Lecture Material

- Derived classes
- Text-mode windowing system
- Resource management
  - *Resource Acquisition Is Initialization* technique
  - *unique_ptr* and *shared_ptr* templates

Derived Classes

- A concept does not exist in isolation.
- When trying to describe the concept of "car" leads us to introduce the concepts of:
  - wheels
  - engines
  - drivers
  - pedestrians
  - trucks
  - ambulances
  - road
  - gasoline
  - speeding tickets
- To represent concepts we use classes. How do we represent relationships between concepts?
  - to express the hierarchical features, i.e. commonality between classes, we use derived classes

Derived Classes

- Consider building a program dealing with people employed by a firm.

```
struct Employee {
 string first_name,family_name;
 char middle_initial;
 Date hiring_date;
 short department;
 // ...
};
```

```
struct Manager {
 Employee emp;
      // manager's employee record
 set<Employee*> group; // people managed
 short level;
 // ...
};
```

- A manager is also an employee; the *Employee* data is stored in the *emp* member of a *Manager* object. This may be obvious to a human reader, but there is nothing that tells the compiler and other tools that *Manager* is also an *Employee*. The correct approach is to explicitly state that a *Manager* is an *Employee*, with a few pieces of information added:

```
struct Manager : public Employee {
 set<Employee*> group;
 short level;
// ...
};
```

Derived Classes

- The *Manager* is derived from *Employee*, and conversely, *Employee* is a base class for *Manager*.
- Derivation is often represented graphically by a pointer from the derived class to its base class indicating that the derived class refers to its base. This relationship is also called inheritance.

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

Derived Classes

- A *Manager* is (also) an *Employee*, so a *Manager** can be used as a *Employee**.

```
void f(Manager m1,Employee e1)
{
 list<Employee*> elist;
 elist.push_front(&m1) ;
 elist.push_front(&e1) ;
 // ...
}
```

- The opposite conversion, from *Employee** to *Manager**, must be explicit.

```
void g(Manager mm, Employee ee)
{
 Employee* pe= &mm;        // ok: every Manager is an Employee
 Manager* pm= &ee;         // error: not every Employee is a Manager
 pm->level = 2;            // disaster: ee doesn't have a 'level'
 pm = static_cast<Manager*>(pe) ; // brute force: works because pe points
                                  // to the Manager mm
 pm->level = 2; // fine: pm points to the Manager mm that has a 'level'
}
```

Member Functions

- Simple data structures, such as *Employee* and *Manager*, are really not that interesting and often not particularly useful. We need to give the information as a proper type that provides a suitable set of operations that present the concept, and we need to do this without tying us to the details of a particular representation.

```
class Employee {
    string first_name, family_name;
    char middle_initial;
    // ...
 public:
    void print() const;
    string full_name() const
    { return first_name+ ´ ´ +middle_initial+ ´ ´ + family_name; }
    // ...
};

class Manager : public Employee {
    // ...
 public:
    void print() const;
    // ...
};
```

Member Functions

- A member of a derived class can use the public and protected members of its base class as if they were declared in the derived class itself.

```
void Manager::print() const
{
 cout << "name is" << full_name() << ´\n´;
 // ...
}
```

- However, a derived class cannot use a base class' private names:

```
void Manager::print() const
{
 cout << " name is" << family_name << ´\n´; // error!
 // ...
}
```

Member Functions

- The cleanest solution is for the derived class to use only the public members of its base class.

```
void Manager::print() const
{
 Employee::print() ;      // print Employee information
 cout << level;  // print Manager-specific information
 // ...
}
```

- Note that :: must be used because *print()* has been redefined in *Manager*. Without it, we would find the program involved in an unexpected sequence of recursive calls.

```
void Manager::print() const
{
 print() ;        // oops!
 // print Manager-specific information
}
```

Constructors and Destructors

- Some derived classes need constructors. If a base class has constructors, then a constructor must be invoked. Default constructors can be invoked implicitly. However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called.

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
 public:
    Employee(const string& n, int d) ;
 // ...
};
class Manager : public Employee {
    set<Employee*> group; // people managed
    short level;
    // ...
 public:
    Manager(const string& n, int d, int lvl) ;
    // ...
};
```

Constructors and Destructors

- Arguments for the base class' constructor are specified in the definition of a derived class' constructor.

```
Employee::Employee(const string& n, int d) : family_name(n) , department(d)
 // initialize members
{
 // ...
}
Manager::Manager(const string& n, int d, int lvl)
 :  Employee(n,d) ,          // initialize base
    level(lvl)               // initialize members
{
 // ...
}
```

- A derived class constructor can specify initializers for its own members and immediate bases only; it cannot directly initialize members of a base.

