balloon... bang!  
8  
pricking the balloon... bang!  
bang!
```

306

## Inheritance

We've seen encapsulation and polymorphism, and we'll see them again. Let's have a closer look at inheritance.

- a class has
  - ▲ a name
  - ▲ private data, private methods
  - ▲ public data, public methods

} *data and methods in a class are called members of the class*

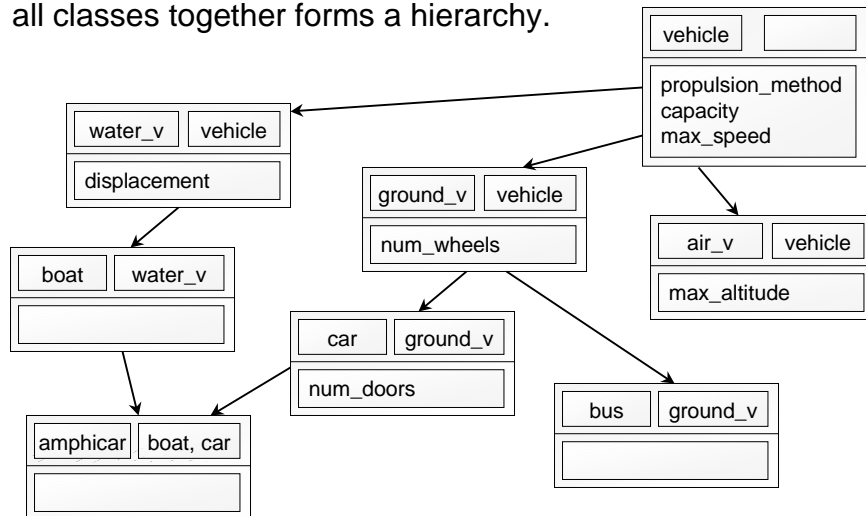
- ▲ a base class (or *superclass*)
  - a name
  - private data, private methods
  - public data, public methods
  - a base class (or *superclass*)
    - a name
    - private data, private methods
    - public data, public methods
    - a base class (or *superclass*)

307



## Inheritance Hierarchies

The parent→child (base class→derived class) relationships for all classes together forms a hierarchy.



308

## A Note on Inheritance Hierarchies

- Some object-oriented languages allow classes to be defined independent of any other class (not derived from any base class)
  - ▲ some classes are *free* (do not belong to a class hierarchy)
- Some object-oriented languages require that all classes be derived from some existing base class
  - ▲ all classes belong to a hierarchy
  - ▲ the top-most class in a hierarchy is maximally general and is predefined in the language
    - there is no reason why there can't be more than one top-most class, though the distinctions between top-most classes reveals biases of the language designers

309

## (Multiple) Inheritance Example (C++)

```
class vehicle {
private: int propulsion_method, capacity, max_speed;
public:
    void set_propulsion_method(int p) {
        propulsion_method = p;
    }
    void set_capacity(int c) {
        capacity = c;
    }
};

class ground_v : public vehicle {
private: int num_wheels;
public:
    void set_num_wheels(int w) {
        num_wheels = w;
    }
};

class water_v : public vehicle {
private: int displacement;
public:
    void set_displacement(int d) {
        displacement = d;
    }
};

class car : public ground_v {
private: int num_doors;
public: car() {
    set_num_wheels(4);
    set_capacity(6);
    set_propulsion_method(GAS_ENGINE);
}
};

class boat : public water_v {
public: boat() {
    set_displacement(SMALL);
}
};

