

Associative Containers

Associative Containers

- Elements are stored and retrieved by a **key**
- The two primary associative containers are **map** and **set**
 - The elements in a **map** are *key-value* pairs. The key serves as index into the map, and the value represents the actual data
 - A **set** contains only keys and supports efficient queries to whether a given key is present
- An object of **map** or **set** type may contain only a single element with a given key. If we need to have multiple elements with a single key, then we can use *multimap* or *multiset*

The *pair* type

- A simple library type defined in the *utility* header
- A *pair* holds two data values of possibly different types. Like the containers, pair is a template type

```
using std::pair;
```

```
pair<string,string> a;
```

```
pair <string,int> b;
```

```
pair<string,string> c ("Hello","World");
```

```
a.first;           //first and second are PUBLIC data members of pair
```

```
a.second;
```

Associative Containers

- Associative containers share many but not all of the operations of sequential containers. They do not have the *front*, *push_front*, *pop_front*, *back*, *push_back* and *pop_back* operations
- Default, copy, and iterators range constructors
- The relational operators
- The *begin end rbegin* and *rend* operations
- The class-associated types such as *size_type*, etc. (Note that the *value_type* for *map* is pair type)
- The *swap* and assignment operator (but not the *assign* functions.)
- The *clear* and *erase* operations
- The basic size operations

Elements are ordered by key

- The associative container type defines additional operations, and redefines the meaning of return type operations that are common with the sequential containers

Important: when we iterate across an associative container, we are guaranteed that the elements are accessed in key order, irrespective of the order in which the elements were placed in the container

map

- Map is a collection of key-value pairs, and it is often referred to as an *associative array*. It is like the built-in array types or vectors, in which the key can be used as an index to fetch a value
- Whenever we use an associative container, its key type must have an associated *comparison function*. The comparison function must define a *complete weak ordering* over the key type.
 - By default, the library uses the > operator, so the operator for the key type should "do the right thing"

```
#include<map>
using std::map;
map<K,V> m;           //creates an empty map
map<K,V> m2 (m);      //creates m2 as copy of m; m and m2 must have the same
                      //key and value types
```

Types define by map

- `map<K,V>::key_type`
 - The type of the keys (K)
- `map<K,V>::mapped_type`
 - The types of the values associated with the keys (V)
- `map<K,V>::value_type`
 - A pair whose first element has type `const map<K,V>::key_type` and second has type `map<K,V>::mapped_type`

When learning the *map* interface, it is essential to remember that the *value_type* is a *pair* and that we can change the value but not the key member of that pair

Dereferencing and Subscripting a *map*

```
map<string,int> word_count;    //empty map
//get an iterator to an element in word_count map
map<string,int>::iterator map_it = word_count.begin ();

/*map_it is a reference to a pair <const string,int> object
cout << map_it ->first;        //prints the key
cout << " " << map_it -> second; //prints the value
map_it -> first = "new key";    //ERROR: key is const
map_it ->second = 5;            //OK

map<string,int> word_count;    //empty map
//insert default initialized element with key AA
word_count["AA"] = 1;
cout << word_count["AA"] << endl; //output: 1
++word_count["AA"];
cout << word_count["AA"] << endl; //output: 2
```

Important: Subscribing a map behaves differently from subscribing an array or *vector* : using an index that is not already present *adds* an element with that index (key) to the map

Dereferencing and subscribing a map

```
map<string,int> word_count;    //empty map
string word;

while (cin >> word) {
    ++word_count[word];
}
```

Unlike *vector* and *string*, the type returned by *map* subscript operator *differs* from the type obtained by dereferencing a *map* iterator

When we use a subscript to add an element to a *map*, the value part of the element is value-initialized. Often, we immediately assign to that value, which means that we have initialized and assigned the same object

Using map::insert

```
map<string,int> m;  
map<string,int>::iterator iter;  
  
pair<map<string,int>::iterator,bool> p (m.insert (make_pair ("aaa",1)));  
cout << p.second << endl;    //output is 1  
  
pair<map<string,int>::iterator,bool> q  
    (m.insert ( map<string,int>::value_type ("aaa",1)));  
cout << q.second << endl;    //output is 0
```

Finding and retrieving a map element

```
map<string,int> word_count;  
int occurs = word_count["AA"];
```

Using a subscript has an important side effect: If that key is not already in the map, then subscript inserts an element with that key

m.find(k) Returns an iterator to the element indexed by k, if there is one, or returns an off-the-end iterator if k is not present

```
int occurs = 0;  
map<string,int>::iterator it=word_count.find ("A");  
if (it != word_count.end ())  
    occurs = it->second;
```

Removing elements from map

```
map<int,double> m;  
map<int,double>::iterator iter = m.begin ();  
int k;  
m.erase (k);           //removes element with key k from m  
m.erase (iter);
```

Iterating across a map

```
// for each element in the map  
for (map<string, int>::const_iterator  
map_it = word_count.begin(); map_it != word_count.end(); ++map_it) {  
    // print the element key, value pairs  
    cout << map_it->first << " occurs "  
    << map_it->second << " times" << endl;  
    ++map_it; // increment iterator to denote the next element  
}
```

The *set* Types

- A *set* is just a collection of keys (that serve also as values), and it is most useful when we want to know whether a value is present
 - Example: a business might define a *set* to hold the names of individuals who have issued bad checks to the company
- With two exceptions, *set* supports the same operations as *map*:
 - All the common container operators
 - Constructors similar to those of *map*
 - The insert operations
 - The *find* operation
 - The *erase* operations
- The exceptions are that *set*
 - Doesn't provide a subscript operator
 - Doesn't define a *mapped_type*

The *multimap* and *multiset* types

- Both *map* and *set* can contain a single instance of each key. The types *multimap* and *multiset* allow multiple instances of the same key
 - For example, in a phone directory, someone might wish to provide a separate listing for each phone number associated with an individual
- The operations supported on *multimap* and *multiset* are the same as those on *map* and *set*, except that *multimap* doesn't support subscribing
- Note: the operations that are common to *map* and *multimap* (or *set* and *multiset*) change in various ways to reflect the fact that there can be multiple keys

```
#include<map>
using std::multimap;

multimap<string,string> authors;
//add data to authors
authors.insert(pair<string,string>("Dvora Omer", "book1"));
authors.insert(pair<string,string>("Dvora Omer", "book2"));
authors.insert(pair<string,string>("Dvora Omer", "book3"));

string search_item ("Dvora Omer");

//how many entries are there for this author
typedef multimap<string,string>::size_type sz_type;
sz_type numEntries = authors.count (search_item);

//get iterator for the first entry of this author
multimap<string,string>::iterator iter= authors.find(search_item);
//loop through the number of entries there are for this author
for (sz_type cnt = 0; cnt != numEntries; ++cnt, ++iter) {
    cout << iter->second << endl;           // print each title
}
```

Iterating

We are guaranteed that iterating across a *multiset* or a *multimap* returns all the elements with a given key in a sequence

A different iterator-oriented solution

```
typedef multimap<string,string>::iterator authors_it;  
authors_it beg = authors.lower_bound (search_item);  
authors_it end = authors.upper_bound (search_item);  
  
// loop through the number of entries there are for this author  
  
while (beg != end) {  
    cout << beg -> second << endl;    //print each title  
    ++beg;  
}
```

If the element is not in the container, then *lower_bound* and *upper_bound* will return iterator referring to the end of the container

Copy control

```
class A {  
    ...  
private:  
    B *fieldPtr;  
}
```

What should the copy constructor of class A do?

Managing pointer members

Three options:

1. The pointer member can be given normal pointerlike behavior
 - such classes will have all the pitfalls of pointers but will require no special copy control
2. The class can implement a so called "**smart pointer**" behavior. The object to which the pointer points is shared, but the class prevents dangling pointers
3. The class can be given **valuelike behavior**. The object to which the pointer points will be unique and managed separately by each class object.

```

class HasPtr {
public:
    HasPtr(int *p, int i) : ptr(p), val (i) {}
    int *get_ptr () const {return ptr;}
    int get_int () const {return val;}
    void set_ptr (int *p) {ptr = p};
    void set_int (int i) { val = i};
    int get_ptr_val () const {return *ptr;}
    void set_ptr_val(int val) const {*ptr = val;}
private:
    int *ptr;
    int val;
};

int obj = 0;
HasPtr ptr1 (&obj, 42); //int* member points to obj, val is 42
//synthesized copy constructor will be called here
HasPtr ptr2 (ptr1);      //int* member points to obj, val is 42

// now the pointers in ptr1 and ptr2 address the same object and int values
// in object are the same

ptr1.set_ptr_val(42); //set object to which both ptr1 and ptr2 point
ptr2.get_ptr_val (); // returns 42

```

Dangling pointers

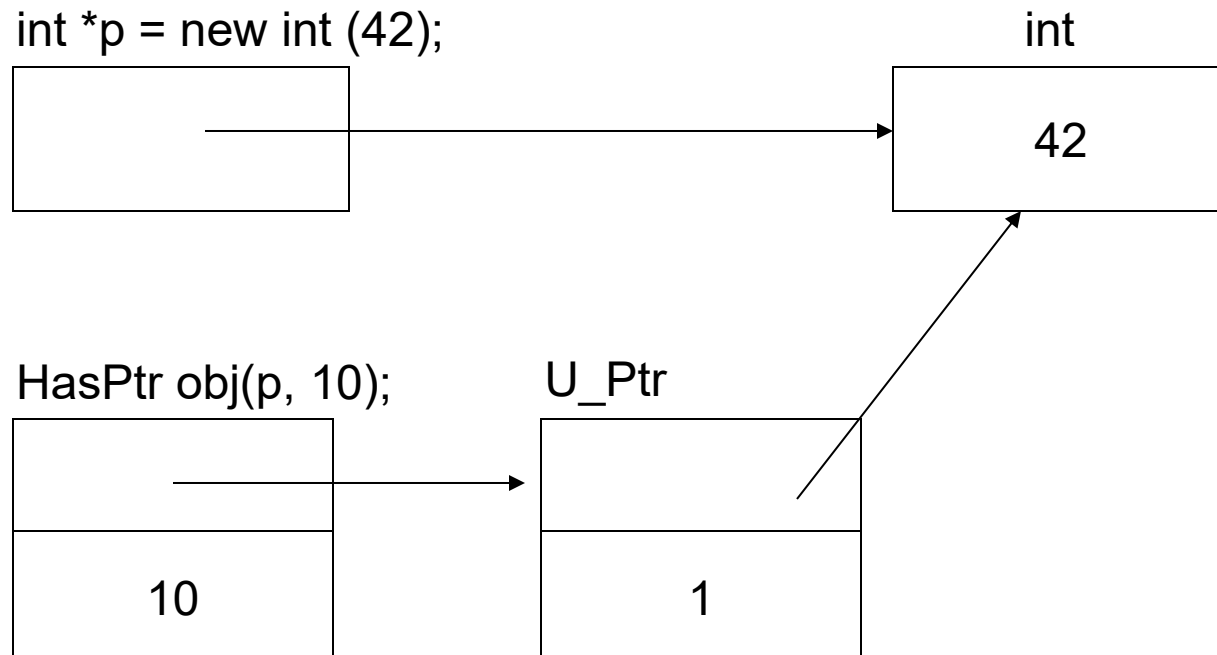
```
int *ip = new int (42); //dynamically allocated int initialized to 42  
HasPtr ptr (ip, 10);    //HasPtr points to same object as ip does  
delete ip;              //object pointed to by ip is freed  
ptr.set_ptr_val (0);    //Disaster! The object to which HasPtr points was freed
```

Smart pointer classes

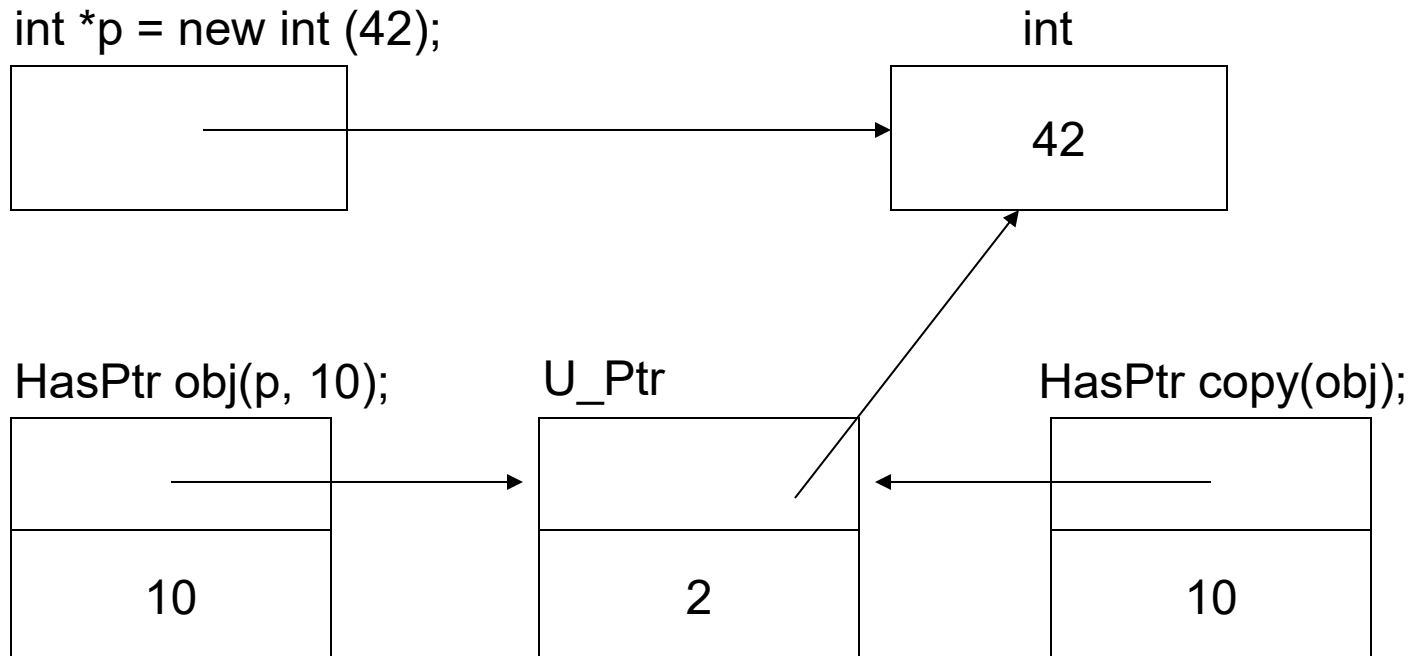
```
//private class for use by HasPtr only
class U_Ptr {
    friend class HasPtr;
private:
    int *ip;
    size_t use; //called reference count (a.k.a. use count)
    U_Ptr(int *p): ip(p),use(1) {}
    ~U_Ptr {delete ip;}
};

class HasPtr {
public:
    //HasPtr owns the pointer; p must have been dynamically allocated
    HasPtr (int *p, int i): ptr (new U_Ptr(p)), val (i) { }
    // copy members and increments the use count
    HasPtr (const HasPtr &orig): ptr(orig.ptr), val(orig.val) { ++ptr->use;}
    HasPtr& operator=(const HasPtr&);
    // If use count goes to zero, delete the U_Ptr object
    ~HasPtr () { if (--ptr->use == 0) delete ptr;}
private:
    U_Ptr *ptr; //points to use-counted U_Ptr class
```

The picture



The picture (contd.)



Assignment and use (reference) counts

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++rhs.ptr->use; //increment use count on rhs first
    if (--ptr->use == 0)
        delete ptr;      //if reference count goes to 0 on this object, delete it
    ptr = rhs.ptr;        //copy the U_Ptr object
    val = rhs.val;        //copy the int member
    return *this;
}
```

Note: the assignment operator guards against self-assignment by incrementing the use count of *rhs* before decrementing the use count in the underlying U_Ptr object

Changing other members

```
class HasPtr {  
public:  
    // copy control and constructors as before  
    // accessors must change to fetch value from U_Ptr object  
    int *get_ptr() const {return ptr->ip;}  
    int get_int () const {return val;}  
    //change the appropriate data member  
    void set_ptr(int *p) {ptr->ip = p;}  
    void set_int(int i) {val = i;}  
    //return or change the value pointed to, so ok for const objects  
    int get_ptr_val () const {return *ptr->ip;}  
    void set_ptr_val (int i) const {*ptr ->ip = i;}  
private:  
    U_Ptr *ptr;           //points to use-counted U_Ptr class  
    int val;  
};
```

Defining valuelike classes

- Classes with ***value semantics*** define objects that behave like the arithmetic types: when we copy a valuelike object, we get a new, distinct copy
 - changes made to the copy are not reflected in the original and vice versa
 - the *string* class is an example of a valuelike class

```

class HasPtr {
public:
    //no point to passing a pointer if we're going to copy it anyway
    //store pointer to a copy of the object we're given
    HasPtr(const int &p, int i) : ptr(new int (p)), val (i) {}
    HasPtr(const HasPtr &orig): ptr(new int (*orig.ptr)), val (orig.val)
        {}
    HasPtr& operator=(const HasPtr&);
    ~HasPtr() {delete ptr;}
    int get_ptr_val() const {return *ptr;}
    int get_int () const {return val;}
    void set_ptr (int *p) {ptr = p;}
    void set_int (int i)  {val = i;}
    int *get_ptr() const {return ptr;}
    void set_ptr_val(int p) const {*ptr = p;}
private:
    int *ptr;           // points to an int
    int val;
};

```

```
HasPtr& HasPtr::operator=(const HasPtr &rhs){
{
    //note: Every HasPtr is guaranteed to point at an actual int;
    // We know that ptr cannot be a zero pointer
    //no need to check for self assignment
    *ptr = *rhs.ptr; // copy the value pointed to
    val = rhs.val;   // copy the int
    return *this;
}
```

Design Patterns

Chapter notes of the book:

Design Patterns (Elements of Reusable Object-Oriented Software)

By Gamma, Helm, Johnson and Vlissides

Object-Oriented design

- Designing object-oriented software is hard
- Designing *reusable* object-oriented software is even harder
- The design should be specific to the problem at hand
 - but also general enough to address future problems and requirements
 - We want to avoid redesign, or at least minimize it
- Expert designers know what ***not*** to do
 - Do not solve every problem from first principles !
- There are recurring patterns of classes and communicating objects in many object oriented systems
 - Such patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately re-usable
 - e.g., "represent states with objects", "decorate objects so you can easily add/remove features"

A design pattern

- Name
- Problem
 - when to apply the pattern
- Solution
 - the elements that make up the design, their relationships, responsibilities, and collaborations
- Consequences
 - Results and trade-offs of applying the pattern

Catalog of patterns

- **Purpose** of a pattern
 - Creational
 - process of object creation
 - Structural
 - composition of classes/objects
 - Behavioral
 - characterize the ways in which classes or objects interact and distribute responsibility
- **Scope** of a pattern
 - pattern applies to
 - classes - static, mainly via sub-classes and inheritance
 - objects - dynamic

Singleton

- A creational pattern (at object level)
- Intent: Ensure a class only has one instance, and provide a global point of access to it
- Motivation
 - One printer spooler in the system
 - There could be many printers
 - One file system
 - One window manager

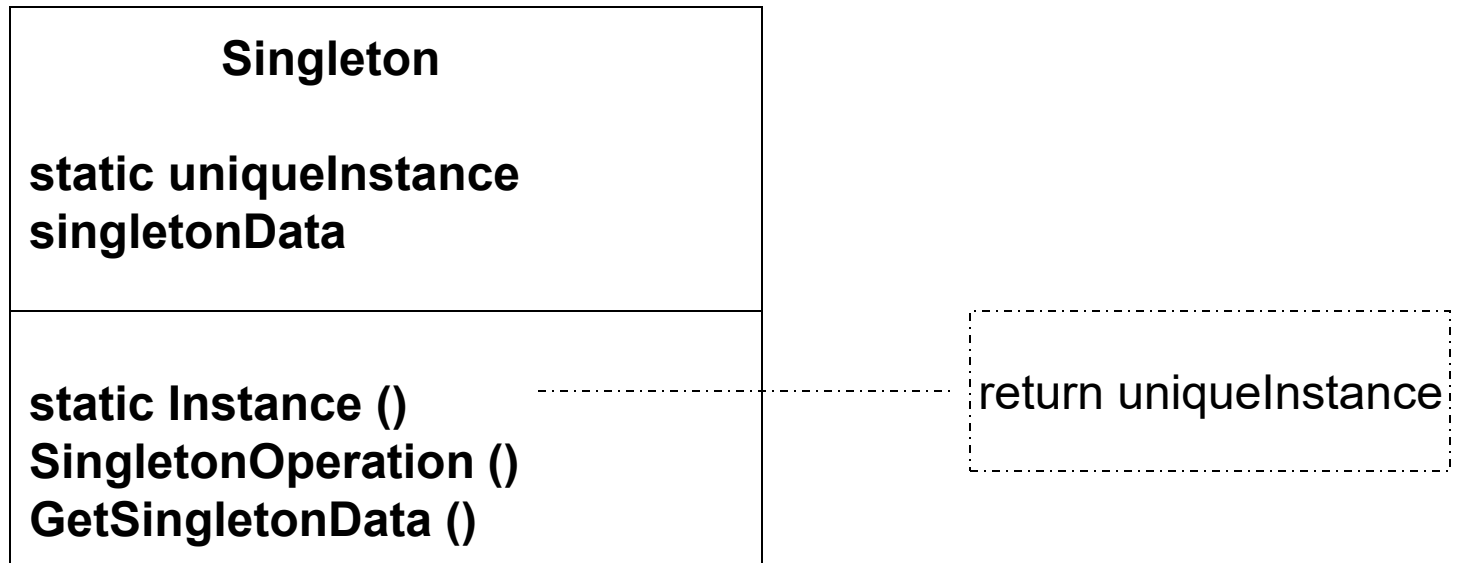
Solutions

- Keep a global variable (wrong !)
 - Makes the object accessible
 - Doesn't keep you from instantiating multiple objects
- Make the class itself responsible for keeping track of its sole instance (right !)
 - The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance

When to use Singleton ?

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- When the sole instance should be extensible by inheritance, and clients should be able to use an extended instance without modifying their code

Structure



Singleton.h

```
class Singleton {  
public:  
    static Singleton* Instance ();  
protected:  
    Singleton ();  
private:  
    static Singleton* _instance;  
};
```

Singleton.cc

```
Singleton* Singleton::_instance = 0;  
  
Singleton* Singleton::Instance () {  
    //lazy initialization  
    if (_instance == 0) {  
        _instance = new Singleton;  
    }  
    return _instance;  
}
```

Consequences

- Benefits of the Singleton class
 - Controlled access to sole instance
 - Reduced name space
 - Compare to global variables
 - Permits refinement of operations and representations
 - The Singleton class may be subclassed
 - Permits a variable number of instances

Adapter

- Structural pattern
- Also known as ***wrapper***
- Intent: convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible types

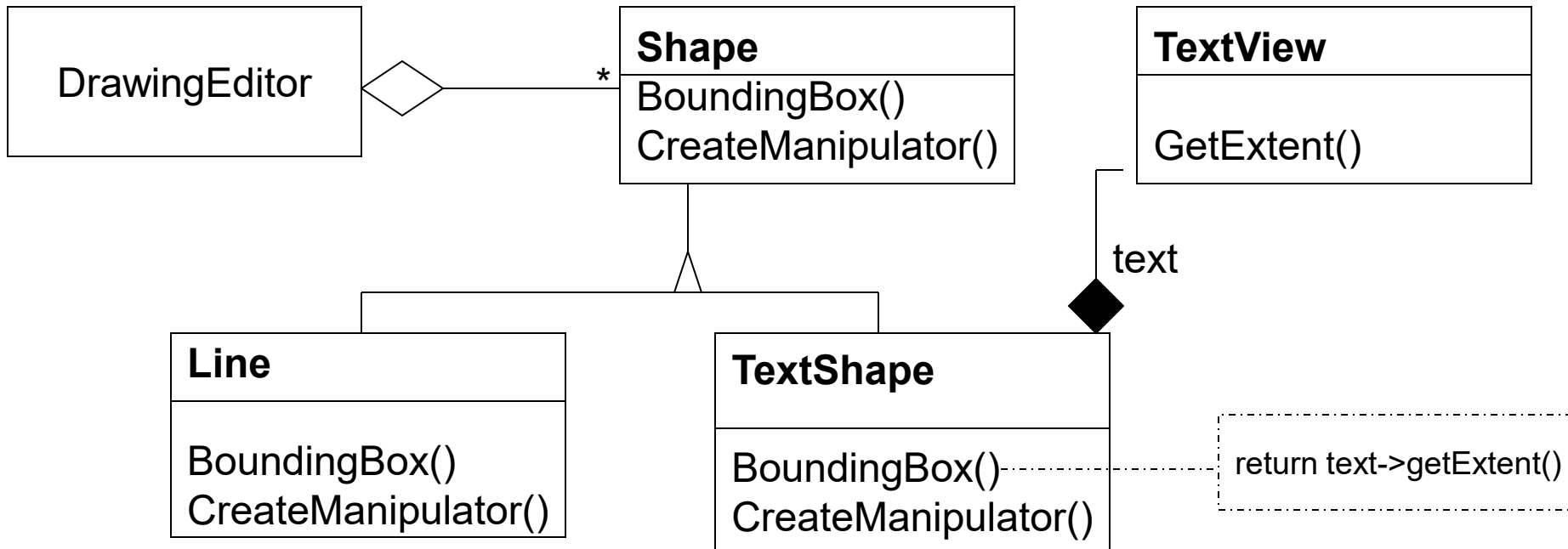
Motivation for using an adapter

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires
- Example:
 - Drawing editor that lets users draw and arrange graphical elements into pictures and diagrams
 - The interface for graphical objects is defined by an abstract class called *Shape*
 - The editor defines a derived class of *Shape* for each kind of graphical object
 - *LineShape* and *PolygonShape* are rather easy to implement, but *TextShape* is more difficult
 - An off-the-shelf user interface toolkit might already provide a sophisticated *TextView* class for displaying and editing text
 - We would like to reuse *TextView* to implement *TextShape*, but the toolkit wasn't designed with *Shape* classes in mind
 - So we cannot use *TextView* and *Shape* objects interchangeably

What can we do ?

- We could change the *TextView* class so that it conforms to the *Shape* interface
 - It's not an option unless we have the toolkit's source code
 - Even if we did, it wouldn't make sense
- The correct solution:
 - Define *TextShape* so that it **adapts** the *TextView* interface to *Shape's*
 - Two ways:
 - Inheriting *Shape's* interface and *TextView's* implementation (class-based solution)
 - Composing a *TextView* instance within a *TextShape* and implementing *TextShape* in terms of *TextView's* interface (object-based solution)

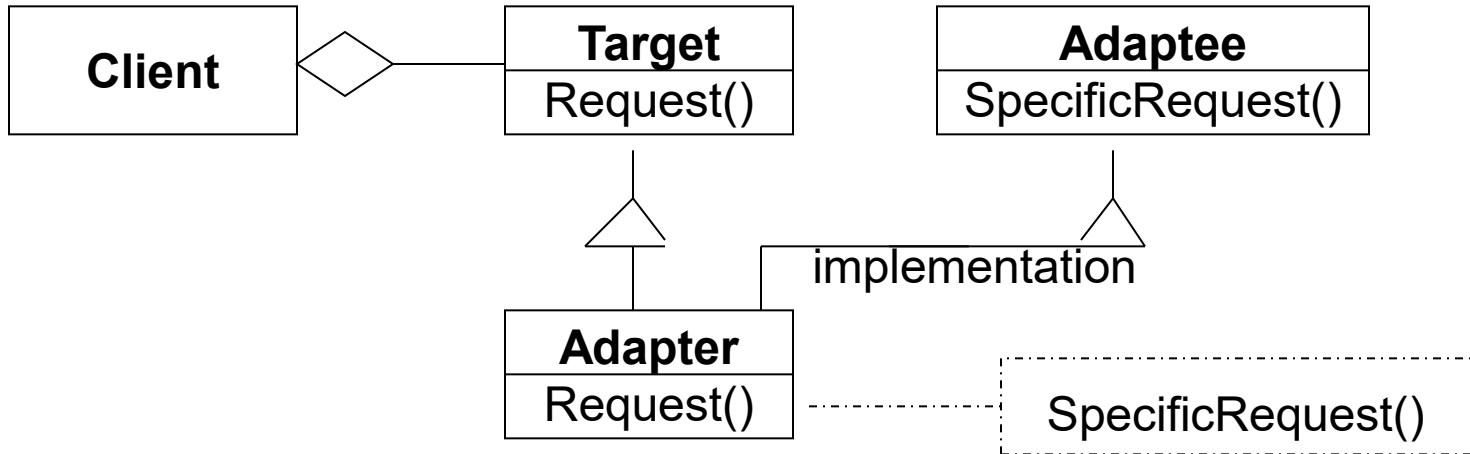
Object Adapter



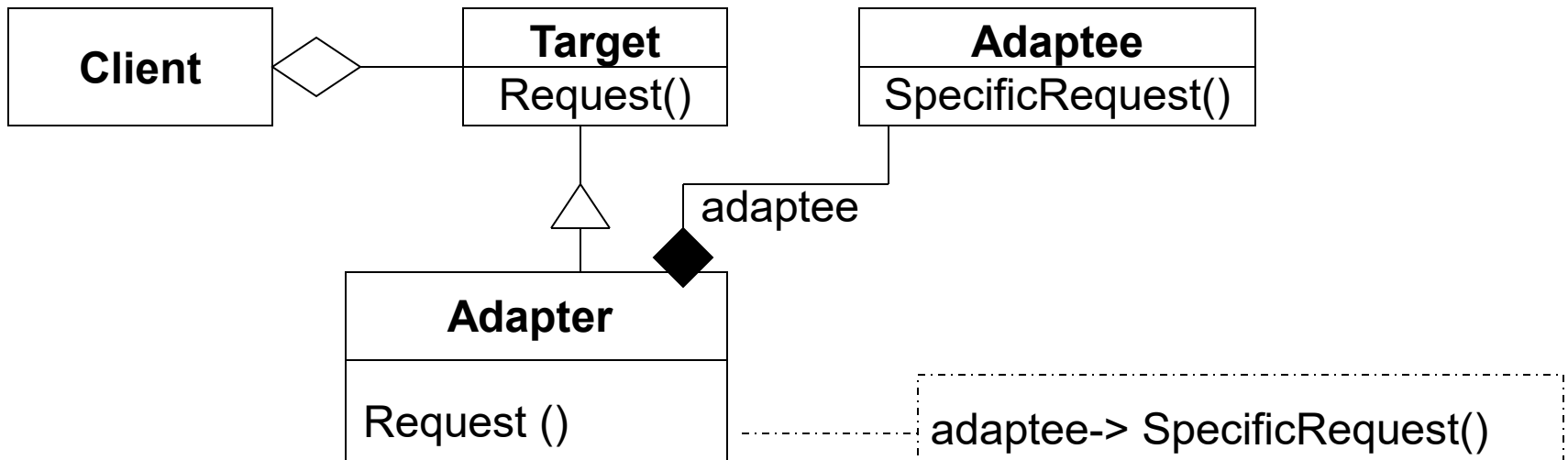
Applicability

- Use the adapter pattern when
 - You want to use an existing class, and its interface doesn't match the one you need
 - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible types
 - (for object adapter only) you need to use several existing derived classes, but it's impractical to adapt their interface by deriving every one. An object adapter can adapt the interface of its parent class

Class Adapter



Object Adapter



What does it mean "implementation inheritance" ?

- The derived class uses the inherited class in its implementation but does not expose the fact of the inheritance as part of its interface
- private inheritance in C++

Private inheritance

All the members of the base class are
private in the derived class

Private inheritance (contd.)

