Chapter 11

Abstract Data Types and Encapsulation Concepts

The big picture in Stroustrup's view

- **Procedural**

  Decide which procedure you want; use the best algorithms you can find

- **Data hiding**

  Partition the program so that data is hidden in the modules

- **Data abstraction**

  Decide which types you want, provide a full set of operations for each type

- **OO Programming**

  Decide which classes you want, provide a full set of operations for each class, make commonality explicit by using inheritance

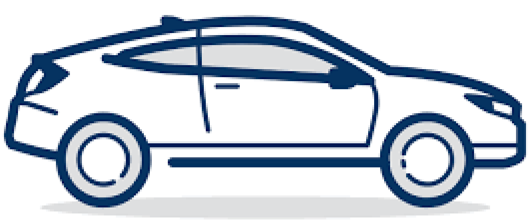
•Bjarne Stroustrup. [What is "Object-Oriented Programming?](#) in Proceedings of the First European Software Festival, 1988.

Chapter 11 Topics

- The Concept of Abstraction
- Introduction to Data Abstraction
- Design issues for Abstract Data Types (ADTs)
- Language Examples
- Parameterized ADT and other Encapsulation constructs

The Concept of Abstraction

- An abstraction
  - is a view or representation of an entity that includes only the most significant attributes
    - Removing/hiding irrelevant details
    - generalization
  - is a weapon against the complexity of program



DATA ABSTRACTION

- Two fundamental types of abstraction
  - process abstraction → subprograms
  - data abstraction →abstract data types
    - Type extension with user-defined types

#include <iostream>

Process abstraction

```
#include <iostream>

using namespace std;

int main()
{
     int sectionA[5] = {99, 95, 79, 89, 77};
     for (int endPos=4; endPos>0; endPos--)
     {
          for (int index=0; index<endPos; index++)
          {
               if (sectionA[index]>sectionA[index+1])
               {
                    int temp = sectionA[index];
                    sectionA[index] = sectionA[index+1];
                    sectionA[index+1]=temp;
               }
          }
     }

     int sectionB[5] = {12, 10, 32, 1, 80};
     for (int endPos=4; endPos>0; endPos--)
     {
          for (int index=0; index<endPos; index++)
          {
               if (sectionB[index]>sectionB[index+1])
               {
                    int temp = sectionB[index];
                    sectionB[index] = sectionB[index+1];
                    sectionB[index+1]=temp;
               }
          }
     }

     cout << "The highest grades for sections A nd B were: "
           << sectionA[4] << "  "  << sectionB[4] << "  "    << endl;

     return 0;
}
```

Process abstraction

```
#include <iostream>
using namespace std;

void sort(int a[], int aSize)
{
     for (int endPos=aSize - 1; endPos>0; endPos--)
     {
          for (int index=0; index<endPos; index++)
          {
               if (a[index]>a[index+1])
               {
                    int temp = a[index];
                    a[index] = a[index+1];
                    a[index+1]=temp;
               }
          }
     }
}

int main()
{
     int sectionA[5] = {99, 95, 79, 89, 77};
     sort(sectionA,5);

     int sectionB[5] = {12, 10, 32, 1, 80};
     sort(sectionB,5);

     int sectionC[5] = {9, 0, 18, 33, 4};
     sort(sectionC,5);

     cout << "The highest grades for sections A,B, and C were: "
           << sectionA[4] << "  "   << sectionB[4] << "  "
           << sectionC[4] << "  " << endl;
    return 0;
}
```

Abstract Data Type

- An ADT should satisfy the following conditions:
  - The declarations of types and the protocols of the operations on objects of the type are contained a single syntactic unit.
  - The representation and implementation details are hidden from the program units that use these objects.

Data Abstraction

- Interface and implementation
  - An interface that prescribes to the client how to create variables of the defined type and how to invoke procedures and functions that manipulate the objects of the defined type.
  - An implementation that prescribes how the abstract data type is represented and carries out its operations.