class amphicar : public car, public boat {
public: amphicar() {
    car::set_capacity(4);
    set_displacement(TINY);
}
};
```

310

## Access Control

In addition to its own private and public data and methods, a class has access to *some* of the data and methods of its base class. There are various levels of access modification to limit the inheritance of names throughout the hierarchy.

- *public members*
  - ▲ obviously a derived class has access to public members in its base class (*all* clients have access to these)
- *private members*
  - ▲ some languages *may* allow a derived class automatic access to its private members
- *protected members*
  - ▲ some languages have a third level of access control that allows derived classes access to certain non-public members

311

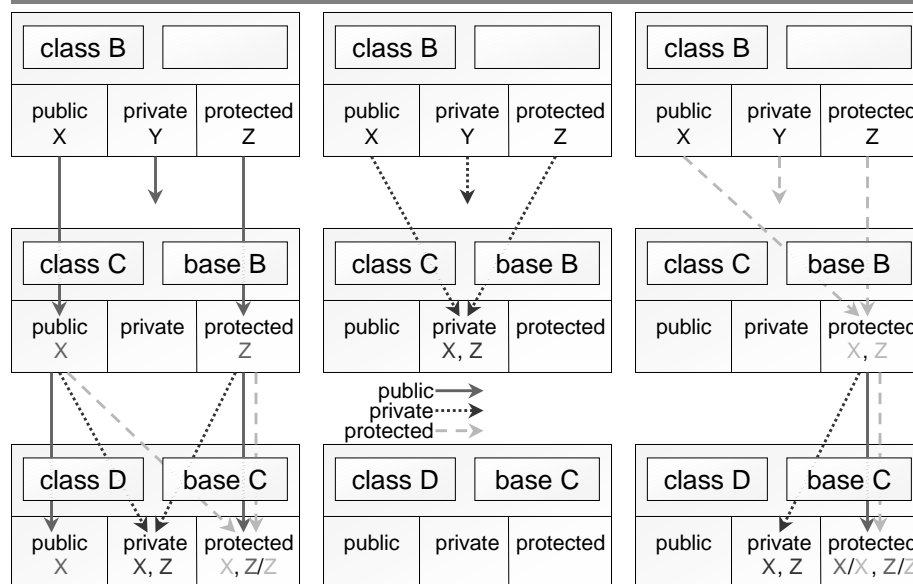
## Inheritance Control

The public and private access modifiers on members restrict the visibility of those members. Some languages have an extra level of access control on inheritance. Assume we have a base class (B), a derived class (C) and a class derived from C (D):

- *public inheritance*
  - ▲ members inherited from B to C are *also* inherited by D
- *private inheritance*
  - ▲ members inherited from B to C are *not* inherited by D
- *protected inheritance*
  - ▲ members inherited from B to C are *also* inherited by D, but the inherited members in D *may be* more restricted than in B or C

312

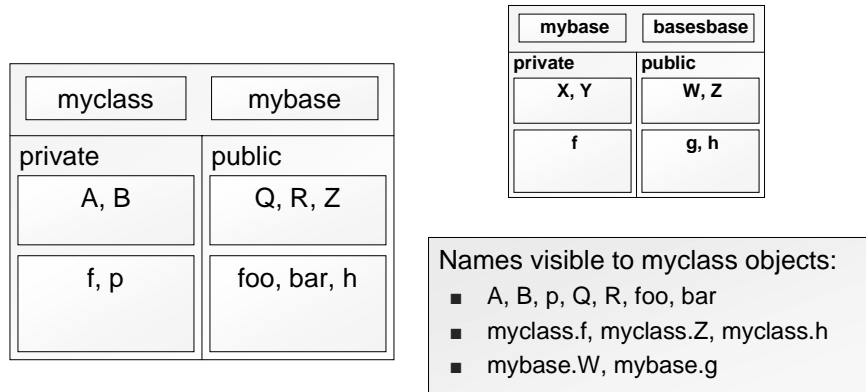
## Member Access and Inheritance Control



313

## Naming and Access

A class has its own members and access to some of its base class' members. But what if a derived class defines members with the same name as members in the base class?



314

## Polymorphism: Overriding

When a class redefines an inherited method, the new method *overrides* the inherited method.

- normally the overridden method is some real method defined for the general case of the base class
- the overriding method is a more specific version for a particular derived class

```
class cube {  
    protected: float edge;  
    public:  
        float get_area() {  
            return edge*edge*6.0;  
        }  
};
```

```
class open_cube : public cube {  
    public:  
        float get_area() {  
            return edge*edge*10.0;  
        }  
};
```



*How is this polymorphism?*

315

## Polymorphism: Abstract Methods

Sometimes, there may be a very general base class whose derived classes all require a particular method, but there is no meaningful implementation at the general level of base class.

- a method declared in a base class having no definition in the base class ( $\therefore$  overriding definitions in derived classes *only*) is called an *abstract method*
- a class containing an *abstract method* is an *abstract class*
- instances of abstract classes are not allowed!
  - ▲ all instances must be instances of one of the derived classes

❗ The book sometimes (§11.2.3) calls abstract methods virtual methods, but they are not the same as **virtual functions** in C++!

❗ The book sometimes (§11.2.3) calls abstract methods virtual methods, but they are not the same as **virtual functions** in C++!

316

## Abstract Methods: An Example...

