Abstract Data Types

• Programmer-created data types that specify
  – values that can be stored (type of data)
  – operations that can be done on the values

• The user of an abstract data type (ADT) does not need to know any implementation details (e.g., how the data is stored or how the operations on it are carried out)
Abstraction in Software Development

• Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented.

• We have already encountered this:
  – To use the `pow` function, a programmer needs to know what inputs it expects and what kind of results it produces.
  – The programmer does not need to know how it works.
Abstraction and Data Types

• **Abstraction**: a definition that captures general characteristics without details
  
ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral

• **Data Type**: defines the kind of values that can be stored and the operations that can be performed on it
Object-Oriented Programming

- **Procedural programming** uses variables to store data, and focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.

- **Object-oriented programming** is based on objects that encapsulate the data and the functions that operate on it.
Object-Oriented Programming Terminology

- **object**: software entity that combines data and functions that act on the data in a single unit
- **attributes**: the data items of an object, stored in member variables
- **member functions (methods)**: procedures/functions that act on the attributes of the class
• **data hiding**: restricting access to certain members of an object; The intent is to enforce data security by allowing only member functions to directly access and modify the object’s data

• **encapsulation**: the bundling of an object’s data and procedures into a single unit. A class encapsulates both variables and functions together in a single unit.
Object Example

Square

<table>
<thead>
<tr>
<th>Member variables (attributes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int side;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setSide(int s)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>side = s;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

| int getSide()               |
| {                           |
|     return side;           |
| }                           |

Square object’s data item: side
Square object’s functions: setSide - set the size of the side of the square, getSide - return the size of the side of the square
Why Hide Data?

- Protection – Member functions provide a layer of protection against inadvertent or deliberate data corruption.
- Need-to-know – A programmer can use the data via the provided member functions. As long as the member functions return correct information, the programmer needn’t worry about implementation details.
C++ Classes

- The **class** is a generalization of the C **structure**
- A programmer-defined data type used to define objects
- A pattern for creating objects
- Can contain:
  - data
  - function (method) prototypes or full implementations
Introduction to Classes

• Class declaration format:

```python
class className
{
    declaration;
    declaration;
};
```

Notice the required ;
Access Specifiers

- Used to control access to members of the class.
- Each member is declared to be either private, protected, public, friend
  
  - **private** members of a class can only be called by (accessed by) other members of the class (or from their friends); in general, data will be private.
  
  - **protected** members are accessible from members of their same class (and from their friends) but also from members of their derived classes.
class Square
{
private:
    int side;
public:
    void setSide(int s)
    {
        side = s;
    }
    int getSide()
    {
        return side;
    }
};
More on Access Specifiers

• Can be listed in any order in a class
• Can appear multiple times in a class
• If not specified, the default is private
Creating and Using Objects

- An **object** is an instance of a class
- It is defined just like other variables
  ```
  Square sq1, sq2;
  ```
- It can access members using dot operator
  ```
  sq1.setSide(5);
  cout << sq1.getSide();
  ```
Types of Member Functions

- **Accessor, get, getter function**: uses, but does not modify a member variable
  
  ex: `getSide`

- **Mutator, set, setter function**: modifies a member variable

  ex: `setSide`
Defining Member Functions

• Member functions are part of a class declaration
• Can place the entire function definition inside the class declaration
  or
• Can place just the prototype inside the class declaration and write the function definition after the class. This is the approach we will use.
Defining Member Functions Inside the Class Declaration

• Member functions defined inside the class declaration are called **inline functions**

• Only very short functions, like the one below, should be inline functions

```cpp
int getSide()
{
    return side;
}
```
class Square
{
    private:
        int side;
    public:
        void setSide(int s)
        {
            side = s;
        }
        int getSide()
        {
            return side;
        }
};
• Put a function prototype in the class declaration

• In the function definition, precede the function name with the class name and `scope resolution operator (::)

```cpp
int Square::getSide()
{
    return side;
}
```
Conventions and a Suggestion

Conventions:

• Member variables are usually **private**
• Accessor and mutator functions are usually **public**
• Use ‘get’ in the name of accessor functions, ‘set’ in the name of mutator functions

Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data
Tradeoffs of Inline vs. Regular Member Functions

• When a regular function is called, control passes to the called function
  – the compiler stores return address of call, allocates memory for local variables, etc.