```
class Base {
public:
    void basemem ();          //public member
protected:
    int i;                    //protected member
};

class Private_derived : private Base {
public:
    int use_base () {return i;}      //ok: derived class can access i
};

class Derived_from_private : public Private_derived {
public:
    //error: Base::i is private in Private_derived
    int use_base () {return i;}
};

Private_derived d;
d.basemem ();      //error: basemem is private in the derived class
```

Back to adapter - consequences

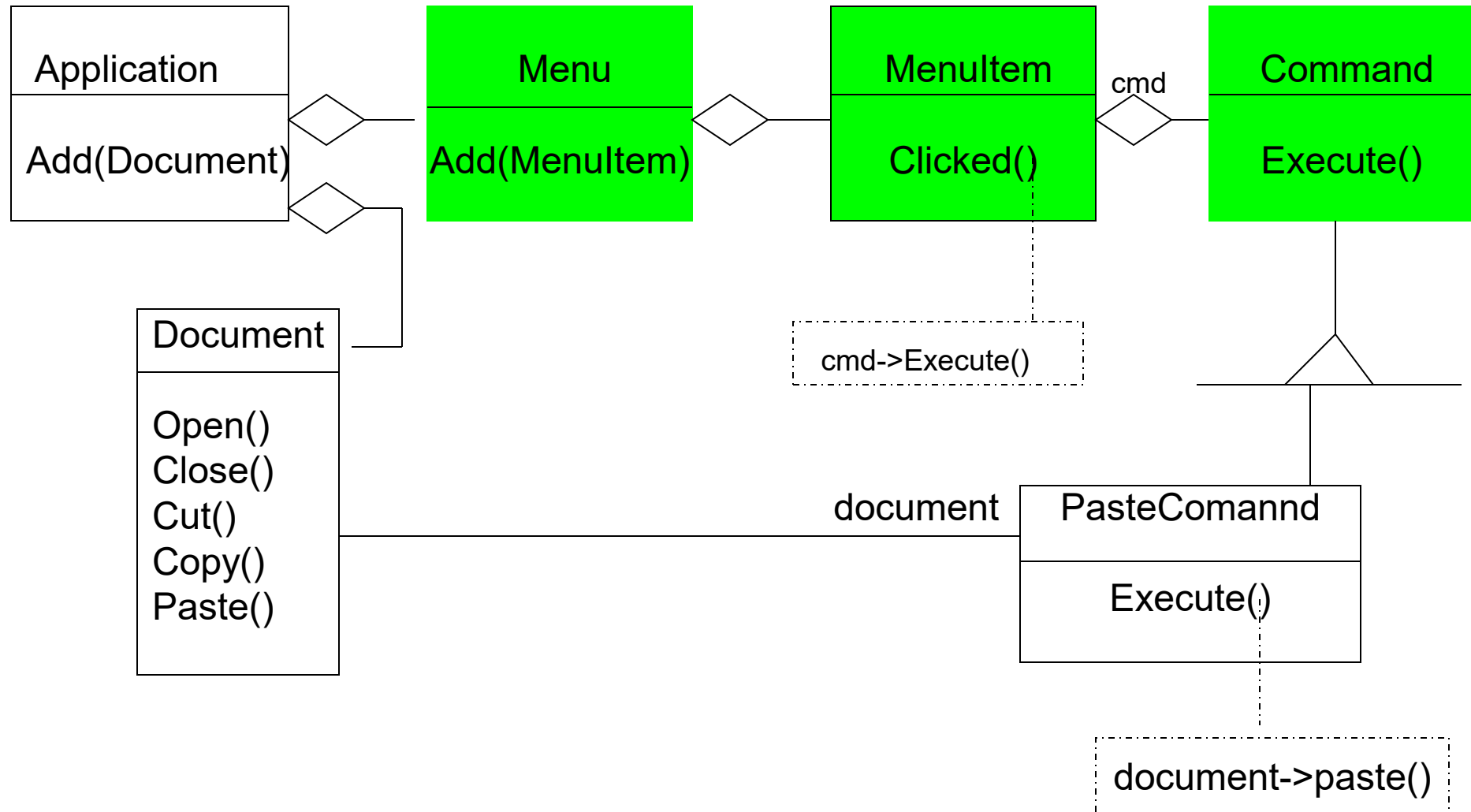
- Class and object adapters have different trade-offs
- A class adapter
 - Adapts *Adaptee* to *Target* by committing to a concrete *Adaptee* class. As a consequence, a class adapter won't work when we want to adapt a class and all its derived classes
 - Lets *Adapter* override some of *Adaptee*'s behavior, since *Adapter* is a derived class of *Adaptee*
 - Introduces only one object, and no additional pointer indirection is needed to get to the adaptee
- An object adapter
 - Lets a single *Adapter* work with many *Adaptees* – that is, the *Adaptee* itself and all of its derived classes (if any). The *Adapter* can also add functionality to all *Adaptees* at once
 - Makes it harder for *Adapter* to override *Adaptee* behavior. It will require derivation from *Adaptee* and making *Adapter* refer to the derived class rather than to the *Adaptee* itself

Command

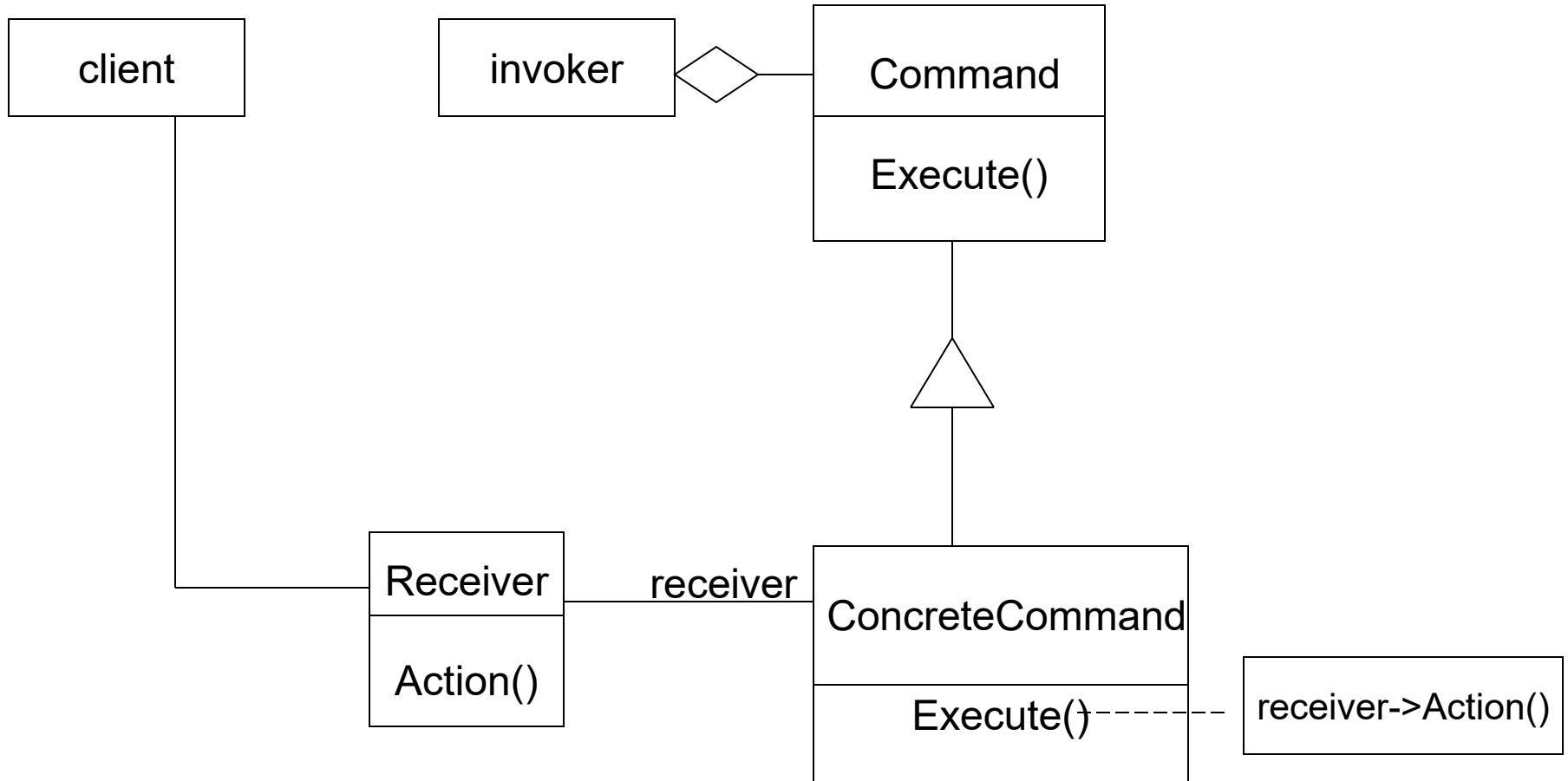
- Behavioral, at the object level
- Also known as Action, Transaction
- Intent:
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, and support undoable operations

Motivation for using Command

- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
 - UI toolkit includes objects like buttons and menus that carry out a request in response to user input. But, the toolkit cannot implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which objects. As toolkit designers we have no way of knowing the receiver of the request or the operations that will be carried out



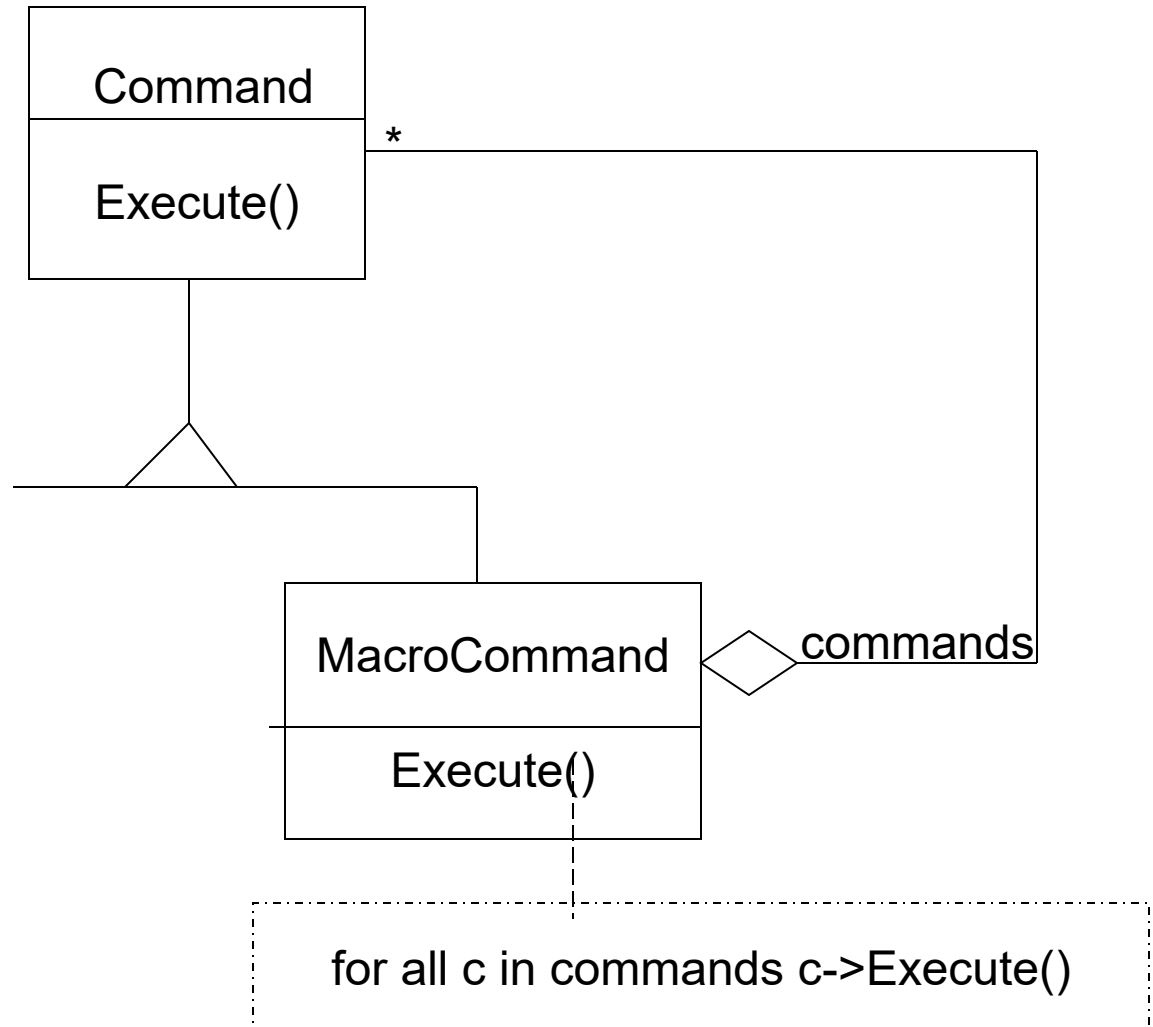
Structure



What's going on here ?

- The *Command* pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object
 - This object can be stored and passed around like other objects
- The key to the pattern is an abstract *Command* class, which declares an interface by executing operations
 - In the simplest form, this interface includes an abstract *Execute* operation
 - Concrete *Command* derived classes specify a receiver-action pair by storing the **receiver** as an instance variable and by implementing *Execute* to invoke the request
 - The receiver has the knowledge required to carry out the request

Multiple commands



The core idea

The Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it

This gives us a lot of flexibility in designing our user interface

We can for example replace commands dynamically, which would be useful for implementing context-sensitive menus

Applicability

- Use the *Command* pattern when you want to
 - Parameterize objects by an action to perform, as MenuItem objects did.
 - Commands are object-oriented replacement for callbacks
- Specify, queue, execute requests at different times
 - A command object can have a lifetime independent of the original request
 - If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object from the request to a different process and fulfill the request there
- Support undo.
 - The *Command's* Execute operation can store state for reversing its effects in the *Command* itself
 - The *Command* interface must have an added *Unexecute* operation that reverses the effects of a previous execute
- Support logging changes so that they can be reapplied in case of a system crash
- Structure a system around high-level operations built on primitives operations.
 - Such a structure is common in information systems that support *transactions*

Consequences

- *Command* decouples the object that invokes the operation from the one that knows how to perform it
- *Commands* are first-class objects. They can be manipulated and extended like any other object
- It's easy to add new *Commands* because you don't have to change existing classes

What to expect from design patterns?

- A common design vocabulary
 - Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.
 - Design patterns make a systems seem less complex by letting you talk about it at a higher level of abstraction than that of a design notation or programming language

How can they help us?

- Make it easier to understand existing systems
- Provide solutions to common problems

All in all, help us better design object oriented systems, and become better designers and programmers

Sequential containers

Containers

- Container is a generalized name for entities that hold collections of objects of the same type
- The container classes share a **common interface**, and this interface became a standard not only for STL, but also for user defined containers
- Each container type offers a different set of *time* and *functionality tradeoffs*
- The advantage of the common interface is that often one container type can be substituted by another container type without changing our code beyond the need to change type declarations

Sequential containers

- In sequential containers, elements are stored and accessed by position
 - e.g. vector
- The *order of elements* in a sequential container is independent of the value of the elements. Instead, the order is determined by the order (and the way) in which elements are added to the container

Sequential Containers	
vector	Supports fast random (direct) access
list	Supports fast insertion/deletion
deque	Double-ended queue
Sequential container adaptors	
stack	LIFO
queue	FIFO
priority_queue	priorities

Basic containers vs. adaptors

- Effectively, an adaptor adapts an underlying container type by defining a ***new interface*** in terms of operations provided by the original container
- Containers define only a small set of basic operations that form a kind of *hierarchy*:
 - Some operations are supported by all container types
 - Other operations are common to only sequential/non-sequential containers
 - Still others are common to only subset of sequential/non-sequential containers

Creating a sequential container

```
#include<vector>
#include<list>
#include<deque>
```

```
vector<string> svec;
list<int> iList;
deque<Sales_item> items;
```

```
vector<string> svec2(svec); //ok
vector<int> ivec (svec); // error: vector svec is not vector<int>
```

```
// initialize slist with copy of each element in svec
list<string> slist (svec.begin (), svec.end ());
```

```
//initialize half_sdeq with first half of svec
deque<string> half_sdeq (svec.begin (), svec.begin() + svec.size ()/2);
```

Constraints on types that a container can hold

- The element type must support *assignment*
- We must be able to *copy* objects of the element type
- Container operations may impose additional requirements
 - constructors that take a single parameter specifying the size of the container -> requires a default constructor of the type
 - If the element type doesn't support the additional requirement, then we can define a container of that type, but may not use the particular operation

Iterators and iterator ranges

- The concept of **iterator range** is fundamental to the standard library
 - An iterator range is denoted by a pair of iterators that refer to two elements; or to one past the "element", in the same container
 - These two operators mark left-inclusive range [*first*,*last*) of elements from the container
 - If the iterators are not equal, then it must be possible to reach *last* by repeatedly incrementing *first*.

Some operations invalidate iterators

Some container operations change the internal state of a container or cause the elements in the container to be moved

- Such operations invalidate all iterators that refer to the elements that are moved and **may invalidate other iterators as well !**
- Using an invalidated iterator is similar to using a dangling pointer !

Sequential container operations

- Each sequential container defines a set of useful typedefs and supports operations that let us
 - Add elements to the container
 - Delete elements from the container
 - Determine the size of the container
 - Fetch the first, and the last elements of the container, if any

size_type	unsigned integral type large enough to hold size of largest possible container
iterator	Type of iterator of this container type
const_iterator	Type of the read-only iterator
reverse_iterator	Iterator that addresses elements in reverse order
const_reverse_iterator	Reverse-order, read-only iterator
difference_type	Signed integral type large enough to hold difference between two iterators
value_type	element type
reference	synonym to value_type&
const_reference	synonym for const value_type&

Adding elements to a sequential container

c.push_back(t)	Adds element with value <i>t</i> to the end of <i>c</i>. Returns void
c.push_front(t)	Adds element with value <i>t</i> to the front of <i>c</i>. Returns void. Valid only for list or deque.
c.insert(p,t)	Inserts element with value <i>t</i> before the element referred to by iterator <i>p</i>
c.insert(p, b, e)	Insert elements in the range denoted by <i>b</i> and <i>e</i> before the element referred to by <i>p</i>

When we add an element to a container, or initialize a container by providing a range of elements, the elements' values are copied to the container. There is no subsequent connection between the object in the container and the value from which it was copied

Remember: inserting an element might invalidate iterators !

Relational operators

- Each container supports the relational operators that can be used to *compare* two containers (of the same type).
- Comparing two containers is based on the pairwise comparison of the elements of the containers
- The comparison uses the same relational operator as defined by the element type

Accessing elements

c.back ()	Returns a reference to the last element in <i>c</i>. Undefined if <i>c</i> is empty.
c.front ()	Returns a reference to the first element in <i>c</i>. Undefined if <i>c</i> is empty.
c.end ()	Returns a "reference" to one-after-the-last "element" in <i>c</i>.
c[n]	Returns a reference to an element indexed by <i>n</i>. Valid only for vector or deque.
c.at(n)	Returns a reference to the element indexed by <i>n</i>. Valid only for vector or deque.

Erasing elements

c.pop_back ()	Removes the last element in c. Returns void. Undefined if c is empty.
c.pop_front(t)	Removes the first element in c. Returns void. Undefined if c is empty. Valid only for <i>list</i> or <i>deque</i>.
c.erase(p)	Removes element referred to by iterator <i>p</i>. Returns an iterator referring to the element after the one deleted.
c.erase(b,e)	Removes the range of elements denoted by iterators <i>b</i> and <i>e</i>.
c.clear ()	Removes all the elements in c.

Remember: removing an element might invalidate iterators !

Assignment and Swap

c1=c2	Deletes elements from <i>c1</i> and copies elements from <i>c2</i> to <i>c1</i>. <i>c1</i> and <i>c2</i> must be the same type.
c1.swap(c2)	Swap contents: After the swap <i>c1</i> has elements that were in <i>c2</i> and vice versa
c.assign(b,e)	Replaces the elements in <i>c</i> by those in the range denoted by iterators <i>b</i> and <i>e</i>

The important thing about swap is that it doesn't delete or insert any elements and is guaranteed to run in *constant* time. No elements are moved, and so iterators are not invalidated !

How a vector grows

- When we insert or push an element into a container, the size of that container increases by one. The library takes care of allocating memory to hold the new element.
- Ordinarily we shouldn't care how a library (or user-defined) type works: all we should care about is how to use it (i.e., its interface).
- However, in the case of *vectors*, a bit of implementation leaks into its interface. To support fast random access, vector elements are stored *contiguously* – each element is adjacent to the previous element.

Deciding which container to use

- Whether elements are stored contiguously has other significant impacts on:
 - The costs to add and delete elements from the middle of the container
 - The costs to perform non sequential access to elements of the container

The degree to which a program does these operations on the object collection in question should be used to decide which type of container to use
- The *vector* and *deque* types provide fast non-sequential access to elements *at the cost* of making it expensive to add or remove elements anywhere other than the end of the container
- The *list* type supports fast insertion and deletion anywhere but *at the cost* of making non-sequential access to elements expensive

Rules of thumb

- If the program requires random access to elements, use a *vector* or *deque*
- If the program needs to insert or delete elements in the middle of the container, use a *list*
- If the program needs to insert or delete elements in the front and the back, but not in the middle, of the container, use a *deque*
- If we need to insert elements in the middle of the container only while reading input and then need random access to the elements, consider reading them into a *list* and then reordering the *list* as appropriate for subsequent access and copying the reordered list into a *vector*

Software Engineering (094219)

Spring 2015

Faculty of Industrial Engineering and Management

Instructor

Prof. Oren Kurland

Administration

- Moodle
 - login to Moodle at least once ASAP
 - messages from course staff
 - syllabus, lecture notes, notes for tutorial sessions, HWs, etc.
- Grade
 - HWs: (TAKEF) 4-5 assignments; worth 16%-20% all together; each is worth 3%-5%
 - Final exam; TAKEF; 80%-84%
 - To pass the course you must pass the final exam
 - You must pass all prerequisite courses (“drishtot kdam”) to take the course.
 - If you take both Moed A and Moed B of the final exam, the grade of Moed B counts
 - Special exam (“moed mehuad”) is only for those who served in the army (miluimnikim)
- Textbook
 - C++ Primer (4th edition), by Lippman, Lajoie, and Moo

Course staff

- Instructor: Prof. Oren Kurland
 - Office hours: Monday 13:00-13:30, Bloomfield 308
- TA in charge: Ira Leviant
- TAs teaching sessions:
 - Ira Leviant
 - Rotem Suarez

קוד אתי

אסור בתכלית האיסור להגיש שיעורי בית המכילים חומר שאינו מקורי שלכם. כל העתקה, גדולה או קטנה, בין אם בשעורי הבית, או בבחינה, בין אם ע"י שימוש בחומר של סטודנטים אחרים בקורס, סטודנטים משנים קודמות, או מקורות אחרים (לדוגמה האינטרנט), תוביל לכישלון מידי בקורס ולהופעה בפני וועדת המשמעת של הטכניון.

תקנון משמעת של הפקולטה להנדסת תעשייה וניהול והטכניון

על הסטודנטים בקורס לקיים את התקנון המשמעת של הטכניון באופן מלא. בפרט, ייאכפו בקורס הנהלים הבאים:

- כניסה לכיתה באיחור תעשה אך ורק דרך הדלת האחורית של הכיתה. על סטודנט המאחר לשעור להיכנס בשקט ולהתיישב בירכתי הכיתה על מנת שלא להפריע למהלך השיעור.
- אכילה מותרת בחלקה האחורי של הכיתה בלבד.
- דיבור במהלך ההרצאה ייעשה באישור המרצה בלבד.
- אין להשתמש בטלפון סלולארי בכיתה ואין לקרוא עיתון במהלך השיעור.

Tentative topics

- Basics of C++
- Object oriented programming and design
- Generic programming
- Design patterns
- Advanced topics

Developing software vs. writing programs

Program

- Typically short
- Developed by individual
- Used by few people
- Minor debugging is enough to get things going
- Can be coded immediately

Software

- Typically long
- Developed by a team
- Used by some customers
- Requires analysis and design
- Requires long-term maintenance

Running example

- A bookstore keeps a file of transactions, each of which records the sales of a given book
- Each transaction contains:
 - ISBN (unique identifier) of the book
 - the number of copies sold
 - the price at which each copy was sold
- Periodically, the bookstore owner reads the file and computes
 - the number of copies of each title that was sold
 - the total revenue from that book
 - the average sales price

Write a program for doing these computations

How shall we start ?

- Basic features of C++
- Write, compile, execute
- Some building blocks
 - variables
 - input/output
 - data structure
 - test whether 2 records have the same ISBN
 - design a loop that will process all records in the transaction file
- Let's go!

Some basics

- Basic program (resides in file prog.cpp, for example)

```
int main() {  
    //add your declarations and commands  
    return 0;  
}
```

Standard input and output

cin: standard input (type of istream)

cout: standard output (type of ostream)

cerr: standard error (type of ostream)

```
#include <iostream>
/* Simple main function: read two numbers and write their sum */
int main ()
{
// prompt user to enter two numbers
std::cout << "Enter two numbers:" << std::endl;
int v1, v2;
std::cin >> v1 >> v2;
std::cout << "The sum of " << v1 << " and " << v2 << " is " << v1 +
    v2 << std::endl;
return 0;
}
```

Expressions and statements

- $v1+v2$ is an *expression*
 - *expression* is composed of one or more *operands* that are combined by an *operator*
 - simplest form of an *expression* consists of a single literal constant (e.g., 5 or '5') or variable (e.g., $v0$)
 - every expression yields a *result*
- $v0 = 5;$ is a *statement*
 - in particular it's called an *expression statement*
 - $v0 = 5$ is an expression
 - $;$ // null statement
 - $\text{int } x;$ is a *declaration statement*
- A *block* is a *compound statement*
 - {
 - //statements
 - }

Initialization, comments, control structure

- variables:
 - `int val1 = 0;` *// val1 is initialized*
 - `int val2;` *// val2 is uninitialized*
- comments in C++
 - *// my comment*
 - */* my comment */*
- control structure
 - statements are executed sequentially

```
int sum = 0, val = 1;  
while (val <= 10) {  
    sum += val; // sum = sum + val; compound assignment operator  
    ++val;      // val = val + 1; prefix increment operator  
}
```

The for loop

```
int sum = 0;  
for (int val = 1; val <= 10; ++val) {  
    sum += val;  
} // no need for parentheses in this case
```

If statement

```
int v1, v2;  
// read values for v1 and v2  
if (v1 < v2) {  
    std::cout << "v2 is larger" << std::endl;  
}  
else if (v1 == v2) {  
    std::cout << "v1 equals v2" << std::endl;  
}  
else {  
    std::cout << "v1 is larger" << std::endl;  
}
```

Reading an unknown number of inputs

```
#include<iostream>  
int main {  
    int sum=0, value;  
    while (std::cin >> value)  
        sum += value;           //sum = sum + value  
    std::cout << "Sum is:" << sum << std::endl;  
    return 0;  
}
```

- Istream becomes invalid when hitting the end of a file
to simulate end of file: Ctrl-d on unix, Ctrl-z on windows
- endl: flushing the stream (alternative: std::cout.flush ())

Introduction to classes (1)

- We need a data structure to represent a transaction
 - in C we would use *structure*, in C++ *class*
- **class** is a data type
 - an instance of a class is object
 - `class CLASS_NAME;`
`CLASS_NAME a;`
 - a is an object of type CLASS_NAME
 - a is a variable of type CLASS_NAME (analog to `int x`)
 - a class supports (i) operations, and (ii) containment of data members

Introduction to classes (2)

- To use a class we need to know
 - the name of the class
 - where is the class defined
 - operations it supports
- **important**: a user of a class doesn't have to know how the class keeps the data or implements the operations it supports !
 - we only need to know the operations supported by the class
 - recall the cin/cout example

Sales_item class

- Stores: ISBN, number of copies sold, revenue, average sales price for the book
- What operations does it support ?
- The class definition resides in a header file that should be included by any program using the class

Operations on Sales_item

- Operations on objects of type Sales_item:
 - + operator: add two Sales_item objects
 - >> operator: read a Sales_item object
 - << operator: write a Sales_item object
 - = operator: assign one Sales_item object to another
 - same_isbn function: determine whether two Sales_item objects refer to the same book

Operations on Sales_item

```
#include <iostream>
#include "Sales_item.h"
int main () {
    Sales_item item;
    // read ISBN, number of copies sold, and sales price
    std::cin >> item;
    // write ISBN, number of copies sold, total revenue, average price
    std::cout << item << std::endl;
    return 0;
}
```

Input: x587 4 25

Output: x587 4 100 25

Adding Sales_item objects

```
#include <iostream>
#include "Sales_item.h"
int main () {
    Sales_item item1, item2;
    std::cin >> item1 >> item2; // read the two items
    std::cout << item1 + item2 << std::endl; // print the sum
    // equivalent to:
    // Sales_item s = item1+item2; std::cout << s << std::endl;
    return 0;
}
```

input:

ax1 3 20

ax1 2 5

output:

ax1 5 70 14

Member functions

- What happens if the input is two Sales_item objects that do not have the same ISBN ?

