

Interfaces, Templates, and Exceptions

Sections 2.2.5-2.4

API and interface

- ❑ Classes should provide an **application programming interface (API)**, or simply **interface**, that specifies how other objects can interact with objects of that class.
- ❑ In the ADT-based approach, an **interface** is specified as a type definition with a collection of member functions, with the arguments for each member function being of specified types.
- ❑ Java provides a language construct for specifying interfaces, but C++ does not.

Informal Interfaces

- We will use **informal interfaces** which look like a C++ specification but aren't exactly valid C++ constructs.

```
class Stack {  
public:  
    bool isEmpty() const;  
    void push(int x);  
    int pop();  
};
```

- Note that this does not contain any data members or function bodies. It is just a documentation aid.

Abstract Classes

- ❑ An **abstract class** is a C++ class that can only be used as a base class for inheritance—it cannot be used to create instances.
- ❑ A class becomes abstract when one or more of its functions are declared as **abstract** (also known as **pure virtual**). This is done as follows:

```
class Shape {  
    virtual void draw() = 0;  
    // ...  
}
```

Abstract Classes

- ❑ C++ does not allow the creation of an object that has one or more pure virtual functions.
- ❑ Thus, any subclass that expects to be instantiated must **override** all pure virtual functions of its superclass.
- ❑ We can use abstract classes in C++ to achieve most of the effect of an **interface**.

```
class Stack {  
public:  
    virtual bool isEmpty() const = 0;  
    virtual void push(int x) = 0;  
    virtual int pop() = 0;  
};
```

Concrete classes

- The opposite of **abstract** is **concrete**.

```
class ConcreteStack : public Stack {  
public:  
    virtual bool isEmpty() { ... };  
    virtual void push(int x) { ... };  
    virtual int pop() { ... };  
private:  
    // ...  
};
```

Templates

- Another mechanism for polymorphism in C++ is **templates**.
- I.e., templates allow us to write code that works for a variety of different types.
- Consider:

```
int integerMin(int a, int b)
{ return (a < b ? a : b); }
```

```
double doubleMin(double a, double b)
{ return (a < b ? a : b); }
```

Function Templates

- We can write a single function that works for both of these by using a template:

```
template <typename T>  
T generalMin(T a, T b) {  
    return (a < b ? a : b);  
}
```

- The **type parameter** T takes the place of an actual type in the declaration of the function.

```
cout << generalMin(3, 4) << ' '  
     << generalMin(1.1, 3.1) << ' '  
     << generalMin('t', 'g') << endl;
```


Class Templates

- ❑ C++ also allows classes to be templated.
 - ❑ The Standard Template Library (STL), a common library of C++ data structures, uses class templates extensively.
-

```
template <typename T>
```

```
class BasicVector {
```

```
public:
```

```
    BasicVector(int capac = 10);
```

```
    T& operator[(int i) { return a[i]; }
```

```
    // ...
```

```
private:
```

```
    T* a;
```

```
    int capacity;
```

```
}
```

Class Templates

Outside of the class body, one can specify member functions such as:

```
template <typename T>
BasicVector<T>::BasicVector(int capac) {
    capacity = capac;
    a = new T[capacity];
}
```

And one can instantiate the class by specifying the type:

```
BasicVector<int> iv(5);
BasicVector<double> dv(20);
BasicVector<string> sv(10);
```

Class Templates

- ❑ Any type can be used as the argument for a template specified with **typename**.
- ❑ This includes templated types!

```
BasicVector<BasicVector<int> > xv(5);    // this is a vector of vectors
// ...
xv[2][8] = 15;
```

- ❑ Here each element of the 5-element vector was initialized with the default capacity of 10.

Exceptions

- ❑ **Exceptions** are unexpected events that happen during the execution of a program, such as an error condition or an unexpected input.
- ❑ C++ allows the programmer to create an object that represents the exception.
- ❑ This object is then **thrown** by the code that encounters the unexpected event, and then **caught** by other code that **handles** the event.
- ❑ If no code catches the exception, the program is terminated.
- ❑ The **C++ run-time environment** can throw exceptions; an example is when a program runs out of memory.

Exceptions

- ❑ An alternative to exceptions is to have a function return special **error codes** or **condition codes** to indicate that it encountered unexpected circumstances.
- ❑ Exceptions are more modern but you will still encounter many **library functions** that return condition codes.
- ❑ In C++, any object can be thrown, but it is the best practice to define object classes for exceptions, often with **Exception** or **Error** as part of its name.