```
Manager::Manager(const string& n, int d, int lvl):
 family_name(n) , // error: family_name not declared in Manager
 department(d) , // error: department not declared in Manager
 level(lvl)
{
 // ...
}
```

- Class objects are constructed from the bottom up: first the base, then the members, and then the derived class itself. They are destroyed in the opposite order: first the derived class itself, then the members, and then the base. Members and bases are constructed in order of declaration in the class and destroyed in the reverse order.

Constructors and Destructors

- Copying of class objects is defined by the copy constructor and assignments

```
class Employee {
 // ...
 Employee& operator=(const Employee&) ;
 Employee(const Employee&) ;
};
void f(const Manager& m)
{
 Employee e = m; // construct e from Employee part of m
 e = m;          // assign Employee part of m to e
}
```

- Because the *Employee* copy functions do not know anything about *Managers*, only the *Employee* part of a *Manager* is copied. This is commonly referred to as slicing and can be a source of surprises and errors.
- If an assignment operator is not defined explicitly, it will be generated automatically by a compiler. It means, that the assignment operator cannot be inherited.
- In default-generated assignement operator first the base class is assigned (using its assignment operator), then the members, fiels by field.

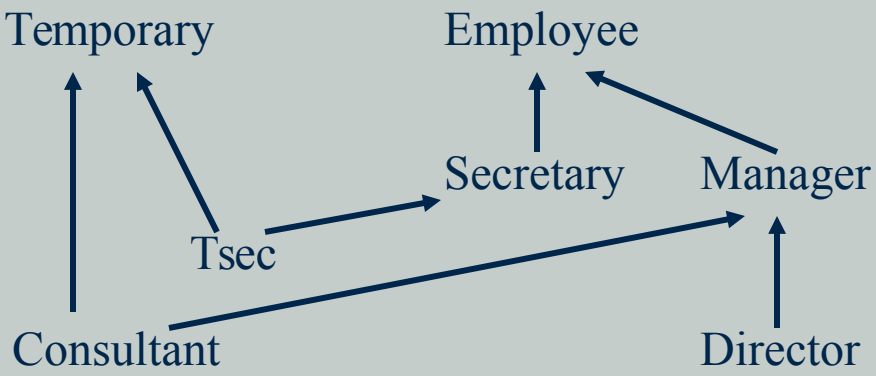
Class Hierarchies

- A derived class can itself be a base class.

```
class Employee{ /* ... */ };
class Manager : public Employee{ /* ... */ };
class Director : public Manager{ /* ... */ };
```

- Such a set of related classes is traditionally called a class hierarchy. Such a hierarchy is most often a tree, but it can also be a more general directed acyclic graph structure.

```
class Temporary{ /* ... */ };
class Secretary : public Employee{ /* ... */ };
class Tsec : public Temporary, public Secretary{ /* ... */ };
class Consultant : public Temporary, public Manager{ /* ... */ };
```



Type Fields

- Pointers to base classes are commonly used in the design of container classes such as set, vector, and list.

```
void print_list(const list<Employee*>& elist)
{
 for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
   (*p)->print(); //oops! Prints only the Employee part
}
```

- The list can contain pointers to employees and managers. The preceding function will print-out only the information about the employee part. The *level* field will not be printed for managers.
- What can we do to make function work according to our expectations?
- The solution to the problem requires an answer to the question: what is the type of the object pointed by pointer?

Type Fields

- Given a pointer of type *base**, to which derived type does the object pointed to really belong? There are four fundamental solutions to the problem:
  - Ensure that only objects of a single type are pointed to
  - Place a type field in the base class for the functions to inspect
  - Use *dynamic_cast*
  - Use virtual functions
- Pointers to base classes are commonly used in the design of container classes such as set, vector, and list. In this case, solution 1 yields homogeneous lists, that is, lists of objects of the same type. Solutions 2, 3, and 4 can be used to build heterogeneous lists, that is, lists of (pointers to) objects of several different types. Solution 3 is a language-supported variant of solution 2. Solution 4 is a special typesafe variation of solution 2. Combinations of solutions 1 and 4 are particularly interesting and powerful; in almost all situations, they yield cleaner code than do solutions 2 and 3.

Type Fields

- Why the type field in a class should be avoided? Let us consider an example:

```
struct Employee {
 enum Empl_type {M,E };
 Empl_type type;
 Employee() : type(E) { }
 string first_name, family_name;
 char middle_initial;
 Date hiring_date;
 short department;
 // ...
};
struct Manager : public Employee {
 Manager() { type =M; }
 set<Employee*> group; // people managed
 short level;
 // ...
};
```

Type Fields

- Let us write a function that prints information about each *Employee*:

