Chapter 15: Polymorphism and Virtual Functions

From Lewis Carrol, Through the Looking Glass:

“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean – neither more nor less.”

“The question is,” said Alice, “whether you can make words mean so many different things.”

“The question is,” said Humpty Dumpty, “which is to be master– that’s all.”

15.1 Basic Concepts

15.1.1 Definitions

Simple derivation. The Hangman program uses the simple form of derivation: two application classes are derived from the same base class. This kind of derivation serves two important purposes:

- To factor out the common parts of related classes so that the common code does not need to be written twice. An example of this is the BaseWord class with its derivatives, Alphabet, and Hangword.
- To facilitate reuse of library classes and templates. The base class will contain a generally-useful data structure and its functions. The derived class will contain functions that are specific to the application. An example of this is the RandString class, which is derived from an instantiation of the FlexArray template.

Polymorphic derivation. In this chapter, we introduce virtual functions and two complex and powerful uses for derived classes that virtual functions support: abstraction and polymorphism.

- A virtual function is a function in a base class that forms part of the interface for a set of derived classes.
  It is declared virtual in the base class and may or may not have a definition in that class. It will have definitions in one or more of the derived classes. The purpose of a virtual function is to have one name, one prototype, and more than one definition so that the function’s behavior can be appropriate for each of the derived classes.
- A pure virtual function is a function that has no definition in the base class.
- An abstract class is a class with one or more pure virtual functions.
- A polymorphic class is a base class that supports a declared set of public virtual functions, together with two or more derived classes that define methods for those functions. Data members and non-virtual functions may be defined both in the base class and the derived classes, as appropriate. We say that the derived classes implement the polymorphic interface class.

15.1.2 Virtual functions.

A virtual function is shared by the classes in a derivation hierarchy such as the Employee classes in Figure 15.1.2 (Note: In this sketch, three shapes have been drawn on top of the ordinary rectangular class boxes. This is not proper UML; the shapes will be used later to explain polymorphism.)

We create a virtual function when the same task must be done for all objects of all types in the hierarchy, but the method for doing this task depends on the particular representation of an object. For example, suppose the function calculatePay() must be defined for all employees, but the formula for the calculation is different for union members and professional staff. We want a function with one name that is implemented by three or more defining methods. For any given function call, we want the appropriate function to be called. For example, suppose we have declared four objects:
Then if we call A.calculatePay() or M.calculatePay() we want the calculation to be made using the formula for professional staff. If we call C.calculatePay() or D.calculatePay(), we want the union formula used.

We implement this arrangement by declaring (and defining) calculatePay() as a virtual function in the Employee class, with or without a general method in that class. We also define calculatePay() in each of the derived classes, to make the appropriate calculations for the specific type of employee. The general method might do the parts of the calculation that are common to all classes. When calculatePay() is called, the system must select one of the methods to use. It does this by looking at the specific type of the object in the function call. This issue is explored more fully in the section on polymorphic classes.

Syntax. For examples, look at print() in Container, Linear and Queue, later in this chapter.

- The prototype for a virtual function starts with the keyword `virtual`. The rest of the prototype is just like any other function.
- The prototype for a pure virtual function ends in `=0` instead of a semicolon. This means that it has no definition within the class.
- You must not write the keyword `virtual` before the definition of a virtual function that is outside the class.
- Any class with one or more virtual functions must have a virtual destructor.

How it works. Every object that could require dynamic dispatch must carry a type tag (one byte) at run time that identifies its relationship to the base class of the class hierarchy. (1st subclass, 2nd subclass, etc.) This is necessary if even one function is declared virtual, either in the class itself or in a parent class.

The run-time system will select the correct method to use for each call on a virtual function. To do so, it uses the type tag of the implied parameter to subscript the function’s dispatch table. This table has one slot for each class derived from the class that contains the original virtual declaration. The value stored in slot $k$ is the entry address for the method that should be used for the $k$th subclass derived from the base class. This is slightly slower than static binding because there is one extra memory reference for every call on every virtual function.

15.2 Polymorphic Classes

The purposes of polymorphism.

- To define an extensible set of representations for a class. One or two may be defined at first, more added later. It is a simple way to make an application extensible; you can build part of it now, part later, and be confident that the parts will work together smoothly.
- To allow a data structure to contain a mixture of items of different but related subtypes, such as the linked list of employees illustrated in Figure 15.2. The data objects in this list are the Employees defined in section 15.1.2; the shape of each object indicates which subtype of Employee it belongs to.
- To support run-time variability of types within a restricted set of related types, and the accompanying run-time dispatch (binding) of methods. The most specific appropriate method is dispatched. This lets us create different varieties of objects depending on input that is entered at run time.
Dispatching. Dispaching a function call is simple when the implied argument belongs to a derived class and a method for the virtual function is defined in that class. However, this is not always the case. For example, Figure 15.2 is a list of pointers to polymorphic Employees. Suppose a virtual print() function is defined in Employee and given a method there. Also, methods are defined in Accountant and Manager; these print the special data then call the Employee::print() to print the rest. However, no method is defined in Clerk because the general Employee method is appropriate. Now suppose we want to print the four Employees on the list and write this:

```c
Employee* p;
for (p=Staff.Empls; p!=NULL; p=p->Next) p->Data->print(cout);
```

When we execute this loop, the Employee::print() function will be called four times. If it were an ordinary function, Employee::print() would be executed four times. However, that does not happen because Employee::print() is virtual, which means that the most appropriate function in the class hierarchy will be executed when Employee::print() is called:

1. The system will dispatch Manager::print() to print M because M is a Manager.
2. To print C and D, Employee::print() will be used because the Clerk class does not have a print function of its own.
3. To print the Accountant, A, the system will dispatch Accountant::print().

In all cases, the most specific applicable method is selected at run time and dispatched. It cannot be done at compile time because three different methods must be used for the four Employees.

Two limitations. Virtual functions cannot be expanded inline because the correct method to apply is not known until run time, when the actual type of the implied parameter can be checked. For this reason, using a large number of virtual functions can increase execution time.

If you have a function that you want a derived class to inherit from its parent, do not define another function with the same name in the derived class, even if its parameter list is different.

Two design guidelines. If you have a polymorphic base class:

1. ... and you allocate memory in a derived class, you must make the base-class destructor virtual.
2. You can use the C++ dynamic_cast operator for safe run-time polymorphic casting.

15.3 Creating a Polymorphic Container

Overview of the demo program. The major example in this chapter is a polymorphic implementation of linear containers. The class Container is an abstract class from which linear containers (lists, queues) and non-linear containers (trees, hash tables) could be derived. Container supplies a minimal generic interface for container classes (those that can be used to store collections of data items).

In this chapter and the next, we focus on linear containers. The class Linear is the base class for a polymorphic family of list-based containers. From it we derive several data structures: Stack and Queue in this chapter, List and Priority Queue in the next. All of these conform to the Container interface and Linear implementation strategy but present different insertion and/or deletion rules. We use the containers in this chapter to store objects of class Exam, consisting of a 3-letter name and an integer.

The classes Linear and Cell should be defined as templates so that they do not depend on the type of the data in the list. Templates were not used here because they would complicate the structure of the program and the important issue here is the structure of a polymorphic class. In a real application, both would be used.
What to look for.

- A virtual function can be initially declared with or without a definition. For example, `pop()` is declared without a definition in `Container`, but `insert()` is declared with a definition in `Linear`.
- A virtual function can be redefined (directly or indirectly) in a derived class, as `Queue::insert()` redefines `Linear::insert()`. When redefinition is used, the derived-class method is often defined in terms of the inherited method, as `Stack::print()` is defined in terms of `Linear::print()`.
- If a function is declared virtual, then it is virtual in all derived classes.
- When a virtual function is executed on a member of the base class, the method defined for that function in the appropriate derived class is dispatched, if it exists. Thus, when `insert()` is called from `Linear::put()` to insert a new Cell into a Queue, `Queue::insert()` is dispatched, not `Linear::insert()`. If `Linear::insert()` were not virtual, `Linear::insert()` would be executed.

15.3.1 Container: An Abstract Class

```cpp
class Container {
    public:
    virtual ~Container() {} // Put Item into the Container.
    virtual void put(Item*) = 0;
    virtual Item* pop() = 0; // Remove next Item from Container.
    virtual Item* peek() = 0; // Look but don't remove next Item.
    virtual ostream& print(ostream&) = 0; // Print all Items in Container.
};
```

A container is a place to store and retrieve data items of any sort which have been attached to cells. Each container has its own discipline for organizing the data that is stored in it. In this section we develop a generic linear container class from which special containers such as stacks or queues can be derived. We use the program to illustrate the concepts, syntax, and interactions among a polymorphic base class and its implementation classes.

The Container class presented here supplies implementation-independent functions that are appropriate for any container and any contents type. Every container must allow a client program to put data items into the container and get them back out. A print() function is often useful and is necessary for debugging.

15.3.2 Linear: A Polymorphic Class

A linear container is one that has a beginning and an end, and the data inside it is arranged in a one-dimensional manner. It might be sorted or unsorted. The class Linear defines a simple, unsorted, linear container that is implemented by a linked list, using a helper class named Cell. It defines a set of list-handling function that can be written in a generic way so that they will apply to most or all linked-list linear containers. All of these functions are protected except the public interface functions that were inherited from Container. The other functions are not public because they allow a caller to decide how and where to put things into or take things out of the list. This kind of control is necessary for defining Stack and Queue but not safe for public use.

