CS2040C Data Structures and Algorithms

Abstract Data Types

Outline

- Abstraction in Programs
- Abstract Data Types
  - Definition
  - Benefits
- Abstract Data Type Examples
- *Appendices*
  - *A: Modular Design in C++*
  - *B: Exception*

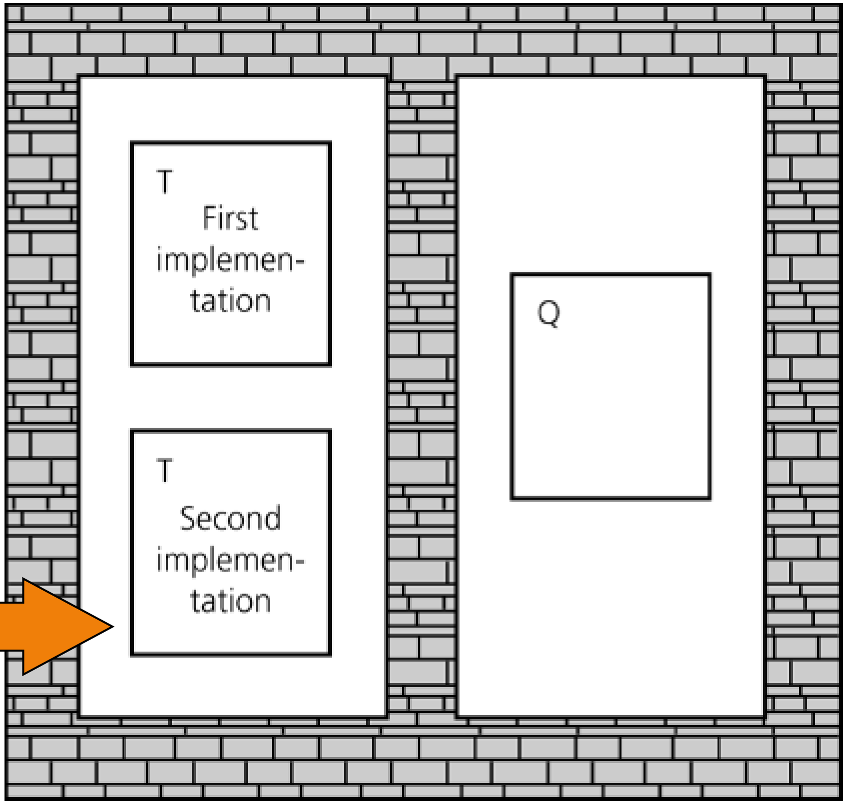
Abstraction

- A modular program is easier to write, read and modify
- Write specifications for each module before implementing it
- Isolate the implementation details of a module from other modules

Abstraction

- Isolated tasks: the implementation of task T does not affect task Q
- Q does not know how task T is performed
- Q must know what task T is and how to initiate it

Makes it easy to substitute new, improved versions of how to do a task later



Abstraction

- The process of isolating implementation details and extracting only **essential property** from an entity
- Program = data + algorithms
- Abstraction in a program:
  - **Data abstraction**
    - What operations are needed by the data
  - **Functional abstraction**
    - What is the purpose of a function (algorithm)

Typical Operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection

The details of the operation vary from application to application, but the overall theme is the **management of data**

Data Abstraction

- When we talk about a program doing something, proper abstraction means we should decide *what* is done and not *how* it is done

- We can do the same thing about the data used in the program

- Data Abstraction: What operations a data collection supports and not how it supports it

Abstract Data Type (ADT)

- **Abstract Data Type** (ADT**):**
  - End result of data abstraction
  - A collection of ***data*** together with a set of ***operations*** on that data
  - ADT = Data + Operations
- ADT is a language independent concept
  - Different language supports ADT in different ways
  - In C++, the class construct is the best match
- Important Properties of ADT:
  - **Specification:**
    - The supported operations of the ADT
  - **Implementation:**
    - Data structures and actual coding to meet the specification

Definition of a Data Structure

- is a construct that can be defined within a programming language to store a collection of data
  - For example, arrays are data structures built into C++.
  - You can also invent other data structures. For example, you want a data structure to store both the names and salaries of a group of employees
    - In C++, you can define