```
void print_employee(const Employee* e)
{
switch (e->type) {
 case Employee::E:
   cout << e->family_name << ´\t´ << e->department << ´\n´;
   // ...
   break;
 case Employee::M:
   { cout << e->family_name << ´\t´ << e->department << ´\n´;
   // ...
   const Manager* p = static_cast<const Manager*>(e) ;
   cout << " level" << p->level << ´\n´;
   // ...
   break;
   }
 }
}
```

- We can use it to print a list of *Employees*:

```
void print_list(const list<Employee*>& elist)
{
 for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
   print_employee(*p) ;
}
```

Type Fields

- This works fine, especially in a small program maintained by a single person.
- It depends on the programmer manipulating types in a way that cannot be checked by the compiler.
- Finding all tests on the type field buried in a large function that handles many derived classes can be difficult.
- Any addition of a new kind of *Employee* involves a change to all the key functions in the system – the ones containing the tests on the type field.

Virtual Functions

- Virtual functions overcome the problems with the type field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class.
- The compiler will guarantee the correct correspondence between objects and the functions applied to them.

```
class Employee {
   string first_name, family_name;
   short department;
    // ...
 public:
   Employee(const string& name, int dept) ;
   virtual void print() const;
   // ...
};
```

- To allow a virtual function declaration to act as an interface to functions defined in derived classes, the argument types specified for a function in a derived class cannot differ from the argument types declared in the base, and only very slight changes are allowed for the return type.

Virtual Functions

- A virtual function must be defined for the class in which it is first declared

```
void Employee::print() const
{
 cout << family_name << '\t' << department << '\n';
 // ...
}
```

- A virtual function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one. When deriving a class, simply provide an appropriate function, if it is needed.

```
class Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
 public:
    Manager(const string& name, int dept, int lvl) ;
    void print() const;
    // ...
};
void Manager::print() const
{
    Employee::print() ;
    cout << "\tlevel" << level << '\n';
    // ...
}
```

Virtual Functions

- The global function *print_employee()* is now unnecessary because the *print()* member functions have taken its place. A list of *Employees* can be printed like this:

```
void print_list(set<Employee*>& s)
{
 for (set<Employee*>::const_iterator p = s.begin() ; p!=s.end() ; ++p)
   (*p)->print() ;
}
```

or even

```
void print_list(set<Employee*>& s)
{
 for_each(s.begin() ,s.end() ,mem_fun(&Employee::print)) ;
}
```

- The following code

```
int main()
{
 Employee e("Brown",1234) ;
 Manager m("Smith",1234,2) ;
 set<Employee*> empl;
 empl.insert(&e) ;
 empl.insert(&m) ;
 print_list(empl) ;
}
```

will print out:

```
Smith 1234
   level 2
Brown 1234
```

Polymorphism

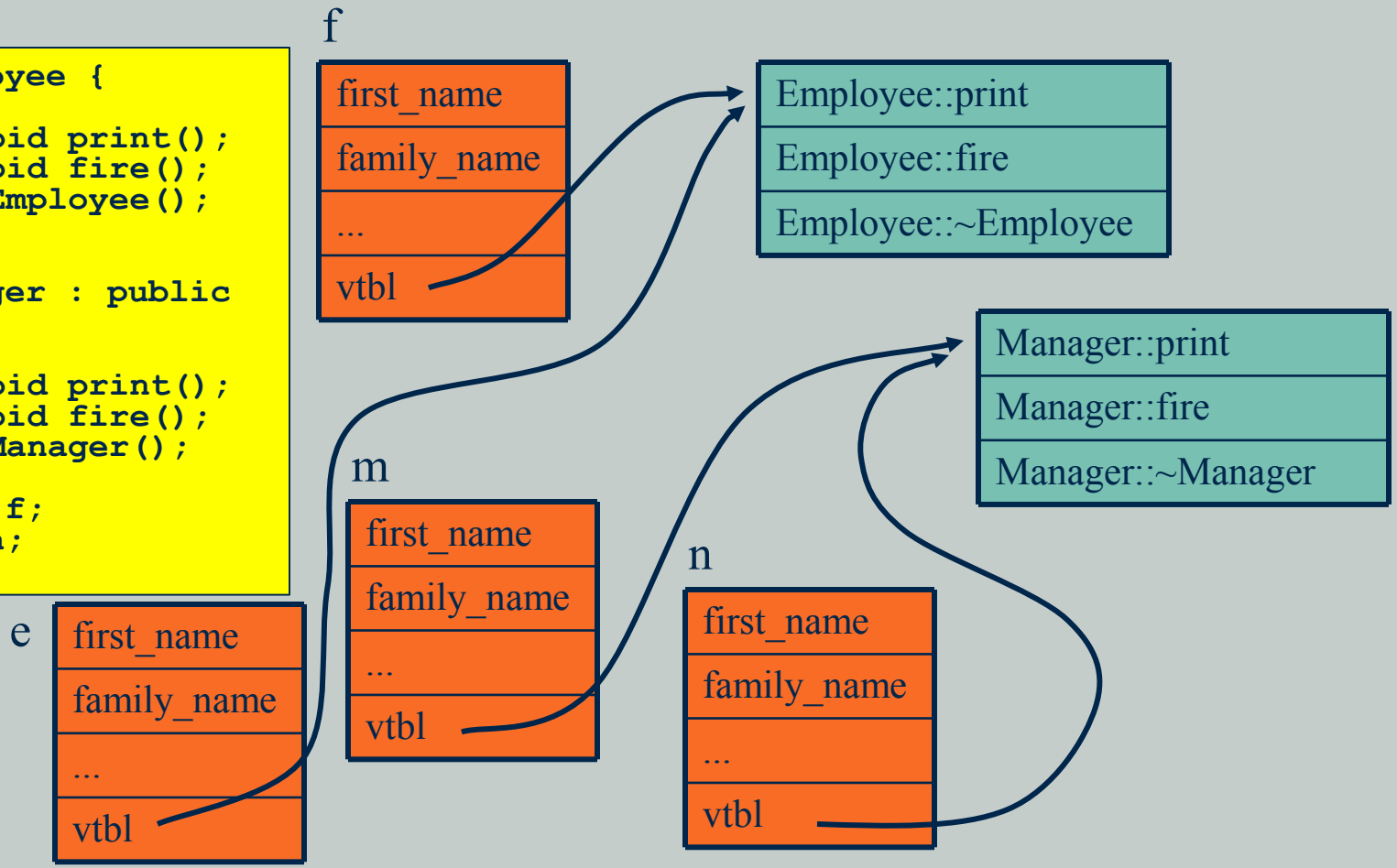
- Getting "the right" behavior from *Employee's* functions independently of exactly what kind of *Employee* is actually used is called polymorphism. A type with virtual functions is called a polymorphic type.
- To get polymorphic behavior in C++, the member functions called must be virtual and objects must be manipulated through pointers or references. When manipulating an object directly (rather than through a pointer or reference), its exact type is known by the compilation so that runtime polymorphism is not needed.
- Clearly, to implement polymorphism, the compiler must store some kind of type information in each object of class *Employee* and use it to call the right version of the virtual function *print()*. In a typical implementation, the space taken is just enough to hold a pointer.
- Calling a function using the scope resolution operator :: as is done in *Manager::print()* ensures that the virtual mechanism is not used.

Polymorphism

- Typical implementation of virtual functions consists in adding to evey object of a class containing at least one virtual function the pointer to the virtual function table
- This table contains pointers to all the virtual functions of the classes the given object belongs to

```
class Employee {
 //...
 virtual void print();
 virtual void fire();
 virtual ~Employee();
};

class Manager : public
 Employee {
 //...
 virtual void print();
 virtual void fire();
 virtual ~Manager();
};
Employee e,f;
Manager m,n;
```



Abstract Classes

- Many classes resemble class *Employee* in that they are useful both as themselves and also as bases for derived classes.
- Some classes, such as class *Shape*, represent abstract concepts for which objects cannot exist. A *Shape* makes sense only as the base of some class derived from it. This can be seen from the fact that it is not possible to provide sensible definitions for its virtual functions:

```
class Shape {
   public:
   virtual void rotate(int) { error("Shape::rotate") ; } // inelegant
   virtual void draw() { error("Shape::draw") ; }
   // ...
};
```

- Trying to make a shape of this unspecified kind is silly but legal:

```
Shape s; // silly: ``shapeless shape''
```

- It is silly because every operation on *s* will result in an error.

Abstract Classes

- A better alternative is to declare the virtual functions of class *Shape* to be pure virtual functions. A virtual function is "made pure" by the initializer = 0 :

```
class Shape{ // abstract class
 public:
   virtual void rotate(int) = 0;  // pure virtual function
   virtual void draw() = 0;       // pure virtual function
   virtual bool is_closed() = 0;  // pure virtual function
   // ...
};
```

- A class with one or more pure virtual functions is an abstract class, and no objects of that abstract class can be created:

```
Shape s; // error: variable of abstract class Shape
```

Abstract Classes

- An abstract class can be used only as an interface and as a base for other classes.