Advantages of Data Abstraction

- Provide a way of organizing programs
- Improve the reliability
- Allow the implementation to be changed without affecting user code
- Promote two important design goals: low coupling and high cohesion.
  - Coupling refers to the degree of dependency between two modules. Cohesion refers to the degree to which a single module forms a meaningful unit.

Two related terms

- Encapsulation
  - The process of hiding (encapsulating) all the details of how a piece of software was written and telling only what a client needs to know in order to use the software
  - Data and operations are bundled into one single unit.

- Information hiding
  - Hide the fine details of what is inside the “capsule.”

- ADTs support encapsulation and information hiding.

Design Issues

- What is form of encapsulation?
- How access controls are provided?
- What kind of operations should be supported?
- Can ADT be parameterized?

LANGUAGE EXAMPLES

What is form of encapsulation?

- The class is the basic/minimum encapsulation device
  - Data members and member functions
  - Class instances can be stack dynamic or heap dynamic

Encapsulation Constructs

Header file:

```
// This is foo.h
#ifndef FOO_H
#define FOO_H
// header file content goes here
#endif
```

-- Method protocols
(for public member function,
friend functions and ordinary functions
-- data members

Implementation file:

```
// This is foo.cpp
#include "foo.h"
// implementation goes here
```

-- Method implementations

A Stack class header file

C++

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private:
  int *stackPtr;
  int maxLen;
  int topPtr;
public:
   Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

```
// Very simplified stack template
template <typename T>
class Stack {
private:
    std::vector<T> data;   // use vector internally
public:
    void push(const T& value) { data.push_back(value); }
    void pop() { data.pop_back(); }
    T& top() { return data.back(); }
    bool empty() const { return data.empty(); }
    size_t size() const { return data.size(); }
};
```

The code file for Stack


```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int[100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
       cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

How access controls are provided? C++

- Public clause
  - the marked data member or member function can be accessed by any client.
- Private clause
  - the marked data member or member function can only be accessed from within the class
- Protected clause
  - the marked data member or member function can only be accessed from within the class or a subclass.
- Friend clause
  - allow two modules to have public access to each other's private components.

Access modifier

C++

The code snippet signifies the fact that goo objects can access the private members of foo objects.

friend clause:

```
class foo {
    friend class goo;
    ...
};

class goo
    ...
};
```

```
#include < iostream.h >

class CPP_Tutorial
{
    int private_data;
    friend class friendclass;
public:
    CPP_Tutorial()
    {
        private_data = 5;
    }
};

class friendclass
{
public:
    int subtractfrom(int x)
    {
        CPP_Tutorial var2;
        return var2.private_data - x;
    }
};

int main()
{
    friendclass var3;
    cout << "Added Result for this C++ tutorial: "<< var3.subtractfrom(2)<< endl
}
```

Access modifier (continued)

C++

```
#include <iostream.h>
//Declaration of the function to be made as friend for the C++ Tutorial sample
int AddToFriend(int x);
class CPP_Tutorial
{
   int private_data;
   friend int AddToFriend(int x);
public:
   CPP_Tutorial()
  {
     private_data = 5;
   }
};
int AddToFriend(int x)
{
   CPP_Tutorial var1;
   return var1.private_data + x;
}
int main()
{
   cout << "Added Result for this C++ tutorial: "<< AddToFriend(4)<<endl;
}
```

Can ADT be parameterized?

C++

```
#include <iostream>
#include <string>
using namespace std;

template <class T>
class Pair
{
  public:
    Pair();
    Pair(T firstValue, T secondValue);
    void setFirst(T newValue);
    void setSecond(T newValue);
    T getFirst() const;
    T getSecond() const;
  private:
    T first;
    T second;
};

template <class T>
Pair<T>::Pair(T firstValue, T secondValue):first(firstValue),second(secondValue){}
template <class T>
Pair<T>::Pair(){}
template<class T>
void Pair<T>::setFirst(T firstValue){first = firstValue;}
template<class T>
void Pair<T>::setSecond(T secondValue){second = secondValue;}

template<class T>
T  Pair<T>::getSecond()const {return second;}

template<class T>
T  Pair<T>::getFirst()const {return first;}
```

Can ADT be parameterized?

C++