```
#include <iostream>
#include "Sales_item.h"
int main () {
    Sales_item item1, item2;
    std::cin >> item1 >> item2; // read the two items
    if (item1.same_isbn(item2)) {
        std::cout << item1 + item2 << std::endl;
        return 0; // success
    }
    else {
        //failure
        std::cerr << "Items must refer to the same ISBN" << std::endl;
        return -1;
    }
}
```

Member functions (methods)

- `item1.same_isbn(item2)`
 - `same_isbn` is a member function of the class `Sales_item`
 - `same_isbn` is called on the object `item1`, which is of type `Sales_item`, with the argument `item2`

```
#include <iostream> // header for standard library file
#include "Sales_item.h" //header for user defined file
int main () {
    Sales_item total, item;
    // is there any data to process
    if (std::cin >> total) {
        while (std::cin >> item) {
            if (total.same_isbn (item)) {
                total = total + item;
            }
            else { // no match, print and assign to total
                std::cout << total << std::endl;
                total = item;
            }
        } // end of loop
        std::cout << total << std::endl;
    } // end of check if there is data
    else { // no input at all
        std::cout << "Warning: No Data?" << std::endl;
        return -1;
    }
    return 0;
}
```


Key Concept: classes define behavior

- The author of `Sales_item` defined *all* the actions that can be performed on objects of type `Sales_item`
 - what happens when an object of type `Sales_item` is created ?
 - what happens when the operators (e.g., `+`, `>>`) are applied to an object of type `Sales_item`?
- In general, only operations defined by a class can be used on objects of the class type

Intermediate summary

- Goal of the course:
 - teach the principles of programming tools that enable to construct software
 - case study: C++
- What have we seen ?
 - basic program, variables, control structures
 - basic input/output
 - class
 - creating and using objects of a given class

Variables and Types

What is a variable ?

- A ***variable*** provides a named storage that the program can manipulate
- Each variable in C++ has a ***type***, *which determines*
 - the size and layout of the variable's storage (variable's memory)
 - the range of values that can be stored in this memory area
 - the set of operations that can be applied to the variable
- The name of a variable is its *identifier*
- For C++ programmers, "variables" and "objects" are synonyms

Primitive built-in types

Type	Meaning	Minimum Size
bool	boolean	NA
char	character	8 bits
wchar_t	wide character	16 bits
short	short integer	16 bits
int	integer	16 bits
long	long integer	32 bits
float	single-precision	6 sig. digits
double	double-precision	10 sig. digits
long double	extended precision	10 sig. digits

Literals

- `int x = 5; int x = 024; int x = 0x14;`
- `bool test = false; bool test = true;`
- By default, floating point literals are of type `double`
 - `float f = 3.14F; float f = 3.13f` single precision
 - `.001f = 1E-3F`
- `char c = 'a'; c = '2'; c = ','; c = ' '`
 - non-printable characters: `char c = '\n';`
 - `c = '\\'`
- `wchar_t wc = L'a'`

String literals

- Array of constant characters
 - "Hello World"
- 'A' takes one character, while "A" takes two (represented as an array with two cells)

Expressions

- C++ has two types of expressions
 - *lvalue*: an expression that can be on both sides of an assignment
 - e.g., variables
 - `x*y = 0`; //error: arithmetic expression is not an lvalue
 - `0 = 1`; //error: literal constant is not an lvalue
 - *rvalue*: an expression that can be only on the right-hand side of an expression
 - e.g., numerical literals (e.g., 5)
 - `x = x + 5`
 - common pitfall
 - ```
If (x = 5) {
 // some statements
}
```



# Defining and initializing objects

- `int ival(0);` // direct initialization
- `int ival = 0;` // copy initialization
- In C++ initialization is not an assignment !
  - initialization happens when a variable is created and gives that variable its initial value
  - assignment involves obliterating an object's current value and replacing that value with a new one
  - subtle differences between copy- and direct-initialization when initializing objects of a class type

# Initializing objects

- Using values of uninitialized variables causes run-time errors
- Built-in types: only one way to initialize
- Objects of class type: some initializations can be done only using direct-initialization
- Initialization in classes
  - *constructor* (a member function of the class)

```
#include<string>
/* ----- */
std::string titleA = "C++ primer";
std::string titleB("C++ primer");
std::string all_nines(3, '9'); // all_nines = "999"
```

# Initializing objects

What happens if we define an object without initialization ?

- ***default constructor***
- `std::string empty; //empty=""`;
- some class types don't have a default constructor
  - definitions for these classes must provide explicit initializer(s)

# Declarations and definitions

- **Definition** of a variable (e.g., `int x; int x = 5;`)
  - allocates storage and may also specify an initial value
  - there must be one and only one definition of a variable in a program
- **Declaration**
  - makes known the type and name of the variable to the program
  - a variable can be declared multiple times in a program
- A definition is also a declaration
- A declaration that is not a definition
  - `extern int i; //declaration`
  - `extern double pi = 3.14; //definition; can be done only outside a function`
  - `extern double pi = 3.14;`  
`double pi; // error: redefining pi`
- In C++ a variable must be defined exactly once and must be defined or declared before it is used
- A variable that is used in more than one file
  - one file contains the definition
  - other files may contain a declaration

# Scope of a name

- **Scope** is the context used to distinguish the names
- Most scopes in C++ are delimited by curly braces {}

```
#include <iostream>
int main () {
 int sum = 0;
 for (int val = 1; val <=10; ++val)
 sum += val;
 return 0;
}
```

main: global scope

sum: local scope

val: statement scope

# Scope of a name

```
#include<iostream>
#include<string>

std::string s1 = "hello"; // s1 has global scope
int main () {
 std::string s2 = "world";
 std::cout << s1 << " " << s2 << std::endl; // "prints hello world"
 int s1 = 42; // s1 is local and hides the global s1
 std::cout << s1 << " " << s2 << std::endl; // "prints 42 world"
 return 0;
}
```

- define a variable near the point at which it is first used to improve readability
- constraint: a variable must be defined in or before the outermost scope in which it will be used

# The *const* qualifier

- for (int index = 0; index != 512; ++index)
- int bufSize = 512;
  - for (int index = 0; index != bufSize; ++index)
- const int bufSize = 512;
  - bufSize = 0; // error: attempt to write to const object
- const int i; // error: i is uninitialized const

```
// file_1.cc
int counter;
// file_2.cc
extern int counter;
++counter;
```

```
// file_1.cc
extern const int counter = 400;
// file_2.cc
extern const int counter;
for (int index = 0; index < counter; ++index)
```

Const variables declared at global scope are local to the file in which they are defined (as opposed to nonconst variables that are extern by default)

# References

- A **reference** is an alternative name for an object (i.e., alias), and is mainly used as a formal parameter for functions
- A reference is a **compound type** (i.e., a type that is defined in terms of another type)

```
int iVal = 1024;
int &refVal = iVal; // refVal refers to iVal
int &refVal2; // error: a reference must be initialized
int &refVal3 = 10; // error: initializer must be an object

refVal += 2; // now iVal == 1026
int ii = refVal; // now ii == 1026
```

A reference must be initialized using an object of the same type, and it remains bound to that object as long as the references exists



# Const references

A ***const reference*** is a reference that may refer to a const object (const reference = reference to a const)

```
const int ival = 1024;
const int &refVal = ival;
int &ref2 = ival; // error: nonconst reference to a const object
refVal = 3; // error

const int &r = 42; // legal for const references only
```

nonconst reference may be attached only to an object of the same type

const reference may be bound to an object of a different but related type  
or to an rvalue

# Typedef and enum

- A ***typedef*** lets us define a synonym for a type
  - used for (i) hiding implementation, and (ii) emphasizing purpose and allowing a single type to be used for more than one purpose

```
typedef double wages;
typedef wages salary;

wages hourly, weekly;
```

- ***enum*** provides a method for grouping sets of integral constants
  - each enum defines a new type

```
enum forms {shape=1; sphere, cylinder, polygon}

forms formA = sphere; // OK
forms formB = 3; // error
forms formC = formA ; // OK
```

# Class types

- In C++ we define our own data types by defining a ***class***
  - the general notion of classes will be a main topic in this course
- A class defines
  - the data that an object of its type contains
  - the operations that can be executed by objects of its type
- We have already used the Sales\_item class

# Class design starts with its interface !

- ***Interface:*** the operations that the class will provide (**what** an object should do)
  - signatures of member functions
- ***Implementation:*** (**how** an object should do what it has to do)
  - code of member functions
  - data needed to represent an object
  - signatures and code of auxiliary functions

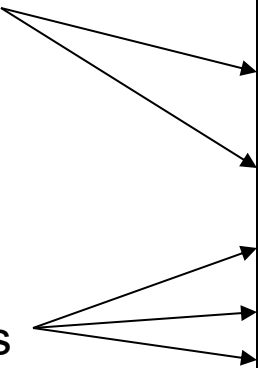
# Sales\_item: from *interface* to *implementation*

- *Interface*:
  - use the addition operator, +, to add 2 Sales\_item objects
  - use the input operator, >>, to read a Sales\_item object
  - use the output operator, <<, to write a Sales\_item object
  - call the same\_isbn function to determine if 2 Sales\_item objects refer to the same book
- Essential parts of the *implementation*:
  - keep track of how many copies of a book were sold
  - keep the revenue for a book
  - given the above we can calculate average price for a book

# Defining the Sales\_item class

Access  
labels

data  
members



```
class Sales_item {
 public:
 // operations on Sales_item objects will go here
 private:
 std::string isbn; //cannot initialize here ...
 unsigned units_sold;
 double revenue;
}; // don't forget this semicolon !
```

The *public* section of a class:

members that can be accessed by any part of the program;  
these members form the *interface* of the class

The *private* section of a class:

members that can be accessed only by code of the class; these  
members form the *implementation* of the class

# Modularity

- Till now all our code was in a single file
- ***separate compilation***
  - enables to compose a program from several files
- ***header files*** (fileName.h)
  - declarations
  - definitions of only
    - classes
    - const objects whose value is known at compile time
    - inline functions
- ***source files*** (fileName.cc)
  - source code for the methods of a class
  - source code for general functions (e.g., main)
- Sales\_item.h, Sales\_item.cc
- More details in the practical session

# Brief summary

- Types are fundamental to all programming in C++
- Each type defines the storage requirements and the operations that may be performed on all objects of that type
- Types can be
  - built-in vs. user-defined (i.e., classes)
  - non-const vs. const
  - basic vs. compound (e.g., references)
- The standard library (STL) uses the class facility to provide a set of higher-level *abstractions* such as IO streams and string
- C++ is a statically typed language: variables and functions must be declared before they are used
- A variable can be declared many times but defined only once
- Initialize variables when you define them



# Library types

# Standard and Abstract data types

- Standard built-in data types in C++ that are borrowed from C are all **low-level** types that represent low-level abstractions (numbers, characters, etc.)
- The standard library of C++ provides higher level ***abstract data types (ADTs)***
  - ADTs model complex concepts
  - We care about the operations supported by the ADTs (not their implementation)
- **string**
  - variable length strings
- **vector**
  - sequences of objects of a specific type

# Namespace using declaration

- recall the use of `std::cin`, `std::cout`

```
#include<string>
#include<iostream>

using std::cin;
using std::string;
using std::endl;
int main ()
{
 string s; //ok
 cin >> s; //ok
 cout << s; //error
 std::cout << s << endl; //ok
}
```

**Note: from now on the slides will not necessarily specify the `#include` and `using` directives, but those must be used, of course, when writing real programs !**

# Library string type

```
#include<string>
using std::string;
using std::cout;

string s1;
string s2 (s1);
string s3 ("value")
string s4 (n,'c');

cin >> s1;
cout << s1 << endl;
cin >> s1 >> s2;
cout << s1 << s2;

string word;
while (cin >> word) //reading an unknown number of strings
 cout << word << endl;

while (getline(cin, line)) // stores characters until \n
 cout << line << endl;
```

# Operations on strings

```
#include<string>
using std::string;

string s1 ("abcd");
bool b = s1.empty (); // b == false
string::size_type size = s1.size (); // size = 4; size_type is unsigned
string::size_type n = 1;
char c = s1[n]; // c == b; last character in s1 is s1[s1.size()-1]

string s2 = "1234";
string s3;
s3 = s1;
s3 = s1 + s2; // s3 == "abcd1234";
s3 = s1 + "," + s2; // s3 == "abcd,1234"
s3 = "1234" + "abcd"; // error, can't add string literals

bool b = (s1 == s2); // b == false
string s4 = "defg";
b = (s4 > s1); // b == true; "defg" > "abcd"

// <, <=, >=, !=, +=
```

# Characters of a string

```
string s = "abcd";
s[0] == 'a';
s[s.size()-1] == 'd';

for (string::size_type ix = 0; ix != s.size (); ++ix)
 s[ix] = '*';

// Note: the library is not required to check
// the value of an index
```

# Vector

- A ***vector*** is a collection of objects of a single type, each of which has an associated integer index
- A vector is a ***container***
- A vector is a ***class template***
  - templates enable us to write a single class or function definition that can be used on a variety of types
    - define a vector that holds strings, ints, or objects of our own class types

```
#include <vector>
using std::vector;
vector<int> ivec; // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales items
```

# Defining and initializing vectors

```
vector<T> v1; // vector that holds objects of type T; default
 // constructor: v1 is empty
vector<T> v2(v1); // v2 is a copy of v1
vector<T> v3(n,i); // v3 has n elements with value i
vector<T> v4(n); // v4 has n copies of value initialized object
 // for int, the value is 0; for objects: default constructor

vector<int> ivec1;
vector<int> ivec2(ivec1); //OK
vector<string> svec(ivec1);; // error: svec holds strings, not ints
vector<int> ivec4 (10,-1); // 10 elements, each initialized to -1
vector<string> svec (10,"hi"); // 10 strings, each initialized to "hi"
```

vectors grow dynamically; good practice: define an empty vector and add elements to it



# Operations on vectors

```
v.empty (); // returns true if v is empty
v.size (); // returns number of elements in v
v.push_back (t); //adds element with value t to the end of v
v[n]; //returns element at position n in v;
v1 = v2; //replaces elements in v1 by a copy of elements in v2
```

---

```
vector::size_type //error vector<int>::size_type // OK
```

---

```
string word;
vector<string> text;
while (cin >> word)
 text.push_back(word);
```

---

```
vector<int> ivec(10,2);
for (vector<int>::size_type ix = 0; ix != ivec.size (); ++ix) //efficient
 ivec[ix] = 0; //subscripting returns an lvalue
```

---

```
vector<int> ivec; ///empty vector
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
 ivec[ix] = ix; //disaster !! An element must exist in order to
 // subscript it; elements are not added when we subscript
```

# Introduction to iterators

- An ***iterator*** is a type that lets us examine the elements in a container and navigate from one element to another
- The library defines an iterator type for each of the standard containers, including vector
- Iterators are more general than subscripts; modern C++ programs tend to use iterators rather than subscripts to access container elements, even on types such as vectors

```
vector<int>::iterator iter;
```

# Working with iterators

```
vector<int> ivec (10,5);
for (vector<int>::size_type ix = 0; ix != ivec.size (); ++ix)
 ivec[ix] = 0;
```

```
//equivalent loop using iterators
for (vector<int>::iterator iter = ivec.begin (); iter != ivec.end (); ++iter)
 *iter = 0; //dereference operator, value returned is an lvalue
```

# const\_iterator

dereferencing a plain iterator we get a nonconst reference

dereferencing a const\_iterator the value returned is a reference to a const object

```
vector<string> text;
for (vector<string>::const_iterator iter = text.begin (); iter != text.end (); ++iter)
{
 cout << * iter << endl;
 *iter = " "; //error: *iter is const
}
```

when using the const\_iterator type, we get an iterator whose own value can be changed but that cannot be used to change the underlying element value

**do not confuse the const\_iterator with an iterator that is const !**

```
const vector<string>::iterator cit = text.begin ();

*cit = "hello"; //ok

++cit; // error: cannot change the value of cit
```

# More on const

```
const vector<int> nines(10,9);
//error: cit2 could change the element it refers to and nines is const
const vector<int>::iterator cit2 = nines.begin ();

//OK: it can't change an element value, so it can be used with a const vector<int>
vector<int>::const_iterator it = nines.begin ();

*it = 10; // error: *it is const
++it; // ok: it is not const so we can change its value
```

**Remember**

|                                          |                                                 |
|------------------------------------------|-------------------------------------------------|
| <b>vector&lt;int&gt;::const_iterator</b> | <b>//an iterator that cannot write elements</b> |
| <b>const vector&lt;int&gt;::iterator</b> | <b>//an iterator whose value cannot change</b>  |

# Iterator arithmetic

```
iter + n // size_type
iter - n // difference_type

iter1 - iter2 //difference_type
//both iter1 and iter2 must refer to elements
// in the same vector or the element one past the
// end of the vector
vector<int>::iterator mid = ivec.begin () + ivec.size() / 2;
```

important: any operation that changes the size of a vector makes existing iterators invalid (e.g., push\_back)

# Arrays and Pointers

# Arrays

- Array is a container of objects of a single type
  - like vector
  - each element is accessed by its position in the array
- Drawbacks with respect to vector
  - fixed size
  - offers no help to the programmer in keeping track of how big a given array is
  - programs that rely on built-in arrays rather than using the standard vector are more error-prone and harder to debug
- Why should we study arrays ?
  - speed
  - internal implementation of classes
  - lots of existing code uses arrays



# Arrays (contd.)

- An array is a compound type
- An array consists of a type specifier and a dimension
  - type specifier can denote a built-in data type or a class type
  - dimension must be a constant expression
    - integral literal constants, enumerators, const objects of integral type that are initialized from constant expressions

# Defining and initializing arrays

```
const size_t buf_size = 512, max_files = 20;
size_t staff_size = 27;
const size_t sz = get_size (); //const value not known until run time
char input_buffer [buf_size]; // OK, const variable
string fileTable[max_files+1]; // OK, constant expression
double salaries [staff_size]; // error: nonconst variable
int test_scores[get_size()]; // error: nonconst expression
int vals[sz]; // error: size not known until run time
```

---

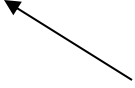
```
const size_t array_size = 3;
int ia[array_size] = {0, 1, 2};
int ia[] = {0,1,2}; //an array of dimension 3
```

# Operations on arrays

- Unlike vector, it's not possible to initialize an array as a copy of another array, nor is it legal to assign one array to another
- By far, the most common causes of security problems are so-called "buffer overflow" bugs;

```
#include<cstddef>
const size_t array_size=10; //use size_t
int ia[array_size], ia2[array_size];
for (size_t ix = 0; ix != array_size; ++ix) {
 ia[ix] = ix;
 ia2[ix] = ia[ix];
}
```

# Pointers

- A pointer is a compound type; it points to an object of some other type
- One role of pointers is being iterators for arrays
- A pointer holds the address of another object
  - `string s("hello world");`
  - `string *sp= &s;    //sp holds the address of s`
    -  address-of operator (can be applied to an lvalue)

# Pointers (contd.)

```
vector<int> *pvec; // pvec can point to vector<int>
int *ip1, *ip2; // ip1 and ip2 can point to an int
string *pstring; // pstring can point to a string
double *dp; // dp can point to a double

//avoid using string* pstring; read right to left
string* ps1, ps2; // ps1 is a pointer to string, ps2 is a string
```

A valid pointer can have one of three states: 1. holds an address 2. points one past the end of an object 3. has zero value (points to no object)

```
int ival = 1024;
int *pi = 0; // pi initialized to address no object
int *pi2 = &ival; // pi2 initialized to address of ival
int *pi3; // OK but dangerous, pi3 is uninitialized
pi = pi2; // pi and pi2 address the same object, i.e., ival
pi2 = 0; // pi2 now addresses no object
```

# Initialization of pointers

- Do not use uninitialized pointers
- If possible, don't define a pointer until the object to which it should point has been defined
  - if you must define a pointer separately from pointing it at an object, initialize it to point to NULL(=0) (so it can be tested)
- Values that pointers can take
  - constant expression with value 0
  - address of an object of an appropriate type
  - the address one past the end of another object
  - another valid pointer of the same type

```
int ival;
int zero = 0;
```

```
int *pi = ival; //error: pi initialized from int value of ival
pi = zero; //error: pi assigned int value of zero
pi = 0; // ok: directly initialize to literal constant 0
int *pi = NULL; //ok: equivalent to int *pi = 0; NULL is defined in cstdlib
```

---

```
double dval;
double *pd = &dval; //OK: initializer is address of a double
double *pd2 = pd; //OK: initializer is a pointer to a double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: attempt to assign address of a double to int
```

---

```
double obj = 3.14;
void *pv = &obj; //OK: void* can hold the address value of any data pointer type
pv = pd; // pd can be a pointer to any type; we can only pass, return and
 // assign a void*
```

# Operations on pointers

- dereference yields an lvalue

```
string s("hello world");
string *sp = &s;
cout << *sp; // prints "hello world"
*sp = "goodbye";
cout << s; // prints "goodbye"

string s2 = "some value"
sp = &s2; // sp now points to s2
```

- comparing pointers and references
  - a reference always refers to an object (recall that we must initialize a reference)
  - assigning to a reference changes the object to which the reference is bound, it doesn't rebind the reference to another object



# Using pointers to access array elements

```
int ia [] = {0,2,4,6,8}; //when we use ia, it's automatically converted into
 // a pointer to the first element in the array
```

```
int *ip = ia; //ip points to ia[0]
```

```
ip = &ia[4]; // ip points to last element in ia
```

```
ip = ia; // ok: ip points to ia[0]
```

```
int *ip2 = ip +4; // ok:ip2 points to ia[4], the last element in ia
 // adding an integral value to a pointer results in a pointer
```

```
int *ip3 = ia + 10; // error: ia has only 4 elements
```

```
int last = *(ia+4); // ok: initializes last to 8, the value of ia[4]
```

```
// equivalent to int last = ia[4];
```

```
last = *ia + 4; // ok: last = 4, equivalent to ia[0] + 4
```

---

```
int *p = &ia[2]; //ok: p points to the element indexed by 2
```

```
int j = p[1]; //ok: p[1] equivalent to *(p+1) , p[1] is the same element as ia[3]
```

```
int k = p[-2]; // ok: p[-2] is the same element as ia[0]
```

# pointers are iterators for arrays

```
const size_t arr_sz = 5;
int int_arr[arr_sz] = {0, 1, 2, 3, 4};

for (int *pbegin = int_arr, *pend = int_arr + arr_sz; pbegin != pend; ++pbegin)
 cout << *pbegin << ' '; //print the current element
```

# Pointers to const objects

```
const double *cptr; // cptr may point to a double that is const
 // const qualifies the type of the object to which
 // cptr points, not cptr itself
```

```
*cptr = 42; // error: *cptr might be const
```

```
const double pi = 3.14;
double *ptr = π // error: ptr is a plain pointer
const double *cptr = π // ok: cptr is a pointer to const
```

```
const int universe = 42;
const void *cpv = &universe; //ok: cpv is const
void *pv = &universe; //error: universe is const
```

```
// a pointer to a const object can be assigned the address of a nonconst
// object
```

```
double dval = 3.14; // dval is a double; its value can be changed
cptr = &dval; // ok: but can't change dval through cptr
```

# Const pointers

- these are pointers whose own value we may not change
- we must initialize a const pointer when we create it

```
int errNumb = 0;
int *const curErr = &errNumb; //curErr is a constant pointer

curErr = curErr; // error: curErr is const
```

const pointer to a const object

```
const double pi = 3.15159;
const double *const pi_ptr = π
```

# Dynamically allocating arrays

The main reason to use dynamically allocated arrays is that the size of an array might not be known at compile time

```
int x = new int; // creating a variable of type int
int *pia = new int [10]; //pia points to the first element of the array
 // array is uninitialized
int *pia = new int [10] (); //array is initialized with zeros

const int *pci_bad = new const int [100]; //error: uninitialized const array
const int *pci_ok = new const int [100] (); //ok:value-initialized array
const string *pcs = new const string[100]; //ok, because string has default
 // constructor

delete x;
delete [] pia; // The [] is very important ! avoid memory leaks
```

# Functions

# Defining and calling a function