```
class Point{ /* ... */ };
class Circle : public Shape {
 public:
   void rotate(int) { }                    // override Shape::rotate
   void draw() ;                           // override Shape::draw
   bool is_closed() { return true; }       // override Shape::is_closed
   Circle(Point p, int r) ;
 private:
   Point center;
   int radius;
};
```

Abstract Classes

- A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class.
- This allows us to build implementations in stages:

```
class Polygon : public Shape{ // abstract class
 public:
   bool is_closed() { return true; }       // override Shape::is_closed
                                  // ... draw and rotate not overridden ...
};
Polygon b;  // error: declaration of object of abstract class Polygon
class Irregular_polygon : public Polygon {
   list<Point> lp;
 public:
   void draw() ;                  // override Shape::draw
   void rotate(int) ;             // override Shape::rotate
   // ...
};
Irregular_polygon poly(some_points) ; // fine (assume suitable constructor)
```

Abstract Classes

- An important use of abstract classes is to provide an interface without exposing any implementation details.
- An operating system might hide the details of its device drivers behind an abstract class:

```
class Character_device {
 public:
   virtual int open(int opt) = 0;
   virtual int close(int opt) = 0;
   virtual int read(char* p, int n) = 0;
   virtual int write(const char* p, int n) = 0;
   virtual int ioctl(int ...) = 0;
   virtual ~Character_device() { } // virtual destructor
};
```

- We can then specify drivers as classes derived from *Character_device*, and manipulate a variety of drivers through that interface.
- Every class having at least one virtual function should have the virtual destructor.

Example - Text-mode Windowing System

- A windowing system operating in text mode based on *ncurses* library
- The *ncurses* library allows to portably control the cursor position
- The program uses only a small subset of functions offered by *ncurses*

```
//screen.h
void init_screen ();
void done_screen ();
void gotoyx (int y, int x);
int ngetch ();
void getscreensize (int &y, int &x);
```

```
//ncurses.h
int printw(char *fmt [, arg] ...);
int refresh(void);
```

Example - Text-mode Windowing System

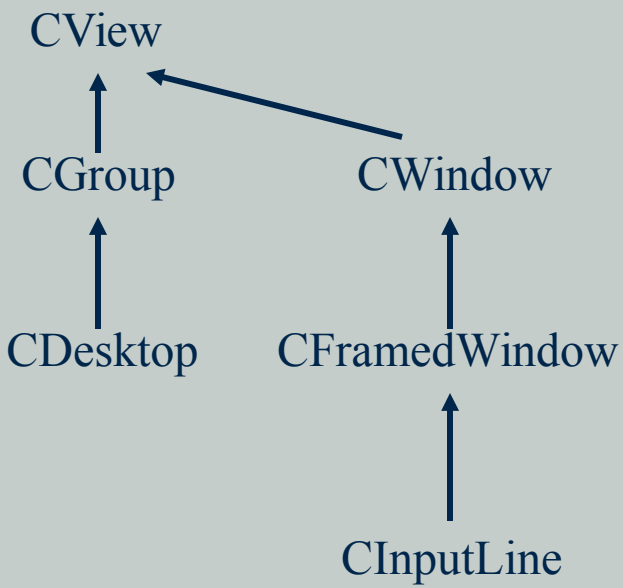
- In *cpoint.h* auxillary classes *CPoint* i *CRect* are defined:

```
struct CPoint
{
  int x;
  int y;
  CPoint(int _x=0, int _y=0): x(_x), y(_y) {};
  CPoint& operator+=(const CPoint& delta)
  {
    x+=delta.x;
    y+=delta.y;
    return *this;
  };
};

struct CRect
{
  CPoint topleft;
  CPoint size;
  CRect(CPoint t1=CPoint(), CPoint s=CPoint()): topleft(t1), size(s) {};
};
```

Example - Text-mode Windowing System

- The class hierarchy used in program



Example - Text-mode Windowing System

- *CView* - an object visible on screen
  - The *geom* field describing dimensions and posistion of a view
  - The *paint* printing the window's contents
  - The *handleEvent* functions handling events
  - Virtual destructor

```
class CView
{
protected:
  CRect geom;
public:
  CView (CRect g):geom (g)
  {
  };
  virtual void paint () = 0;
  virtual bool handleEvent (int key) = 0;
  virtual void move (const CPoint & delta)
  {
    geom.topleft += delta;
  };
  virtual ~CView ()  {};
};
```

Example - Text-mode Windowing System

- Event-driven environment
- Reacts only on external event
  - keypress
  - redraw command
- The objects do not do anything "on their own".

Example - Text-mode Windowing System

- *CWindow* - movable window

```
class CWindow:public CView
{
protected:
  char c;
public:
  CWindow (CRect r, char _c = '*'):CView (r), c (_c) {};
  void paint ()
  {
    for (int i = geom.topleft.y; i < geom.topleft.y + geom.size.y; i++)
      {
        gotoyx (i, geom.topleft.x);
        for (int j = 0; j < geom.size.x; j++)
          printw ("%c", c);
      };
  };
  bool handleEvent (int key)
  {
    switch (key)
      {
      case KEY_UP:
        move (CPoint (0, -1));
        return true;
      case KEY_DOWN:
         move (CPoint (0, 1));
        return true;
      case KEY_RIGHT:
        move (CPoint (1, 0));
        return true;
      case KEY_LEFT:
        move (CPoint (-1, 0));
        return true;
      };
    return false;
  };
};
```

Example - Text-mode Windowing System

- *CFramedWindow* - a window with a frame