Since Linear is derived from Container, it inherits all of the function prototypes of Container. Since these functions are equally appropriate for use with an array or a linked list, Linear could be implemented either way. We commit to a linked list implementation when we define the Linear constructor in line 37. The code for `reset()` (line 39), `end()` (line 40), and all the functions in the .cpp file also rely on a linked list representation for the data.
15.3. CREATING A POLYMORPHIC CONTAINER

```cpp
// Linear Containers
// A. Fischer June 12, 2001 file: linear.hpp

#ifndef LINEAR_H
#define LINEAR_H

#include "contain.hpp"
#include "cell.hpp"
#include "tools.hpp"

class Linear: public Container {
protected: // -----------------------------------------------------------------------------
    Cell* head; // This is a dummy header for the list.

private: // -----------------------------------------------------------------------------
    Cell* here; // Cursor for traversing the container.
    Cell* prior; // Trailing pointer for traversing the container.

protected: // -----------------------------------------------------------------------------
    Linear(): head(new Cell), here(NULL), prior(head) {};
    virtual ~Linear();
    void reset() { prior = head; here = head->next; }
    bool end() const { return here == NULL; }
    void operator ++();
    //virtual void insert( Cell* cp );
    void insert( Cell* cp );
    virtual void focus() = 0;
    Cell* remove();
    void setPrior(Cell* cp){ prior = cp; here = prior->next; }

public: // -----------------------------------------------------------------------------
    void put(Item* ep) { if (ep) insert( new Cell(ep) ); }
    Item* pop();
    Item* peek() { focus(); return *here; }
    //virtual ostream& print( ostream& out );
    ostream& print( ostream& out );
    inline ostream& operator<<(ostream& out, Linear& s) {return s.print(out); }

#endif
```

The two private data members in this class, together with the functions reset(), end(), and ++ permit us to start at the beginning of the container and visit each member sequentially until we get to the end. The reset() function sets the pointer named here to the first item in the container and sets prior to NULL. The ++ operator sets prior = here and moves here to the next item. At all times, these two pointers will point to adjacent items in the container (or to NULL). The end() function returns true if here has passed the last item in the container.

Three of the functions inherited from Container are no longer virtual, so the run-time dispatcher will not look in the Stack or Queue class for an overriding definition when put(), pop() or peek() is called from within Linear. One would expect put() to be virtual, since stacks and queues need different methods for putting an item into the container. However, put() simply wraps the item in a Cell, then delegates the actual insertion operation to insert(), which is virtual so that the specific and appropriate version of insert() in the derived class (Stack, Queue, etc.) will always be dispatched. We say that the Linear class collaborates with Stack or with Queue to handle the insertion task.

In a similar way, remove(), which is not virtual collaborates with focus(), which is virtual to focus the deletion on the proper element of the list. Since this is a pure virtual function, Linear is an abstract class that cannot be instantiated. The print() function is virtual for the same reason: to pass the responsibility to a derived class that is the experts on how printing should be done. The destructor is virtual because C++ requires a virtual destructor in classes that have virtual functions.

The Container class refers to Cells but does not define Cell. From that class, all we know is that Cell is
a helper class that must be used when information is placed into the Container. At that stage, a Cell could be
the traditional linked-list cell or it could be just a typedef-synonym for Item*, appropriate for use with an
array-based container. In the Linear class, we must commit to one representation or another, and we do commit
to using a linked list. The definition given here for Cell is the usual two-part structure containing an Exam*
and a Cell*. Nothing in Cell is virtual and everything is inline.

```cpp
// Linear Containers
// A. Fischer June 12, 2001 file: linear.cpp
#include "linear.hpp"

Linear::~Linear () {
    for (reset(); !end(); ++*this) {
        delete prior->data;
        delete prior;
    }
    delete prior->data;
    delete prior;
}

// Move index to next Item in the container.
void
Linear::operator ++() {
    if (!end()) {
        prior = here;
        here = here->next;
    }
}

// Put an Item into the container between prior and here.
// Assumes that here and prior have been positioned already.
void
Linear::insert(Cell* cp) {
    cp->next = here;
    here = prior->next = cp;
}

// Take an Item out of the container. Like pop or dequeue.
// Assumes that here and prior have been positioned to the desired Item.
Cell*
Linear::remove() {
    if (! here ) return NULL;
    Cell* temp = here; // Grab cell to remove.
    here = prior->next = temp->next; // Link around it.
    return temp;
}

// Remove a Cell and return its Item.
Item*
Linear::pop(){
    focus(); // Set here and prior for deletion point.
    Cell* temp = remove(); // Remove first real cell on list.
    if (!temp) return NULL; // Check for empty condition.
    Item* answer = *temp; // Using cast coercion from Cell to Item.
    delete temp; // Take contents out of cell, delete Cell.
    return answer; // Return data Item.
}

// Print the container's contents.
ostream&
Linear::print (ostream& out ) {
    out << " <\n"
    for (reset(); !end(); ++this) out << "\t" <<*here;
    return out << " ]\n";
};
```
As in previous linked list definitions, this Cell class gives friendship to Linear. One slight difference is the friendship declaration on line 122, which is needed for the operator extension on line 139. We need to choose one of three alternatives:

a. Using the friend function declaration,
b. Making the print function public, and
c. Not having an operator extension for class Cell.

In previous versions of this class, we chose strategy (c); this time we choose (a) because it makes the Linear::print function shorter.

15.3.3 Cell: The Helper Class

One new technique is introduced in Cell: we define a cast (line 130) from type Cell to type Item*. This cast can be used explicitly, just like a built-in cast function or it can be used by the compiler to coerce (automatically convert) the type of an argument. For example, Line 100, in Linear::pop(), could be written two ways, as shown below.

Item* answer = (Item*)((Cell*)p)->data(); // Explicit use of a cast from Cell to Item*.
Item* answer = *p; // Using cast coercion from Cell to Item*.

The first uses cast operation explicitly. The second supplies *p in a context where an Item* is required. The compiler will find the cast operator on line 130 and use it to coerce the argument on the right to the type of the variable on the left.

15.3.4 Exam: The Actual Data Class

The linear containers defined above would be appropriate to store any kind of data. The data class used here is very simple but could be much more complex. It is used again in the next chapter where it is extended by
adding comparison functions and a key() function so that the data can be sorted.

```cpp
// Exam: A student's initials and one exam score.
// A. Fischer, October 1, 2000 file: exam.hpp
//===========================================================================
 ifndef EXAM_H
 #define EXAM_H
 #include "tools.hpp"
 typedef int KeyType;
 class Exam // One name-score pair
 {
 private: //---------------------------------------------------------------
 char Initials [4]; // Array of char for student name
 protected://---------------------------------------------------------------
 int Score; // Integer to hold score
 public: //----------------------------------------------------------------
 Exam (const char* init, int sc){
   strncpy( Initials, init, 3);
   Initials[3] = '\0';
   Score = sc;
 }
 ~Exam (){ cerr << " Deleting Score " <<Initials "..."; }
 ostream& Print ( ostream& os ){
   return os <<Initials "": " <<Score " ";
 }
};
//===========================================================================
 ifndef ITEM_H
 #define ITEM_H
 typedef Exam Item;
//===========================================================================
 #ifndef ITEM_H
 #define ITEM_H
 #endif
```

15.3.5 Class diagram.

Figure 15.3.5 is a UML diagram for this program, showing the polymorphic nature of Linear and its relation to Cell, Stack, and Queue.

15.4 Stack: Fully Specific

A Stack is a container that implements a LIFO discipline. In this program, we derive Stack from Linear and, in so doing, we represent a stack by a linear linked list of Cells. From Linear, Stack inherits a head pointer, two scanning pointers (here and prior) and a long list of functions.

Notes on the Stack code.

- The list head pointer is protected, not private in Linear, because some derived classes (for example Queue), need to refer to it. The scanning pointers are private because the derived classes are supposed to use Linear::setPrior() rather than setting the pointers directly. This guarantees that the two scanning pointers are always in the correct relationship to each other, so that insertions and deletions will work properly. It also saves lines of code in the long run, since they are written once in Linear instead of multiple times in the derived classes.
The Stack constructor and destructor are both null functions; they are supplied because it is bad style to rely on the defaults. The program does work perfectly well without either. Five functions are defined explicitly in Stack.

Stack defines focus(), which is abstract in Linear. This is called by Linear::remove() to set prior and here so that the first Cell in the container will be returned. (Last in, first out.)

Since focus(), was the only remaining pure virtual function in Linear. Since we define it here, Stack becomes a fully specified class and can be used to create objects (that is, it can be instantiated.)
• Linear::put(Exam*) calls Insert, which is defined in both Linear and Stack. Because insert() is virtual, the version in Stack will be used to execute the call written in Linear::put(). We want to insert the new Cell at the head of the list because this is a stack. The actual insertion will be done by Linear::insert(), after Stack::insert() positions here and prior to the head of the list by calling reset(). The left side of figure 15.4 illustrates how control passes back and forth between the base class and the derived class during this process.

Figure 15.4: How the classes collaborate during Stack::put (left) and Stack::print (right).

• The redefinition of print() is not absolutely necessary here; all it does is to print a few words ("The stack contains:")) and call Linear::print() to print the contents. For contrast, the Queue class does not define a print function. However, the collaboration between the two print() functions and the << operator is instructive.

In the main program, we never call the print functions directly. All printing is done by statements like cerr << S; But Stack and Queue do not supply definitions for operator<<, so the output task is given (by inheritance) to the definition in Linear (line 54). It, in turn, calls Linear::print(), which is virtual, and dispatches the job to Stack::print() because S is a stack. Stack::print() prints the output label and calls Linear::print() to finish the job. The class name is necessary only in the last call; all other shifts of responsibility are handled by inheritance or virtual dispatching. This activity is diagrammed on the right in Figure 15.4

15.5 Queue: Fully Specific

```
204 // -------------------------------------------------------------------------
205 // Queues: derived from Container<--Linear<--Queue
206 // A. Fischer June 9, 2001 file: queue.hpp
207 // -------------------------------------------------------------------------
208 #ifndef QUEUE_H
209 #define QUEUE_H
210 #include "linear.hpp"
211 // -------------------------------------------------------------------------
212 class Queue : public Linear {
213 private:
214    Cell* tail;
215 public:
216    Queue() { tail = head; }
217    Queue(){}
218    void insert( Cell* cp ) { setPrior(tail); Linear::insert(cp); tail=cp;}
219    void focus(){ reset(); }
220    // -------------------------------------------------------------------------
221 #endif
```
A queue is a container that implements a FIFO discipline. This Queue is a linear linked list of Cells with a dummy header. This declaration of Queue follows much the same pattern as Stack. However, Queue uses the inherited Linear::print(), instead of defining a specific version of its own. This is done for demonstration purposes; we recognize that better and more informative formatting could be achieved by defining a specific local version of print().