```
class geom {
private: char colour;
public:
    void set_colour(char C) { colour = C; }
    char get_colour() { return colour; }
    float get_area() { }
};
```


```
class rectangle : public geom {
private: int len, wid;
public:
    float get_area() {
        return (float) (len*wid);
    }
    void set_lw(int l, int w) {
        len = l; wid = w;
    }
    int get_l() { return len; }
    int get_w() { return wid; }
};
```

```
class triangle : public geom {
private: int h, b;
public:
    float get_area() {
        return (float) (h*b) / 2.0;
    }
    void set_hb(int hi, int ba) {
        h = hi; b = ba;
    }
    int get_h() { return h; }
    int get_b() { return b; }
};
```

317

## Abstract Methods: An Example Used

```
int main() {  
    rectangle r;  
    triangle t;  
  
    r.set_lw(3, 7);  
    cout << r.get_area() << endl;  
  
    t.set_hb(7, 3);  
    cout << t.get_area() << endl;  
  
    return 0;  
}
```



21.0  
10.5  
\_

318

## Polymorphism: Operator Overloading

We already saw user-defined operator overloading in Ada (an *imperative* language with lots of abstraction facilities).

- user-defined operator overloading allows the programmer to associate new functionality with existing operators
  - ▲ +, -, \*, /, :=, %, etc.
  - ▲ with object-oriented languages, operators are overloaded with new method definitions
  - ▲ in languages that require all methods to belong to classes, operator overloading definitions must also be within classes
  - ▲ in languages that allow *free* methods, operator overloading definitions may also be *free*

319

## Overloading Operators: An Example...

```
class day {
private: int d;
public:
    day():d(0) {}
    day(int newd):d(newd) {}
    void set_day(int newd) {
        while(newd > 6)
            newd -= 7;
        d = newd;
    }
    int get_day() {
        return d;
    }
    int operator+(int days) {
        while((d + days) > 6)
            days -= 7;
        return (d + days);
    }
};
```

320

## Overloading Operators: An Example Used

```
int main() {
    day d1, d2(3);

    d1.set_day(15);

    cout << d1.get_day() << endl;

    d1.set_day(d2 + 8);

    cout << d1.get_day() << endl;

    return 0;
}
```



```
K-os
1
4
—
```

321

## Revisiting Arrays

We once defined an *array* as a homogeneous aggregate:

- *aggregate*: collection of many things
- *homogeneous*: the things are all the same

We can arrays of any type:

- ```
{  
    char    ca[20];           // an array of 20 chars  
    int     ia[20];           // an array of 20 ints  
    float   fa[20];           // an array of 20 floats  
    my_rec  ra[20];           // an array of 20 records  
    ...
```

*We can even have arrays of objects (in some languages)*

322

## Heterogeneous Arrays?

Arrays are by definition restricted to elements of all the same type. If we want a collection of different types, we define a record instead (a *heterogeneous aggregate*). But sometimes we want a cross between the two:

- heterogeneous aggregate
  - ▲ so we can have different types in the collection
- ordered collection
  - ▲ so we can access elements iteratively

For example, we can imagine wanting an array of numbers, some short integers, some long integers, some floats...

323

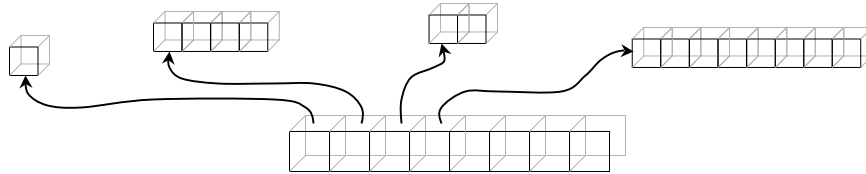


## Point the Way...

Here's how we can do it in a relatively weakly typed language:

```
■ {  
    void *mixedarray[20];  
  
    mixedarray[0] = new int;  
    mixedarray[1] = new float;  
    mixedarray[2] = new long;  
    mixedarray[3] = new long double;  
    ...  
}
```

 *Array elements must be the same type. So how is this possible?*



324

## Arrays of Objects

Some OO languages allow arrays of objects.