```
#include <iostream>
#include <string>
#include "Pair.h"
using namespace std;

int main()
{
  Pair<int> intPair(1,2);
  cout << "A pair of integers: ";
  cout << intPair.getFirst() << " " << intPair.getSecond() << endl;
  Pair<string> stringPair("Template", "Pair");
  cout << "A pair of strings:: ";
  cout << stringPair.getFirst() << " " << stringPair.getSecond() << endl;
  return 0;
}
```

Under the hood

Code specialization: The compiler generates a new representation for every instantiation of a generic type or method. For instance, the compiler would generate code for a list of integers and additional, different code for a list of strings, a list of dates, a list of buffers, and so on.

Template specialization

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

C++ Templates summary

- Implemented in the compiler
  - Requires template source to be in headers
- Glorified macro facility
- Can use template arguments for both classes and straight functions
- Template specialization
  - Specific implementation of a templated type or method

Naming Encapsulations

C++

- **Name clashes - C++ uses namespaces to resolve such ambiguities.**

e.g.

```
// This is a file from one vendor, say vendor A
namespace vendorA {
    class foo {
    ...
    };
}
// This is a file from another vendor, say vendor B
namespace vendorB {
    class foo {
    ...
    };
}
```

```
// Client code
vendorA::foo x; // creates a foo object from Vendor A
vendorB::foo y; // creates a foo object from Vendor B

//using directive:
{
        using namespace vendorA;
        foo x; // creates a foo object from Vendor A
}
```

```cpp
#ifndef NAMESAPCE_EXAMPLE_H
#define NAMESPACE_EAMPLE_H

#include <iostream>
using namespace std;

namespace savitch1
{
    void greeting();
}

namespace savitch2
{
    void greeting();
}

void big_greeting();
#endif
```

```cpp
#include <iostream>
#include "NameSpaceExample.h"
using namespace std;

namespace savitch1
{
 void greeting()
 {
   cout << "hello form namespace savitch 1.\n";
 }
}

namespace savitch2
{
  void greeting()
  {
    cout << "hello from namespace satich 2.\n";
  }
}

void big_greeting()
{
  cout << "a big global hello.\n";
}
```

C++

```cpp
#include <iostream>
#include "NameSpaceExample.h"
using namespace std;

int main()
{
  {
   using namespace savitch2;
   greeting();
  }

  {
   using namespace savitch1;
   greeting();
  }
savitch1::greeting();
  savitch2::greeting();

 big_greeting();
 return 0;
}
```

Summary of C++

- support for ADTs via class
- Support public, private, protected, and friend access modifier/clauses
- Support parametrized ADT via template
- provide effective mechanisms for encapsulation and information hiding

What is form of encapsulation? **Java**

- The class is the basic encapsulation device
  - All user-defined types are classes
    - data →instance variables; operations → methods.
  - All objects are allocated from the heap
  - Java has a second scoping mechanism, package scope, which can be used in place of friends
    - All entities in all classes in a package that do not have access control modifiers are visible throughout the package

Encapsulation Constructs

- Java squeezes both the interface and the implementation into a single file.
- Java supports javadoc $\rightarrow$ API

An Example in Java

```
class StackClass {
      private int [] stackRef;
      private int maxLen, topIndex;
      public StackClass() { // a constructor
            stackRef = new int [100];
            maxLen = 99;
            topPtr = -1;
      };
      public void push (int num) {…};
      public void pop () {…};
      public int top () {…};
      public boolean empty () {…};
}
```

How access controls are provided?

- Public clause
  - the marked data member or member function can be accessed by any client.
- Private clause
  - the marked data member or member function can only be accessed from within the class
- Protected clause
  - the marked data member or member function can be accessed from within the package or a subclass of another package.

Access Modifiers

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Naming Encapsulations

- Java Packages
  - Packages can contain more than one class definition; classes in a package are *partial* friends
  - Clients of a package can use fully qualified name or use the ***`import`*** declaration

Naming Encapsulations (cont.)

Declare the fully-qualified class name