![Figure 15.5: The Queue named Q, newly constructed and after one insertion.](image)

**Notes on the Queue code.**

- Linear has three data members (head, prior and here), and Queue has one (tail). The constructor of each class must initialize the data members in its own class. When a Queue is constructed, all four are initialized to create an empty linked list with a dummy header, as in Figure 15.5. An important issue here is that the constructor of the base class, Linear, is executed first. It creates a dummy cell and attaches it to the head and prior pointers. When control gets to the Queue constructor, the dummy cell will exist and it is easy to attach the tail pointer to it.

- Queue::insert(Item*) set here and prior to the tail of the list so that insertions will be made after the last list cell. As before, Linear::insert() then finishes the job and does the actual insertion. Almost no code is duplicated because the classes collaborate.

- Queue defines focus(), which is abstract in Linear, to set the scanning pointers to the beginning of the Queue. The result is that the earliest insertion will be the next removal (FIFO).

- When an inherited virtual function (such as pop) is called, control passes back and forth between the base class and the derived class, as shown in Figure 15.4.

![Figure 15.6: How the classes collaborate during Queue::pop.](image)

- The output (following main) shows that the queue Q was printed by the inherited function, Linear::print().

- If Linear::insert() is *not* virtual, insertion is done incorrectly for queues. I removed the “virtual” property and got the following output:

```
Putting 3 items on the Queue Q: 11, 22, 33.
>[: Cell 0x00804c98 [Cil: 33 ] , 0x804c478]
Cell 0x00804c478 [Bea: 22 ] , 0x804c458]
Cell 0x00804c458 [Ali: 11 ] , (nil)]
```

Compare it to the correct output at the end of this chapter, where Ali comes first in the queue.
15.6 A Main Program and its Output

```
// Demonstration of derived classes with virtual functions.
#include "tools.hpp"
#include "exam.hpp" // Must precede #include for item.hpp.
#include "item.hpp" // Abstract base type for stacks and queues.
#include "stack.hpp" // Base type is Item == Exam.
#include "queue.hpp" // Base type is Item == Exam.

#include <iostream>
using namespace std;

int main( void ) {
    cerr << "Putting 3 items on the Stack S: 99, 88, 77.\n";
    Stack S;
    S.put( new Exam("Ned", 99) ); //cerr << S << endl;
    S.put( new Exam("Max", 88) ); //cerr << S << endl;
    cerr << " Peeking after second insertion: " <<*S.peek() <<"\n";
    S.put( new Exam("Leo",77) ); cerr << S << endl;

cerr << "Putting 3 items on the Queue Q: 11, 22, 33.\n";
    Queue Q;
    Q.put( new Exam("Ali",11) ); //cerr << Q << endl;
    Q.put( new Exam("Bea",22) ); //cerr << Q << endl;
    cerr << " Peeking after second insertion: " <<<Q.peek() <<"\n";
    Q.put( new Exam("Cil",33) ); cerr << Q << endl;

cerr << "Pop two Exams from Q, put on S. \n";
    S.put(Q.pop()); S.put(Q.pop()); cerr <<"\n" <<S << endl;

cerr << "Put another Exam onto Q: 44.\n";
    Q.put( new Exam("Dan",44) ); cerr << Q << endl;

cerr << "Pop two Exams from S and discard.\n";
    delete S.pop();
    delete S.pop(); cerr <<"\n" << S << endl;
bye();
}
```

To test the linear container classes, we wrote a meaningless main program that instantiates one Stack and one Queue and moves data onto both and from one to the other. The call graph in Figure 15.6 attempts to show the ways that functions are actually called, after dynamic dispatching. In this chart, grey circles that say “V” mark virtual dispatching and circles that say “I” show calls that were made through inheritance.

Enough function calls are made to demonstrate that the classes work properly; the contents of S and Q are printed just often enough to see the data move into and out of each container. Note that several diagnostic output commands were used during debugging and are now commented out.

- The Queue implements a first-in first-out order.
- The Stack implements a last-in first-out order.
- peek() returns the same thing that pop() would return, but does not remove it from the list.
- Stack::print() is used to print the stack but the inherited Linear::print() prints the queue.
- All dynamically allocated objects are properly deleted.

The output:

```
Putting 3 items on the Stack S: 99, 88, 77.
Peeking after second insertion: Max: 88
The stack contains:
] [ Cell 0x0x804c988 [Leo: 77 , 0x804c968]
    Cell 0x0x804c968 [Max: 88 , 0x804c948]
    Cell 0x0x804c948 [Ned: 99 , (nil)]
```
Putting 3 items on the Queue Q: 11, 22, 33.
Peeking after second insertion: Ali: 11
\[
\begin{align*}
\text{Cell} & 0x0x804c9a8 \ [\text{Ali}: 11, 0x804c9c8] \\
\text{Cell} & 0x0x804c9c8 \ [\text{Bea}: 22, 0x804c9e8] \\
\text{Cell} & 0x0x804c9e8 \ [\text{Cil}: 33, (\text{nil})]
\end{align*}
\]
>

Pop two Exams from Q, put on S.
Deleting Cell 0x0x804c9a8...
Deleting Cell 0x0x804c9c8...
The stack contains:
\[
\begin{align*}
\text{Cell} & 0x0x804c9c8 \ [\text{Bea}: 22, 0x804c9a8] \\
\text{Cell} & 0x0x804c9a8 \ [\text{Ali}: 11, 0x804c988] \\
\text{Cell} & 0x0x804c988 \ [\text{Leo}: 77, 0x804c968] \\
\text{Cell} & 0x0x804c968 \ [\text{Max}: 88, 0x804c948] \\
\text{Cell} & 0x0x804c948 \ [\text{Ned}: 99, (\text{nil})]
\end{align*}
\]
>

Put another Exam onto Q: 44.
\[
\begin{align*}
\text{Cell} & 0x0x804c9e8 \ [\text{Cil}: 33, 0x804ca08]
\end{align*}
\]
>

Pop two Exams from S and discard.
Deleting Cell 0x0x804c9c8... Deleting Score Bea...
Deleting Cell 0x0x804c9a8... Deleting Score Ali...
The stack contains:
\[
\begin{align*}
\text{Cell} & 0x0x804c988 \ [\text{Leo}: 77, 0x804c968] \\
\text{Cell} & 0x0x804c968 \ [\text{Max}: 88, 0x804c948] \\
\text{Cell} & 0x0x804c948 \ [\text{Ned}: 99, (\text{nil})]
\end{align*}
\]
>

**Termination.** After printing the termination message, the objects Q and S will go out of scope and be deallocated. An output trace can serve as part of a proof that deallocation is done correctly and fully, without crashing. The output trace from main is given below, with a call graph (Figure 15.6) showing how control moves through the destructors of the various classes.

**Normal termination.**

Deleting Cell 0x0x804c928... Deleting Score Cil...
Deleting Cell 0x0x804c9e8... Deleting Score Dan...
Deleting Cell 0x0x804ca08... Deleting Score Leo...
Deleting Cell 0x0x804c918... Deleting Score Max...
Deleting Cell 0x0x804c948... Deleting Score Ned...
Deleting Cell 0x0x804c948...

Figure 15.7: A call graph for the linear container program.
Figure 15.8: Terminating the linear container program.
Chapter 16: Abstract Classes and Multiple Inheritance

An abstract class declares a set of behaviors (function prototypes) that all of its descendents must follow (or implement). Documentation accompanying the abstract class must explain the purpose of each function and how each interacts with other parts of the class. Derived classes must implement all of the functions, and should obey the guidelines explained in the documentation.

16.1 An Abstract Class Defines Expectations

Abstract classes. When a base class includes even one prototype for a pure virtual function, it is an abstract class which cannot be used to create objects. However, such classes do have a purpose and they are important in the process of developing a large system. An abstract class lets us define and enforce a common interface, or behavior, for a set of related classes. Class derivation, combined with methods defined in the derived class(es) make the abstraction useful.

An abstract class specifies a set of virtual representation-dependent function prototypes for which definitions will be required in future derived classes. By doing so, it enables a large system to be developed in a top-down style.

The first development step is to define the major modules. The interface that each module provides is then written. The compiler will ensure that no necessary part of A is forgotten, and that all functions conform to the prototypes that were promised.

Large systems developed by teams of programmers are built this way. First, each major system component is identified. As soon as its role in the system is clear, its public interface can be defined in the form of an abstract class declaration. The abstract class forms a contract for the programming team that will develop the component, and different people (or teams) will work on different subsystems simultaneously.

Suppose a system designer has specified modules named A, B, and C. Since all the prototypes provided by module A are defined, programmers working on modules B and C can begin to write code that calls the functions of A, even before A is fully implemented. Derivation is used to connect the abstract interfaces to their implementation modules. When modules are complete, the subsystems can be easily integrated because the established prototypes guarantee that functions in one module will be able to call functions in another module that were programmed by a different person.

Definitions and rules.

- The opposite of abstract is concrete. A concrete class can have virtual functions, but all of those functions must have methods defined within the class itself or its ancestor classes.
- An abstract class cannot be instantiated, that is, used to construct any objects.
- Any class that has one or more pure virtual functions is called an abstract class.
- Abstract classes is polymorphic if more than one concrete class is derived from it.

16.2 Abstraction Example: an Ordered Type

Two abstract classes are used in this chapter's program: Container (from the previous chapter) and Ordered.

- Container defines the interface that should be presented by any container class: a way to put data into the container (put()), and find it when needed (get()), take it out of the container (remove()), write the contents of the container to a stream (print()).
Ordered defines prototypes for functions that are needed when you sort data items: comparison functions, sentinels, and a way to access the key field of the data.