```
class CFramedWindow: public CWindow
{
public:
  CFramedWindow (CRect r, char _c = '\''):CWindow (r, _c) {};
  void paint ()
  {
    for (int i = geom.topleft.y; i < geom.topleft.y + geom.size.y; i++)
   {
        gotoyx (i, geom.topleft.x);
        if ((i == geom.topleft.y)||(i == geom.topleft.y + geom.size.y - 1))
          {
            printw ("+");
            for (int j = 1; j < geom.size.x - 1; j++)
              printw ("-");
            printw ("+");
          }
        else
          {
            printw ("|");
            for (int j = 1; j < geom.size.x - 1; j++)
              printw ("%c", c);
            printw ("|");
          }
      }
  };
};
```

Example - Text-mode Windowing System

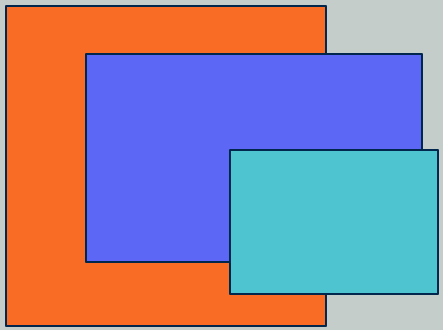
CInputLine - text input window

```
class CInputLine:public CFramedWindow
{
  string text;
public:
  CInputLine (CRect r, char _c = ',':CFramedWindow (r, _c) {};
  void paint ()
  {
    CFramedWindow::paint ();
    gotoyx (geom.topleft.y + 1, geom.topleft.x + 1);
    for (unsigned j = 1, i = 0; (j + 1 < (unsigned) geom.size.x) &&
                                (i < text.length ()); j++, i++)
      printw ("%c", text[i]);
  };
  bool handleEvent (int c)
  {
    if (CFramedWindow::handleEvent (c))
      return true;
    if ((c == KEY_DC) || (c == KEY_BACKSPACE))
      {
        if (text.length () > 0)
          {
            text.erase (text.length () - 1);
      return true;
          };
      }
    if ((c > 255) || (c < 0))
      return false;
    if (!isalnum (c) && (c != ' '))
      return false;
    text.push_back (c);
    return true;
  }
};
```

Example - Text-mode Windowing System

CGroup - a group of objects

```
class CGroup:public CView
{
  list < CView * >children;
public:
  CGroup (CRect g):CView (g) {};
  void paint ()
  {
    for (list < CView * >::iterator i = children.begin ();
         i != children.end (); i++)
      (*i)->paint ();
  };
 //...
};
```



Example - Text-mode Windowing System

- *CGroup* - a group of objects (contd.)

```
class CGroup:public CView
{
 list < CView * >children;
public:
 CGroup (CRect g):CView (g) {};
 void paint ();
 bool handleEvent (int key)
 {
    if (!children.empty () && children.back ()->handleEvent (key))
      return true;
    if (key == '\t')
      {
        if (!children.empty ())
          {
            children.push_front (children.back ());
            children.pop_back ();
          };
        return true;
      }
    return false;
 }
 //...
};
```

Example - Text-mode Windowing System

- *CGroup* - a group of objects (contd.)

```
class CGroup:public CView
{
 list < CView * >children;
public:
 CGroup (CRect g):CView (g) {};
 void paint ();
 bool handleEvent (int key);
 void insert (CView * v)
  {
    children.push_back (v);
  };
 ~CGroup ()
 {
    for (list < CView * >::iterator i = children.begin ();
                                          i != children.end (); i++)
      delete (*i);
 };
};
```

Example - Text-mode Windowing System

- *CDesktop* - the entire screen

```
class CDesktop:public CGroup
{
public:
CDesktop ():CGroup (CRect ())
  {
      int y, x;
      init_screen ();
      getscreensize (y, x);
      geom.size.x = x;
      geom.size.y = y;
  };
~CDesktop ()
  {
    done_screen ();
  };
void paint()
{
  for (int i = geom.topleft.y;
   i < geom.topleft.y + geom.size.y; i++)
    {
     gotoyx (i, geom.topleft.x);
     for (int j = 0; j < geom.size.x; j++)
       printw (".");
    };
  CGroup::paint();
}
```

```
int getEvent ()
  {
    return ngetch ();
  };
void run ()
  {
    int c;
    paint ();
    refresh ();
    while (1)
      {
        c = getEvent ();
        if (c == 27)
          break;
        if (handleEvent (c))
          {
            paint ();
            refresh ();
          };
      };
  };
};
```

Example - Text-mode Windowing System

- The *main* function