• Code for an inline function is copied into the program in place of the call when the program is compiled
  – This makes a larger executable program, but
  – There is less function call overhead, and possibly faster execution
Constructors

• A constructor is a member function that is often used to initialize data members of a class
• Is called automatically when an object of the class is created
• It must be a public member function
• It must be named the same as the class
• It must have no return type
Constructor – 2 Examples

**Inline:**
```cpp
class Square {
    ...
    public:
        Square(int s) {
            side = s;
        }
    ...
};
```

**Declaration outside the class:**
```cpp
Square(int);  //prototype
//in class
Square::Square(int s) {
    side = s;
}
```
Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```c
Square();
Square(int);
```
Constructors and Destructors

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of the class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition includes a return value -- not even void.
The Default Constructor

• Constructors can have any number of parameters, including none

• A default constructor is one that takes no arguments either due to
  – No parameters or
  – All parameters have default values
The Default Constructor

• If you do not declare any constructors, the compiler provides a default constructor with base values assigned to the instance variables.

• If a class has any programmer-defined constructors, the compiler no longer provides an implicit default constructor; therefore, the class must provide a programmer-defined constructor.
Constructors and Destructors

In the class specification below, the Box class includes two constructors: Box() and Box(int w, int l, int h)

// box.h  (specification file)
class Box  {
   public:
      Box();
      Box(int w, int l, int h);
      ~Box();
      int  volume();
      void set_values(int w, int l, int h);
   private:
      int width;
      int length;
      int height;
};

int main()  {
   Box b1;
   Box b2(4, 3, 5);
   Box *b3 = new Box(5, 7, 2);
   b1.set_values(5, 7, 3);
   cout << "b1 volume = 
      << b1.volume()
      << endl;
   cout << "b2 volume = 
      << b2.volume()
      << endl;
   cout << "b3 volume = 
      << b3->volume()
      << endl;
}
Passing Objects to Functions

• A class object can be passed as an argument to a function

• When passed by value, the function makes a local copy of the object. Original object in calling environment is unaffected by actions in function

• When passed by reference, function can use ‘set’ functions to modify the object.
Notes on Passing Objects

- Using a value parameter for an object can slow down a program and waste space.

- Using a reference parameter speeds up program, but allows the function to modify data in the parameter.
Notes on Passing Objects

• To save space and time, while protecting parameter data that should not be changed, use a `const` reference parameter

```cpp
void showData(const Square &s) // header
```

• In order for the `showData` function to call `Square` member functions, those functions must use `const` in their prototype and header:

```cpp
int Square::getSide() const;
```
Returning an Object from a Function

• A function can return an object
  
  ```
  Square initSquare(); // prototype
  s1 = initSquare(); // call
  ```

• The function must define an object
  – for internal use
  – to use with `return` statement
Returning an Object Example

Square initSquare()
{
    Square s;    // local variable
    int inputSize;
    cout << "Enter the length of side: ";
    cin >> inputSize;
    s.setSide(inputSize);
    return s;
}
The *this* Pointer and Constant Member Functions

- *this* pointer:
  - Implicit parameter passed to a member function
  - Points to the object calling the function
Using the **this** Pointer

Can be used to access members that may be hidden by parameters with the same name:

```cpp
class SomeClass
{
    public:
        SomeClass();
        SomeClass(int num);
        void setNum(int num);

    private:
        int num;

};
```
Using the **this** Pointer

In this example, `num` is the private data member and `num` is a parameter; therefore, the **this** pointer is needed in order to access the private data member

```cpp
SomeClass::SomeClass
{
    this->num = num;
}

void setNum(int num)
{
    this->num = num;
}
```
Using the **this** Pointer

Note that **this** is a pointer and, therefore, pointer notation must be used with it:

```c++
this->num = num;
```
Using separate files

Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design.
Using Separate Files

• class specification file
  – A header file that contains the class declaration
  – Name the file *classname*.h (for example, Square.h)

• class implementation file
  – A .cpp file that contains the member function definitions.
  – Name the file *classname*.cpp (for example, Square.cpp)
    This file should `#include` the class specification file.
  – This file is also referred to as a source file.
Using Separate Files

• In addition to the specification file and the implementation file, you will also need a main function (client code) for testing every method of your class. We generally call this program the test driver.

• Must `#include` the class specification file and be compiled and linked with the class implementation file.
Include Guards

- Used to prevent a header file from being included twice
- Format:
  
  ```
  #ifndef symbol_name
  #define symbol_name
  . . . (normal contents of header file)
  #endif
  ```
- `symbol_name` is usually the name of the header file, in all capital letters:
  
  ```
  #ifndef SQUARE_H
  #define SQUARE_H
  . . . SQUARE_H
  #endif
  ```
Class should be designed to provide functions to store and retrieve data

In general, input and output (I/O) should be done by functions that use class objects, rather than by class member functions