---

```
// ------------------------------------------------------------------------
// Ordered base class -- An abstract class
// A. Fischer June 8, 1998 file: ordered.hpp

#ifndef ORDERED_H
#define ORDERED_H
#include <limits.h>
#include <iostream.h>

// ------------------------------------------------------------------------
class Ordered {
public:
  virtual ~Ordered(){}
  virtual KeyType key() const =0;
  virtual bool operator < (const KeyType&) const =0;
  virtual bool operator == (const KeyType&) const =0;
};
#endif
```

Specifying and enforcing requirements. The purpose of a container is to store a collection of items. The data stored in an item is not important; we use the class name Item as a representative of any kind of object that a containers might store. However, a few properties of an Item are essential for use with a sorted container:

- The Item must contain a key field and a key() function that returns the key.
- The operators < and == must be defined to compare two Items. Items will be compared using one or both of these operators. They will be stored in the container in ascending order, as defined by the operator <.
- A programmer who creates an Ordered class must supply the appropriate typedef for KeyType and appropriate definitions for the operators and sentinels that define the minimum and maximum possible values for a KeyType object.

The first two properties can be specified by defining an abstract class, Ordered, that gives prototypes (but no definitions) for the three required functions. We can enforce these requirements in a data class by deriving the data class from Ordered. When we do this, we instruct the compiler to guarantee that the derived data class does implement every function listed by Ordered. If one of the functions is missing, the compiler will give an error comment.

16.3 Multiple Inheritance

A class may be derived from more than one parent class. (We must #include the header files for each parent class.) The purpose of such multiple inheritance is:

- Simple form: to inherit properties from one parent and constraints from another.
- General form: inherit properties from two parent classes.

The syntax is a simple extension of ordinary derivation. We use it here (line 30) to combine the properties of the Exam class from the previous chapter with the abstract Ordered class. The new class is a wrapper for Exam that provides more functions than the original class but does not duplicate the ones that Exam supplies (Print() and operator<<).

16.3.1 Item: The Data Class

In the previous chapter, we used a typedef to make Item a synonym for the Exam class. In this chapter, we do more with Item: we use multiple inheritance to add constraints and functionality to the original Exam class.
16.3. MULTIPLE INHERITANCE

---

// Class declaration for data items.
// A. Fischer, May 29, 2001 file: item.hpp

#ifndef ITEM_H
#define ITEM_H
#include <limits.h>

typedef int KeyType;
#include "exam.hpp"
#include "ordered.hpp"

class Item : public Exam, public Ordered {
 public:
 static const KeyType max_sentinel = KeyType(INT_MAX);
 static const KeyType min_sentinel = KeyType(INT_MIN);

 Item(char* init, int sc): Exam(init, sc){}
 ~Item() { cerr <<"Deleting Item " << key() <<"\n"; }

 KeyType key() const { return Score; }
 bool operator==(const KeyType& k) const { return key() == k; }
 bool operator< (const KeyType& k) const { return key() < k; }
 bool operator< (const Item& s) const { return key() < s.key(); }

#endif

---

Figure 16.1: Inheriting both functionality and constraints.

Notes on the Item class. Two comparison operators and the key() function are required for Item because it was derived from Ordered. We define these three functions (lines 38–40) in such a way that Score (inherited from Exam) is the key field and the exams will be sorted in ascending order by Score. A third comparison function (line 41) is added for the convenience of client classes.

The Item class also defines two constants that are often needed for sorting algorithms: the maximum and minimum values of type KeyType. Line 24 tells us that KeyType is a synonym for int; the int values used here are supplied by the file <limits.h> in the standard library.

The Item constructor does nothing but pass its arguments through to the Exam constructor because it has no variables of its own that need initialization. The virtual destructor in the Ordered class is necessary to avoid warning comments in the Item class. For example, without that seemingly useless function, we get a warning comment about line 42:

item.hpp:42: warning:
'class Item' has virtual functions but non-virtual destructor

With these definitions, Item fulfills all the inherited obligations, the class is concrete and can be used...
normally. This class will be used with Linear and Cell from the prior chapter to build two new container classes: List and Priority Queue.

16.4 Linear Containers You Can Search

In this chapter we develop two new container classes from Linear. In a stack or a queue, all insertions and deletions are at one of the ends of the container; we never need to locate a spot in the middle. In contrast, a priority queue requires all insertions to be made in sorted order, and a simple list requires a search whenever an item is removed. To develop these classes in a general way, we assume that each Cell will contain an Item that is derived from Ordered, and we use the functions promised by Ordered to define three new functions in the Linear class:

\[
\begin{align*}
\text{bool Linear:: operator < ( Cell* cp )} & \{ \text{return (*cp->data < *here->data); } \\
\text{bool Linear:: operator < ( KeyType k )} & \{ \text{return *here->data < k; } \\
\text{bool Linear:: operator== ( KeyType k )} & \{ \text{return *here->data == k; } 
\end{align*}
\]

These functions allow us to search or sort a linear container according to the key field of the Item.

16.4.1 PQueue: a Sorted Linear Container

Notes on the PQueue code Items are deleted from a priority queue at the head of the list, just like an ordinary queue. Preparation for a deletion is easy because Linear provides the reset() function to position its pointers at the head of the list.

In a priority queue, items are inserted in priority order in the list and removed from the head of the list. To do the insertion, we must scan the list to locate the correct insertion spot: the item at prior should have higher priority (a higher key number than the new item) and the item at here the same or lower priority. The loop on lines 60–62 performs such a scan using the < operator and list traversal functions (reset(), end(), and ++) that are provided by Linear. Each reference to *this calls an operator defined by Linear and inherited by PQueue. When the right place is found, control is returned to Linear to do the actual insertion.

The only other functions here are a null constructor and a null destructor. Neither is necessary because the compiler will supply them by default. (Compare this class to List, below, in which the constructor and destructor have been omitted.) We write explicit functions because it is good style.
16.4.2 List: An Unordered Container

Notes on the List code  The List class provides a container with no special rules for insertion, deletion, or internal order. Since the order of items in the list does not matter, we use the easiest possible insertion method: insertion at the head, as in Stack. However, removing an item creates two new problems: how can we specify which item to remove, and how can we find it? The removal function required by Container does not have a parameter, but to remove an item from a List, we must know the key of the desired item. We solve the problem here by making the required focus() function interactive; it asks the operator to input a key. In a real program, the List class would probably have another public function that could be called with a parameter.

Once we know the key to remove, we search the list sequentially for a matching key. Since the list is unsorted, we must search the entire list before we know whether or not the key is in the list. If it is not, the pointer here will be NULL when we return from this function to Linear::remove(); the remove() will pass on the NULL it received to its caller, main().

```cpp
67 // ------------------------------------------------------------------------
68 // Unsorted list: derived from Container--->Linear--->List
69 // A. Fischer June 9, 2001 file: list.hpp
70 // ------------------------------------------------------------------------
71 #ifndef LIST_H
72 #define LIST_H
73 #include "linear.hpp"
74 #include "item.hpp"
75 #endif
76 // ------------------------------------------------------------------------
77 class List : public Linear {
78     public:
79         void insert( Cell* cp ) { reset(); Linear::insert(cp); }
80     // --------------------------------------------------------------------
81     void focus(){
82         KeyType k;
83         cout <<"What key would you like to remove? " ;
84         cin >> k;
85         for (reset(); !end(); ++*this) if (*this == k) break;
86     }
87 }
88 #endif
```

Figure 16.2: UML for all the Linear classes.
16.4.3 The Main Program

This main program has only one purpose: to test the two new classes. It puts meaningless data into two containers, takes some back out, and prints the results. The test results shown on the next page prove that insertions into the PQueue are implemented in the correct order, and that removal from the middle of the list works correctly. Other test runs verified that removal from head and tail of list, and attempted removal of a key that did not exist, also work properly. Trace comments after termination show that 9 cells (7 data cells and 2 dummy headers) and 7 Items were deleted by the destructors.

The output.

Print the empty List L.
<[
]
Putting 3 items onto List L: 99, 77, 88.
<[
Cell 0x0x33650 [Max: 18 , 0x33630]
Cell 0x0x33630 [Leo: 37 , 0x33610]
Cell 0x0x33610 [Ned: 29 , 0x0]
]> 
Putting 3 items onto PQueue P: 22, 11, 44.
<[
Cell 0x0x336b0 [Dan: 44 , 0x33670]
Cell 0x0x33670 [Bea: 22 , 0x33690]
Cell 0x0x33690 [Ali: 11 , 0x0]
]> 
Remove one item from L and queue on P. 
What key would you like to remove? 37
16.5. C++ Has Four Kinds of Casts

16.5.1 Static Casts

A static cast is an ordinary type conversion. It converts a value of one type to a value with approximately the same meaning in another type. The conversion can be a lengthening (short to long), a shortening (int to char), or a change in representation (float to int). The C and C++ languages support the built-in type conversions shown in Figure 16.5.1. These are called “static” casts because the compiler finds out that they are needed at compile time and generates unconditional conversion code at that time.

```
char          int              * int is used if it can represent
short int    * unsigned int all values of the original type.
int bit field

any type integer  float    double    long double
```

Figure 16.3: Built-in type conversions in C and C++.

**Explicit casts.** A static cast can be called explicitly using ordinary C syntax. In addition, C++ has two new ways to call a cast:

```
int k, m, *ip;
float f, *fp;
f = (float)k;                  // traditional C syntax.
```
f = float(k); // function-call syntax.
f = static_cast<float>(k); // explicit C++ syntax.

Coercion. Coercion, or automatic type conversion, happens when a function call is encountered, and the type of an argument does not match the declared type of its parameter. In this case, the argument will be converted to the parameter type, if the compiler has a method for doing so. Coercion is also used to make operands match the type-requirements of operators. In C, coercion is limited to primitive types. However, in C++, it can also apply to a class type, say Cls:

- If an object of type T is used where a Cls object is needed, and the class Cls contains a constructor with one parameter of type T, the constructor will be used to coerce the T value to a value of type Cls.

- If an object of type Cls is used where a T object is needed, and the class Cls contains a cast operator whose result is type T, the cast operator will be used to coerce the Cls object so that the context makes sense.