```
int max (int v1, int v2)
```

```
{
```

```
 if (v1 >= v2) {
```

```
 return v1;
```

```
 }
```

```
 else {
```

```
 return v2;
```

```
 }
```

```
}
```

```
#include <iostream>
```

```
int main () {
```

```
 int i,j
```

```
 cin >> i >> j;
```

```
 cout << "The larger number is" << max(i,j) << endl;
```

```
 int z = max (i,j);
```

```
}
```

parameters

arguments

# Defining and calling a function (contd.)

- Names defined inside a function body are accessible only within the function itself
  - such variables are referred to as **local variables**
  - in the previous example, i and j were local to the function main
- Parameters of the function are initialized by the arguments passed to the function when the function is called
  - an argument is an expression
  - we must pass exactly the same number of arguments as the function has parameters
  - the type of each argument must have the same type or a type that can be implicitly converted to the parameter type
- The return type of a function can be a built-in type (e.g., int, double) , class type, or a compound type such as int& or string\*
  - the return type can be void
  - the return type cannot be a built in array
  - functions must specify a return type

```
Date &calendar (const char*)
void process ();
bool is_present ();
```



# Parameters and arguments

- void process () is equivalent to  
void process (void)
- No two parameters can have the same name
  - a variable local to a function may not use the same name as that of one of the parameters
- Names are optional, but in a function definition, normally all parameters are named
- C++ is a ***statically typed language***: arguments of every call are checked during compilation

```
int max (int v1, int v2) {
 //something
}
max ("hello",1); //error: wrong argument types
max (1); //error: too few arguments
max (1,2,3); //error: too many arguments
max (3.14, 6.3) //ok : type conversion, but equivalent to max(3,6);
 // probably a compiler warning
```

# Argument passing

- ***Non reference parameters*** are initialized by copying the corresponding argument
  - the function has no access to the actual arguments of the call, i.e., changes made to the parameter are made to the local copy, and once the function terminates, the local values are gone
- ***Pointer parameters***
  - The argument pointer is copied

```
void reset (int *ip)
{
 *ip = 0; //changes the value of the object to which ip points
 ip = 0; // changes only the local value of ip; the argument is unchanged
}
```

```
int i = 42;
int *p = &i;
cout << "i: " << *p << '\n'; // prints i:42
reset (p); // changes *p but not p
cout << "i: " << *p << endl; //ok: prints i: 0
```

```
void use_ptr (const int *p)
{
 //use_ptr may read but not write to *p
 *p = 0; //error
}
```

```
// We can call use_ptr on either an int* or a const int*; we can pass only
// on an int* to reset (passing const int* won't compile)
```

# Reference parameters

- Limitations of copying arguments
  - when we want the function to change the value of an argument
  - when we want to pass a large object as an argument
    - time and space for copying are too expensive
  - when there is no way to copy the object

copying the arguments here doesn't work ! Use reference parameters

```
void swap (int v1 ,int v2)
{ int tmp = v2;
 v2 = v1;
 v1 = tmp;
}
```

```
void swap (int &v1 ,int &v2)
{ int tmp = v2;
 v2 = v1;
 v1 = tmp;
}
```

# Reference parameters (contd.)

- Another use of reference parameters is to return an additional result to the calling function (recall that functions can return a single value)

```
int max (int v1, int v2, bool &isEq) {
 if (v1 != v2) {
 isEq = false;
 if (v1 > v2) return v1;
 else return v2;
 }
 else {
 isEq = true;
 return v1;
 }
}
```

# Const reference parameters

```
bool isShorter (const string &s1, const string &s2)
{
 return s1.size () < s2.size ();
}
```

1. Because the parameters are references the arguments are not copied
2. Because the parameters are const references, isShorter may not use the references to change the arguments

**Important: reference parameters that are not changed should be references to const. Plain, nonconst reference parameters are less flexible. Such parameters may not be initialized by const objects, or by arguments that are literals or expressions that yield rvalues**

```

int incr (int &val)
{
 return ++val;
}
int main()
{
 short v1 = 0;
 const int v2 = 42;
 int v3 = incr(v1); //error: v1 is not an int
 int v3 = incr (v2); //error: v2 is const
 int v3 = incr (0); //error: literals are not lvalues
 int v3 = 5;
 int v4 = incr (v3); // OK: v3 is a nonconst object of type int
}

```

---

```

string::size_type find_char (string &s, char c)

```

```

//should have declared const string&

```

```

{
 string::size_type i = 0;
 while (i != s.size () && s[i] != c)
 ++i;
 return i;
}

```

```

if (find_char("Hello World",'o')) //fails at compile time

```

# Passing a reference to a pointer

```
void ptrswap (int *&v1, int * &v2)
{
 // v1 is a reference to a pointer to an object of type int
 int *tmp = v2;
 v2 = v1;
 v1 = tmp;
}
```

Avoid passing vectors by copying; either pass references or iterators

```
void print (vector<int>::const_iterator beg, vector<int>::const_iterator end)
{
 while (beg != end)
 {
 cout << *beg;
 beg++;
 if (beg != end) cout << " "; // no space after last element
 }
 cout << endl;
}
```



# The return statement

```
void do_swap (int &v1, int &v2)
{
 //swap values of v1 and v2
 return; // no need to use that
}
```

If a function is defined to return some type then the compiler checks that the return type is correct

The compiler cannot guarantee that there is indeed a return (especially after loops that have a return inside them); in such cases the run time behavior is undefined

The main function is allowed to terminate without a return: compiler inserts "return 0"

# Returning a reference

```
const string &shorterString (const string &s1, const string &s2)
{
 if (s1.size () < s2.size ())
 return s1;
 else
 return s2;
}
```

**Never return a reference to a local object  
(and never return a pointer to a local object)**

```
const string &manip (const string& s)
{
 string ret = s;
 return ret; //wrong: returning reference to a local object
 //compiler prints a warning
}
```

# Function declarations

- A function must be declared before it is used
- We can declare a function separately from its definition
- Function declaration consists of a return type, the function name, and parameter list
  - The above three are called the ***function prototype***
    - ***function prototypes*** provide the interface between the programmer who defines the function and the programmers who use it
  - parameter list contains the types of the parameters but needn't name them
- Function declarations go in header files
  - the source file that ***defines*** the function should include the header that ***declares*** the function

```
void print (int *array, int size);
```

# Default arguments

```
string screenInit (string::size_type height = 24, string::size_type width = 80,
char background = ' ');
```

**//correct invocation of this function**

```
string screen;
```

```
screen = screenInit ();
```

```
screen = screenInit (66);
```

```
screen = screenInit (66,256);
```

```
screen = screenInit (66,256,'3');
```

**//we can omit only trailing arguments**

```
screen = screenInit (, , '?'); //error, can omit only trailing arguments
```

```
screen = screenInit ('?'); //calls screenInit ('?',80,' ');
```

**//because char is an integral type it's legal to pass a  
// char to an int parameter and vice versa**

1. Order the parameters so that those least likely to use a default value appear first and those most likely to use a default value appear last
2. Default arguments ordinarily should be specified with the declaration for the function and placed in an appropriate header file. (We can specify default arguments in either the function definition or declaration. However, only once in a file.)

# Local objects

- Names have scope (where the name is known), objects have ***lifetimes*** (time during the program's execution that the corresponding object exists)
- ***Automatic Objects***: parameters, local variables
  - are objects that exist only while a function is executed
  - are created and destroyed on each call to a function
    - an automatic object will be destroyed at the end of the block in which it is defined

# Inline functions

- Calling a function can be a slow process
  - registers are saved (on call) and restored (on return)
  - arguments are saved before the call and restored after the return
  - program branches to a new location
- ***Inline functions*** avoid call overhead
  - an inline function is (usually) expanded "in line" at each point in the program in which it is invoked
- inline mechanism is meant to optimize small, straight-line functions that are called frequently
- inline specification is only a request to the compiler
- inlines should be defined in header files

```
inline const string &shorterString (const string &s1, const string &s2)
{
 return s1.size () < s2. size () ? s1 : s2;
}
cout << shorterString (s1, s2) << endl; // will be expanded to
// cout << (s1.size () < s2.size () ? s1 : s2) << endl;
```

# Class member functions

- Function *prototype* (= return type, function name, (possibly empty) comma-separated list of parameters) must be defined within the class body
  - function body may be defined inside the class or outside the class

```
class Sales_item {
public:
 // operations on Sales_item object
 double avg_price () const; //function declaration
 bool same_isbn (const Sales_item &rhs) const
 { return isbn == rhs.isbn; } //implicitly treated as inline
 //private members as before
private:
 std::string isbn;
 unsigned units_sold;
 double revenue;
};
```

# Member functions

- A member function may access the ***private*** members of its class
- Member functions have an extra, implicit parameter that binds the function to the object on which the function is called
  - the implicit parameter is called ***this***
    - when a member function is called, the *this* parameter is initialized with the address of the object on which the function is invoked
    - `total.same_isbn (item)` is "translated" by the compiler to `Sales_item::same_isbn (&total,item);`
- Inside a member function we needn't explicitly use the *this* pointer to access the members of the object on which the function was called

```
bool same_isbn (const Sales_item &rhs) const
{
 return isbn == rhs.isbn;
 //equivalent to return this->isbn == rhs.isbn;
}
```



# Const member functions

- The *const* (as in `avg_price () const`) modifies the type of the implicit *this* parameter
- when we call `total.same_isbn(item)` the implicit *this* parameter will be a `const Sales_item*` that points to `total`

A const member function cannot change the object on whose behalf the function is called

Important: a const object or a pointer or reference to a const object may be used to call only const member functions. It's an error to try to call a nonconst member function on a const object or through a pointer or reference to a const object

# Defining member functions outside the class

File: Sales\_item.cc

```
#include "Sales_item.h"
double Sales_item::avg_price () const
{
 if (units_sold)
 return revenue/units_sold;
 else
 return 0;
}
```



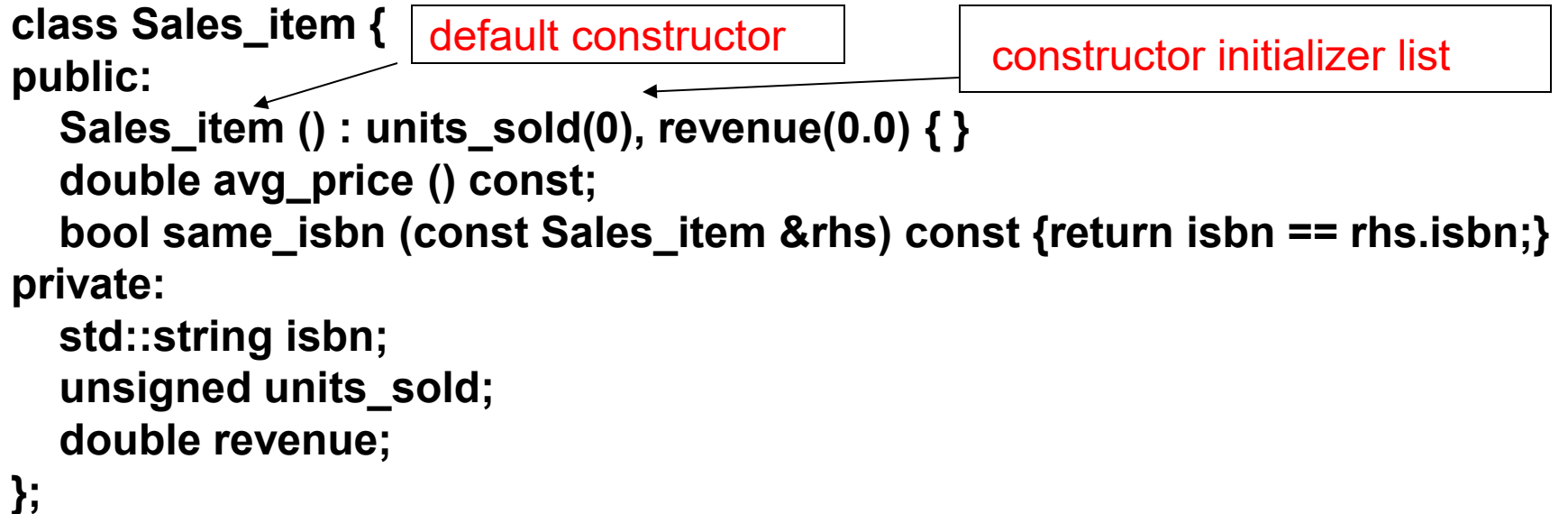
scope operator

# Constructor

- A special member function that has the same name as its class
- Doesn't have a return type
- Takes a parameter list and has a function body
- A class can have multiple constructors
  - each must differ from the others in the number or types of its parameters
- Constructors should ensure that every data member is initialized
- ***Default constructor***: takes no arguments

```
vector<int> vi; //default constructor: empty vector
string s; //default constructor: empty string
Sales_item item; //default constructor: ??
```

```
class Sales_item {
public:
 Sales_item () : units_sold(0), revenue(0.0) { }
 double avg_price () const;
 bool same_isbn (const Sales_item &rhs) const {return isbn == rhs.isbn;}
private:
 std::string isbn;
 unsigned units_sold;
 double revenue;
};
```



## The initializer list

specifies initial values for one or more data members of the class

follows the constructor parameter list

isbn will be initialized using the string default constructor

# Synthesized default constructor

- The compiler created default constructor is known as the ***synthesized default constructor***
  - it initializes each member using the same rules as are applied for variable initializations
    - members that are of class type are initialized using the default constructor of the member's class
    - built-in types: if the object is defined at global scope (outside of any function) they will be initialized to 0 (otherwise (local scope) they will be uninitialized)
- Classes with members of built-in or compound type should usually define their own default constructors to initialize those members

# The big picture of classes

file: Sales\_item.h

```
#ifndef SALES_ITEM_H
#define SALES_ITEM_H
#include<string>

class Sales_item {
public:
 Sales_item ();
 double avg_price () const;
 bool same_isbn (const Sales_item &rhs) const
 {return isbn == rhs.isbn;}
private:
 std::string isbn;
 unsigned units_sold;
 double revenue;
};

#endif
```

```

#include "Sales_item.h"
Sales_item::Sales_item () : units_sold(0), revenue(0.0) {
}
double Sales_item::avg_price () const
{
 if (units_sold)
 return revenue/units_sold;
 else
 return 0;
}

```

file: Sales\_item.cc

file: main.cc

```

#include "Sales_item.h"
#include <iostream>
int main () {
 Sales_item s1;
 const Sales_item s2 (); const Sales_item s2;
 Sales_item &s3 = s1;
 Sales_item *ps = new Sales_item ();
 // do stuff
 std::cout << s1.avg_price () << " " << s3.avg_price () << " "
 << ps -> avg_price () << " " << (*ps).avg_price ();
}

```

# Overloaded functions

- Two functions that appear in the same scope are ***overloaded*** if they have the same name but have different parameter lists
- 1+3, 1.0+3.0 (the '+' operator is overloaded)
- The main function may not be overloaded

```
Record lookup (const Account&); //find by account
Record lookup (const Phone&); //find by phone
Record lookup (const Name&); //find by name
```

```
r1 = lookup(acct); //call version that takes an account
r2 = lookup(phone); //call version that takes a phone
```



# Distinguish overloading from redeclaring a function

- If the return type and parameter list of two functions declarations match exactly, then the second declaration is treated as a redeclaration of the first
- If the parameter lists of two functions match exactly but the return type differs, then the second declaration is an error
  - functions cannot be overloaded based only on differences in the return type

# Examples

**Record lookup (const Account&);**  
**bool lookup (const Account&);**      **//error: only return type is different**

**Record lookup (const Account &acct);**  
**Record lookup (const Account&);**      **//parameter names are ignored, redeclaration**

**Record lookup (const Phone&, const Name&);**  
**//default argument doesn't change the number of parameters**  
**Record lookup (const Phone&, const Name& = "");**

**//const is irrelevant for non-reference parameters**  
**Record lookup(Phone);**  
**Record lookup (const Phone);**      **//redeclaration**

# Overloading and scope

```
/* For illustration purposes only !
 * bad style to define a local variable in a function
 * with the same name as a global name the function wants to use
 */

string init ();
void fcn ()
{
 int init = 0; //init is local and hides global init
 string s = init (); //error: global init is hidden
}
```

If we declare a function locally, that function hides rather than overloads the same function declared in an outer scope. This means that declarations for every version of an overloaded function must appear in the same scope.

**Important:** it's a bad idea to declare a function locally.  
Function declarations should go in header files.

**//Example of a very bad programming practice**

**void print (const string&);**

**void print (double);                   //overloads the print function**

**void print (int);                    //overloads the print function**

**void fooBar (int ival)**

```
{
 void print (int); //new scope: hides previous instances of print

 print ("Value: "); //error: print (const string&) is hidden
 print (ival); //ok: print (int) is visible;
 print (3.14); // ok: calls print (int); print (double) is hidden
}
```

**//The right way to do that**

**void print (const string&);**

**void print (double);                //overloads the print function**

**void print (int);                    //another overloaded instance**

**void fooBar (int ival)**

```
{
 print ("Value: "); //ok: calls print(const string&)
 print (ival); //ok: print (int)
 print (3.14); // ok: calls print (double)
}
```

# Classes

# Recap

- A class defines a new type and a new scope
- We've seen the classes: vector, string, Sales\_item
- Each class defines zero or more members
  - members can be either data, functions, or type definitions
  - all members must be declared inside the class
- Access modifiers
  - members defined in the **public** section are accessible to all code that uses the type
  - members defined in the **private** section are accessible to other class members
- Constructor
  - a special member function that has the same name as the class
  - initializer list: Sales\_item(): units\_sold(0), revenue(0.0) { }
- Member functions (**methods**)
  - must be declared
  - optionally may be defined inside the class (then they are inline)
  - implicit parameter *this*
  - double avg\_price () const; //const must appear in both the declaration and definition

# The big picture of classes

file: Sales\_item.h

```
#ifndef SALES_ITEM_H
#define SALES_ITEM_H
#include<string>

class Sales_item {
public:
 Sales_item ();
 double avg_price () const;
 bool same_isbn (const Sales_item &rhs) const
 {return isbn == rhs.isbn;}
private:
 std::string isbn;
 unsigned units_sold;
 double revenue;
};

#endif
```

```

#include "Sales_item.h"
Sales_item::Sales_item () : units_sold(0), revenue(0.0) {
}
double Sales_item::avg_price () const
{
 if (units_sold)
 return revenue/units_sold;
 else
 return 0;
}

```

file: Sales\_item.cc

file: main.cc

```

#include "Sales_item.h"
#include <iostream>
int main () {
 Sales_item s1;
 const Sales_item s2;
 Sales_item &s3 = s1;
 Sales_item *ps = new Sales_item ();
 // do stuff
 std::cout << s1.avg_price () << " " << s3.avg_price () << " "
 << ps -> avg_price () << " " << (*ps).avg_price ();
}

```



# Data abstraction and encapsulation

- The fundamental ideas behind classes are **data abstraction** and **encapsulation**
- **Data abstraction** is a programming and a design technique that relies on the separation of **interface** and **implementation**. The class designer must worry about **how** her class is implemented, while programmers using this class can think *abstractly* about **what** the type does rather than **how** the type works
- **Encapsulation** describes the technique of combining lower-level elements to form a new, higher-level entity.
  - Encapsulated elements **hide** the details of their implementation
  - A function is an example of encapsulation: the actions performed by the function are encapsulated in the larger entity that is the function itself
    - We may have no access to the statements of the function
  - A class is an encapsulated entity
    - It represents an aggregation of several members, and most well-designed class types hide the members that implement the type
- **vector**: data abstraction and encapsulation
  - To use it we think about its interface (data abstraction)
  - We have no access to the details of how the type is represented nor to any of its implementation (encapsulation)
- **array**: neither abstract nor encapsulated

# Access labels enforce abstraction and encapsulation

- Members defined after a **public** label are accessible to all parts of the program
  - the data abstraction view of a type is defined by its public members
- Members defined after a **private** label are not accessible to code that uses the class
  - the private sections encapsulate (e.g., hide) the implementation from code that uses the type

# Benefits of data abstraction and encapsulation

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object
- The class implementation might evolve over time in response to changing requirements or bug reports without requiring change in user-level code
- Technical (important) point:  
Changing a class definition in a header file effectively changes the text of every source file that includes that header's code. Thus, re-compilation of these source files is needed.

# Some more details on classes

```
class Screen {
public:
 // interface member functions
 typedef std::string::size_type index;
private:
 std::string contents;
 std::string size_type cursor;
 std::string::size_type height, width;
};

// Screen::index; nobody has to know/care that we are using a string
// to represent the content of the screen
```

```
int main () {
 Screen::index x;
}
```

# Overloading member functions

```
class Screen {
public:
 // interface member functions
 typedef std::string::size_type index;
 //return a character at the cursor
 char get () const {return contents[cursor];}
 //return a character at a given position
 char get (index ht, index wd) const;
private:
 std::string contents;
 std::string size_type cursor;
 std::string::size_type height, width;
};

Screen myScreen;
char c = myscreen.get (); //calls Screen::get ()
ch = myScreen.get (0,0); // calls Screen::get (index, index)
```

# Inline member functions

Screen.h

```
class Screen {
public:
 // interface member functions
 typedef std::string::size_type index;
 // implicitly inline when defined inside the class declaration
 char get () const {return contents[cursor];}
 // explicitly inline;
 inline char get (index r, index c) const {
 index row = r * width;
 return contents [row+c];
 }
private:
 std::string contents;
 std::string size_type cursor;
 std::string::size_type height, width;
};
```

```
class LinkScreen { // here the class is declared but not defined yet
 Screen window;
 LinkScreen *next;
 LinkScreen *prev;
};
```

# Constructors

- The job of a constructor is to ensure that the data members of each object start out with sensible values
- \_INITIALIZER list
- Constructor cannot be declared as a const function
- Constructors may be overloaded

Sales\_item.h

```
class Sales_item {
public:
 Sales_item(const std::string&);
 Sales_item(int);
 Sales_item();
};
```

main.cc

```
Sales_item empty;
Sales_item primer_3("0-201-B24d");
Sales_item primer_4(33444);
```



# Initializer list

2 steps: 1. initialization 2. general computation

**//prefer this way !!!**

```
Sales_item::Sales_item(const string &book):isbn(book), units_sold (0), revenue(0.0)
{
}
```

**//possible, but less preferred**

```
Sales_item::Sales_item(const string &book)
{
 isbn = book;
 units_sold = 0;
 revenue = 0.0;
}
```

# More on initialization

- Members that are const or reference types **must** be initialized in the constructor initializer list

```
class ConstRef {
public:
 ConstRef (int ii);
private:
 int i;
 const int ci;
 int &ri;
};
//no explicit constructor initializer: error ri is uninitialized
ConstRef::ConstRef (int ii)
{
 i = ii; // ok
 ci = ii; //error: cannot assign to a const
 ri = i; // assigns to ri which was not bound to an object
}
//ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii):i(ii),ci(ii),ri(ii) { }
```

# More on initialization (2)

- It is a good idea to write constructor initializers in the same order as the members are declared. Moreover, when possible, avoid using members to initialize other members
- Initializers may be any expression
  - e.g., `Sales_item(const std::string &book, int cnt, double price):isbn(book), units_sold(cnt), revenue(cnt*price)`

# Default constructor

- The default constructor is used whenever we define an object but do not supply an initializer
- If the class defines even one constructor then the compiler will not generate the default constructor
  - **Remember**: the compiler generates a default constructor automatically only if a class defines **no** constructors
- The ***synthesized default constructor*** initializes members using the same rules as those that apply for how variables are initialized
- **Remember**: if a class contains data members of built-in or compound type, then the class should not rely on the synthesized default constructor
- In practice, it is almost always right to provide a default constructor if other constructors are being defined

**Do not use this:** `Sales_item myObj ();` // this is a function declaration !!!

**Use this:** `Sales_item myObj;`

**Or use this:** `Sales_item myobj = Sales_item ();`

**Or use this:** `Sales_item *sPTR = new Sales_item ();`

# Type conversion

```
class Sales_item {
public:
 Sales_item (const std::string &book); //defines an implicit conversion
};
```

```
string null_book = "9-999-999-999";
//ok: builds a Sales_item with 0 units_sold and revenue and an isbn equal to
// null_book
Sales_item item;
item.same_isbn(null_book);
```

## suppressing implicit conversions:

```
class Sales_item {
public:
 //explicit is used only in the declaration of the constructor
 explicit Sales_item (const std::string &book); //defines an implicit conversion
};
```

```
item.same_isbn(null_book); //error: string constructor is explicit
item.same_isbn(Sales_item(null_book)); //ok: explicit constructor
```

# Friends

- Sometimes it's convenient to let specific nonmember functions access the private members of a class while still preventing general access
  - overloaded operators, such as the input or output operators, often need access to the private data members of the class

```
class Screen {
 friend class Window_mgr; //prefer the following option:
 // friend Window_mgr& Window_Mgr::relocate(Window_Mgr::index,
 Window_Mgr::index, Screen&);
 private:
 std::string::size_type height, width;

};
Window_Mgr& Window_Mgr::relocate(Screen::index r, Screen::index c,
 Screen &s)
{
 s.height += r;
 s.width += c;
 return *this;
}
```

# Static class members

- Sometimes, all the objects of a particular class type need access to a global object
  - e.g., keep count of how many objects of a particular class have been created at any point of the program
  - making the object global violates encapsulation ! (general user code can modify the value)
  - solution: ***class static member***
- Static data members and functions exist independently of any objects of the class
  - remember: nonstatic data members and functions exist in each object of the class type
  - each static data member and a function is associated with the class and not with the objects of the class
  - static member functions has no *this* parameter
    - they may directly access the static members of the class but may not directly use the nonstatic members
    - static member functions may not be declared as const

# Advantages of using class static members

- The name of a static member is in the scope of the class, thereby avoiding name collisions with members of other classes or global objects
- Encapsulation can be enforced. A static member can be a private member, a global object cannot
- It is easy to see by reading the program that a static member is associated with a particular class. This visibility clarifies the programmer's intentions



Account.h

```
class Account {
public:
 // interface functions here
 void applyint ()
 { amount += amount * interestRate;}
 static double rate () {return interestRate;}
 static void rate(double); // sets a new rate
private:
 std::string owner;
 double amount;
 static double interestRate;
 static double initRate ();
};
```

Account.cc

```
#include "Account.cc"
double Account::interestRate =
 initRate ();
void Account::rate(double newRate)
{
 interestRate = newRate;
}
```

```
Account ac1;
Account *ac2 = &ac1;
//equivalent ways to call the static member rate function
double rate;
rate = ac1.rate (); //through an Account object or reference
rate = ac2->rate (); //through a pointer to an Account object
rate = Account::rate (); //directly from the class using the scope operator
```

# Static data members

- Static data members must be defined exactly once outside the class body
  - static data members are not initialized through the class constructor(s) and instead should be initialized when they are defined
- **Remember:** the best way to ensure that the object is defined exactly once is to put the definition of *static* data members in the same file that contains the definitions of the class noninline member functions

# Integral const static members

- One exception to the previous slide: const static members of integral type can be initialized within the class body as long as the initializer is a constant expression

Account.h

```
class Account {
public:
 static double rate () {return interestRate;}
 static void rate(double); // sets a new rate
private:
 //interest posted every 30 days
 static const int period = 30;
 double daily_tbl (period); // ok: period is
 // a constant expression
};
```

Account.cc

```
//can use this definition
//but standard doesn't require
//it anymore
const int Account::period;
```

# Copy control

# Copy Control

When we define a new type, we specify - explicitly or implicitly – what happens when objects of that type are **copied** (*copy constructor*), **assigned** (*assignment operator*) and **destroyed** (*destructor*)

# Copy constructor

- Takes a single parameter that is usually (const) reference to an object of the class type itself
- Can be implicitly invoked by the compiler
  - like the default constructor
- The copy constructor is used to
  - explicitly or implicitly initialize one object from another of the same type
  - copy an object to pass it as an argument to a function
  - copy an object to return it from a function
  - initialize the elements in a sequential container
  - initialize the elements in an array from a list of element initializers

# Defining copy constructor

**// in the Sales\_item.h file we will have:**

```
class Sales_item {
public:
 // other functions
 Sales_item(const Sales_item &orig);
private:
 std::string isbn;
 int units_sold;
 double revenue;
};
```

**// in the Sales\_item.cc file we will have**