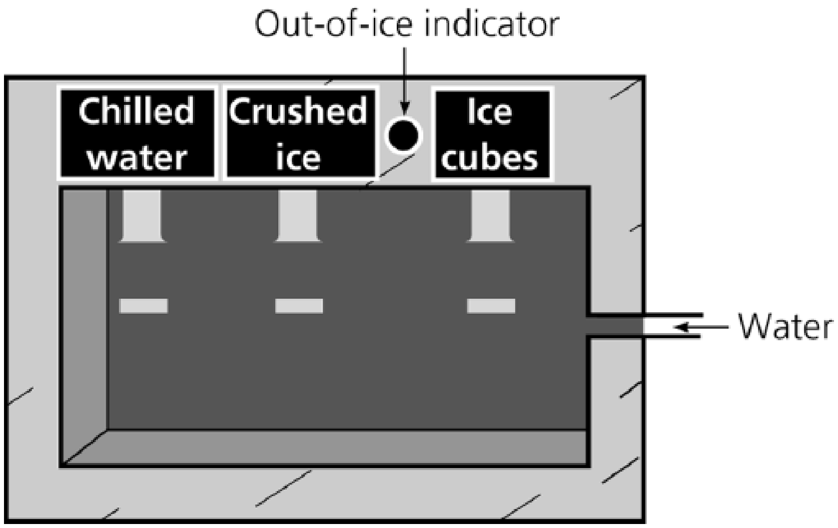
```
const int MAX_NUMBER = 500;
string names[MAX_NUMBER];
double salaries[MAX_NUMBER];
// employee names[i] has a salary of salaries[i]
```

- When a program needs data operations that are not directly supported by a language, you need an ADT
- You should first design the ADT by carefully specifying the operations before implementation

A water dispenser as an ADT

- Data: water
- Operations: chill, crush, cube, and isEmpty
- Data structure: the internal structure of the dispenser
- Walls: made of steel

  The only slits in the walls:
  - Input: water;
  - Output: chilled water, crushed ice, or ice cubes.



Crushed ice can be made in many different ways.
We don’t care how it was made

- Using an ADT is like using a vending machine.

ADT: Specification and Implementation

- Specification and implementation are disjointed:
  - **One** specification
  - **One or more** implementations
    - **Using different data structure**
    - **Using different algorithm**
- Users of ADT:
  - Aware of the specification **only**
    - Usage only based on the specified operations
  - Do not care / Need not know about the actual implementation
    - i.e. Different implementations do **not** affect the user

Abstraction as Wall: **Illustration**

```
int main()
{
   int result;

   result = factorial( 5 );

   ... ...
}
```

User of **`factorial( )`**

- **`main()`** needs to know
  - **`factorial()`**'s purpose
  - Its parameters and return value
- **`main()` does not** need to know
  - factorial() internal coding
- Different **`factorial()`** coding
  - Does not affect its users!
- We can build a wall to shield **`factorial()`** from **`main()`**!

```
int factorial( int n )
{
   if (n == 0)
      return 1;

   return n * factorial(n-1);
}
```

Implementation 1

```
int factorial( int n )
{
   int i, result = 1;

   for (i = 2; i <= n; i++)
      result *= i;

   return result;
}
```

Implementation 2

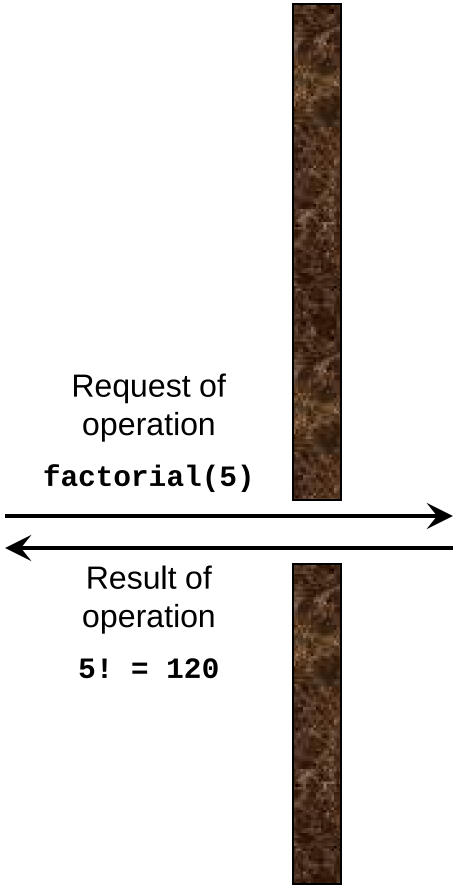
Specification as Slit in the Wall

```
int main()
{
   int fac5;

   fac5 = factorial(5);

   ... ...
}
```