16.5.2 Reinterpret Casts

A reinterpret cast is a type trick performed with pointers. It relabels the pointer’s base type without changing any bits of either the pointer or its referent. This is like putting lamb’s clothing on a wolf. Using a reinterpret cast, a program can access a value of one type using a pointer of a different type, without compiler warnings. (See Figure 16.5.2, line 147.) This lets us perform nonsensical operations such as adding incompatible values. For example, the reinterpret cast in the following program lets us relabel the integer 987654321 as a float, then add 1.0 to it to produce garbage:

140 #include <iostream.h>
141 using namespace std;
142 // --------------------------------------------------------------- File: "assign.cpp"
143 int main( void )
144 {
145   int   my_int = 987654321;
146   int*  p_int = & my_int;
147   float* p_float = (float*) p_int;
148   float answer = *p_float + 1.0;
149   cerr <<answer <<"\n";
150 }

The answer is garbage: 1.0017, not 987654322 because The integer value and the floating point 1.0 were added to each other without any type conversion. The primary applications for reinterpret casts are hash functions and input conversion functions like strtod() and strtol().

Alternative syntax. Ordinary C syntax or C++ syntax with angle brackets can be used to invoke a reinterpret cast:

\[ fp = (float*)p_int \] // ordinary C syntax.
\[ fp = reinterpret_cast<float*>(ip); \] // explicit C++ syntax.

16.5.3 Const Casts

A const cast provides a way to remove the const property from a pointer variable just long enough to change the value of its referent. This lets us use a constructor (or any class function) instead of a ctor to initialize a const class member.
#include <iostream.h>

int main( void )
{
    int w = 99;
    const int* cip = &w;
    cout << " &w is " << &w << " referent of cip is " << cip << endl;
    cout << " w= " << w << " *cip= " << *cip << endl;
    * const_cast<int*>(cip) = 33;
    //*cip = 33;
    cout << " w= " << w << " *cip= " << *cip << endl;
}

When we write *cip = 33; without a const cast, we get a const violation error:

    const.cpp: In function 'int main ()':
    const.cpp:165: assignment of read-only location

With a const cast, we are permitted to change the location and we get output:

    &w is 0xbffff714 referent of cip is 0xbffff714
    w= 99  *cip= 99
    w= 33  *cip= 33

## 16.5.4 Dynamic Casts

Dynamic casts are used with polymorphic classes, and can cast either pointers or references. Dynamic casts can move either upward or downward on the derivation tree. In Figure 16.6 a cast from B to D would be a downward cast, a cast from B to A would be an upward cast.

Suppose class D is derived from class B, as in the figure, and suppose p is a pointer (or a reference) to an object of class D. Then we can use a dynamic cast to relabel p as a B pointer (or reference). An upward dynamic cast (from D to B) is always meaningful because any derived-class object includes a base-class object as part of itself.

According to my compiler, no downward casts are permitted at all. According to the Schildt text, some downward casts are permitted, but they are more complex and require a run-time check. Here is what Schildt says: If a pointer, p, has type A* it could be pointing at an object of any one of the four types A, B, C, or D. A down-cast of p from A* to B* would be meaningful if p’s referent were actually type B or D, because these classes actually have all the members required by type B. However, such a down-cast would not be meaningful if p’s referent were actually type A or C because some of B’s members would be missing. It could cause a run-time crash if such a cast were permitted. To prevent such problems, the legality of every down-cast is checked at run time. A bad pointer down-cast will return a NULL result. A bad reference down-cast will throw an exception.

### When do I use a dynamic-cast?

You don’t need to use an explicit dynamic cast to move up a derivation tree. The dynamic cast is done automatically whenever you use a derived-class object in a base-class context, or set a base-class pointer to point at a derived-class object. For this reason, it is hard to find a good use for an explicit dynamic cast in a simple program.

The rules for using dynamic casts are complicated by private parts, private derivation, and multiple inheritance. Some examples are given in the next section to illustrate these issues and show what kind of dynamic casts are and are not permitted in a multiple-inheritance situation.

## 16.6 Virtual Inheritance and Dynamic Casts

As long as derivation is used singly—so that a derived class has only one parent with data members—inheritance works smoothly, the storage model is easy to implement, and it is not hard to understand how it works. However, when a class inherits data members from two parents, two serious problems can occur:
The `this` pointer points at the beginning of the entire object, consisting of the parts of the first parent, followed by the parts of the second, and finally, the parts of the derived class. To apply a function inherited from the second class, the compiler must compute where `this` should point for that part of the object. Dynamic casts are related to this problem.

A class could inherit the same grandparent from two parents. If the grandparent has data members, are two copies of each inherited? Virtual inheritance exists to prevent this. When used in this way, the word `virtual` has nothing to do with virtual functions.

16.6.1 Virtual Inheritance

The keyword `virtual` can be used in a derivation declaration to prevent inheriting the same members from two parents. It is only relevant when multiple inheritance will be used, and when there is more than one path through the UML diagram from an ancestor class that has data members to some derived class. A class may have both virtual and nonvirtual base classes. When virtual derivation is used, an object of a derived class will have exactly one copy of the members of each ancestor. If derivation is not virtual, each derivation path will produce its own copy of any common ancestor, resulting in two data members with the same name in the derived-class object.

The simplest situation that illustrates the rules for virtual derivation is a class hierarchy in the shape of a “donut”. The following set of four classes creates the “donut” shown in Figure 16.5. Each class has either one or two data members, a constructor, and a `dump` function. Data is made public to make it easier to show what is going on.

The data diagrams on the right show how storage would be allocated for objects of each of the four classes. The data member of A is inherited by classes B and C and becomes the first member of objects bb and cc. class D inherits all of the data members of B and all of the data members of C. Nonetheless, dd has only one sub-object of class A, because virtual derivation was used to derive B and C from A.

Naming rules. Two rules govern the meaning of a name in a donut situation:

- A class can inherit two members with the same name from different parent classes. When this happens, the name is `ambiguous` and you must use the `::` to denote which one you want. For example, within class D, you would write B::a to refer to the member named `a` inherited from B or C::a for the member inherited from C.
- If there are more than three levels in an inheritance hierarchy, the virtual property must be redeclared at each level that has multiple inheritance.

Figure 16.5: Double (donut) inheritance.
16.6. VIRTUAL INHERITANCE AND DYNAMIC CASTS

// ------------------------------------------------- File: "donut.hpp"
#ifndef DONUT_H
#define DONUT_H
#include <iostream>
#include <iomanip>
using namespace std;
class A { //----------------------------------------- Grandparent Class
  public:
    double x;
    A(): x(11.1) {}
    virtual ~A(){};
    virtual void dump(){ cerr <<" A::x = " << x <<"\n"; }
  }

class B: virtual public A { //----------------------- First Parent of D
  public:
    int a;
    double x;
    B(): a(20), x(22.2) {}
    virtual ~B(){};
    virtual void dump(){
      A::dump();
      cerr <<" B::x = " << x <<" B::a = " << a <<"\n"; }
  }

class C: virtual private A { //--------------------- Second Parent of D
  public:
    int a;
    C(): a(30) {}
    virtual ~C(){};
    virtual void dump(){ A::dump(); cerr <<" C::a = " << a <<"\n"; }
  }

class D: public B, public C { //---------------------- Grandchild Class
  float f;
  public:
    D(): f(44.4) {}
    void dump(){
      A::dump();
      B::dump();
      C::dump();
      cerr <<" D::f = " << f <<"\n"; }
  }
#endif

- A is a parent class of B, and both define a member named x. In classes A and C, only A::x is visible, so writing x in these contexts will always mean A::x. For functions in classes B and D, both A::x and B::x are visible, but B::x dominates A::x because it is “closer” to these classes. The dominant member will be used when x is written without the double colon. To refer to the non-dominant member, the full name A::x must be used.

The following brief program shows how the naming works and how visibility interacts with private inheritance. Two lines are commented out because they caused compilation errors:

- Line 204 produced this compiler error:
  
  double A::x is inaccessible within this context in 'C' due to private inheritance.

- Line 210 was ambiguous. Neither B::a nor C::a is dominant here because the two classes B and C are equally close ancestors of D. To use either one in class D, we must qualify the member name as in lines 208 and 209.
CHAPTER 16. ABSTRACT CLASSES AND MULTIPLE INHERITANCE

Here is the output:

&cc.a = 0xbffff864 value= 30
&dd.x = 0xbffff850 value= 22.2
&dd.A::x = 0xbffff840 value= 11.1
&dd.B::a = 0xbffff84c value= 20
&dd.C::a = 0xbffff834 value= 30

16.6.2 Dynamic Casts on the Donut

The final program in this chapter demonstrates dynamic casts and polymorphism in the context of our donut-shaped derivation tree.

Here is the output:

&cc.a = 0xbffff864 value= 30
&dd.x = 0xbffff850 value= 22.2
&dd.A::x = 0xbffff840 value= 11.1
&dd.B::a = 0xbffff84c value= 20
&dd.C::a = 0xbffff834 value= 30

16.6.2 Dynamic Casts on the Donut

The final program in this chapter demonstrates dynamic casts and polymorphism in the context of our donut-shaped derivation tree.