```
Sales_item::Sales_item (const Sales_item &orig): isbn(orig.isbn),
 units_sold (orig.units_sold),
 revenue (orig.revenue)
{ }
```

# Forms of object definition

- `string null_book = "9-999"; // copy initialization`
- `string dots (10, '.'); //direct-initialization`
- `string empty_copy = string (); //copy-initialization`
- `string empty_direct; //direct-initialization`
- Direct initialization
  - directly invokes the constructor matched by the arguments
- Copy initialization
  - first uses the indicated constructor to create a temporary object and then uses the copy constructor to copy that temporary into the one we are creating
- `ifstream file1("filename"); //ok:direct initialization`
- `ifstream file2 = "filename"; //error: copy constructor is private`
- The following is OK only if the `Sales_item(const string&)` constructor is not ***explicit***
  - `Sales_item item = string ("9-999");`



# More uses of copy constructors

- `string make_plural (size_t, const string&, const string);` // copy constructor used to copy the return value; second parameter is a reference so it won't be copied, third parameter will be copied using copy constructor
- `vector<string> svec (5);` // default string constructor and five string copy constructors invoked
  - general rule: unless you intend to use the default initial value of the container elements, it is more efficient to allocate an empty container and add elements as the values for those elements become known

# Synthesized copy constructor

- If we don't define a copy constructor the compiler synthesizes one for us
  - a copy constructor is synthesized even if we define other constructors
- Memberwise initializes the new object as a copy of the original object
- Usually it's a better practice to not rely on the compiler and define our own copy constructor
  - we **must** define our own copy constructor when
    - there is a data member that is a pointer or that represents another resource that is allocated in the constructor
    - there is some bookkeeping that must be done whenever a new object is created
- Preventing copies
  - remember: if we don't define a copy constructor, the compiler will synthesize one
  - to prevent copies, a class must explicitly declare its copy constructor as ***private***

# The assignment operator

- Sales\_item trans, accum;
- How do we make this happen:
  - trans = accum;
- ***Overloaded operators***
  - Functions that have the name *operator* followed by the symbol for the operator being defined
  - An operator function has a return type and a parameter list
    - The parameter list must have the same number of parameters (including the implicit *this* parameter if the operator is a member) as the operator has operands
  - Most operators may be defined as member or nonmember functions
    - when an operator is a member function, its first operand is implicitly bound to the *this* pointer
    - the assignment operator **must** be a member of the class for which it is defined

```
class Sales_item {
public:
 //other members as before
 Sales_item& operator=(const Sales_item &);
};

// (almost) equivalent to the synthesized assignment operator
Sales_item& Sales_item::operator=(const Sales_item &rhs)
{
 if (this == &rhs) return *this; //check for self assignment
 isbn = rhs.isbn;
 units_sold = rhs.units_sold;
 revenue = rhs.revenue;
 return *this;
}
```

- Classes that can use the synthesized copy constructor usually can use the synthesized assignment operator as well.
- If a class needs a copy constructor, it will also need an assignment operator

# The destructor

- Special member function that can be used to do whatever resource deallocation is needed
  - complements the constructor
- The destructor is called automatically whenever an object of its class is destroyed

```
Sales_item *p = new Sales_item ();
{
 Sales_item item(*p); //copy constructor copies *p into item
 delete p; //destructor called on object pointed by p
 // exit local scope; destructor called on item
}
```

# The destructor (contd.)

- **Note:** the destructor is not run when a reference or a pointer to an object goes out of scope
  - the destructor is only run when a pointer to a dynamically allocated object is *deleted* or when an actual object (not a reference to the object) goes out of scope

```
Sales_item *p = new Sales_item[10]; //dynamically allocated
vector<Sales_item> vec;
vec.push_back (p[0]);
vec.push_back (p[1]);
delete [] p; //array is freed
//vec goes out of scope; destructor run on each element
//elements in the container are always destroyed in reverse order
}
```

# Writing a destructor

- Sales\_item allocates no resources and therefore doesn't need its own constructor
- The destructor is a member function with the name of the class prefixed by a tilde (~)
  - it has no return value and takes no parameters
  - cannot be overloaded (because it takes no parameters)
  - we can provide only one destructor
  - even if we write our own destructor, the synthesized destructor is still run
    - destroys each non static member in the reverse order from that in which the object was created (invokes destructors of data members if needed)

```
class Sales_item {
public:
 ~Sales_item () {}
 // other members as before
}
```

# When to write an explicit destructor?

**Rule of three:** if a class needs a destructor, it will also need the assignment operator and a copy constructor



# Object Oriented Programming (OOP)

# Overview

- Many applications are characterized by concepts that are related but slightly different
  - e.g., our bookstore might offer different pricing strategies for different books
- Object-oriented programming (OOP) is a good match to this kind of applications
  - through ***inheritance*** we can define types that model the different kinds of books
  - through ***dynamic binding*** we can write applications that use these types but that can ignore type-dependent differences

# Object-Oriented Programming

Data Abstraction (C++:classes)

Inheritance (C++:class derivation)

Dynamic binding

# Two different types, but

## Hourly employee

get\_name

get\_ID

set\_name

set\_ID

print\_check

get\_hourRate

set\_hourRate

get\_hours

set\_hours

## Salaried employee

get\_name

get\_ID

set\_name

set\_ID

print\_check

get\_salary

set\_salary

# there is quite a lot of similarity

## Hourly employee

get\_name

get\_ID

set\_name

set\_ID

print\_check

get\_rate

set\_rate

get\_hours

set\_hours

## Salaried employee

get\_name

get\_ID

set\_name

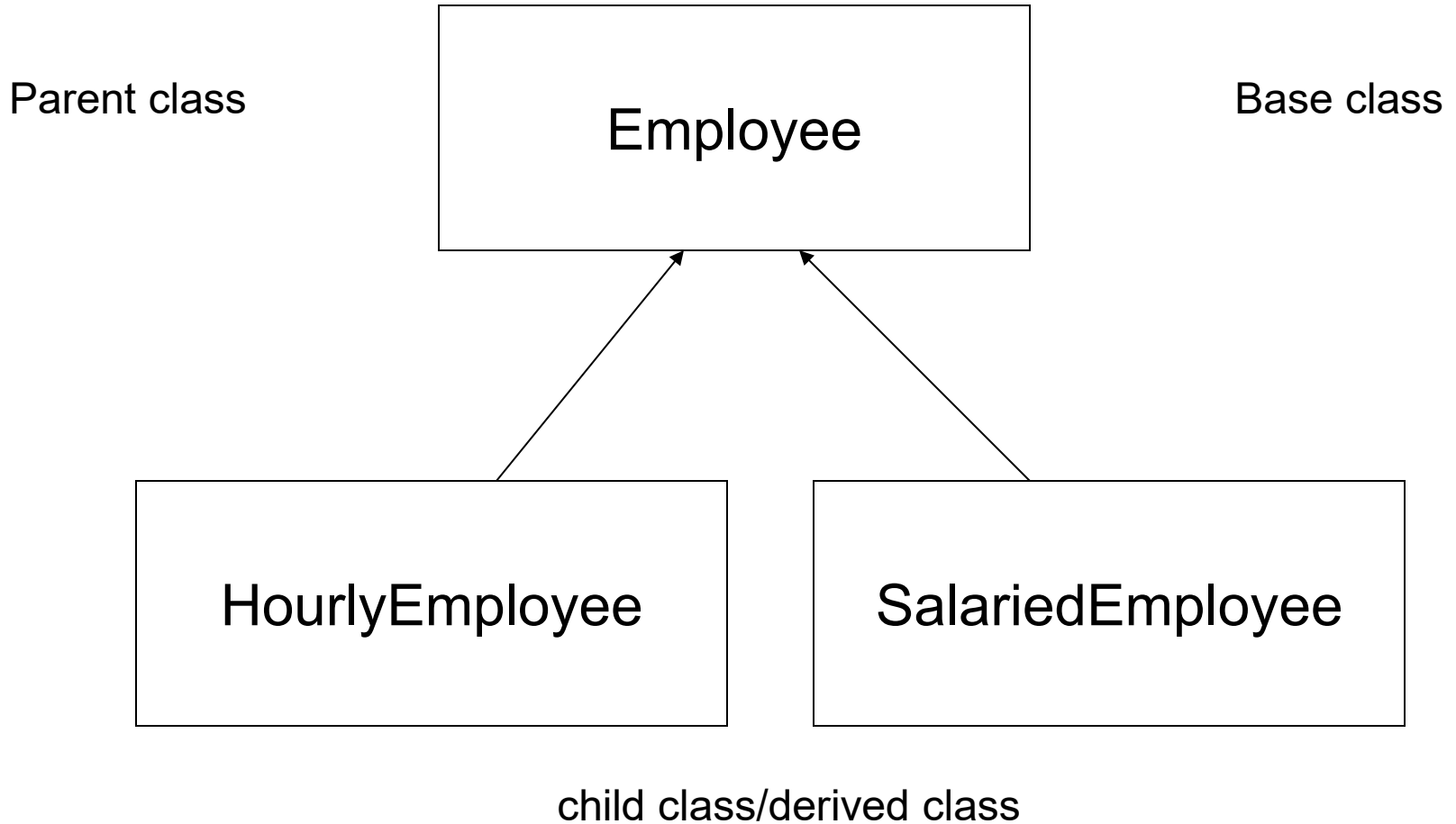
set\_ID

print\_check

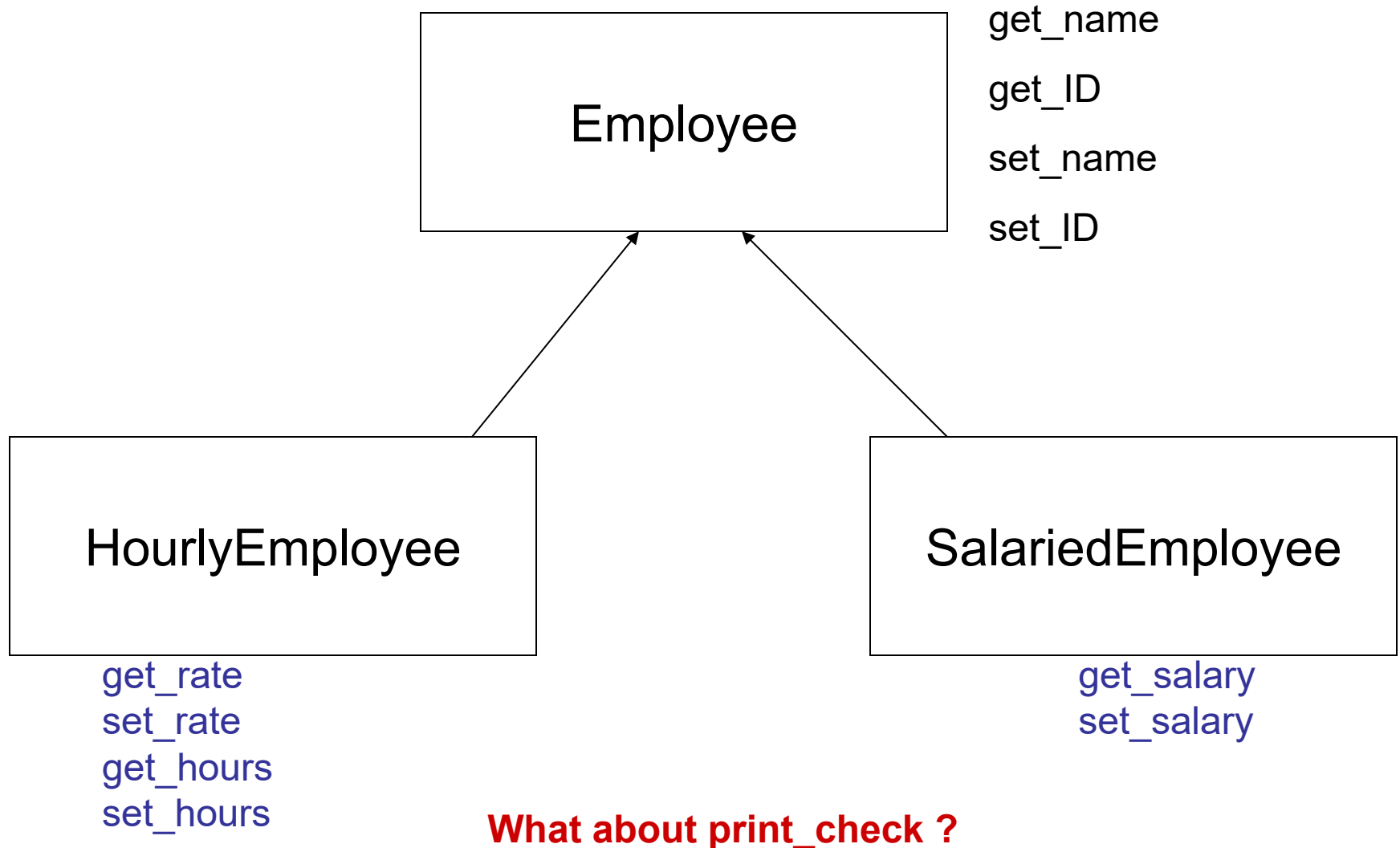
get\_salary

set\_salary

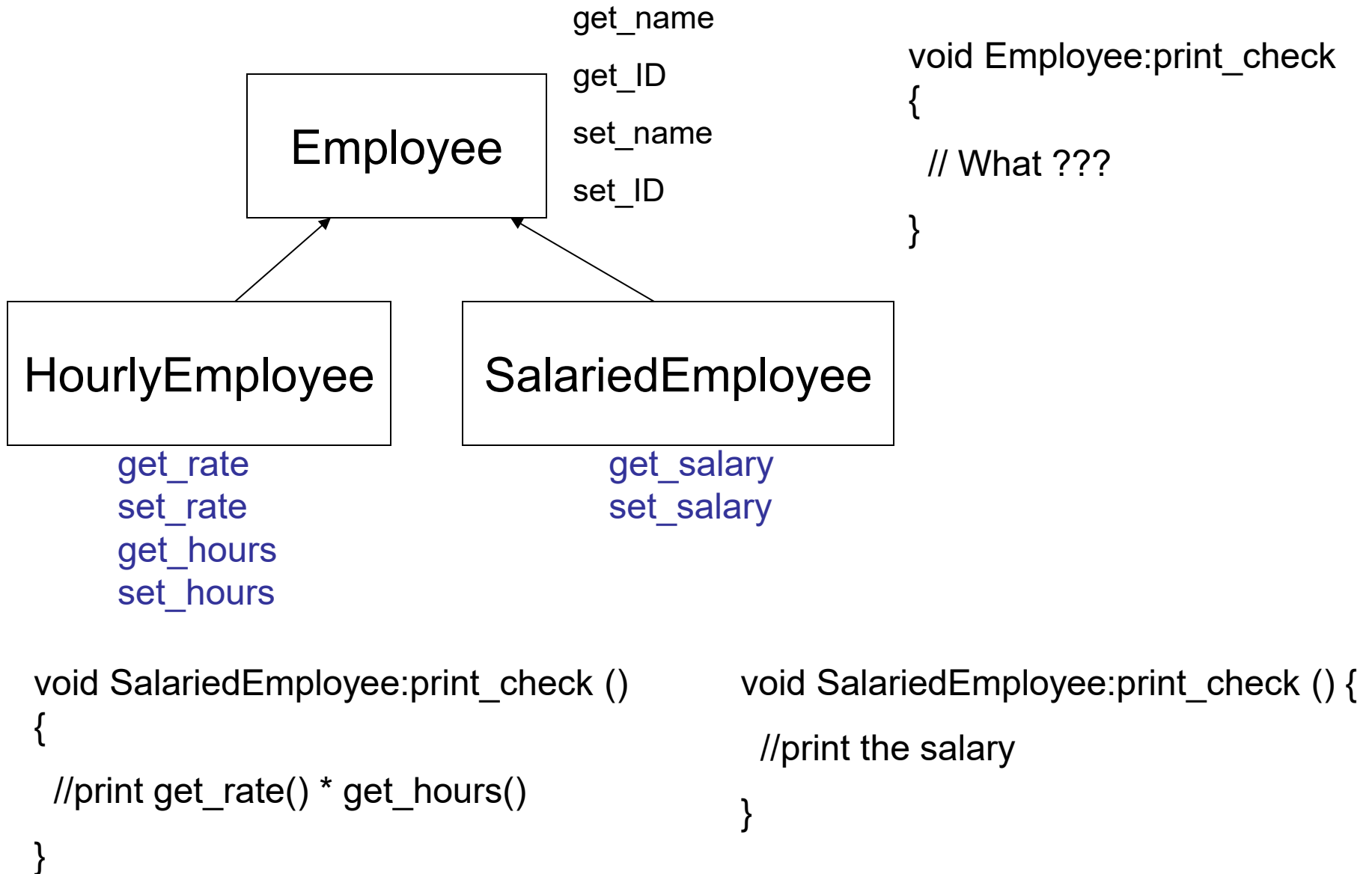
# Inheritance



# Inheritance (contd.)



# How about that ?





# The Employee class

```
class Employee {
public:
 Employee ();
 Employee (const string &name, const string &ID);
 string get_name () const;
 string get_ID () const;
 void set_name (const string &newName);
 void set_ID (const string &newID);
 void print_check () const;
private:
 string _name;
 string _ID;
};
```

```
class Employee {
public:
 Employee ();
 Employee (const string &name, const string &ID);
 string get_name () const;
 string get_ID () const;
 void set_name (const string &newName);
 void set_ID (const string &newID);
 void print_check () const;
private:
 string _name;
 string _ID;
};

class HourlyEmployee: public Employee {
public:
 HourlyEmployee ();
 HourlyEmployee (const string &name, const string &ID, double rate, double hours) ();
 void set_rate (double new_rate);
 void set_hours (double new_hours);
 double get_rate () const;
 double get_hours () const;
 void print_check () const;
private:
 double _rate;
 double _hours;
};
```

```
class Employee {
public:
 Employee ();
 Employee (const string &name, const string &ID);
 string get_name () const;
 string get_ID () const;
 void set_name (const string &newName);
 void set_ID (const string &newID);
 void print_check () const;
private:
 string _name;
 string _ID;
};
```

```
class SalariedEmployee: public Employee {
public:
 SalariedEmployee ();
 SalariedEmployee (const string &name, const string &ID, double salary) ();
 void set_salary (double salary);
 double get_salary () const;
 void print_check () const;
private:
 double _salary;
};
```

```

class SalariedEmployee: public
Employee {
public:
 SalariedEmployee ();
 SalariedEmployee (
 const string &name, const string &ID,
 double salary) ();
 void set_salary (double salary);
 double get_salary () const;
 void print_check () const;
private:
 double _salary;
};

```

```

int main () {
 SalariedEmployee s ("Oded",
 "233245",5000.0);
 HourlyEmployee h ("Moshe",
 "29229",50.0,20.0);
 cout << s.get_name () <<
 h. get_name () << endl;
}

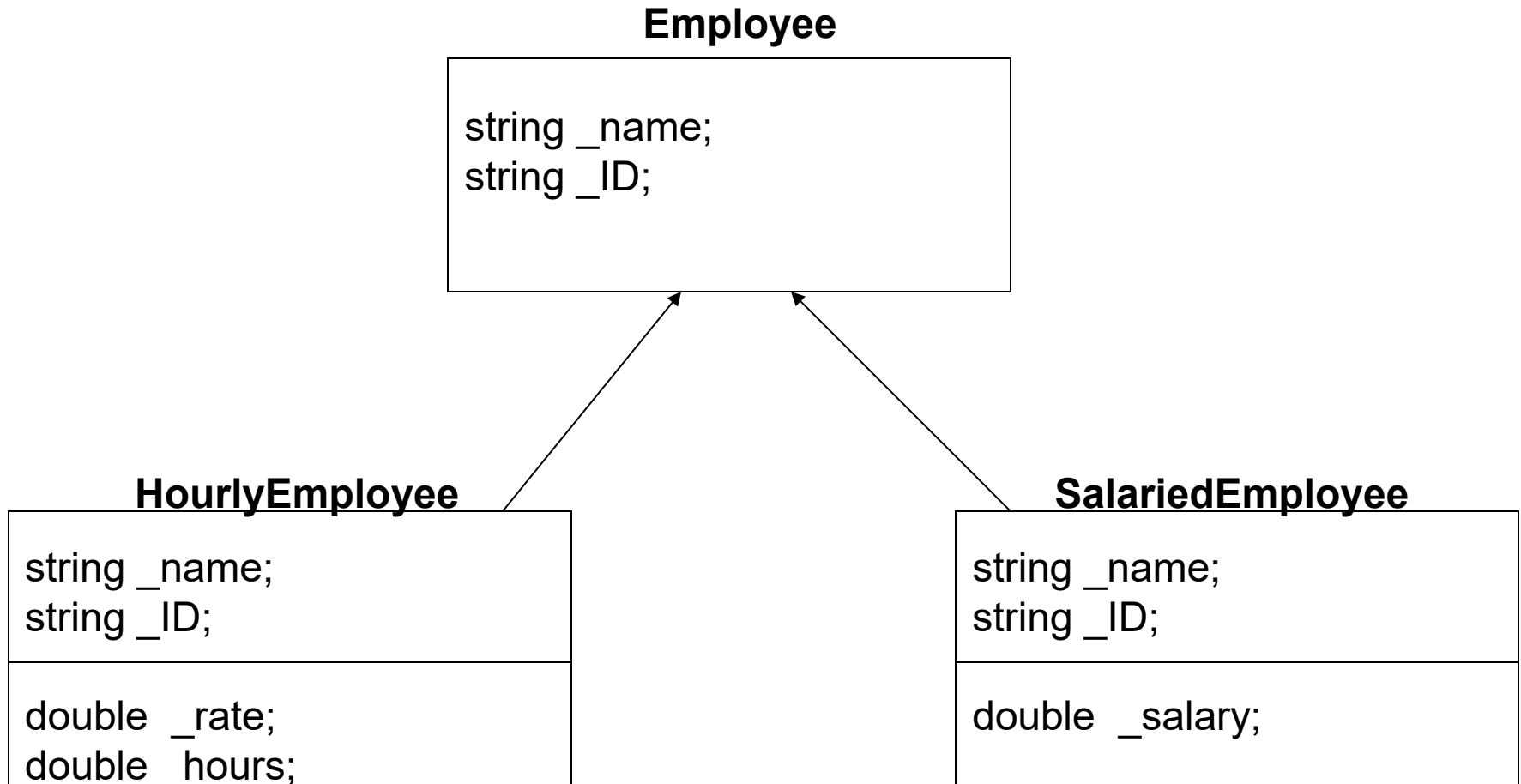
```

```

class HourlyEmployee: public
Employee {
public:
 HourlyEmployee ();
 HourlyEmployee (const string &name,
 const string &ID, double rate, double
 hours) ();
 void set_rate (double new_rate);
 void set_hours (dboule new_hours);
 double get_rate () const;
 double get_hours () const;
 void print_check () const;
private:
 double _rate;
 double _hours;
};

```

# The structure of the objects



# HourlyEmployee Constructors

```
HourlyEmployee::HourlyEmployee (): Employee (), _rate (0.0), _hours(0.0)
{ }
HourlyEmployee::HourlyEmployee (const string &name, const string &ID,
 double rate, double hours): Employee (name,ID), _rate (rate), _hours(hours)
{ }
```

- Constructors are not inherited in the derived class, but you can (and have to) invoke a constructor of the base class as the first thing on the initialization list of the derived class constructor
- If B is derived from A, and C is derived from B, then when an object of class C is created, first a constructor of A is called, then a constructor of B is called, and only then the remaining actions of the C constructor are taken
- The constructor initializer list for a derived-class constructor may initialize only the members of the derived class; it may not directly initialize its inherited members. Instead, a derived constructor indirectly initializes the members it inherits by including its base class in its constructor initializer list

# HourlyEmployee Methods

```
HourlyEmployee::HourlyEmployee (): Employee (), _rate (0.0), _hours(0.0)
{}
HourlyEmployee::HourlyEmployee (const string &name, const string &ID,
 double rate, double hours): Employee (name,ID), _rate (rate), _hours(hours)
{}

void HourlyEmployee::print_check () const {
 //compute and print
}
```

# HourlyEmployee destructor

```
HourlyEmployee::~~HourlyEmployee (){ }
```

**Destructors are not inherited as well. The destructor of the derived class needs only to worry about releasing (owned) resources that are added in the derived class**

**If B is derived from A, and C is derived from B, then when an object of class C is "destroyed", first the destructor of C is called, then the destructor of B is called, and finally the destructor of A is called**



# Example

```
class A {
public:
 A::A () {cout <<" A is created " << endl;}
 A::~~A () { cout << "A is dead" << endl;}
};
class B : public A {
public:
 B::B () {cout << "B is created" << endl;}
 B::~~B () {cout << "B is dead" << endl;}
};
class C : public B {
public:
 C::C () {cout << "C is created" << endl;}
 C::~~C () {cout << "C is dead" << endl;}
};

C *p = new C;
delete p;
```

# HourlyEmployee and SalariedEmployee are employees !

- In everyday experience an hourly employee and a salaried employee are both employees, and this relation is naturally supported in C++

Important: You can use an object of a derived class anywhere that an object of its base class is allowed

- you can use an argument of type HourlyEmployee/SalariedEmployee when a function requires an argument of type Employee
  - **but do it only when you pass parameters by reference !**
- you can assign an object of a class HourlyEmployee/SalariedEmployee to a variable of type Employee (but not vice versa!)
  - **but be aware of the consequences (unless you define a reference)**

# Example

```
class A {
};
class B :public A {
};