Sample Exception Classes

```
class MathException {  
    public:  
        MathException(const string& err)  
            : errMsg(err) { }  
        string getError() { return errMsg;}  
    private:  
        string errMsg;  
};
```

```
class ZeroDivideException : public MathException {  
    public:  
        ZeroDivideException(const string& err)  
            : MathException(err) { }  
};
```

Throwing and Catching

- ❑ Exceptions are processed in the context of **try** and **catch** blocks.

```
try {  
    // ...  
    if (divisor == 0)  
        throw ZeroDivideException("divide by zero!");  
}  
catch (ZeroDivideException& zde) {  
    // handle division by zero  
}  
catch (MathException& me) {  
    // handle any other math exception  
}
```

Try-Catch statements

- ❑ A **try-catch statement** consists of a **try block** followed by any number of **catch blocks**.
- ❑ Execution begins in the try block.
- ❑ If the try block finishes without any exceptions being thrown, then the catch blocks are not executed.
- ❑ If the try block throws an exception, then control is immediately transferred to the **first** catch block that matches that exception.
- ❑ If no catch block matches the exception, then the subroutine/function being executed is exited with the thrown exception.

Exception Specification

- ❑ An exception that is not caught ends a function and this can propagate up through many active functions.
- ❑ Therefore, when specifying a function, we should also specify the exceptions it might throw. This lets the programmer and the compiler know which exceptions to expect.
- ❑ This is done with a **throw** declaration as part of the function definition.

```
void calculator() throw(ZeroDivideException,  
                        NegativeRootException) {  
    // ... function body  
}
```

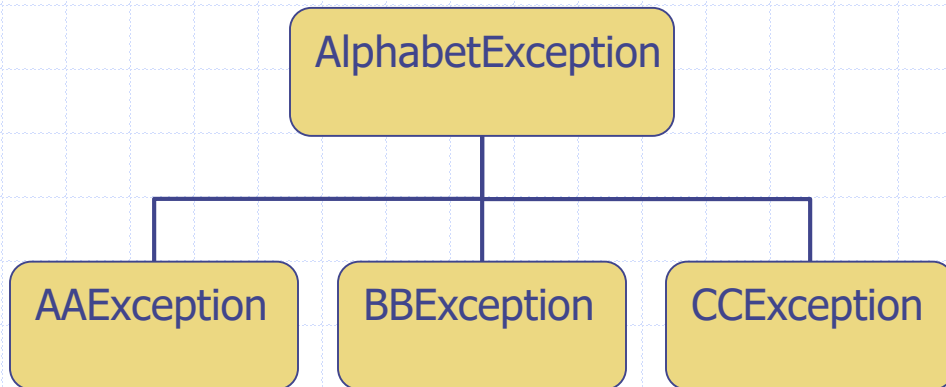
Exception Specification

- ❑ A benefit to specifying exceptions on a function is that it tells us which exceptions the function does not itself need to handle.
- ❑ This is appropriate when other code is responsible for the circumstances that led to the exception.
- ❑ Here's an example of **passing** an exception **through** a function:

```
void getReadyForClass() throw(OutOfMoneyException) {  
    goShopping();                // could throw OutOfMoneyException  
    makeCookiesForTA();  
}
```

Exception Specification

- ❑ A function can declare that it throws as many exceptions as it likes.
- ❑ Such a listing of exceptions can be simplified if several exceptions are derived classes of the same exception. If so, we need only list the appropriate base class.



```
int func() throw(AAException,  
BBException,  
CCException) { ... }
```

```
int func() throw(  
AlphabetException) { ... }
```

Exception Specification

- ❑ A function that does not contain a **throw** declaration is assumed to throw **any** exception.

```
void ICanThrowAnyException();
```

- ❑ A function can declare it throws **no** exceptions by specifying an empty list after the **throw** keyword.

```
void ICanThrowNoExceptions() throw();
```

General Exception Class

```
class RuntimeException {  
  private:  
    string errorMsg;  
  public:  
    RuntimeException(const string& err) {  
      errorMsg = err;  
    }  
    string getMessage() const {  
      return errorMsg;  
    }  
};
```

- This is an example of an **immutable** object.