Here is the output:

&cc.a = 0xbffff864 value= 30
&dd.x = 0xbffff850 value= 22.2
&dd.A::x = 0xbffff840 value= 11.1
&dd.B::a = 0xbffff84c value= 20
&dd.C::a = 0xbffff834 value= 30

#include "donut.hpp"
int main( void )
{
    A aa; // Data member: x
    C cc; // Data members: A::x, C::a
    D dd; // Data members: A::x, B::a, B::x, C::a, D::f
    cerr <<"Dumping aa\n"; aa.dump();
    cerr <<"Dumping cc\n"; cc.dump();
    cerr <<"Dumping the B part of dd\n"; dynamic_cast<B*>(&dd)->dump();
    // Upward casts.  ----------------------------------------------------------
    B* bp = &dd; // Explicit Low->high pointer cast not needed.
    B& br = dd; // Explicit Low->high reference cast not needed.
    cerr <<"Dumping br\n"; br.dump(); // Use the reference variable to dump.
    A* ap = dynamic_cast<A*>(bp); // Upward cast IS OK if derivation is public.
    cerr <<"Dumping ap\n"; ap->dump(); // Show result of dynamic cast.
    //dynamic_cast<A*>(&cc)->dump(); // Upward cast NOT OK if derivation is private.
    // Downward casts.  --------------------------------------------------------
    cerr <<"Dumping dp after up and down casts, D->B->A->D .\n";
    D* dp = dynamic_cast<D*>(bp); // Can dynamic_cast downward where type is true.
    dp->dump(); // Start with D and return to a D.
    cerr <<"Dumping cp after up and down casts, D->B->A->C .\n";
    C* cp = dynamic_cast<C*>(ap); // Can dynamic_cast downward where type is true.
    cp->dump(); // A D object has all the parts of a C object.
    cerr <<"Dumping after down casts, A->C .\n";
    ap = &aa;
    //cp = dynamic_cast<C*>(&aa); // Cannot dynamic_cast down to wrong type.
    cp = dynamic_cast<C*>(ap); // Cannot dynamic_cast down to wrong type.
    cerr <<"No exception was thrown.\n";
    cp->dump();
}
Polymorphism. The file donut.hpp defines a base class named A with two derived classes B, and C, and a class D derived from both B and C. We can create a donut diagram like this with or without polymorphism; a class only becomes polymorphic when it has virtual functions. In this example, Classes A, B, and C are polymorphic because they have virtual \texttt{dump()} functions. This forces us to also define virtual destructors.

Class D is the end of the derivation chain and the last class in the polymorphic family of classes. It is last because its derivation is not virtual and its functions are not virtual. Because these properties end in class D, it should not be used for further derivation.

A class is polymorphic if it has even one virtual function. If a class is polymorphic or is derived from a polymorphic class, the true type of every class object must be stored as part of the object at run time. I call this a “type tag”. Whenever a virtual function is called, this type tag is used to select the most appropriate method for the function. If class is not part of a polymorphic family, no run-time type tag is attached to its objects, and no run-time function dispatching happens.

Virtual derivation

- Lines 216 through 222 create and print three variables. The output is:

```
 Dumping aa
 A::x = 11.1
 Dumping cc
 A::x = 11.1
 C::a = 30
 Dumping the B part of dd
 A::x = 11.1
 B::x = 22.2  B::a = 20
```

- In this example, both B and C are derived virtually from A. If we omit the “virtual” from either one of the lines 163 and 174, or from both, we get this error comment.

```
donut.hpp: In method ‘void D::dump()’:
donut.hpp:37: cannot convert a pointer of type ‘D’ to a pointer of type ‘A’
donut.hpp:37: because ‘A’ is an ambiguous base class
```

Upward dynamic casts.

- Lines 225 and 226 perform implicit upward dynamic pointer and reference casts from class D to class B. Line 227 shows how to use the reference variable. The output is:

```
 Dumping br
 A::x = 11.1
 B::x = 22.2  B::a = 20
```

- On line 222, an explicit cast was used because it was simplest. I tried an implicit cast there but it had the wrong precedence in relation to the \texttt{->} operator: \texttt{(B*)(&dd)->dump()}

- Line 229 shows another explicit dynamic cast, and line 230 prints the result. The output is:

```
 Dumping ap
 A::x = 11.1
```

- Line 231 is commented out because it caused a privacy error:

```
Error in function ‘int main()’ of donut.cpp:
donut.cpp:20: dynamic_cast from ‘C’ to private base class ‘A’
A* ap = dynamic_cast<A*>(&cc);
```

Polymorphic downward dynamic casts.

- Lines 235, 239, 244, and 245 do explicit downward dynamic casts. If Classes A, B, and C did not have virtual functions, all of these lines would cause compile-time errors. The dynamic down-cast uses the run-time type information stored with every polymorphic object. But when a class is not polymorphic, the type tag is not there, and a dynamic down-cast cannot be done. Here is the error comment:

```
donut.cpp:24: cannot dynamic_cast ‘ap’ (of type ‘class A *’) to type ‘class C *’
```
Because donut.hpp does defines a polymorphic class, lines 235, 239, 244, and 245 compile. The first of these lines is fine: we started with an object of type D, up-cast it, then down-cast it again. On line 235, we finished with class D, where we started. Clearly, every step in this casting-process was meaningful. The output is not surprising:

```
Dumping dp after up and down casts, D->B->A->D.
A::x = 11.1
A::x = 11.1
B::x = 22.2  B::a = 20
A::x = 11.1
C::a = 30
D::f = 44.4
```

On Line 239, we downcast the same object to class C, which is also meaningful, since every D object contains a C object as part of itself. Even though the program now thinks it has a C object, the object itself retains its true type identity, and when we dump it, we get class D’s version of dump, just as we did on line 236.

```
Dumping cp after up and down casts, D->B->A->C.
A::x = 11.1
A::x = 11.1
B::x = 22.2  B::a = 20
A::x = 11.1
C::a = 30
D::f = 44.4
```

Line 244 is commented out because it gives a warning message:

```
donut.cpp:33: warning: dynamic_cast of ‘class A aa’ to ‘class C *’ can never succeed
```

This warning happens because aa is an object, not a pointer, and we know that it does not have all the members that a C object needs. No casting magic can create the missing parts.

On line 245, we down-cast an A* instead of an A&. There is no warning here, because the compiler cannot predict the actual type of an object that an A* pointer might point at. The result is a run-time malfunction:

```
Dumping after down casts, A->C.
No exception was thrown.
Bus error
```

The Schidt text says that this downcast should cause an exception to be thrown, and since this program does not attempt to catch exceptions, the program should be terminated. (Exceptions are covered in Chapter 17.) Clearly, since control reached line 246, this did not happen. The compiled code did not check for an illegal downcast; it performed it. The result was a bus error (segmentation error on a second machine) when the print function tried to access a member of the object that never existed.
Chapter 17: Exceptions

From the board game MONOPOLY, the rule to follow when your man lands on the “illegal” square:

Go to jail. Go directly to jail, do not pass GO and do not collect $200.

17.1 Handling Errors in a Program

Function calls, loops, conditionals, switches, and breaks permit the programmer to control and direct the sequence of evaluation of a program and modify the default order of execution, which is sequential. These statements permit controlled, local perturbations in the order of execution. Almost all situations that arise in programming can be handled well using some combination of these statements.

The goto statement is also supported in C, but its use is discouraged because it makes uncontrolled, non-local changes in execution sequence. Use of a goto is even worse in C++; if it is used to jump around the normal block entry or exit code, the normal construction, initialization, and deletion of objects can be short-circuited, potentially leaving the memory in an inconsistent state. Therefore, this statement should simply not be used.

There are some situations in which the ordinary structured control statements do not work well. These involve unusual situations, often caused by hardware or input errors, which prevent the program from making further fruitful progress. In C++, a failure during execution of a constructor is one situation that calls for use of exceptions.

In the old days, these situations would be handled by a variety of strategies:

1. Do nothing. Don’t check for errors; hope they don’t happen. Let the program crash or produce garbage answers if an error happens.

2. The program could identify the error and call an error function. However, this defers the problem instead of solving it. The error function still must do something about the error.

3. Identify the error, print an error comment, and call exit(). (This is equivalent to using the fatal() function in the toolspp library.) However, aborting execution is not permissible in many real-life situations. For example, aborting execution of a program that handles bank accounts could leave those accounts in an inconsistent or incorrect state.

4. One could use assert() to check for errors. This is similar to option (1) but worse because it gives no opportunity to print out information about the error or its cause.

5. The function being executed could return with an error code. The function that called it would need to check for that error code and return to its caller with an error code, and so on, until control returned to the level at which the error could be handled.

This method usually works but distorts the logic of the program and clutters it with a large amount of code that is irrelevant 99% of the time. Using this technique discourages use of functions and modular code.

6. A long-distance goto could be used to return to the top level. Using this strategy, any information about the error would have to be stored in global variables before executing the goto. This is like programming in BASIC. Use of this control pattern destroys the modularity that C programs can otherwise achieve. It is not recommended.
17.1.1 What an Exception Handler Can Do

An exception handler provides an additional control option that is designed for dealing with unusual situations (exceptions) in which the ordinary structured control statements do not work well. (Exceptions can be used in non-error-conditions, but should not be used that way.) An exception handler lets control go . . .

- From the level at which an error is discovered, often deep within the code, at a stage when several functions have been called and have not yet returned . . .
- Carrying as much information as necessary about the error . . .
- To the level at which the error can be handled, often in the main function or a high-level function that handles the overall progress of the program or operator interaction . . .
- And back into normal execution, if that is appropriate.

An exception handler and the exceptions it can handle are defined at a high level. When an exception condition is identified at a lower level, an exception object is created and “thrown” upward. This causes immediate exit from the originating function, with proper clean-up of its stack frame. The function that called it then “has the ball” and can either catch the exception or ignore it (ignoring the exception causes an immediate return to its caller). Thus, control passes backward along the chain of function calls until some function in the chain-of-command “catches” the exception. All stack frames in this chain are deleted and the relevant destructors are run. The programmer should be careful, therefore, not to allow exceptions to be thrown when some objects are half-constructed.

Implementing a compiler and run-time-system that can do this efficiently is a hard problem. Exceptions can come in many types and each type of exception must have a matching handler. An exception is an object that may have as many fields as necessary to contain all the important facts about where and why the exception was thrown. This object must outlive the function that created it (it must be created in dynamic storage, not on the stack) and it must carry an identifying type-tag, so it can be matched to the appropriate exception handler. An uncaught exception will abort a process.

What next? After catching an exception, there are several options for handling it:

1. The catcher can clean up the data and execute return with some appropriate value.
2. The catcher can clean up the data and call the containing function recursively.
3. The catcher can clean up the data and continue from the line that follows the exception handler (not the line that follows the function call that caused the exception).
4. The catcher can comment, get more information from the operator, and carry on in either of the two preceding ways.
5. The catcher can abort the process or return to its caller, as in C.
6. The catcher can fix some data fields and rethrow the same exception or some other exception.