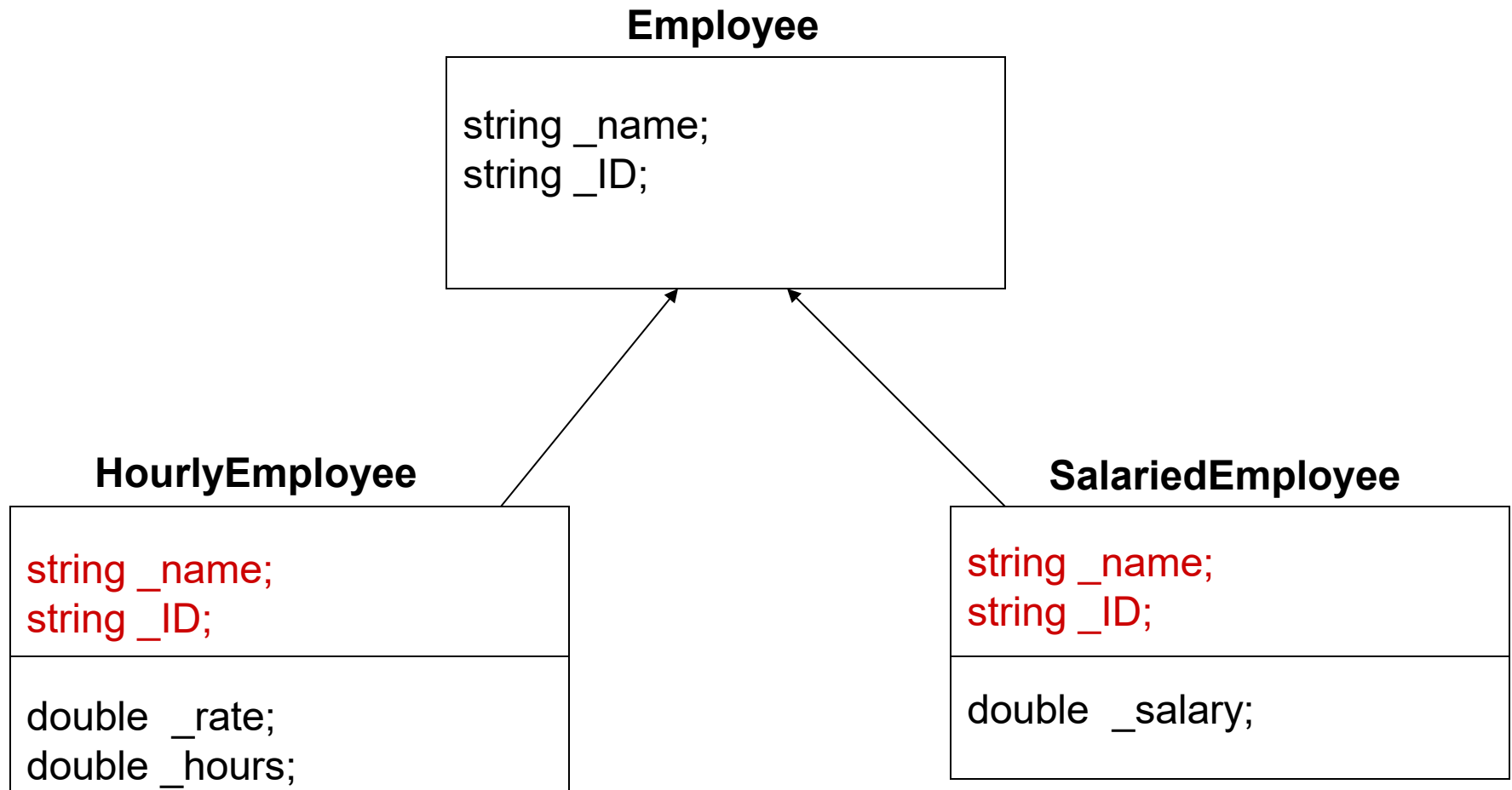
void myFunc (A& a) {
}

int main () {
 B b;
 A &aRef = b;
 A *aPtr = &b;
 A aa = b; // but be carefull!
}
```

# Inheritance vs. Composition

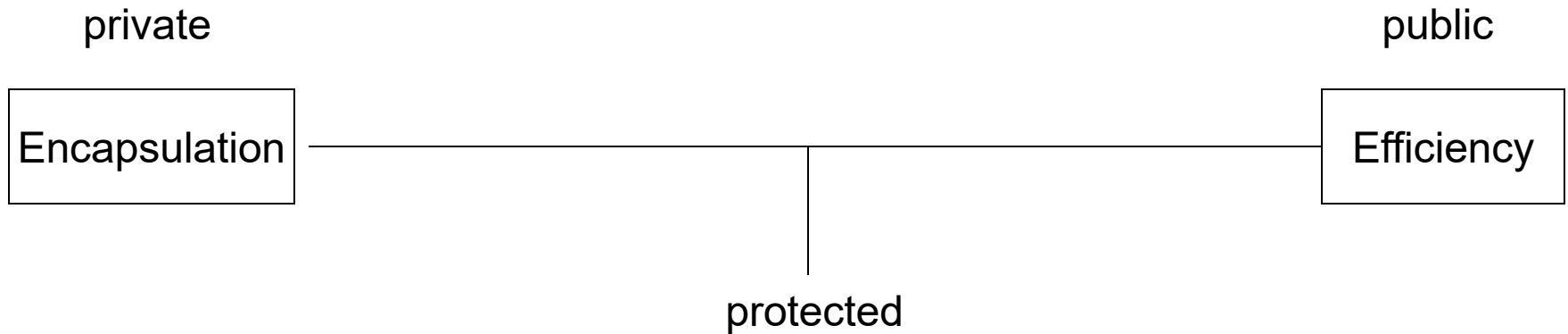
- Inheritance represents an **"IS A"** relationship
  - *class X: public Y* means that every object of type X is a Y
  - A SalariedEmployee is an Employee
- Composition: **"HAS A"** relationship
  - Types related by "Has A" relationship imply membership
  - Employee classes are composed from members representing ID and name

# What is inherited ?



Why is it good that the derived classes have no access to the members `_name` and `_ID` ?

# Visibility in inheritance



Protected acts as *public* to derived classes and as *private* to the rest of the world

With *public inheritance* the access control for the derived parts remains the same as it was in the base class

# Protected members

A member should be made protected if it provides an operation or data that a derived class will need to use in its implementation

# Protected members

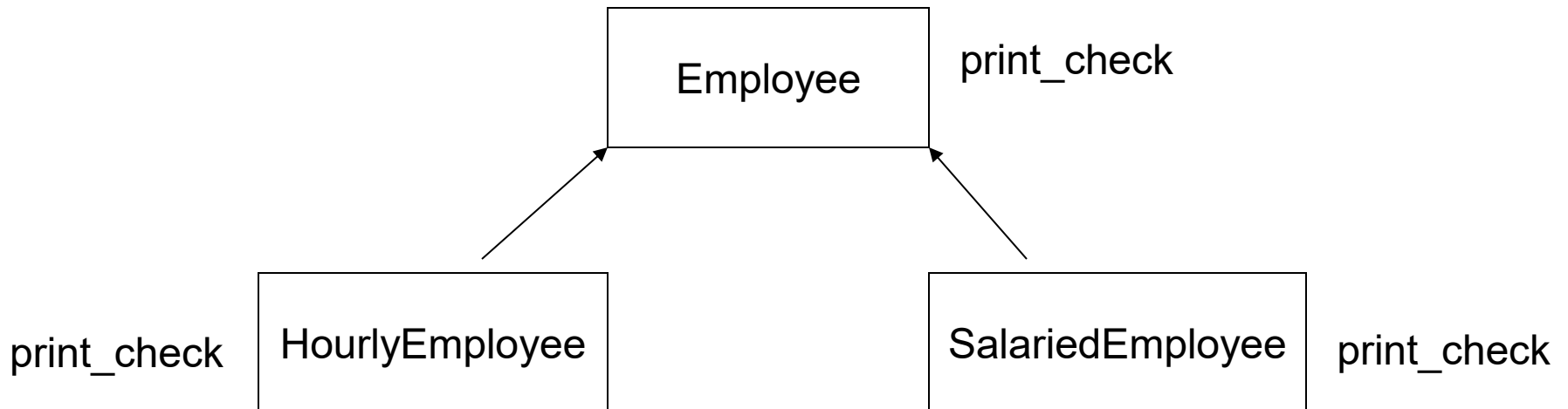
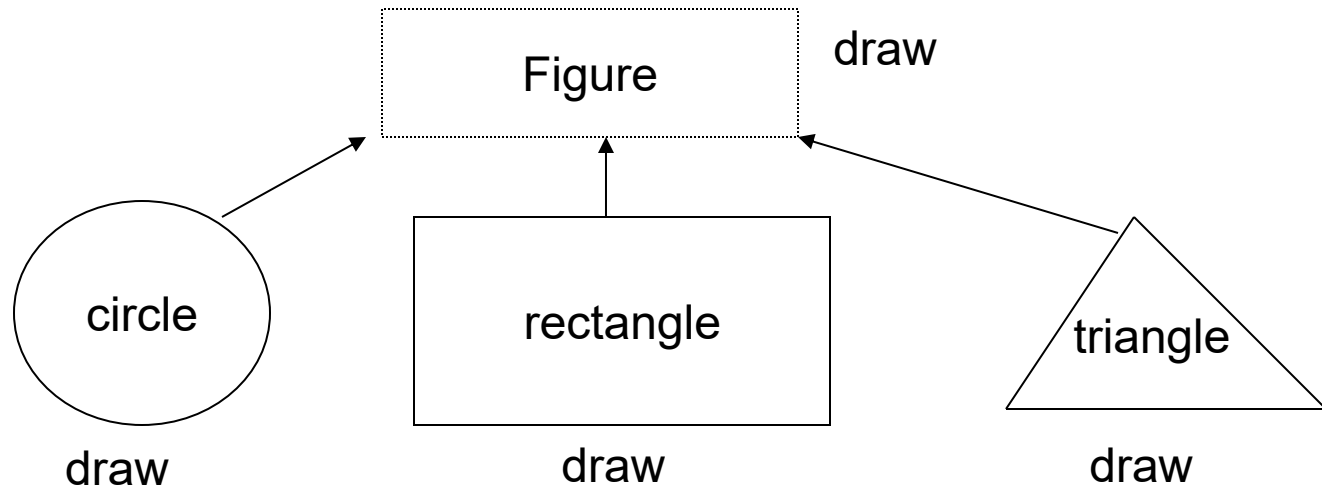
```
class A {
protected:
 int _field;
};

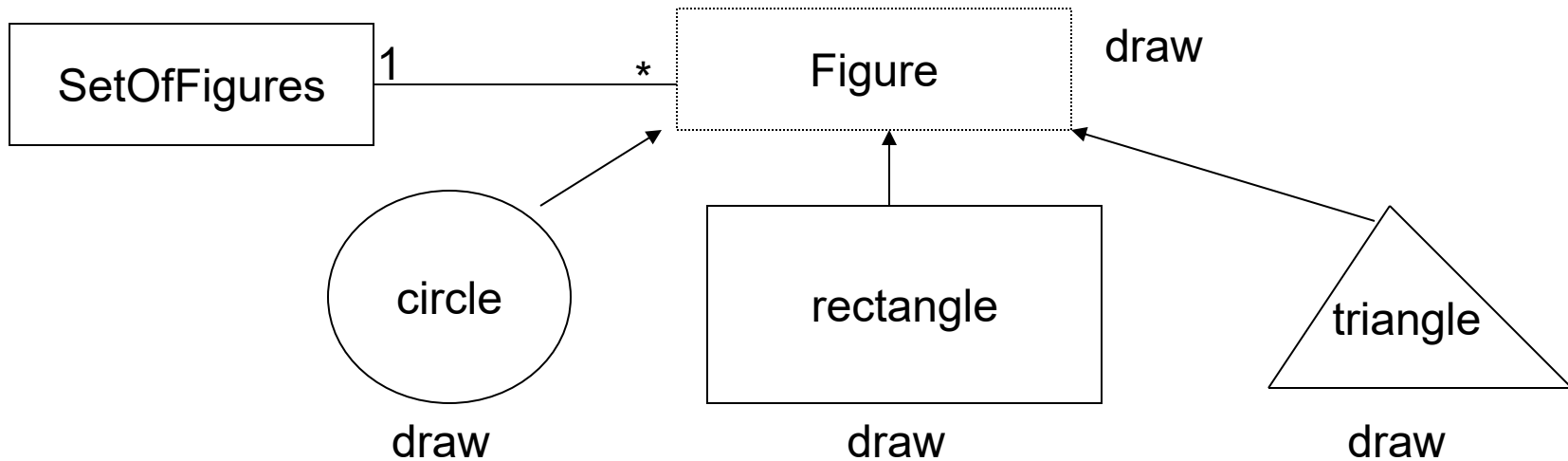
class B : public A{
public:
 void firstFunc () {cout << _field << endl;} // OK
 void myFunc (A& a) {cout << a._field << endl;} // error !!!
 void myFunc (B& b) {cout << b._field << endl;} //OK
};

int main () {
 A a;
 cout << a._field << endl; //error
 B b;
 cout << b._field << endl; //error
}
```



# Polymorphism ("many forms")

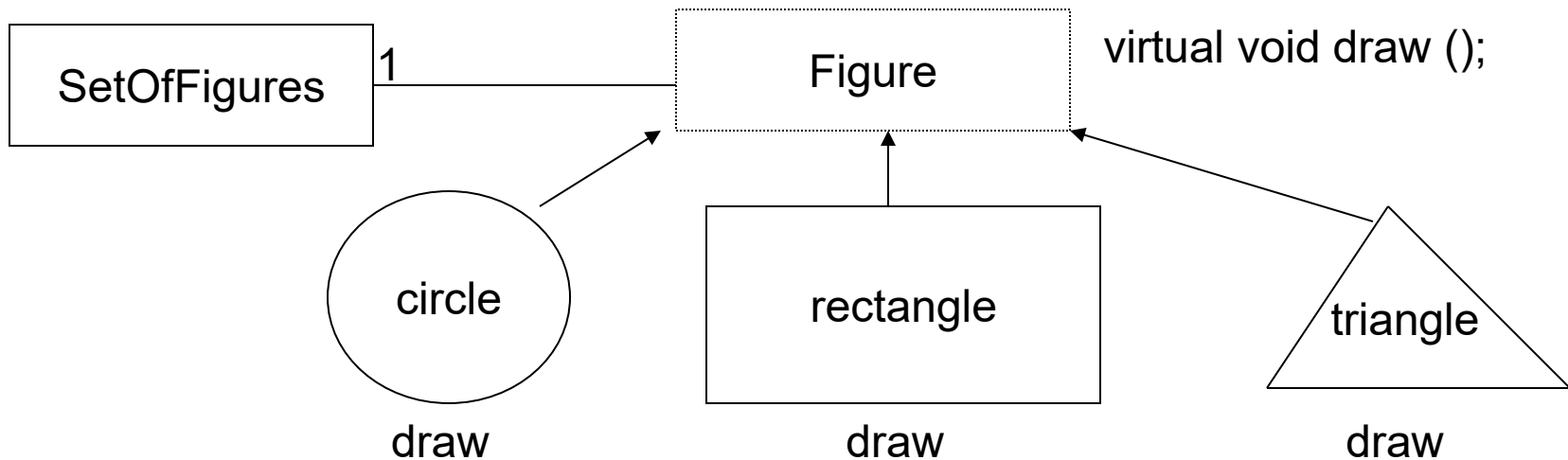




```
class SetOfFigures {
private:
 Figure *figArray [numFigs];
public:
 void drawFigures () {
 for (size_t i = 0; i < numFigs; i++) {
 figArray[i] -> draw ();
 }
 };
};
```

# Polymorphism

- Philosophy:
  - associating multiple meanings to a single name of action
- Object-oriented programming:
  - Associating multiple meanings to one function name by means of a special mechanism known as *late binding*
  - We speak of types related by inheritance as polymorphic types, because in many cases we can use the "many forms" of a derived or base class interchangeably
- In general, a base class should indicate which of its functions it intends for its derived classes to redefine. Functions defined as **virtual** are ones that the base class expects its derived classes to redefine



When you make a call to a virtual function you tell the compiler: "I do not know which function should be invoked here. Wait until it is actually invoked in a program, and then get its implementation from the object on which it is called."

The technique of waiting until run-time to determine which function should be actually invoked is called *late binding* (a.k.a. dynamic binding)

Binding ? a message with a method

# Virtual functions

```
class Sale {
public:
 Sale ();
 Sale (double thePrice);
 virtual double bill () const {return _price;}
 double savings (const Sale& other) const {return bill() – other.bill ();}
protected:
 double _price;
};
```

```
class DiscountSale : public Sale {
public:
 DiscountSale () :Sale (), _discount(0.0) {}
 DiscountSale (double the_price, double the_discount):
 Sale(the_price),_discount(the_discount) {}
 virtual double bill () const {return _price*(1-_discount/100);}
protected:
 double _discount;
};
```

# Late binding

```
Sale s, *ps1, *ps2;
```

```
DiscountSale ds, *pds;
```

```
ps1= new Sale (34);
```

```
ps2 = new DiscountSale (100,20);
```

```
pds = new DiscountSale (150,30);
```

| Object | Static Type  | Dynamic Type |
|--------|--------------|--------------|
| s      | Sale         | Sale         |
| ds     | DiscountSale | DiscountSale |
| *ps1   | Sale         | Sale         |
| *ps2   | Sale         | DiscountSale |
| *pds   | DiscountSale | DiscountSale |

# Late Binding (contd.)

```
Sale s, *ps1, *ps2;
```

```
DiscountSale ds, *pds;
```

```
ps1= new Sale (34);
```

```
ps2 = new DiscountSale (100,20);
```

```
pds = new DiscountSale (150,30);
```

```
s.savings(*pds);
```

```
s.bill ();
```

```
ps1 -> bill (); //virtual function
```

```
ps2->savings (ds);
```

```
pds -> bill (); //virtual function
```

```
ds.bill ();
```

Calls to **virtual** functions are resolved at run-time only if the call is made through a reference or a pointer

(Because only in these cases it is possible for an object's dynamic type to be unknown until run-time.)

# One more example on virtuals

```
class A {
public:
 virtual void print () { cout << "In print of class A" << endl;}
};
class B : public A {
public:
 virtual void print () {cout << "In print of class B" << endl;}
};
void myFunc (A& parA) {
 parA.print ();
}
int main () {
 A* a = new B;
 a->print (); //prints "In print of class B"
 A *aa = new A;
 aa->print (); //prints "In print of class A"
 B b;
 A &refA = b;
 refA.print (); //prints "In print of class B"
 myFunc (b); //print "In print of class B"
}
```



# Name lookup happens at compile time

```
class A {
public:
 void aFunc () { }
};
```

```
class B : public A {
public:
 void bFunc () { }
};
```

```
B b;
```

```
B *bPTR = &b; //ok, static and dynamic types are the same
```

```
A *aPTR = &b; // ok, static and dynamic types differ
```

```
bPTR -> bFunc (); //ok: bPTR has type B*
```

```
aPTR -> bFunc (); //error: aPTR has type A*
```

# Name collisions

- A derived-class member with the same name as a member of the base class hides direct access to the base-class member

```
class A {
public:
 A () : mem (0) { }
protected:
 int mem;
};
class B : public A {
public:
 B (int i): mem (i) { } //initialized B::mem
 int get_mem () const {return mem;} //returns B::mem
protected:
 int mem;
};
int main () {
 B b(42);
 cout << b.get_mem () << endl; //prints 42
}
```

# Collisions of function names

- A member function with the same name in the base and derived class behaves the same way as a data member
  - the derived-class member hides the base-class member within the scope of the derived class
  - the base member is hidden, even if the prototypes of the functions differ !
- **Recall:** functions declared in local scope don't overload functions defined at global scope. Similarly, functions defined in a derived class do **not** overload members defined in the base class.

# Collisions of function names (contd.)

```
class A {
public:
 int memfcn ();
};
class B : public A {
public:
 int memfcn(int); //hides memfcn of A
};
int main () {
 A a;
 B b;
 a.memfcn (); //calls A::memfcn
 b.memfcn (10); //calls B::memfcn
 b.memfcn (); //error: memfcn with no arguments is hidden
 b.A::memfcn (); //ok: calls A::memfcn
}
```

# Function calls resolution

1. Start by determining the static type of the object, reference, or pointer through which the function is called
2. Look for the function in that class. If it's not found, look in the immediate base class and continue up the chain of classes until either the function is found or the last class is searched. If the name is not found in the class or its enclosing base classes, then the call is in error
3. Once the name is found, do normal type-checking to see if this call is legal given the definition that was found
4. Assuming the call is legal, the compiler generates code. If the function is virtual and the call is through a reference or a pointer, then the compiler generates code to determine which version to run based on the dynamic type of the object. Otherwise, the compiler generates code to call the function directly.

# Destructors

```
Sale s, *ps1, *ps2;
```

```
DiscountSale ds, *pds;
```

```
ps1= new Sale (34);
```

```
ps2 = new DiscountSale (100,20);
```

```
pds = new DiscountSale (150,30);
```

```
delete ps1;
```

```
delete ps2;
```

```
delete pds;
```

If we delete a pointer to a base class then the base-class destructor is run and the members of the base are cleaned up.

If we delete a pointer to base, and the object is really of a derived type, then the behavior is undefined

# Destructors (contd.)

- To ensure that the proper destructor is run, the base-class destructor must be defined as *virtual*

```
class Sale {
public:
 virtual ~Sale () { }
};
```

The root class of an inheritance hierarchy should define a virtual destructor even if the destructor has no work to do

# Overriding the virtual mechanism

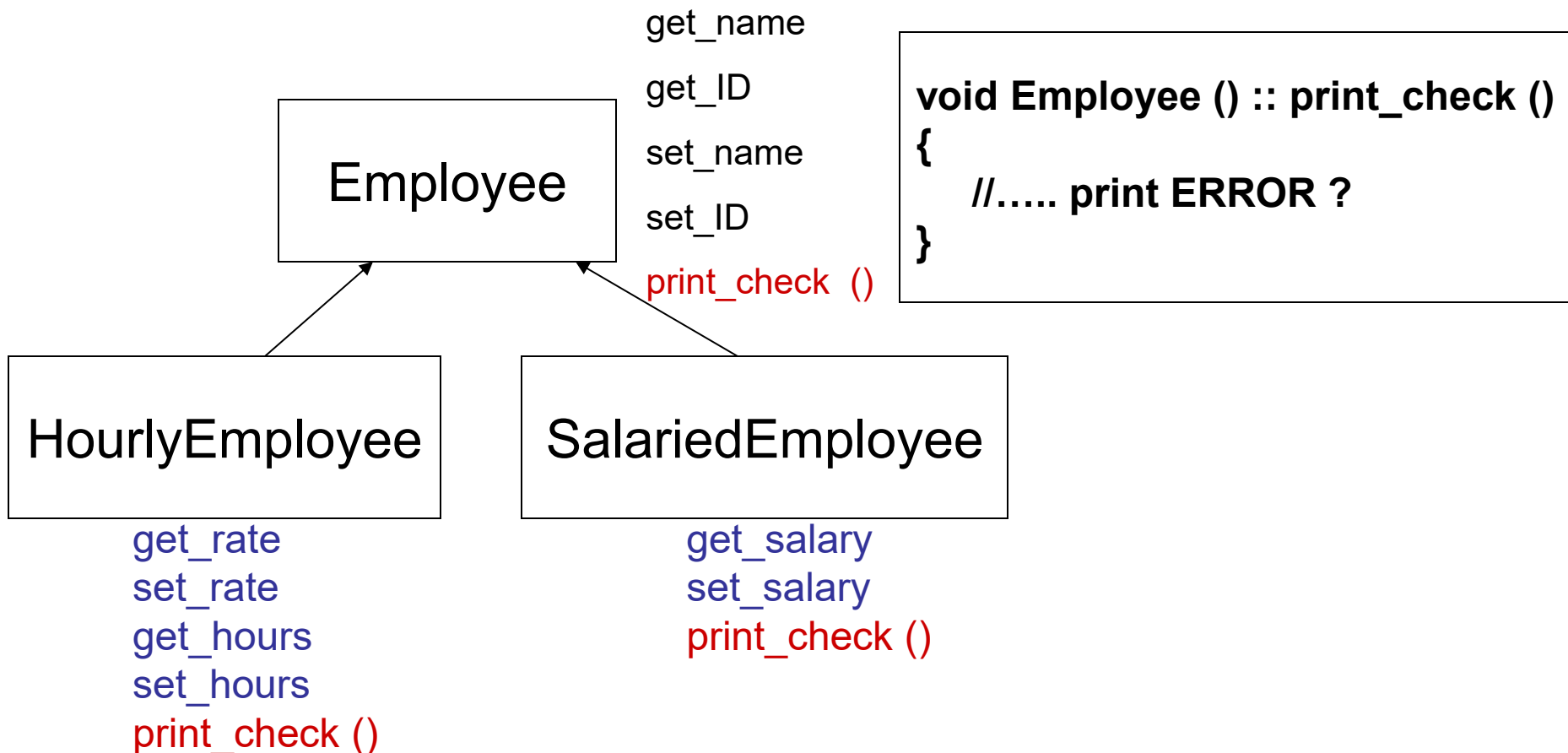
```
class Sale {
public:
 Sale ();
 Sale (double thePrice);
 virtual double bill () const {return _price;}
 double savings (const Sale& other) {return bill() – other.bill ();}
protected:
 double _price;
};

class DiscountSale : public Sale {
public:
 DiscountSale () :Sale (), _discount(0.0) {}
 DiscountSale (double the_price, double the_discount):
 Sale(the_price),_discount(the_discount) {}
 //virtual double bill () const {return _price*(1-_discount/100);}
 virtual double bill () const {return Sale::bill ()*(1-_discount/100);}
protected:
 double _discount;
};
```



# Overriding the virtual mechanism (contd.)

- Only code inside member functions should ever need to use the scope operator to override the virtual mechanism
- The most common reason to use this mechanism is when a derived-class virtual calls the version from the base
  - As we saw on the previous slide
  - In such cases the base class version might do work common to all types in the hierarchy
    - Each type adds only whatever is particular to its own type



print\_check is a part of *Employee*'s interface and it can be (not only redefined but) overridden in the derived classes

Pitfall: implementation of print\_check in Employee is not natural (is the class Employee itself artificial ?)

# Abstract types and pure virtual functions

```
class Employee {
public:
 ...
 virtual void print_check () const = 0;
 ...
};
```

Defining a virtual function as pure (with the =0) indicates that the function provides an interface for subsequent types to override, but that the version in this class will never be called

Users will not be able to create objects of type *Employee*. In general, a class containing or (inheriting) one or more pure virtual functions is an **abstract class**

# The benefits of inheritance

- Code reuse
  - Inheritance allows you to modify or extend an existing package without touching the package code
  - Save programming time, increase reliability, and decrease maintenance time
- Consistency of interface
  - Guarantees that interface to similar objects is indeed similar, making the development process more natural and thus faster
- Polymorphism
  - Different objects of the same type behave in response to the same "message"

# Containers and inheritance

- We would like containers (or built-in arrays) to hold objects that are related by inheritance. However, the fact that in C++ the objects are not *polymorphic* makes it effectively impossible
- A possible solution is to keep a container of pointers
  - The user must insure that the pointed objects stay around for as long as the container does
  - If the objects are dynamically allocated, then the user must ensure that they are properly freed when the container goes away
- The common technique used in C++ to overcome this problem is based on "very smart" pointers called "handle classes" (We might discuss them later in the course)

# Multiple inheritance

# Multiple inheritance

- The ability to derive a class from more than one immediate base class
- A multiply derived class inherits the properties of all its parents
- Can cause tricky design-level and implementation-level problems

# Example: modeling a zoo

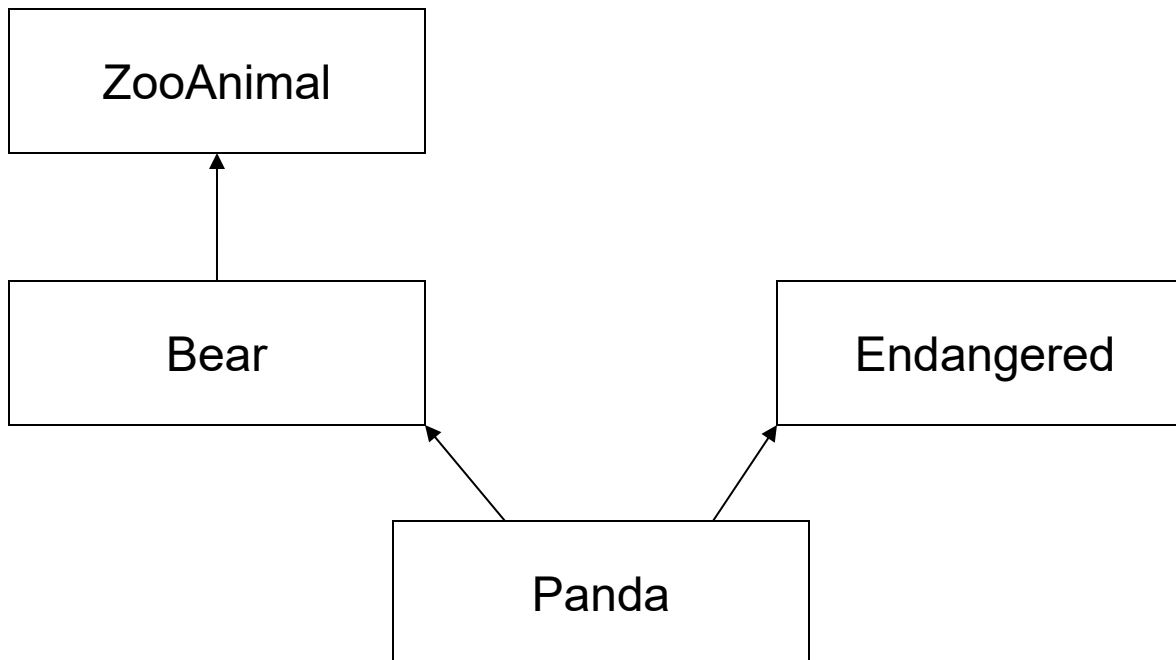
- Individual animals, distinguished by names
  - Ling-ling, Mowgli, Balou
- Each animal belongs to a species
  - e.g., Ling-ling is a giant panda
  - a giant panda is a member of the bear family
  - Each family is a member of the animal kingdom
    - in our case, the limited kingdom of the zoo
- We will define
  - abstract ZooAnimal class to hold information that is common to all zoo animals and provides the public interface
  - a Bear class that will contain information unique to the Bear family
  - and along these lines, more classes



# Inheritance

```
class Panda : public Bear, public Endangered {
};
```

```
Panda ying_yang ("ying_yang");
```



# Construction

```
//explicitly initialized both base classes
Panda :: Panda (std::string name, bool onExhibit):
 Bear (name,onExhibit,"Panda") ,
 Endangered(Endangered::critical)
// implicitly use Bear default constructor to initialize the Bear subobject
Panda::Panda() : Endangered (Endangered::critical)
```

**The base-class instructors are invoked in the order in which they appear in the class *derivation* list !!**

**- and not according to their order in the *initialization* list**

# Destruction

- Destructors are always invoked in the reverse order from which the constructors are run
- In our example the order will be: ~Panda, ~Endangered, ~Bear, ~ZooAnimal

# Conversion

**//operations that take references to base classes of type Panda**

**void print (const Bear&);**

**void highlight (const Endangered&);**

**void func(const ZooAnimal&);**

**Panda ying\_yang ("ying\_yang"); //create a Panda object**

**print (ying\_yang); //passes Panda as a reference to Bear**

**highlight(ying\_yang) ; //passes Panda as a reference to Endangered**

**func(ying\_yang); //passes Panda as a reference to ZooAnimal**

**void print (const Bear&);**

**void print (const Endangered&);**

**Panda ying\_yang ("ying\_yang");**

**print (ying\_yang); //error: ambiguous**

# Virtual functions

```
class ZooAnimal {
public:
 virtual void print ();
 virtual ~ZooAnimal ();
};
class Bear :public ZooAnimal {
public:
 virtual void print ();
 virtual void highlight ();
 virtual void toes ();
 virtual ~Bear ();
};
class Endangered {
public:
 virtual void print ();
 virtual void highlight ();
 virtual ~Endangered ();
};
class Panda : public Bear, public Endangered {
public:
 virtual void print ();
 virtual void highlight ();
 virtual void toes ();
 virtual void cuddle ();
 virtual ~Panda();
};
```

# Virtual functions (contd.)