User of **`factorial( )`**



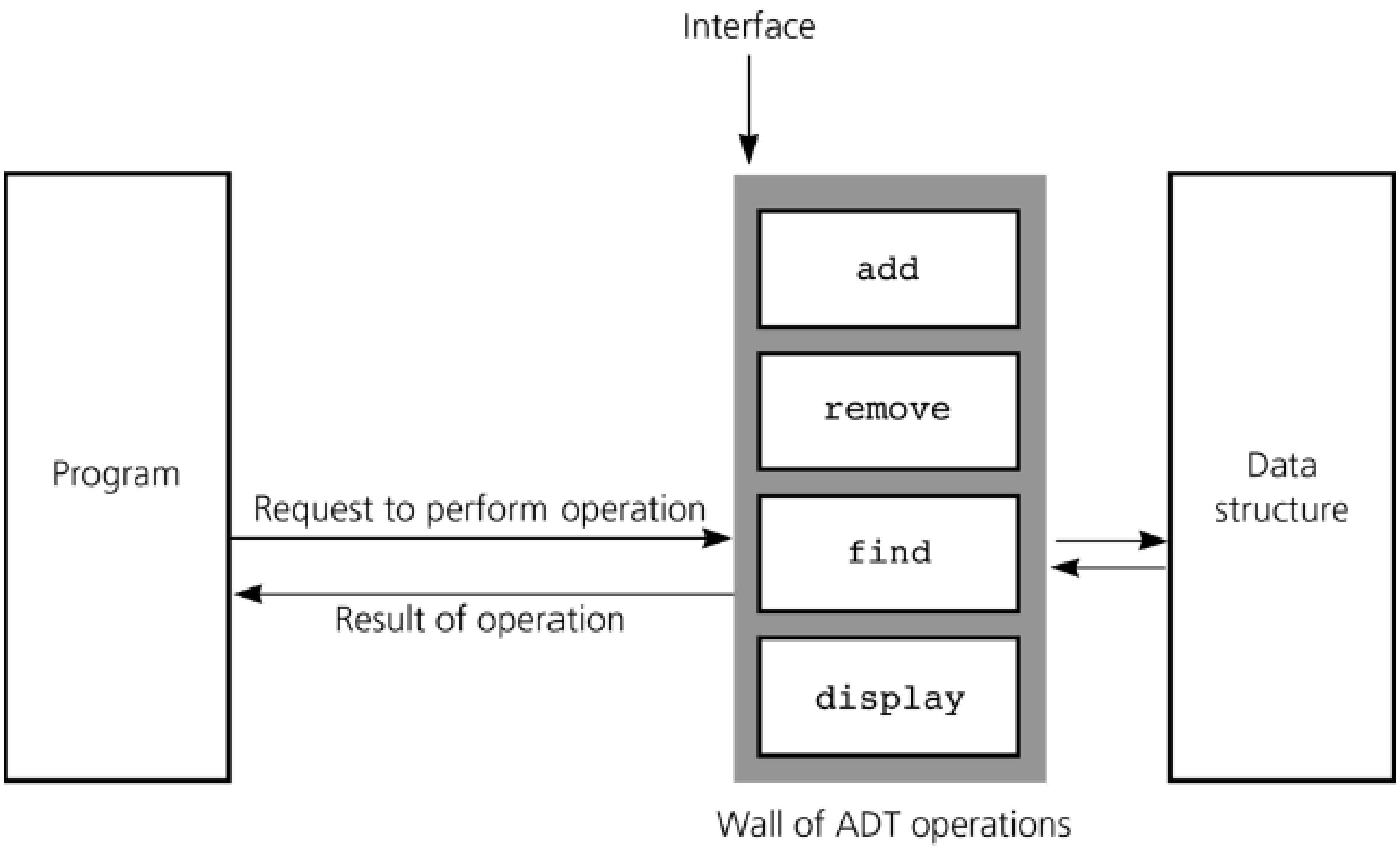

```
int factorial( int n )
{
    ... ... ...
}
```

Implementation

- User only depends on specification
  - Function name, parameter types and return type