17.2 Defining Exceptions in C++

One major principle applies to the use of exceptions: they are not a substitute for proper use of else or while, so the function that throws an exception should not be the function that catches it. Exceptions are intended for global, not local, use. Their proper use is to enable control and information to pass, directly, across classes and through a chain of function calls.

17.2.1 Defining an Exception Type

An exception is an object and its type is defined like any class. It can (but usually does not) have public, protected, and private parts. The exception class definition can be global or it can be contained within the definition of another class. Related exceptions can be created by derivation. For example, the Bad, BadSuit, and
BadSpot classes, below, define three exception types that might be used in a program that plays an interactive
card game and reads input from the keyboard. The class Bad is a base class for the others and defines the
functionality common to all three classes. The UML diagram in Figure 17.1 shows the derivation hierarchy. The
purple circle with a V marks a virtual function.

```
//===========================================================================
// Exception Classes for Playing Card Errors. file: bad.hpp
// Exception demonstration program: March 2009
#pragma once
#include "tools.hpp"

class Bad {
public:
    char spot;
    char suit;
    //---------------------------------------------- Suit and Spot both wrong.
    Bad (char n, char s) : spot(n), suit(s) {};
    virtual ~Bad(){}
    virtual void print(){
        cerr <<" Both spot value and suit are wrong\n"
        <<" Legal spot values are 2..9, T, J, Q, K, A\n"
        <<" Legal suits are H D C S\n";
        pr();
    }
    void pr(){
        cerr <<" You entered "<<spot <<" of " <<suit
        <<". Please reenter. \n";
    }
};

// --------------------------------------------------- Only the suit is wrong.
class BadSuit : public Bad {
public:
    BadSuit (char n, char s) : Bad(n, s) {}
    virtual ~BadSuit(){}
    virtual void print(){
        cerr <<" Legal suits are H D C S\n";
        pr();
    }
};

// ---------------------------------------------- Only the spot value is wrong.
class BadSpot : public Bad {
public:
    BadSpot (char n, char s) : Bad(n, s) {}
    virtual ~BadSpot(){}
    virtual void print(){
        cerr <<" Legal spot values are 2..9, T, J, Q, K, A\n";
        pr();
    }
};
```

17.2. Exception Specifications

In Java, every class must declare the exceptions that might happen within it, unless they are also caught within
it. This requirement is confusing to beginners, but it helps make Java programs more predictable and it leads
to better-informed error handling. In C++, a function may declare a list of potential exceptions, or declare
that it does not throw exceptions. However, such a declaration is not required. Since an uncaught exception
will terminate a program, this can be an important kind of documentation. Note: this is a new feature of C++
and may not be supported by all compilers.

The exception specification follows the parameter list in a prototype and precedes the semicolon. For
example, suppose a function named divide(int, int) could throw two kinds of exceptions. The declaration
would be:
CHAPTER 17. EXCEPTIONS

Figure 17.1: UML diagram for the exception classes.

\begin{verbatim}
// A playing card class and related exception classes.
// Alice E. Fischer, March 2009
#pragma once
#include "tools.hpp"
#include "bad.hpp"

enum SuitType{ spades, hearts, diamonds, clubs, bad };

class Card {
    int  spot_;
    SuitType suit_;
public:
    Card (istream& sin) throw (Bad, BadSpot, BadSuit);
    int  spot(){ return spot_; }
    SuitType suit(){ return suit_; }
    ostream& print(ostream&);
static void instructions(ostream&, int n);
};
\end{verbatim}

We can also declare that a function does not throw exceptions at all. The syntax is:

\begin{verbatim}
int safeFunction (int, int) throw ();
\end{verbatim}

Technically, this declaration is legal even if some function called by safeFunction throws exceptions. However, since we can’t know (let alone control) which library functions throw exceptions, this kind of declaration has questionable value.

17.2.3 Playing card demo.

Next is a simple class that models a playing card. We will use it to demonstrate throwing and catching exceptions. The constructor for this class takes input from the keyboard, a notoriously error-prone data source. Each card is represented by two chars representing the face value (spot) and the suit of the card. Spaces and capitalization are not important and can be used freely. After reading the two chars, the Card constructor validates them and (possibly) throws an exception. The nature of the input error is encoded in the name of the exception, enabling the program to use virtual functions to display a specific and appropriate error comment each time.
17.2.4 Throwing an Exception

The `throw` statement is used to construct an exception and propagate it. One type of exception is commonly used without definition; a function may throw a literal string (type `std::string` or `char*`). Throwing an exception causes control to pass backwards through the chain of function calls to the nearest previous catch clause that handles that particular type of exception. The destructors will be run for all objects in the stack frames between the throw and the catch.

```cpp
68 ////////////////////////////////////////////////////////////////////////////////////////////////
69 // Functions and constants for the Card class.
70 // Alice E. Fischer, March 2009 cards.cpp
71 ////////////////////////////////////////////////////////////////////////////////////////////////
72 #include "cards.hpp"
73 const char* suitlabels[5] = {"spades", "hearts", "diamonds", "clubs", "bad"};
74 const char* spotlabels[16] = {
75    "bad", "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
76    "10", "Jack", "Queen", "King", "Ace"
77    
78 }; 
79
80 // --------------------------------------------------------------------------
81 void Card::instructions(ostream& out, int n) {
82    out << "Please enter " << n << " cards.
"
83       << "Spot codes are 2..9, T, J, Q, K, A \n"
84       << "Suit codes are S H D C \n"
85 }
86
87 // --------------------------------------------------------------------------
88 Card::Card (istream& sin) throw (Bad, BadSpot, BadSuit) {
89    char inspot, insuit;
90    sin >> inspot >> insuit;
91    if (!sin.good()) throw "Low level read error\n";
92    if (inspot >='2' && inspot<='9') spot_ = inspot - '0';
93    else switch( toupper(inspot) ){
94        case 'T': spot_ = 10; break;
95        case 'J': spot_ = 11; break;
96        case 'Q': spot_ = 12; break;
97        case 'K': spot_ = 13; break;
98        case 'A': spot_ = 1; break;
99        default : spot_ = 0;
100    }
101    switch( toupper(insuit) ){
102        case 'S': suit_ = spades; break;
103        case 'H': suit_ = hearts; break;
104        case 'D': suit_ = diamonds; break;
105        case 'C': suit_ = clubs; break;
106        default : suit_ = bad;
107    }
108    if (spot_ == 0 & suit_ == bad) throw Bad(inspot, insuit);
109    if (spot_ == 0) throw BadSpot(inspot, insuit);
110    if (suit_ == bad) throw BadSuit(inspot, insuit);
111 }
112
113 // --------------------------------------------------------------------------
114 ostream&
115 Card::print(ostream& sout) {
116     return sout <<spotlabels[spot_] << of " <<suitlabels[suit_] <<endl;
117 }
```

In this program, line 91 throws a string exception. This will be caught by the general exception handler on line 153. Lines 108–10 throw exceptions from the Bad hierarchy. These will be caught on line 146. The names of
these exceptions are announced on line 88. (This is a worthy but optional form of program documentation. It is not clear how much checking, if any, compiler does with these declarations.)

17.2.5 Catching an Exception

```cpp
#include "tools.hpp"
#include "cards.hpp"
#include "bad.hpp"
define NCARDS 3

int main( void )
{
    Card* hand[NCARDS];
    int k;
    bool success;
    Card::instructions( cout, NCARDS );
    //------------------------------ Here is the single line of active code.
    try {
        success = false;  // Will not be changed if an exception happens.
        //------------------------------ Input one card.
        try {
            cout << "Enter a card (spot code, suit code): 
            hand[k] = new Card(cin); // Input one card.
            success = true; //-- No exception - we have a good card.
            cout << " Card successfully entered into hand: ";
            hand[k]->print(cout);
            ++k;
        }
        //-------------- Check for the three application-specific exceptions.
        catch (Bad& bs) { bs.print(); } // Catch all 3 Bad errors.

        //------------ Now check for general exceptions thrown by system.
        catch (bad_alloc bs) { //-------------- Catch a malloc failure.
            cerr << " Allocation error for card #" <<k <<".\n";
            return 1;
        }

        catch (...) {
            //-------------- Catch everything else.
            cerr << " Last-ditch effort to catch exceptions.\n";
        }
        //---------- Control comes here after the try/catch is finished.
        if(!success) delete hand[k];  // Delete the half-made object.
    }
    cout << "Hand is complete:"
    for (k = 0; k < NCARDS; ++k) { hand[k]->print( cout ); }
}
```

Exceptions are caught and processed by exception handlers. A handler is defined like an ordinary function; the type of its parameter is the type of exception that it will catch.

Code that may generate exceptions (at any nesting level) and wishes to catch those exceptions must be enclosed in a `try` block (lines 137..144). The try block can contain multiple statements of any and all sorts. The exception handlers are written in `catch` blocks that immediately follow the `try` block (lines 146..155). Several things should be noted here:

1. The fields inside an exception object may be used according to the normal rules of public and/or private access. The data can be public because there is no need to protect it.

2. If the handler does not need to access the information in the exception, the parameter name may be omitted.
3. The order in which the handlers are written is important; the general case must come after all related specific cases.

4. A base-class exception handler will catch exceptions of all derived types. When an exception of a derived type is caught, the fields that belong to the derived type are not "visible" to the handler. If the exception is rethrown, all fields (including the fields of the derived type) are part of the rethrown object.

5. An exception handler whose parameter is . . . will catch all exceptions.

In this example, line 139 calls the Card constructor, which can throw exceptions from the Bad class and its derived classes. The handler for those exceptions is on line 146. This line will catch all three kinds of exceptions and process them by calling the print function in the exception class.

Processing the exception. When an exception is caught by line 146, we call the bs.print() to process it. Note that the print function is virtual in the Bad class hierarchy, and has three defining methods. When bs.print() is called, the system will inspect the run-time type-tag attached to bs to find out which class or subclass constructed this particular exception. Then the print() function of the matching class will be called. This is how we get three different kinds of output from one catch clause. (Note the first, second, and third blocks of error comments in the output transcript, below.) If Bad::print() were not virtual, the base-class print() function would always be used.