```
Bear *pb = new Panda ("ying_yang");
pb->print (); //ok, calls Panda::print
pb->cuddle (); //error, not part of Bear interface
pb->highlight (); //ok, calls Panda:highlight ()
delete pb; //ok, calls Panda::~~Panda ()
//the same thing would have happened if Panda object had been assigned
//to a ZooAnimal pointer
```

```
Endangered *pe = new Panda ("ying_yang");
pe->print (); //ok, calls Panda::print
pe->toes (); //error: not part of Endangered interface
pe->cuddle (); //error, not part of Endangered interface
pe->highlight (); //ok, calls Panda::highlight()
delete pe; //ok, calls Panda::~~Panda ()
```

# Copy control

- Each base class is implicitly constructed, assigned, or destroyed, using that base class' own copy constructor, assignment operator or destructor

```
//Assuming Panda uses the default copy control members
Panda ying_yang("ying_yang"); //create a Panda object
Panda ling_ling = ying_yang; //uses copy constructor
```

In the example above the following copy constructors are run:

1. ZooAnimal
2. Bear
3. Endangered
4. Panda

# Class scope

- Name lookup for a name used in member functions starts in the function itself
- If the name is not found locally, lookup continues in the member's class and then in the base class
- Under multiple inheritance, the search simultaneously examines all the base-class inheritance subtrees
  - In our example, the *Endangered* and the *Bear/ZooAnimal* subtrees are examined in parallel
  - If the name is found in more than one subtree, then the use of that name must explicitly specify which base class to use
    - otherwise the use of the name is ambiguous



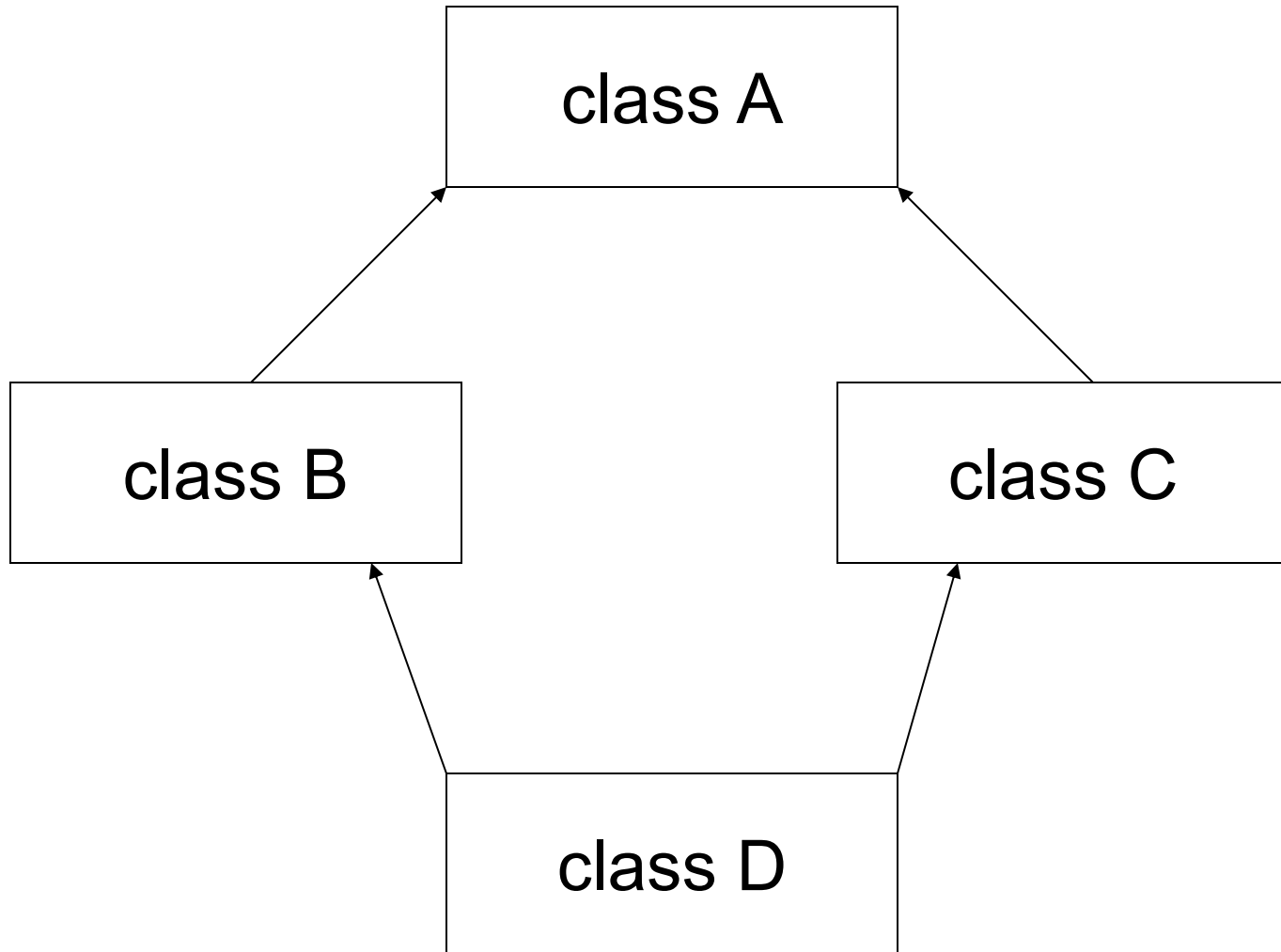
# Ambiguities

- Recall the `print` function defined in both *Bear* and *Endangered*
  - suppose that *Panda* doesn't define `print`, then the statement `ying_yang.print ()` results in a compile time error
  - It's legitimate to have `print` in both base classes. You can avoid ambiguity by specifying which version of *print* you want (e.g., use `ying_yang.Bear::print ()`)
  - an error would have been generated even if the two inherited functions had different parameter lists (similarly if *print* was *private* in one class and *public* in the other class)

## A good programming practice:

```
void Panda::print () const
{
 Bear::print ();
 Endangered::print ();
}
```

# Diamond



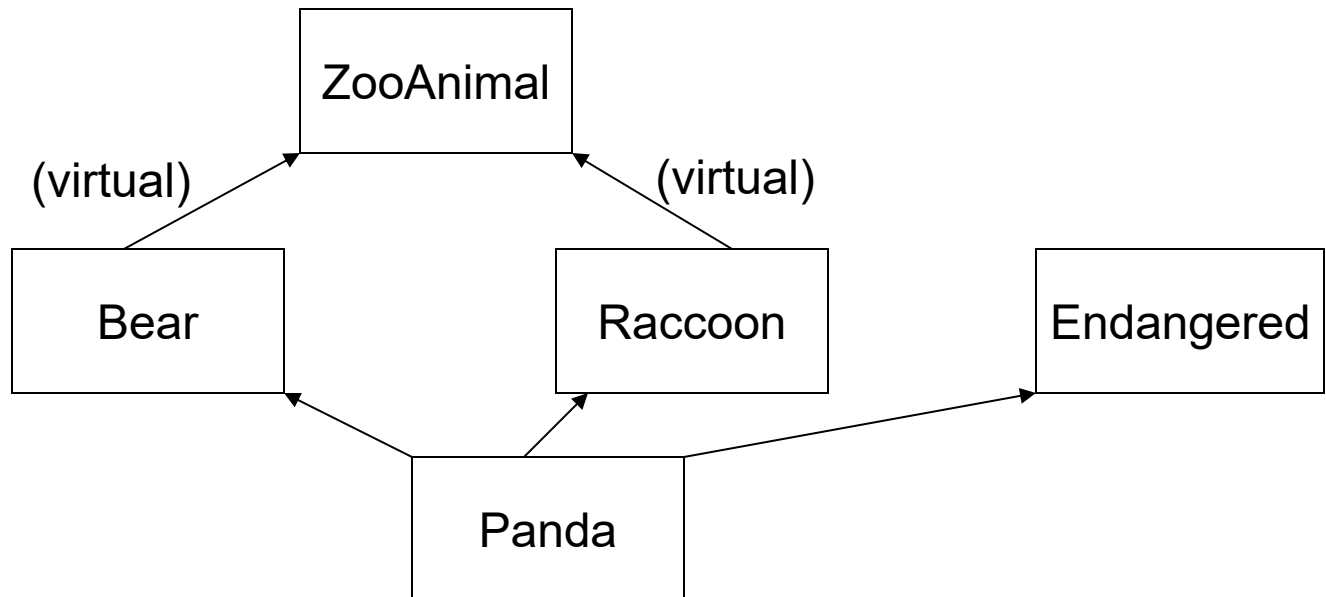
# Virtual inheritance

- A class specifies that it's willing to share the state of its virtual base class
- Only one, shared base class subobject is inherited for a given virtual base regardless of how many times the class occurs as a virtual base within the derivation hierarchy
- The shared base-class subobject is called a ***virtual base class***

```
class B: public virtual A {
};
class C: public virtual A {
};
class D: public B, public C {
};
```

# Back to the zoo example

```
class Bear : public virtual ZooAnimal {
};
class Raccoon : public virtual ZooAnimal {
};
class Panda : public Bear, public Raccoon, public Endangered {
};
```



# Normal conversions to base are supported

```
void dance (const Bear*);
void rummage (const Raccoon*);
void func(const ZooAnimal&);
Panda ying_yang;
dance (&ying_yang); //ok:converts address to pointer to Bear
rummage(&ying_yang); //ok:converts address to pointer to Raccoon
func (ying_yang); //ok:passes ying_yang as a ZooAnimal
```

Members in the shared virtual base can be accessed unambiguously and directly. If a member from the virtual base is redefined along only one derivation path, then that redefined member can be accessed directly. Under a nonvirtual derivation, both kinds of access would be ambiguous

# Special initialization semantics

- Ordinarily each class initializes only its own direct base class
- This initialization strategy fails when applied to a virtual base class
- If normal rules were used, then the virtual base might be initialized multiple times
  - The class would be initialized along each inheritance path that contains the virtual base
- In virtual derivation, the virtual base is initialized by the ***most derived constructor***
  - In our example, when we create a *Panda* object, the *Panda* constructor alone controls how the *ZooAnimal* base class is initialized

```
Bear::Bear(std::string name, bool onExhibit):
ZooAnimal(name, onExhibit, "Bear")
{ }
Raccoon::Raccoon(std::string name, bool onExhibit):
ZooAnimal(name, onExhibit, "Raccoon");
{ }

Panda::Panda(std::string name, bool onExhibit)
 : ZooAnimal(name, onExhibit, "Panda"),
 Bear(name, onExhibit),
 Raccoon(name, onExhibit),
 Endangered(Endangered::critical),
 sleeping_flag (false)
```

```
Bear winnie ("pooh",true);
Raccoon meeko("meeko",true);
Panda yolo ("yolo",true);
```

# Constructing a virtually inherited object

- When a Panda object is created
  - The ZooAnimal part is constructed first, using the initializers specified in the Panda constructor initializer list
  - Next the Bear part is constructed. The initializers for ZooAnimal Bear's constructor initializer list are ignored
  - Then the Raccoon part is constructed, again, ignoring the ZooAnimal initializers
  - Finally the Panda part is constructed
- If Panda's constructor doesn't explicitly initialize the ZooAnimal class, then the ZooAnimal default constructor is used
- Virtual base classes are always constructed prior to nonvirtual base classes regardless of where they appear in the inheritance hierarchy



# Generic programming with templates

# Object-oriented programming

- Relies on a form of polymorphism
  - polymorphism applies at run time to classes related by inheritance
  - code that uses such classes in ways that ignore the type differences among the base and derived classes

# Generic programming

- Lets us write classes and functions that are polymorphic across unrelated types at compile time
  - A single class or function can be used to manipulate objects of a variety of types
- standard library containers, iterators and algorithms are good examples of generic programming
  - The library defines each of the containers, iterators, and algorithms in a type-independent fashion
- In C++, **templates** are the foundation for generic programming

# Templates

- A template is a blueprint or formula for creating a class or a function
  - e.g., STL defines a single class template that defines what it means to be a *vector*; this template is used to generate any number of type-specific *vector* classes (e.g., `vector<int>`, `vector<string>`)

# Example

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare (const string &v1, const string &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
}
int compare (const double &v1, const double &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
}
```

# Defining a function template

- Rather than defining a new function for each type in the last example, we can define a single **function template**
  - **function template** is a type-independent function that is used as formula for generating type-specific version of the function

```
template <typename T> //can also write template <class T>
int compare (const T &v1, const T &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
}
```

# Using a function template

- When we use a function template, the compiler infers what **template argument(s)** to bind to the **template parameter(s)**
- Once the compiler determines the actual template argument(s), it **instantiates** an instance of the function template for us

```
int main ()
{
 //here compiler instantiates int compare (const int&, const int&)
 cout << compare (1,0) << endl;
 //here compiler instantiates int compare (const string&, const string&)
 string s1 = "hi", s2 = "world";
 cout << compare (s1,s2) << endl;
 return 0;
}
```

From **algorithm** abstraction to **data** abstraction



# Defining a class template

To illustrate class templates, we'll implement our own version of the standard library *queue* class.

Our queue must be able to hold objects of different types, so we will define it as a **class template**

Operations supported by queue (i.e., interface):

- *push* : to add an item to the back of the queue
- *pop* : to remove the item at the head of the queue
- *front* : to return a reference to the element at the head of the queue
- *empty*: to indicate whether there are any elements in the queue

# Defining the *Queue* interface

```
template <class Type> class Queue {
public:
 Queue (); //default constructor
 Type &front (); //return element from head of Queue
 const Type &front() const;
 void push (const Type &); //add element to back of Queue
 void pop (); //remove element from head of Queue
 bool empty () const; //true if no elements in the queue
private:
 //...
};
```

# Using class template

```
Queue<int> qi; // Queue that holds ints
Queue<vector<double>>qc; // Queue that holds vectors of doubles
Queue<string> qs; // Queue that holds strings
```

The compiler uses the arguments to instantiate a type-specific version of the class. In the example above, the compiler will instantiate three classes

And now, to some technical details about  
using templates

# Template declarations

```
// declares compare but does not define it
template <class T> int compare (const T&, const T&);
```

```
//actual definition of the template
template <class Type> Type calc (const Type& a, const Type& b)
{/* ... */}
```

```
//error: must precede U by either typename or class
template <typename T, U> T calc (const T&, const U&);
```

# Template type parameters

- Type parameters consist of the keyword *class* or the keyword *typename* followed by an identifier
  - in a template parameter list, these keywords have the same meaning
    - Older programs use *class*; now *typename* is part of standard C++

```
template <class T> T calc (const T& a, const T& b)
{
 T tmp = a;
 return tmp;
}
```

# Designating types inside the template definition

```
template <class Parm, class U>
Parm fcn (Parm* array, U value)
{
 Parm::size_type *p; //If Parm::size_type is a type, then a declaration
 // If Parm::size_type is an object, then multiplication
 // here the compiler will assume multiplication
}
```

↓  
solution

```
template <class Parm, class U>
Parm fcn(Parm* array, U value)
{
 typename Parm::size_type *p;
}
```

# Instantiation

- The process of generating a type-specific instance of a template is known as **instantiation**
- A class template is instantiated when we refer to the actual class type, and a function template is instantiated when we call it
  - Each instantiation of a class template constitutes an independent class type. The below *Queue* instantiation of type *int* has no relationship to nor any special access to the members of any other *Queue* type

```
//class instantiation
```

```
Queue<int> qi;
```

```
Queue<string> qs;
```

```
Queue qs; //error, Queue is not a type, Queue<string> and Queue<int> are
```



# Instantiation (2)

```
int main () {
 compare (1, 0); //ok: binds template parameter to int
 compare (3.14,2.7); //ok: binds template parameter to double
}
```

The program instantiates two versions of *compare*: one where T is replaced by *int* and the other where it is replaced by *double*

# Function-Template explicit arguments

```
template <class T1, class T2, class T3>
T1 sum (T2,T3);
```

```
//ok: T1 explicitly specified; T2 and T3 inferred from argument types
int i; short s;
int val3 = sum<long>(i,s)
```

```
template <class T1, class T2, class T3>
T3 alternative_sum (T2,T1);
```

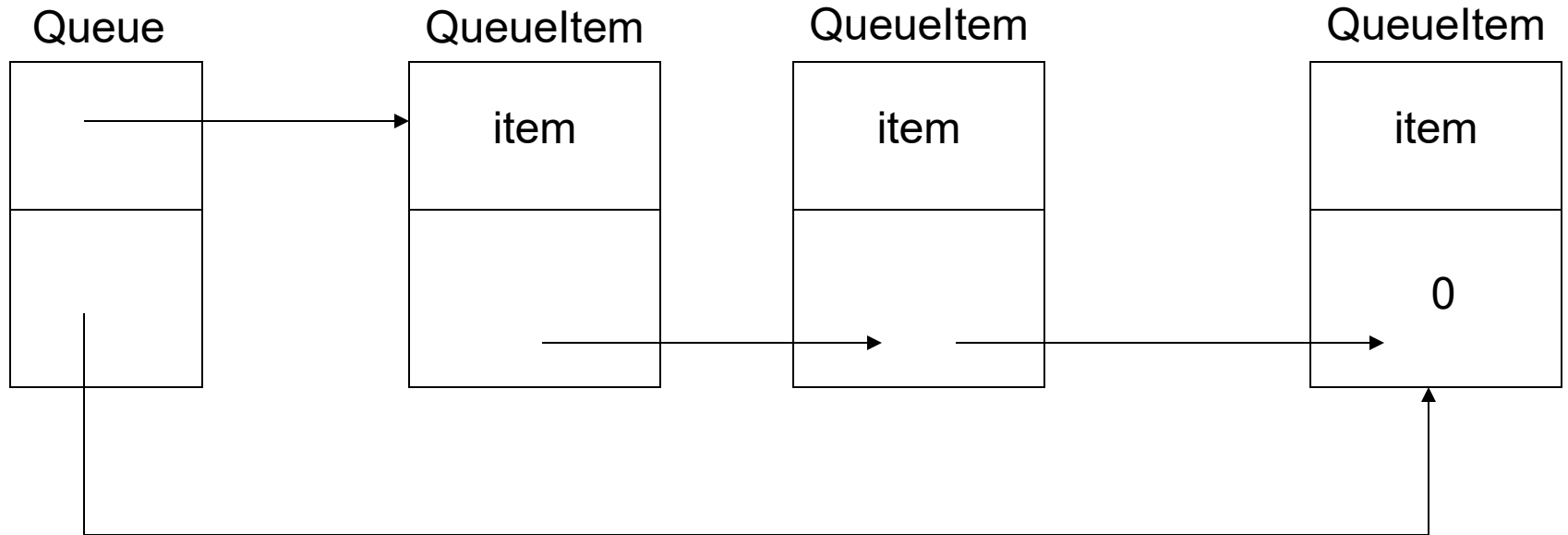
```
//ok: all three parameters explicitly specified
int i; short s;
int val2 = alternative_sum<int, int, short> (i,s);
```

# Class template members

## Re-considering the Queue example

- Class *QueueItem* will represent a node in the *Queue*'s linked list. This class has two data members: *item* and *next*:
  - *item* holds the value of the element in the *Queue*; its type varies with each instance of *Queue*
  - *next* is a pointer to the next *QueueItem* object
- Class *Queue* will provide the interface functions previously described
  - the *Queue* class will also have two data members: *head* and *tail*. These members are pointers to *QueueItem*
- Our *Queue* class will copy the values it's given

# Queue implementation



# QueueItem class

```
template <class Type> class QueueItem {
 //this is a private class: no public section !
 //class Queue will be a friend of class QueueItem
private:
 QueueItem (const Type &t) : item(t), next(0) { }
 Type item; //value is stored in this element
 QueueItem *next; // pointer to next element in the Queue
};
```

Note: inside the scope of a class template, we may refer to the class using its unqualified name

# The *Queue* class

```
template <class Type> class Queue {
public:
 // empty Queue
 Queue (): head(0), tail (0) {}
 // copy control to manage pointers to QueueItems in the Queue
 Queue (const Queue &Q): head (0), tail (0) {copy_elems(Q);}
 Queue& operator=(const Queue&);
 ~Queue () {destroy ();}
 // return element from head of the Queue
 // unchecked operation: front on an empty Queue is undefined
 Type& front () {return head->item;}
 const Type &front () const {return head->item;}
 void push (const Type &); //add element to back of Queue
 void pop (); //remove element from head of queue
 bool empty () const { //true if no elements in the Queue
 return head == 0;
 }
}
```

# Queue (contd.)

**private:**

```
QueueItem<Type> *head; //pointer to first element in Queue
QueueItem<Type> *tail; //pointer to last element in Queue
//utility functions used by copy constructor, assignment, and destructor
void destroy (); // delete all the elements
void copy_elems (const Queue&); //copy elements from parameter
};
```

# Class template member functions

- Class template member function defined outside the class (**but in the .h file !!!**)

```
template <class Type> void Queue<Type>::destroy () {
 while (!empty ())
 pop ();
}
```

```
template <class Type> void Queue<Type>::pop () {
 // pop is unchecked :Popping off an empty Queue is undefined
 QueueItem<Type>* p = head; //keep pointer to head so we can delete it
 head = head -> next; //head now points to the next element
 delete p; //delete old head element
}
```



# Implementation of Queue – contd.

```
template <class Type> void Queue<Type>::push (const Type &val)
{
 //allocate a new QueueItem object
 QueueItem<Type> *pt = new QueueItem<Type>(val);
 //put item into existing Queue
 if (empty ())
 head = tail = pt; //the queue now has only one element
 else {
 tail -> next = pt; //add new element to the end of queue
 tail = pt;
 }
}
```

# The copy function

```
template <class Type> void Queue<Type>::copy_elems(const Queue &orig)
{
 //copy elements from orig into this Queue
 // loop stops when pt ==0, which happens when we reach orig.tail
 for (QueueItem<Type> *pt = orig.head; pt !=0; pt=pt->next)
 push(pt->item); //copy the element
}
```

# Instantiation of class-template member functions

- When we call the *push* member of an object of type *Queue<int>* the *push* function that is instantiated is  
void Queue<int>::push(const int &val)
- Member functions of a class template are instantiated only for functions that are used by the program

note: the compiler doesn't perform template-argument deduction in this case, therefore conversions are allowed

```
Queue<int> qi;
short s = 42;
int i = 42;
qi.push(s); // instantiates Queue<int>::push(const int&)
qi.push(i); // instantiates Queue<int>::push(const int&)
```

# Friend declarations

```
template <class Type> class Bar {
 //grant access to ordinary, nontemplate class and function
 friend class FooBar;
 friend void fcn ();
};
```

```
template<class Type> class Bar {
 // generate access to Foo or temp1_fcn1 parameterized by any type
 template <class T> friend class Foo;
 template <class T> friend void temp1_fcn1(const T&);
 // ...
};
```

```
template <class T> class Foo2;
template <class T> void temp1_fcn2 (const T&);
template <class Type> class Bar {
 // grants access to a single specific instance parameterized by char*
 friend class Foo2<char*>;
 friend void tmp1_fcn2<char*> (char* const &);
};
```

# Making Queue a friend of QueueItem

```
template <class Type> class Queue;
template <class Type> class QueueItem {
 friend class Queue<Type>;
 // ...
};
```

# Static members of class templates

```
template <class T> class Foo {
public:
 static size_t count () {return ctr;}
private:
 static size_t ctr;
};
```

```
template <class T>
size_t Foo<T>::ctr = 0;
```

```
Foo<int> fi, fi2;
size_t ct = Foo<int>::count (); // instantiates Foo<int>::count
ct = fi.count (); //ok: uses Foo<int>::count
ct = fi2.count (); //ok: uses Foo<int>::count
ct = Foo::count (); // error: which template instantiation ?
```

# Generic Algorithms

# Generic Algorithms

- The library containers surprisingly define a small set of containers
- Rather than adding lots of functionality to the containers, the library provides a set of ***algorithms***, most of which are independent of any particular type of container
- These algorithms are ***generic***: They operate on different types of containers and on elements of various types
  - Library containers
  - User defined containers
  - Built-in array types
  - Other types of sequences



## Example: finding an element in a sequence

```
#include<algorithm>
//Use find to search an array
int ia[6] = {2,6,4,5,7,1};
int search_value = 7;
int *result = find (ia, ia+6, search_value);
cout << "The value" << search_value <<
 (result == ia+6 ? "is not present" : "is present") << endl;
```

## Example: finding an element in a vector

```
#include<algorithm>
//Use find to search a vector
vector<int> vec;
int search_value = 7;
vector<int>::const_iterator result = find(vec.begin(),vec.end(),search_value);
cout << "The value" << search_value <<
 (result == vec.end() ? "is not present" : "is present")
 << endl;
```

# Example: finding an element in a list

```
#include<algorithm>
//Use find to search a list
list<int> ls;
int search_value = 7;
vector<int>::const_iterator result = find(ls.begin(), ls.end(),search_value);
cout << "The value" << search_value <<
 (result == ls.end() ? "is not present" : "is present")
 << endl;
```

# How the algorithms work

- Each generic algorithm is implemented (as a template function) independently of the container type
- The algorithms are also largely, but not completely, independent of the types of the elements that the container holds
- To see how the algorithms work, let us look a bit more closely at *find*. Conceptually, the steps that *find* must take are
  1. Examine each element in turn
  2. If the element is equal to the one we want, return an iterator that refers to this element
  3. Otherwise, examine the next element, repeating step 2 until either the value is found or all the elements have been examined
  4. If we have reached the end of the collection and we haven't found the value, return a value that indicates that the value wasn't found

**Note: Nothing in our algorithm depends on the container type !**

# Continued

- Implicitly, the *find* algorithm does have one dependency on the element type: We must be able to compare elements for equality
- More specifically, the requirements of the algorithm are:
  - We need to be able to compare each element to the value we want
  - We need a way to traverse the collection
  - We need to know when we have reached the end of the collection
  - We need a type that can refer both to an element's position within the container or that can indicate the element wasn't found

# Iterators bind algorithms to containers



- We need a way to traverse the collection
- We need to know when we have reached the end of the collection
- We need a type that can refer both to an element's position within the container and that can indicate the element wasn't found

The algorithms achieve type independence by *never* using container operations. All access to the traversal of the elements is done through the ("ordinary" or "insertion") iterators

# A first look at the algorithms

- The library provides more than 100 algorithms. Like the containers, the algorithms have a consistent architecture. We will discuss the *unified principles* used by the library algorithms
- With only a few exceptions, all the algorithms operate over a range of elements (called "input range"), captured by the first two parameters of the algorithm function
- The algorithms differ in how they use the elements in that range
  - Read elements
  - Write elements
  - Rearrange elements

# Read-only algorithms

- A number of algorithms read, but never write to, the elements in their input range. *find* is one such algorithm. Another simple algorithm is *accumulate*: Assuming *vec* is a vector of int values:

```
#include <numeric>
int sum = accumulate (vec.begin(), vec.end (), 42);
```

set sum equal to the sum of elements in vec plus 42

- There are two primary implications of type independence:
  - We must pass an initial starting value, because otherwise *accumulate* cannot know what starting value to use
  - The types of the arguments must match, and the element type should support addition

```
string sum = accumulate (vec.begin(), vec.end (), string("aa"));
```

# A quick glance

```
template<typename _InputIterator, typename _Tp>
_Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init)
{
 for (; __first != __last; ++__first)
 __init = __init + *__first;
 return __init;
}
```



# Algorithms that write container elements

```
fill (vec.begin (), vec.end (), 0);
fill_n (vec.begin(), 10, 0); // be careful
```

# Introducing back\_inserter

#include<iterator>

- An *insert iterator* is an iterator that *adds* elements to the underlying container:
  - When we assign through an ordinary iterator, we assign to the elements to which the iterator refers
  - When we assign through an ordinary iterator, a new element equal to the iterator's argument value is added to the container.
- The *back\_inserter* function is an *iterator adaptor*: it takes an object (container) and generates a new object (*back\_inserter*) that adapts the behavior of its argument

```
vector<int> vec; //empty vector
```

```
fill_n (back_inserter(vec), 10 , 0); //appends 10 elements to vec
```

```
//Assignment through this iterator calls push_back to add an element with
// the given value to the container
```