```
int main ()
{
  CDesktop d;
  d.insert (new CInputLine (CRect (CPoint (5, 7), CPoint (15, 15))));
  d.insert (new CWindow (CRect (CPoint (2, 3), CPoint (20, 10)), '#'));
  d.run ();
  return 0;
};
```

Resource Management

- When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, sets an access control lock, etc., – it is often essential for the future running of the system that the resource be properly released. Often that "proper release" is achieved by having the function that acquired it release it before returning to its caller.

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w") ;
  // use f
  fclose(f) ;
}
```

- This looks plausible until you realize that if something goes wrong after the call of *fopen()* and before the call of *fclose()*, an exception may cause *use_file()* to be exited without *fclose()* being called. We can fix it as follows:

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w") ;
  try {
      // use f
  }
  catch (...) {
      fclose(f) ;
      throw;
  }
  fclose(f) ;
}
```

Resource Management

- The problem with the solution on the previous slide is that it is verbose, tedious, and potentially expensive. Furthermore, any verbose and tedious solution is error-prone because programmers get bored. Fortunately, there is a more elegant solution. The general form of the problem looks like this:

```
void acquire()
{
  // acquire resource 1
  // ...
  // acquire resource n

  // use resources

  // release resource n
  // ...
  // release resource 1
}
```

- It is typically important that resources are released in the reverse order of their acquisition. This strongly resembles the behavior of local objects created by constructors and destroyed by destructors.

Resource Acquisition Is Initialization Technique

- Let's define a *File_ptr* class, which behaves as a *FILE**:

```
class File_ptr {
      FILE* p;
  public:
      File_ptr(const char* n, const char* a) { p = fopen(n,a) ; }
      File_ptr(FILE* pp) { p = pp; }
      ~File_ptr() { fclose(p) ; }
      operator FILE*() { return p; }
};
```

- The program size is significantly reduced

```
void use_file(const char* fn)
{
  File_ptr f(fn,"r") ;
  // use f
}
```

Resource Acquisition and Constructors

- The similar technique can be applied in constructors

```
class X {
        File_ptr aa;
        Lock_ptr bb;
  public:
        X(const char* x, const char* y)
        : aa(x,"rw") ,      // acquire 'x'
          bb(y)             // acquire 'y'
        {}
   // ...
};
```

- The most common resource acquired in an ad-hoc manner is memory.

```
class Y {
    int* p;
    void init() ;
  public:
    Y(int s) {p = new int[s]; init() ; }
    ~Y() { delete[] p; }
   // ...
};
```

```
class Z {
        vector<int> p;
        void init() ;
  public:
        Z(int s) : p(s) { init() ; }
   // ...
};
```

- Usually it is better to employ a standard *vector* template

unique_ptr Template

- The standard library provides the template class *unique_ptr*, which supports the "resource acquisition is initialization" technique.
- The *unique_ptr* template is declared in *<memory>*.
- Basically, an *unique_ptr* is initialized by a pointer and can be dereferenced in the way that a pointer can.
- Also, the object pointed to will be implicitly deleted at the end of the *unique_ptr's* scope.

```
void f()
{
  std::unique_ptr<Shape> a(new Shape());
  a->move(3,3);
  ...
  if(in_a_mess)
     throw Exception()
}
```

unique_ptr Template

- The *unique_ptr* does not have copy constructor nor the copying assignment operator.

```
std::unique_ptr<Shape> a(new Shape());
std::unique_ptr<Shape> b(a); // error - no copy constructor
std::unique_ptr<Shape> c;
c = a; // error - no copying assignment operator
```

- It has a move constructor and move assignment operator

```
std::unique_ptr<Shape> a(new Shape());
std::unique_ptr<Shape> b(std::move(a));
std::unique_ptr<Shape> c;
c = std::move(a);
```

The Move Constructor and Assignment Operator

- The move constructor destroys the original
- Is used by default if the source is an rvalue
  - Something you cannot take the address of, e.g. literal or temporary/anonymous object
- You can force its application with *std::move*

```
class Example {
  public:
  int* val;
  Example(Example&& rhs) : val(rhs.val)
  {
    rhs.val = nullptr;
  }
  Example& operator=(Example&& rhs)
  {
    val = rhs.val;
    rhs.val = nullptr;
    return *this;
  }
  Example(int a): val(new int(a)) {}
  ~Example()  {  delete val; }
};
```