Output The output that follows shows two sample runs of this program with different exception handlers active. The first output is from the program as shown:

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Both spot value and suit are wrong
  Legal spot values are 2..9, T, J, Q, K, A
  Legal suits are H D C S
  You entered m of n. Please reenter.

Enter a card (spot code, suit code): 2m
  Legal suits are H D C S
  You entered 2 of m. Please reenter.

Enter a card (spot code, suit code): 1s
  Legal spot values are 2..9, T, J, Q, K, A
  You entered 1 of s. Please reenter.

Enter a card (spot code, suit code): 2s
  Card successfully entered into hand: 2 of spades

Enter a card (spot code, suit code): kh
  Card successfully entered into hand: King of hearts

Enter a card (spot code, suit code): JC
  Card successfully entered into hand: Jack of clubs

Hand is complete:
2 of spades
King of hearts
Jack of clubs
```

The following output was produced after commenting out the first catch clause (line 146).

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Last-ditch effort to catch exceptions.

Enter a card (spot code, suit code): 3s
  Card successfully entered into hand: ... and so on.
```
17.2.6 Built-in Exceptions

Simple objects like integers and strings can also be thrown and caught. In addition, C++ has about a dozen built-in exceptions (see Schildt, pages 922-924). One of these, bad_alloc changes the way we write programs. In C, it was necessary to check the result of every call on malloc(), to find out whether the system was able to fulfill the request. In C++, such a check is not necessary or helpful. If there is an allocation failure, the run-time system will throw a bad_alloc exception and control will not return to the failed call. If your program does not have a handler for such exceptions, and one occurs, the program will be terminated.

The bad_cast exception is used when a program executes an invalid downward dynamic cast, and bad_exception is thrown when a function violates its own exception specification.

One group of standard exceptions called runtime_errors are beyond the programmer’s control and are used for mistakes in library functions or the run-time system. These include overflow_error, range_error, and underflow_error.

A final group of exceptions called logic_errors are defined by the standard but, so far as I know, not used by the system. They seem to be intended for use by any program when a run-time error is discovered. These include domain_error, invalid_argument, length_error, and out_of_range.

17.2.7 Summary

In the hands of an expert, exception handlers can greatly simplify error handling in a large application. However, they are not easy to use and using them without understanding is likely to lead to errors that are difficult to diagnose and cure. They are also difficult for a compiler to handle and force the compiler to interact with the host system in awkward ways. Virtual print functions are a powerful and simple tool to produce good error comments in situations where multiple faults can occur. Be aware, though, that processing a call on a virtual function takes more time than processing a non-virtual function in the same class.
Chapter 18: Design Patterns

Design patterns are elegant, adaptable, and reusable solutions to everyday software development problems. Each pattern includes a description of a commonly occurring type of problem, a design for a set of classes and class relationships that solve that problem, and reasons why the given solution is wise.

18.1 Definitions and General OO Principles

18.1.1 Definitions

1. Subclass: X is a subclass of Y if X is derived from Y directly or indirectly.

2. Collaboration: two or more objects that participate in a client/server relationship in order to provide a service.

3. Coupling: A dependency between program elements (such as classes) typically resulting from collaboration between them to provide a service. Classes X and Y are coupled if...
   - X has a function with parameter or local variable of class Y.
   - X has a data member that points at something of class Y.
   - X is a subclass of Y.
   - X implements an interface for class Y (Y gives friendship to X).

[Example:] If the Key class calculates the hash-table index, it must know the length of the hash table. But this couples two classes that would not otherwise be coupled.

4. Cohesion: This is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion is a “specialist” with narrow power.

5. System event: A high-level event generated by an external actor; an external input event. For each system event, there is a corresponding operation. For example, when a word-processor user hits the “spell check” button, he is generating a system event indication “perform spell check”.

6. Use case: The sequence of events and actions that occur when a user participates in a dialog with a system during a meaningful process.

18.2 General OO Principles

1. Encapsulation. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another.

2. Narrow interface. Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. [Why] This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.

3. Delegation: a class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert. [Example:] HashTable::find selects one list and delegates the searching task to List::find.
18.3 Patterns

A pattern is a design issue or communication problem... with a solution based on class structure... and guidance on how to apply the solution in a variety of contexts.

18.3.1 GRASP: General Responsibility Assignment Software Patterns

- High cohesion is desirable. [Why?] It makes a class easier to comprehend, easier to maintain, and easier to reuse. The class will also be less affected by change in other classes. [Example:] Cohesion is low if a HashTable class contains code to extract fields from a data record. Cohesion is low if the data class computes a hash index. Cohesion is high if each class does part of the process.

- Low coupling is desirable. Which class should be given responsibility for a task? [A] Assign a responsibility so that its placement does not increase coupling. [Why?] High coupling makes a class harder to understand, harder to reuse, and harder to maintain because changes in related classes force local changes.

- Expert. Who should do what? [A] Each class should do for itself actions that involve its data members. Each class should “take care of” itself and handle its own emergencies. [Why] This minimizes coupling.

- Creator. Who should create (allocate) an object? [A] The class that composes, aggregates or contains it. [Why] This minimizes coupling.

- Don’t Talk to Strangers. That is, don’t “send messages” to objects that are not close to you. Non-strangers are:
  - your own data members.
  - elements of a collection which is one of your own data members.
  - a parameter of the current function.
  - a locally-created object.
  - this (but using this is rarely the right thing to do).

Delegate the operation [Why] This is a generalization of the old rule, don’t use globals. It maximizes the locality of every reference and avoids unnecessary coupling between classes.

[Example] Don’t deal directly with a component of one of your own members. Suppose a hardware store class composes an object of class Inventory, and the Inventory is a flex-array of Item pointers, as shown below. The retail store would be talking to a stranger if its sell function called the sell function in the Item class directly, like this: `Inv.find(currentKey)->sell(5);` The preferred design is to work through the intermediate class. That is, Inventory should provide a function such as `sell(key, int)` that can be called by RetailStore, and `Inventory::sell(key, int)` should call `Inventory::find(key)` followed by `Item::sell(int)`. Evaluation: In the first design, a change in the Item class might force a change in both RetailStore and Inventory. Using the second design, only Inventory is affected.

![Diagram](RetailStore:Inventory:Item)

- RetailStore: Inventory
- Inventory: Itemlist
- + order(int)
- + sell(key, int): void
- + find(key): Item*
- + sell(key, int): void
- Item: + order(int)
- + sell(int)

Figure 18.1: Don’t talk to strangers.

18.4 More Complex Design Patterns

- Adapter. Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires. [Solution:] Define an adapter class that can add, subtract, or override functionality, where necessary. There are two ways to do this; on the left is a class adapter, on the right an object adapter.
18.4. **MORE COMPLEX DESIGN PATTERNS**

- **Indirection.** This pattern is used to decouple the application from the implementation where an implementation depends on the interface of some low-level device. [Why] To make the application stable, even if the device changes.

- **Proxy.** This pattern is like Indirection, and is used when direct access to a component is not desired or possible. What to do? [Solution:] Provide a placeholder that represents the inaccessible component to control access to it and interact with it. The placeholder is a local software class. Give it responsibility for communicating with the real component. [Special cases:] Device proxy, remote proxy. In Remote Proxy, the system must communicate with an object in another address space.

- **Polymorphism:** In an application where the abstraction has more than one implementation, define an abstract base class and one or more subclasses. Let the subclasses implement the abstract operations. [Why] to decouple the implementation from the abstraction and allow multiple implementations to be introduced, as needed.

- **Controller:** Who should be responsible for handling a system event? [A] A controller class. The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

Factors such as the number of events to be handled, cohesion and coupling should be used to decide among the three kinds of controllers described below and to decide how many controllers there should be. A controller class represents one of the following choices:

- The overall application, business, or organization (facade controller).
– Something in the real world that is active that might be involved in the task (role controller).  
[Example:] a menu handler.
– An artificial handler of all system events involved in a given use case (use-case controller).  
[Example:] A retail system might have separate controllers for BuyItem and ReturnItem.

• Bridge: This pattern is a generalization of the Indirection pattern, used when both the application class and the implementation class are (or might be) polymorphic. The bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments. In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.

```
Figure 18.5: Bridging the implementation gap.
```

• Subject-Observer or Publish-Subscribe: Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

[Example:] An OO spreadsheet application contains a data object, several presentation “views” of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

[Solution:] In the following discussion, the SpreadsheetData class is the subject, the views and graphs are the observers. The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects from its list. Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes. We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a get_state() function, that returns a copy of its state.

```
Observer::update() {
    observer_state = SS->get_state();
}
Spreadsheet::notify() {
    ... get_state()
    OL.updateall()
}
ObserverList::updateall() {
    for all x in the list,
        x->update()
}
```

```
Figure 18.6: One subject with three observers
```

The ObserverList class defines an updating interface for objects that should be notified of changes in a subject. The Observer class provides an abstract public function called update() which will be called by
ObserverList whenever `updateall()` is called. This abstract function must be implemented in each concrete observer class.

When the state of the SpreadsheetData subject changes, it executes its inherited `notify()` function, which calls `ObserverList::updateall()`, which notifies all of the observers. Each one, in turn, executes its update function, which calls the subject’s `get_state` function. Changes can then be made locally that reflect the change in the subject’s state.

- **Singleton**: Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance. [Solution:] A single object may be made available to all objects of class `C` by making the singleton a static member of class `C`. A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.

```cpp
static member StringStore& StringStore::getStore(){
    if (instance==NULL) instance = new StringStore;
    return instance;
}
```

![Figure 18.7: How to create a singleton.](image)

[Example] Suppose there were several parts of a program that could use a `StringStore`. We might define `StringStore` as a singleton class. The `StringStore::put` function would be made static and would become a global access point to the class, while maintaining full protection for the class members.

**More patterns?** The seven patterns presented here are some of the earliest and most useful that have been developed. But many, many more design patterns have been identified and published. A professional working in either C++ or Java would do well to study some of the available literature.