```
#include <iostream.h>  
  
class geom {  
protected: int b, h;  
public:  
    float get_area() {  
        return (float)(b*h);  
    }  
};  
  
class rectangle : public geom {  
public:  
    void set_lw(int l, int w) {  
        b = l; h = w;  
    }  
    int get_l() { return b; }  
    int get_w() { return h; }  
};  
  
class triangle : public geom {  
public:  
    float get_area() {  
        return (float) (h*b) / 2.0;  
    }  
    void set_hb(int hi, int ba) {  
        h = hi; b = ba;  
    }  
    int get_h() { return h; }  
    int get_b() { return b; }  
};  
  
int main() {  
    rectangle ra[20];  
  
    ra[0].set_lw(3,5);  
    ra[1].set_lw(4,2);  
    ra[2].set_lw(3,2);  
    cout << ra[0].get_area() << endl;  
    cout << ra[1].get_area() << endl;  
    cout << ra[2].get_area() << endl;  
  
    return 0;  
}
```

K-os

15

8

6

325

## Heterogeneous Arrays of Objects

Of course, we may want an array of geometric shapes (rectangles *or* triangles). What about this:

```
■ {  
    geom ga[20];  
    ...  
}
```

Unfortunately, we can't just use an array of base class objects: in this case, the base class has no *write* methods to set the base and height variables (*b* and *h*). We'll have to do *this* instead:

```
■ {  
    geom *ga[20];  
    ...  
}
```

An array of *pointers* to geom objects!

326

## Arrays of Object Pointers

And here's the code:

```
■ int main() {  
    rectangle r;  
    triangle t;  
    geom *ga[20];  
  
    r.set_lw(4,2);  
    ga[0] = new rectangle;  
    *ga[0] = r;  
  
    t.set_hb(5,3);  
    ga[1] = new triangle;  
    *ga[1] = t;  
  
    cout << ga[0].get_area() << endl;  
    cout << ga[1].get_area() << endl;  
  
    return 0;  
}
```



Aw, crap.

327

## Arrays Gone Awry

Here is the problem with the previous example:

- even though `ga[1]` is supposed to be a `triangle`, the C++ compiler looks at the static declaration of `ga` to determine its class
- since `ga` is an array of pointers to `geom`, the compiler uses the `geom::get_area()` method instead of the overriding `triangle::get_area()` method as desired.

Luckily, C++ has a easy way around it...

328

## C++ Virtual Functions

There's no trick to it, it's just a simple trick!

```
#include <iostream.h>

class geom {
protected: int b, h;
public:
    virtual float get_area() {
        return (float)(b*h);
    }
};

class rectangle : public geom {
public:
    void set_lw(int l, int w) {
        b = l; h = w;
    }
    int get_l() { return b; }
    int get_w() { return h; }
};

class triangle : public geom {
public:
    float get_area() {
        return (float) (h*b) / 2.0;
    }
    void set_hb(int hi, int ba) {
        h = hi; b = ba;
    }
    int get_h() { return h; }
    int get_b() { return b; }
};

int main() {
    rectangle r; triangle t; geom *ga[20];
    r.set_lw(4,2); t.set_hb(5,3);
    ga[0] = new rectangle; *ga[0] = r;
    ga[1] = new triangle; *ga[1] = t;

    cout << ga[0]->get_area() << endl;
    cout << ga[1]->get_area() << endl;

    return 0;
}
```

*the only diff!*

```
K-os
8
7.5
```

329

## Runtime Polymorphism in C++

There are several things to note about the preceding example:

- a pointer to an object of class *C* may also point to an object of any of *C*'s derived classes

```
▲ vehicle *pv;  
  ground_v gv;  
  car c;  
  ...  
  pv = &gv;    ...    pv = &c;
```

- if

- ▲ a pointer *p* is declared to point to class *C* *and*
- ▲ *p* is set to point to an object of a derived class *D* of *C* *and*
- ▲ member function *C::m* is overridden as *D::m* *and*
- ▲ you want to call *D::m* for *p*

then

- ▲ *C::m* must be declared as `virtual`

330

## The Trouble with Arrays

An array is a useful built-in data type, but sometimes it isn't exactly what we need:

- tricking the compiler into allowing heterogeneous elements by using pointers is ugly
- accessing array elements must be done through the array index: a fixed type in most languages (usually integer)
- fixed-size arrays are the norm; variable-size arrays are sometimes possible; fully dynamic arrays are rare

The problems with arrays seem to fall under two kinds:

- arrays are not general enough
- we have to be concerned with too many details

331

## Generic Programming: Container Classes

Both of these problems suggest an object-oriented solution: instead of using an array (and living with its limitations), implement a special class that works like an array without the hassles

- container class
  - ▲ a class that holds a collection of objects and the operations for adding, accessing, modifying and removing objects
  - ▲ often implemented as a kind of linked list with external links

332

## A Simple Container Example: List...

Here is an example of a container class implementing a list of books. It doesn't really matter what a book is, but here it is:

```
■ class book {  
    private: int numpages;  
             char title[45];  
             char author[30];  
    public:  
        book(char *ti, char *au, int np) {  
            strcpy(title, ti);  
            strcpy(author, au);  
            numpages = np;  
        }  
        int get_pages() { return numpages; }  
        char *get_title() { return title; }  
        char *get_author() { return author; }  
};
```

333

## A Simple Container Example: List (continued)

```
class elem {
public: elem *next;
       book *data;
       elem(book *b, elem *link) {
           data = b; next = link;
       }
};
```

- There are two parts to the container
  - ▲ an elem class
    - defining the pointers for each book node in the list
  - ▲ a list class
    - maintaining pointers to the first and current elements in the list
    - implementing methods for adding books, traversing the list and resetting the list

```
class list {
private: elem *first;
         elem *current;
public:
    list() {
        first = NULL;
        current = first;
    }
    void add(book *e) {
        first = new elem(e, first);
        current = first;
    }
    book *get_elem() {
        book *b;
        if(current) {
            b = current->data;
            current = current->next;
            return b;
        }
        else
            return NULL;
    }
    void reset() { current = first; }
};
```

334

## Using the Simple Container

```
int main() {
    book tkmb("To Kill A Mockingbird", "Lee", 336);
    book hhgg("The Hitchhiker's Guide to the Galaxy", "Adams", 216);
    book *b1, *b2;
    list booklist;

    b1 = new book("The Stand", "King", 1135);

    booklist.add(b1);
    booklist.add(&tkmb);
    booklist.add(&hhgg);
    booklist.add(new book("The Ascent of Man", "Bronowski", 342));

    while(b2 = booklist.get_elem()) {
        cout << b2->get_title() << " by ";
        cout << b2->get_author() << " (";
        cout << b2->get_pages() << " pages)\n";
    }

    return 0;
}
```

335

## The Simple Container: Improvement #1

336

## The Simple Container: Improvement #2

- Arrays are built-in. We can declare arrays of any type.
- Our container can only contain books.
- We'll use C++ templates to make our container generic.

```
template <class dataclass>
class elem {
    friend class list<dataclass>;
private: elem *next;
        dataclass *data;
public:
    elem(dataclass *d, elem *link) {
        data = d; next = link;
    }
};
```

```
template <class listclass>
class list {
private: elem<listclass> *first;
        elem<listclass> *current;
public:
    list() {
        first = NULL;
        current = first;
    }
    void add(listclass *d) {
        first = new elem<listclass>(d, first);
        current = first;
    }
    listclass *get_elem() {
        listclass *d;
        if(current) {
            d = current->data;
            current = current->next;
            return d;
        }
        else
            return NULL;
    }
    void reset() { current = first; }
};
```

337

## Using the Improved Container

```
int main() {
    book *b;
    list<book> booklist;
    geom *g;
    list<geom> geomlist;

    booklist.add(new book("The Stand", "King", 1135));
    booklist.add(new book("To Kill A Mockingbird", "Lee", 336));
    booklist.add(new book("The Hitchhiker's Guide to the Galaxy", "Adams", 216));
    booklist.add(new book("The Ascent of Man", "Bronowski", 342));

    while(b = booklist.get_elem()) {
        cout << b->get_title() << " by ";
        cout << b->get_author() << " (";
        cout << b->get_pages() << " pages)\n";
    }

    geomlist.add(new rectangle(3, 2));
    geomlist.add(new rectangle(4, 3));
    geomlist.add(new triangle(6, 3));
    geomlist.add(new triangle(5, 3));

    while(g = geomlist.get_elem())
        cout << g->get_area() << endl;

    return 0;
}
```

338

## Improving the Improved Container

Our container can hold any kind of element. But it's quite restricted in how it lets us access those elements:

- elements are always added to the front of the list
- when we get an element from the front of the list, the current element pointer automatically moves to the next element in the list

Our problems are due to the fact that the mechanisms for accessing list elements are stuck in the class that defines the list. That is, for a given list object

- we are always forced to use the list class access methods
- there is only one current element pointer

339



## Iterators