```
...
world.HelloWorld helloWorld = new world.HelloWorld();
world.moon.HelloMoon helloMoon = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
...
```

Use import

```
import world.*;  // we can call any public classes inside the world package
import world.moon.*;  // we can call any public classes inside the world.moon package
```

```
...
  HelloWorld helloWorld = new HelloWorld();
  HelloMoon helloMoon = new HelloMoon(); //
...
```

Parameterized ADT

- Parameterized ADTs allow designing an ADT that can store any type elements
- In Java, also known as generic classes. In C++, template classes.
- C++ and Java 1.5 and up provide support for parameterized ADTs

Parameterized ADT

```
public class Pair<T>
{
  T first;
  T second;

  public Pair(){}
  public Pair(T firstValue, T secondValue)
  {
     first = firstValue;
     second = secondValue;
  }

  public T getFirst()
  {
    return first;
  }

  public T getSecond()
  {
    return second;
  }

  public void setFirst(T firstValue)
  {
    first =  firstValue;
  }
  public void setSecond(T secondValue)
  {
    second = secondValue;
  }

  public static void main(String [] args)
  {
     Pair<Integer> intPair = new Pair<Integer>(1,2);
     Pair<String> stringPair = new Pair<String>("Template", "Pair");
     System.out.println("A pair of integers: " + intPair.getFirst() + " " + intPair.getSecond());
     System.out.println("A pair of strings: " + stringPair.getFirst() + " " + stringPair.getSecond());
  }
}
```

Features of Java Generics

- Based on Pizza project ([Pizza into Java: Translating theory into practice])
- Implemented in the compiler
  - Does not require source of generic type to be available
  - Compiled code can theoretically run on older JVMs
- Applies to classes and methods within classes
- Type parameter bounds <X extends Widget>
- Mostly used to eliminate downcast

Under the hood

Code sharing: The compiler generates code for only one representation of a generic type or method and maps all the instantiations of the generic type or method to the unique representation, performing type checks and type conversions where needed.

Type Erasure

A process that maps a parameterized type (or method) to its unique byte code representation by eliding type parameters and arguments.

Example (before type erasure):

```
public class Pair<X,Y> {
  private X first;
  private Y second;
  public Pair(X x, Y y) {
    first = x;
    second = y;
  }
  public X getFirst() { return first; }
  public Y getSecond() { return second; }
  public void setFirst(X x) { first = x; }
  public void setSecond(Y y) { second = y; }
}

final class Test {
  public static void main(String[] args) {
    Pair<String,Long> pair = new Pair<String,Long>("limit", 10000L);
    String s = pair.getFirst();
    Long   l = pair.getSecond();
    Object o = pair.getSecond();
  }
}
```

Example (after type erasure):

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( Object x, Object y) {
    first = x;
    second = y;
  }
  public Object getFirst() { return first; }
  public Object getSecond() { return second; }
  public void setFirst( Object x) { first = x; }
  public void setSecond( Object y) { second = y; }
}

final class Test {
  public static void main(String[] args) {
    Pair pair = new Pair("limit", 10000L);
    String s = (String) pair.getFirst();
    Long   l = (Long)    pair.getSeond();
    Object o =          pair.getSecond();
  }
}
```

Summary of Java

C++

- support for ADTs via class
- Support public, private, and protected access modifier
- Support parametrized ADT via generic
- provide effective mechanisms for encapsulation and information hiding

Summary

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding

Exercise 1: Choosing ADT operations

- When you define an ADT, what kinds of operations/functions/methods you mostly like to have?

Exercise 2:

Why does the default copy constructor behave incorrectly?

```
// Node and LinkedList class
class Node {
    public:
        int data;
        Node* next;
};

class LinkedList {
    private:
        Node* headNode;
    public:
        // Here's what the default copy constructor looks like
        LinkedList(const LinkedList& other) {
            this->headNode = other.headNode;
        }
        ...
};
```