```
int main()
{
  Example a(3);
  Example b = std::move(a);
  Example c(Example(4));
  Example d;
  d = std::move(b);
}
```

unique_ptr and Exception Safety

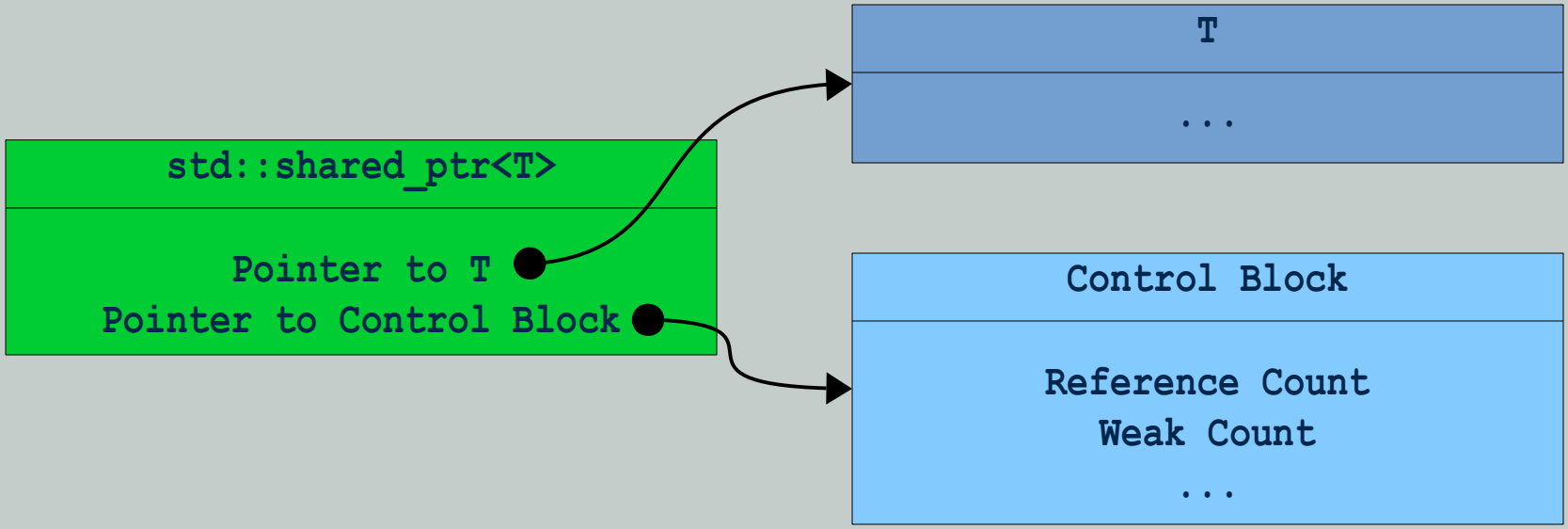
- *unique_ptr* still requires some care to make it exception-safe
- The line below can cause problems:

  fun(unique_ptr<Example>(new Example), genexc())
- If the evaluation order is as follows:
  - new *Example* object is constructed
  - *genexc* is called
  - *unique_ptr* constructor is invoked

  exception in *genexc* will cause a memory leak
- That's why we have *make_unique*
  - fun(make_unique<Example>(), genexc());
- The line above is fully exception-safe

shared_ptr Template

- Due to lack of the copy constructor in *unique_ptr* we can have only one pointer to a given object
- If we need more, we have to use *shared_ptr*
- Due to reference counting, the last *shared_ptr* to a given object destroys it



shared_ptr Template

- We can pass shared pointers around without worrying about ownership ...

```
std::shared_ptr<Example> getPtr() {
  std::shared_ptr<Example> a(new Example());
  std::shared_ptr<Example> b = a;
  return a;
}

int main() {
  std::shared_ptr<Example> c = getPtr();
}
```

- unless we create a cycle:

```
int main() {
  std::shared_ptr<Example> p1 = std::make_shared<Example>();
  std::shared_ptr<Example> p2 = std::make_shared<Example>();
  p1->ptr=p2;
  p2->ptr=p1;
  return 0;
}
```

- The code above introduces a memory leak

weak_ptr Template

- We can use *weak_ptr* to break a cycle:

```
int main()
{
  std::weak_ptr<Example> w1;
  {
    std::shared_ptr<Example> p1 = std::make_shared<Example>();
    w1 = p1;
    std::shared_ptr<Example> p2 = w1.lock();
    assert(p2 != nullptr);
  }

  std::shared_ptr<Example> p3 = w1.lock();
  assert(p3 == nullptr);
}
```

- *weak_ptr* is a non-owning pointer and before we can use it we have to convert it to *shared_ptr*
  - due to this fact, it can dangle, i.e. point to deleted object

Smart Pointers and Arrays

- Do not use a smart pointer to hold a pointer to an array:

```
void g()
{
 shared_ptr<int> p=new int[100];
       // error, delete instead of delete[] invoked on exit from g
}
```

- Use vector template for this purpose

```
void g()
{
  vector<int> a(100);
  int* p=&(a[0]); // guaranteed to point to array of 100 integers
}
```