```
template <class dataclass>
class elem {
    friend class list<dataclass>;
    friend class iterator<dataclass>;
private: elem *next;
        dataclass *data;
public:
    elem(dataclass *d, elem *link) {
        data = d; next = link;
    }
};

template <class listclass>
class list {
    friend class iterator<listclass>;
private: elem<listclass> *first;
        elem<listclass> *last;
public:
    list() {
        first = NULL;
        last = first;
    }
    void add(listclass *d) {
        first = new elem<listclass>(d, first);
    }
};

template <class itclass>
class iterator {
private: elem<itclass> *first;
        elem<itclass> *current;
public:
    itclass *get_elem() {
        itclass *d;
        if(current) {
            d = current->data;
            current = current->next;
            return d;
        }
        else
            return NULL;
    }
    void reset(list<itclass> *l) {
        first = l->first;
        current = first;
    }
};
```

340

## Using the Iterator

```
int main() {
    book *b;      list<book> booklist;      iterator<book> bookit;
    geom *g;      list<geom> geomlist;      iterator<geom> geomit;

    booklist.add(new book("The Stand", "King", 1135));
    booklist.add(new book("To Kill A Mockingbird", "Lee", 336));
    booklist.add(new book("The Hitchhiker's Guide to the Galaxy", "Adams", 216));
    booklist.add(new book("The Ascent of Man", "Bronowski", 342));

    bookit.reset(&booklist);
    while(b = bookit.get_elem()) {
        cout << b->get_title() << " by ";
        cout << b->get_author() << " (";
        cout << b->get_pages() << " pages)\n";
    }

    geomlist.add(new rectangle(3, 2));
    geomlist.add(new rectangle(4, 3));
    geomlist.add(new triangle(6, 3));
    geomlist.add(new triangle(5, 3));

    geomit.reset(&geomlist);
    while(g = geomit.get_elem())
        cout << g->get_area() << endl;

    return 0;
}
```

341

## An Even More General List

```
template <class dataclass>
class elem {
    friend class list<dataclass>;
    friend class iterator<dataclass>;
private: elem *next, *prev;
        dataclass *data;
public:
    elem(dataclass *d, elem *link){
        data = d; next = link;
        prev = NULL;
        if(next)
            next->prev = this;
    }
};
```

```
template <class listclass>
class list {
    friend class iterator<listclass>;
private: elem<listclass> *first;
        elem<listclass> *last;
public:
    list() {
        first = NULL;
        last = first;
    }
    void add(listclass *d) {
        first = new elem<listclass>(d, first);
        if(!first->next)
            last = first;
    }
};
```

342

## An Even More General Iterator

```
template <class itclass>
class iterator {
private: elem<itclass> *first;
        elem<itclass> *current;
public:
    itclass *get_elem() {
        if(current)
            return current->data;
        else
            return NULL;
    }
    void fwd() {
        if(current) current = current->next;
    }
    void rev() {
        if(current) current = current->prev;
    }
    void rewind(list<itclass> *l) {
        first = l->first;
        current = first;
    }
    void fastforward(list<itclass> *l) {
        first = l->first;
        current = l->last;
    }
};
```

343

## Using the Even More General List and Iterator

```
int main() {
    geom *g1, *g2;
    list<geom> geomlist;
    iterator<geom> geomit1, geomit2;

    geomlist.add(new rectangle(3, 2));
    geomlist.add(new rectangle(4, 3));
    geomlist.add(new triangle(6, 3));
    geomlist.add(new triangle(5, 3));

    geomit1.rewind(&geomlist);
    geomit2.fastforward(&geomlist);
    while((g1 = geomit1.get_elem()) && (g2 = geomit2.get_elem())) {
        cout << g1->get_area() << "\t";
        cout << g2->get_area() << "\n";
        geomit1.fwd();
        geomit2.rev();
    }

    return 0;
}
```

344