# Iterators connect algorithms and containers

```
vector<int> vec ;
```

```
fill_n (back_inserter(vec), 2, 7); // vec now contains 7 7
```

```
fill_n (vec.begin(), 2, 3); //vec now contains 3 3
```

```
fill_n (back_inserter(vec), 2, 5); // vec now contains 3 3 5 5
```

```
fill_n (vec.rbegin (), 2 , 8); // vec now contains 3 3 8 8
```

# Insert iterators

- The *back\_inserter* function is an example of an *inserter*.
  - *back\_inserter* creates an iterator that uses *push\_back*
  - *front\_inserter* creates an iterator that uses *push\_front*
  - *inserter* creates an iterator that uses *insert*. In addition to a container, *inserter* takes a second argument: an iterator that indicates the position ahead of which insertion should begin

```
list<int> ilst, ilst2, ilst3; //empty lists
//after this loop ilst contains 3 2 1 0
for (list<int>::size_type i =0; i !=4; ++i)
 ilst.push_front(i);
//after copy ilst2 contains: 0 1 2 3
copy(ilst.begin(), ilst.end (), front_inserter(ilst2));
//after copy ilst3 contains: 3 2 1 0
copy(ilst.begin(), ilst.end (), inserter(ilst3,ilst3.begin()));
```

# Algorithms that reorder container elements

- Suppose we want to analyze the words in a set of stories. For example, we might want to know how many words contain six or more characters
  - We want to count each word only once
  - We would like to print the words in size order
  - We want the words in alphabetic order within a given size
- Assume that we have read our input and stored the text of each book in a *vector* of *strings* named *words*
- To solve the problem we need to
  - Eliminate duplicate copies of each word
  - Order the words based on size, and then on alphabet
  - Count the words whose size is 6 or greater

# Eliminating duplicates (easy)

```
// sort words alphabetically so we can find duplicates
sort(words.begin (), words.end ());
```

```
//eliminate duplicate words:
```

```
//unique re-orders words so that each word appears once
```

```
// in the front portion of words and returns an iterator
```

```
// one past the unique range
```

```
vector<string>::iterator end_unique = unique(words.begin(), words.end ());
words.erase (end_unique, words.end());
```

**Algorithm never directly change the size of a container. If we want to add or remove elements, we must use a container operation**

# Defining utility functions

```
// comparison function to be used to sort by word length
bool isShorter (const string &s1, const string &s2) {
 return s1.size () < s2.size ();
}
```

```
//determine whether a length of a word is 6 or more
const string::size_type L = 6;
bool GT6 (const string &s) {
 return s.size () >= L;
}
```

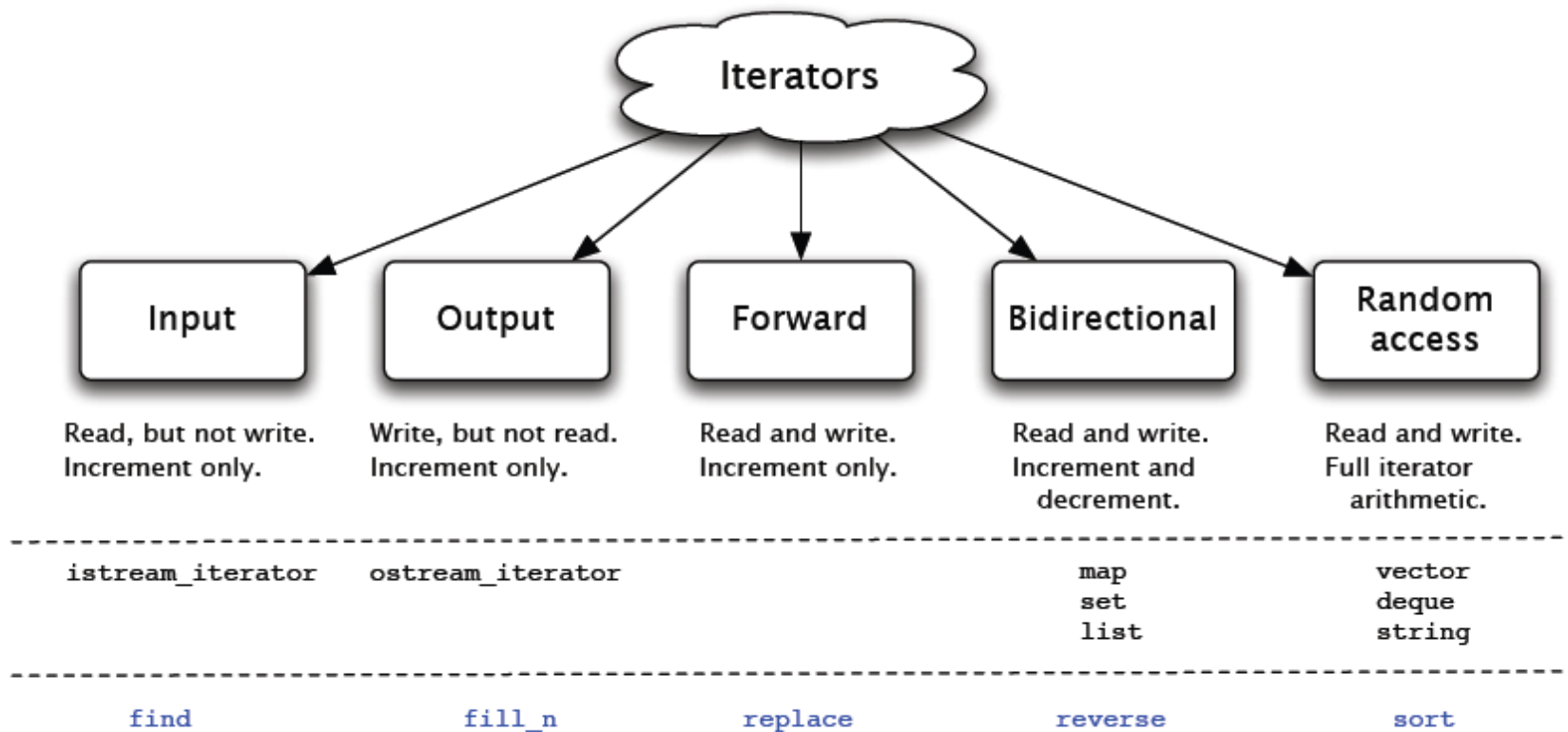
# Sorting algorithm

```
// sort words alphabetically so we can find duplicates
sort (words.begin (), words.end ());
vector<string>::iterator end_unique = unique (words.begin(), words.end ());
words.erase (end_unique, words.end ());
```

```
//sort words by size, but
// maintain alphabetic order for words of the same size
stable_sort (words.begin (), words.end (), isShorter);
vector<string>::size_type wc = count_if (words.begin(),words.end(),GT6);
```



# The five iterator categories



# Exceptions

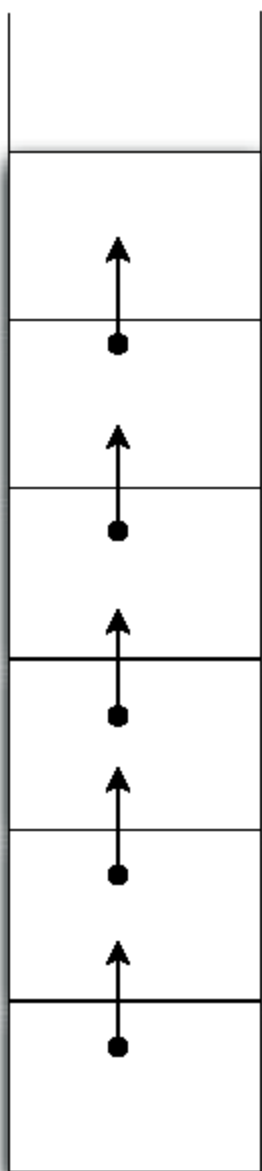
# Errors

- Errors happen a lot in running programs
- ***Excpetions*** are run-time anomalies (running out of memory, encountering unexpected input); thus, they exist outside the "normal" functioning of the program
- Long-lived interactive programs (Web-based applications, production line drivers, etc.) can devote as much as 90% of their code to **error detection** and **error handling**

# Traditional error handling

Suppose an error is encountered in a certain function call. What should this function do ?

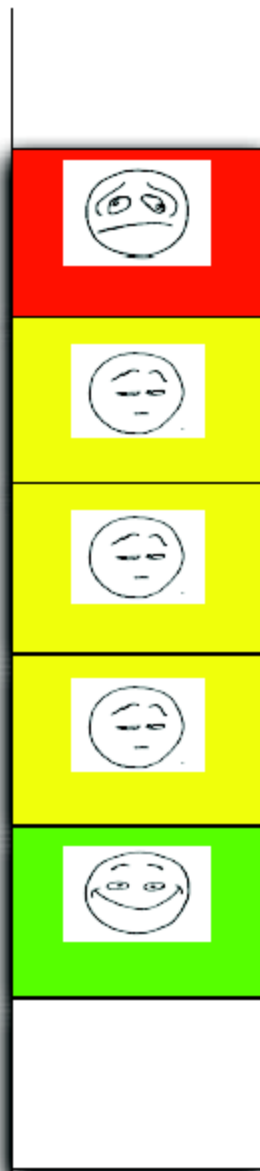
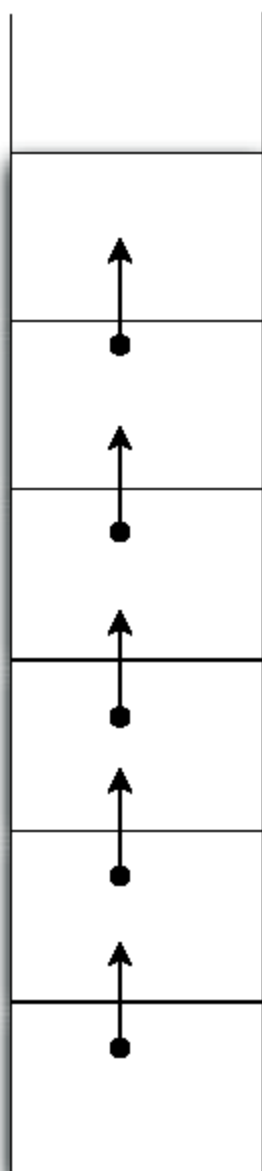
- Terminate the program
- Return a value and leave the program in an illegal state
- Return a value representing this error
  - need to check return value for error at every function call while errors are typically rare
  - the immediate function caller may not be responsible for this error but must be aware of it
- Call a function that "knows" what to do with this error



Error of type X!!!

*I have no idea  
what to do!*

I know how to handle  
errors of type X  
that may encounter  
in this course of action!



Error of type X!!!  
*I have no idea  
what to do!*

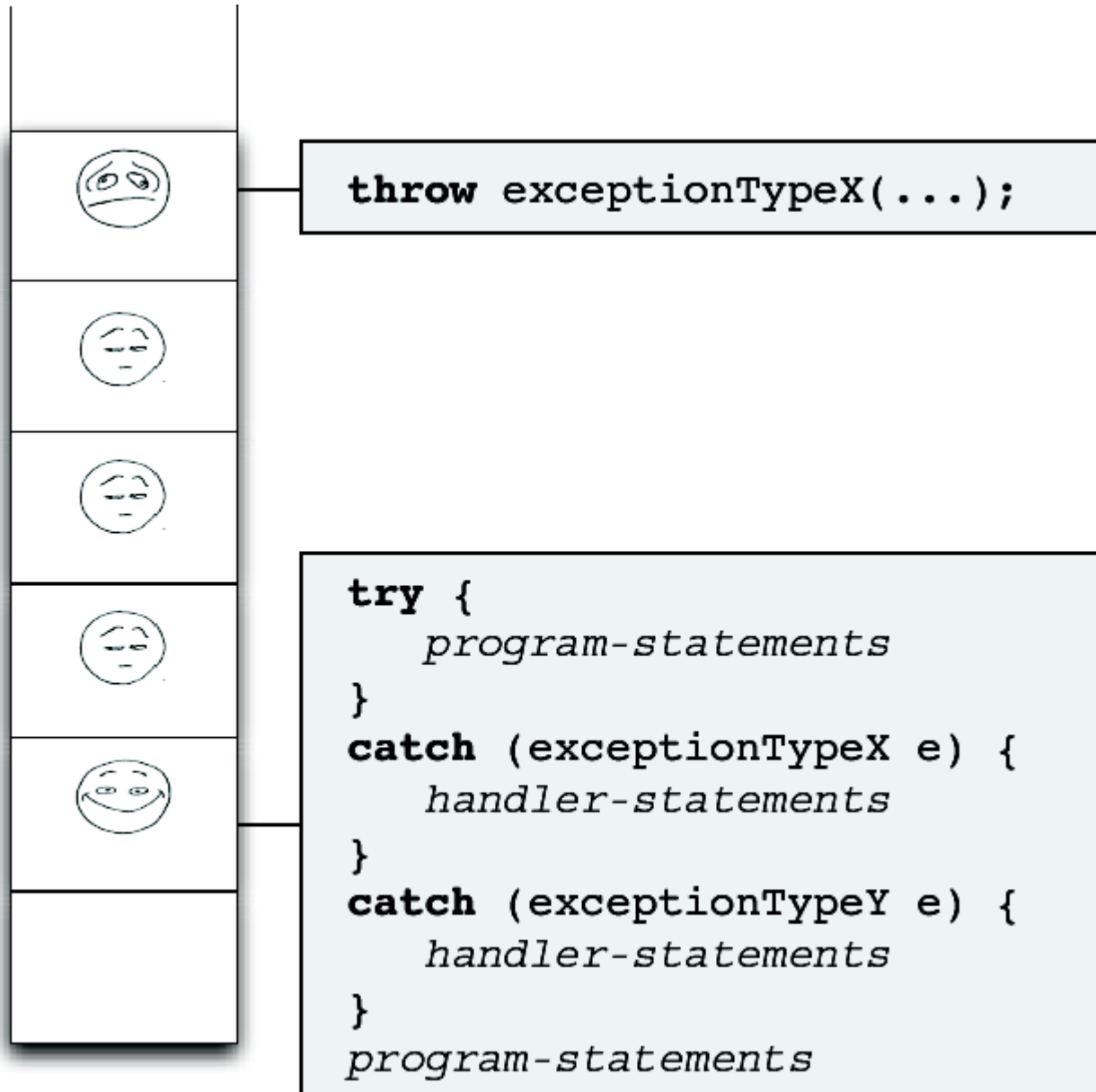
What is error of type X?

What is error?

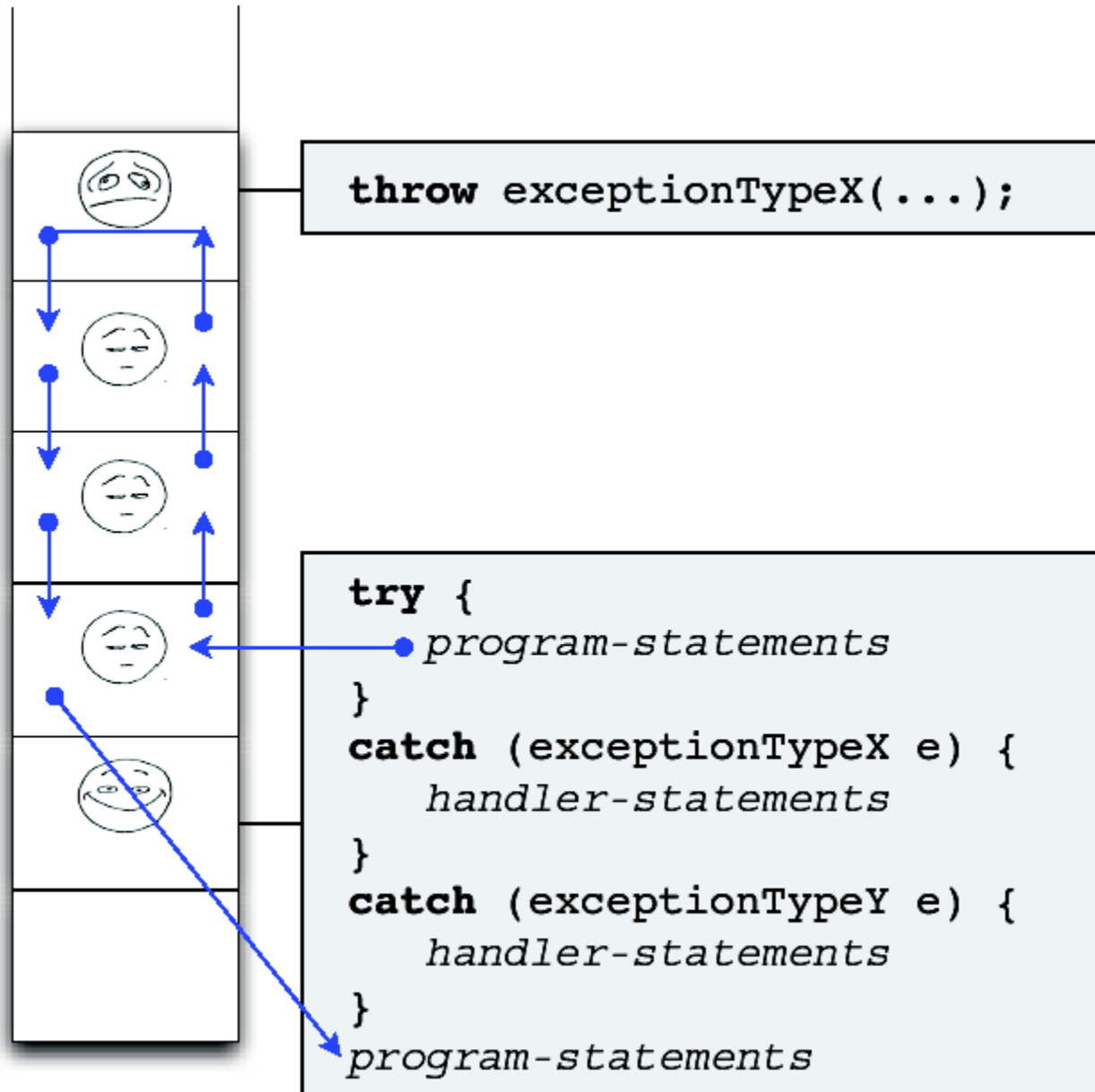
What is error of type X?

I know how to handle  
errors of type X  
that may encounter  
in this course of action!

# Exceptions in C++



# Exceptions in C++





# Exception handling

- Allows independently developed parts of a program to communicate about and handle problems that arise during execution of the program
  - e.g., one part of the program can detect a problem that another part of the program cannot resolve
- Exceptions let us separate problem detection from problem resolution
  - the part of the program that detects a problem needn't know how to deal with it

# Example

```
class Sales_item {
public:
 void add (const Sales_item &other) {
 if (!same_isbn(other.isbn)) {
 throw runtime_error ("Data must refer to same ISBN");
 }
 /* all the rest of the stuff */
 }
}
```

```
int main {
 Sales_item item1, item2;
 cin >> item1 >> item2;
 try {
 item1.add(item2);
 }
 catch (const runtime_error &e) {
 cerr << e.what () << "Try again" << endl;
 }
}
```

# Another example

```
void func1 () {
 try {
 func2 ();
 }
 catch (const runtime_error &e) {
 cerr << e.what () << "An error has occurred" << endl;
 }
 cout << "Continuing execution" << endl;
}
```

```
void func2 () {
 cout << "print something" << endl;
 func3 ();
 cout << "print something else" << endl;
}
```

```
int func3 () {
 throw runtime_error ("Error in func3");
 return 0;
}
```

# Throwing an exception of class type

- An exception is ***raised*** by ***throwing*** an object
  - the type of the object determines which handler will be invoked
  - the selected handler is the one nearest in the call chain that matches the type of the object
- Exceptions are thrown and caught in ways similar to how arguments are passed to functions
- An exception can be an object of any type that can be passed to a **nonreference** parameter
  - i.e., you must be able to copy this object
- There are no exceptions of array type
  - if we throw an array we actually throw a pointer to the first element of the array

# Contd.

- When a *throw* is executed, the statements following the *throw* are not executed
  - Control is transferred from the *throw* to the matching *catch*
  - Functions along the call chain are prematurely exited
  - Storage that is local to a block that throws an exception isn't around when the exception is handled -> the object that is thrown is not stored locally
- The *throw* statement initializes an *exception* object
  - Initialized as a copy of the object that is thrown
  - Passes to the corresponding *catch* and destroyed after the exception is completely handled

# Exception objects and inheritance

When an exception is thrown, the static, compile-time type of the thrown object determines the type of the exception object

# Stack unwinding

- When an exception is thrown
  - execution of current function is suspended; a search begins for a matching *catch* clause
- Is the *throw* located inside a *try* block ?
  - If so, the *catch* clauses associated with that *try* are examined
    - If a matching one is found the exception is handled
    - If no matching *catch* is found, the calling function is exited and the search continues in the function that called this function
- When a *catch* completes, execution continues at the point immediately after the last *catch* clause associated with that *try* block

# Stack unwinding

During stack unwinding, the memory used by the local objects is freed and destructors for local objects of class type are run

- Destructors should never throw exceptions
- A constructor can throw an exception and then the members that have been constructed will be destructed

**Uncaught exceptions terminate the program !**



# Catching an exception

```
try {
 //statements
}
catch (const A& a) {
}
catch (B b) {
}
```

# Catching an exception

- The *catch* that is selected is the first *catch* found that can handle the exception
  - Therefore, in a list of *catch* clauses, the most specialized *catch* must appear first
- The types of the exception and the *catch* specifier must match exactly with only a few possible differences
  - Conversion from nonconst to const
  - Conversions from derived type to base type are allowed
  - An array is converted to a pointer to the type of the array
  - No other conversions are allowed
- The exception-specifier type might be a reference
  - recall that the exception object itself is a copy of the object that was thrown
  - If the exception-specifier is not a reference then the exception object is copied into the *catch* parameter (like function parameter passing)
- **Good practice:** a *catch* clause that handles an exception of a type related by inheritance ought to define its parameter as a reference

# *Catch* clause - rule

Multiple *catch* clauses with types related by inheritance must be ordered from the most derived type to the least derived

# Rethrow

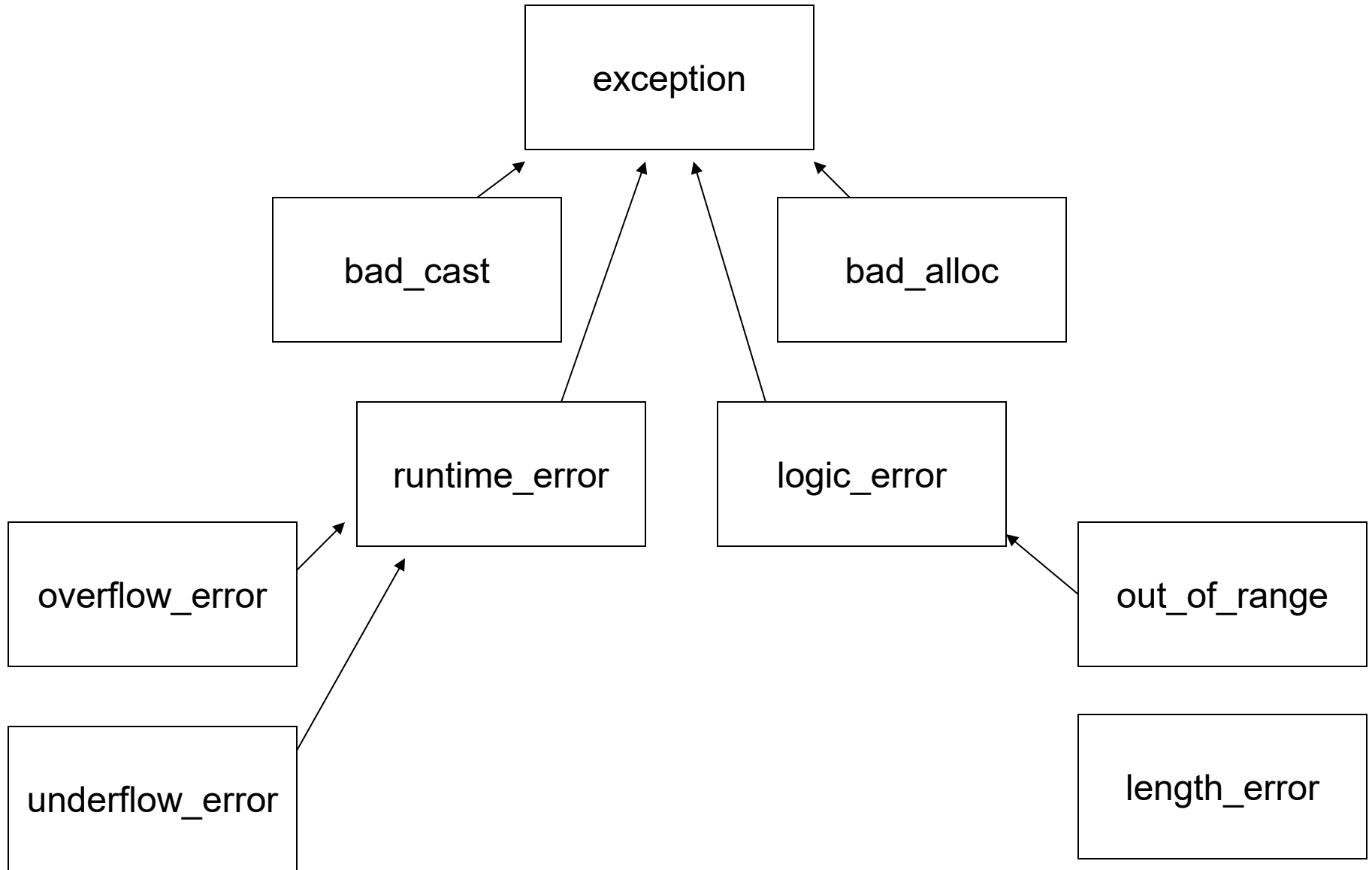
```
catch (my_error &eObj) { // specifier is a reference type
 eObj.status = severeErr; //modifies the exception object
 throw; //this is a re-throw, the status member of the
 // exception object is severErr
}
catch (otherErr eObj) {
 eObj.status = badErr; //modifies local copy only
 throw; // the status member of the exception rethrown
 // is unchanged
}
```

# The catch-all handler

```
//matches any excpetion that might be thrown
catch (...) {
 //place your code here
}
```

Note: if *catch(...)* is used in combination with other *catch* clauses, it must be last; otherwise, any *catch* clause that followed it could never be matched

# Standard exception class hierarchy



# Using programmer-defined exception types

```
#include<stdexcept>
using std::runtime_error;
using std::logic_error;
using std::string;

class out_of_stock : public runtime_error {
public:
 explicit out_of_stock (const string &s): runtime_error(s) { }
}

class isbn_mismatch: public logic_error {
public:
 explicit isbn_mismatch (const string &s) : logic_error (s) { }
 isbn_mismatch (const string &s, const string ¶m):
 logic_error(s), _param(param) { }
 virtual ~isbn_mismatch () throw;
private:
 const string _param;
}
```

```
class Sales_item {
public:
 void add (const Sales_item &other) {
 if (!same_isbn(other.isbn)) {
 throw isbn_mismatch ("Data must refer to same ISBN");
 }
 /* all the rest of the stuff */
 }
}
```

```
Sales_item item1, item2, sum;
while (cin >> item1 >> item2) {
 try {
 sum.add (item1);
 sum.add (item2);
 }
 catch (const isbn_mismatch &e) {
 cerr << e.what () << endl;
 }
}
```



# Exception specification

- Looking at an ordinary function declaration, it's not possible to determine what exceptions the function might throw
  - it can be useful to know whether and which exceptions a function might throw in order to write appropriate catch clauses
- An ***exception specification*** specifies that if the function throws an exception, the exception it throws will be one of the exceptions included in the specification, or it will be a type derived from one of the listed exceptions

# Defining an exception specification

```
void func (int) throw (runtime_error);
//if func throws an exception, it will be a runtime_error or an exception
// of type derived from runtime_error
```

```
void myFunc () throw ();
//myFunc doesn't throw any exceptions
```

If a function declaration doesn't specify an exception specification, the function can throw exceptions of any type

The compiler cannot and doesn't attempt to verify exception specifications at compile time, therefore the practical utility of exception specifications is often limited

```
void f () throw (){
 throw exception (); //violates exception specification
 //the compiler will not complain !!!
}
```