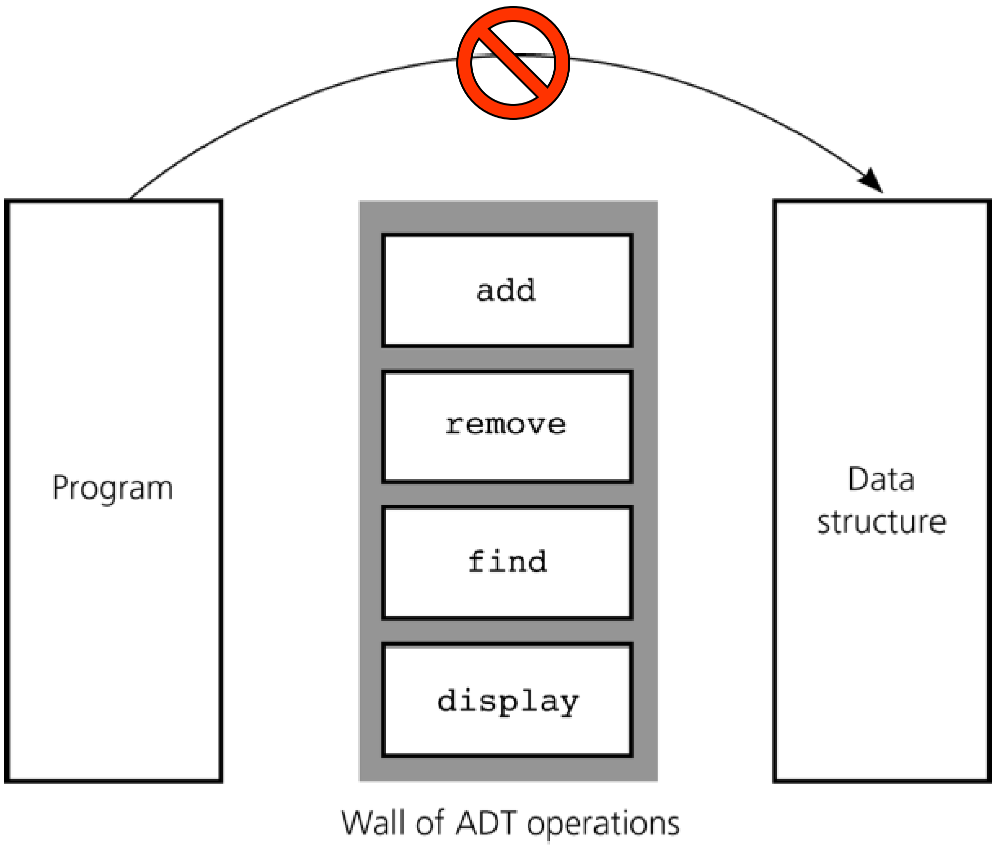
A wall of ADT operations

- ADT operations provide:
  - Interface to data structure
  - Secure access



Violating the Abstraction

- User programs **should not**:
  - use the underlying data structure directly
  - depend on implementation details



Abstract Data Types: When to use?

- When you need to operate on data that are not directly supported by the language
  - E.g. Complex Number, Module Information, Bank Account etc.

- Simple Steps:
  1. Design an abstract data type
  2. Carefully specify all operations needed
     - Ignore/delay any implementation related issues
  3. Implement them

Abstract Data Types: Advantages

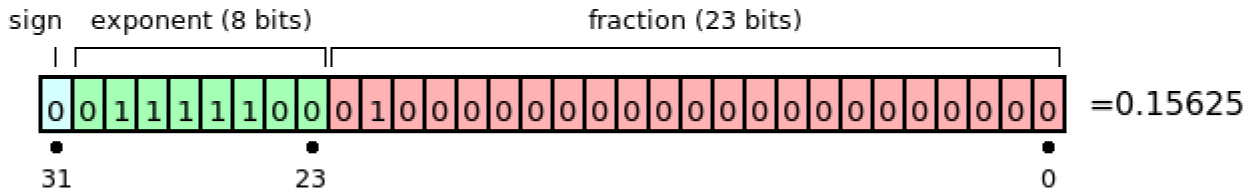
- Hide the unnecessary details by building walls around the data and operations
  - So that changes in either will not affect other program components that use them
- Functionalities are less likely to change
- Localise rather than globalise changes
- Help manage software complexity
- Easier software maintenance

ADT Examples

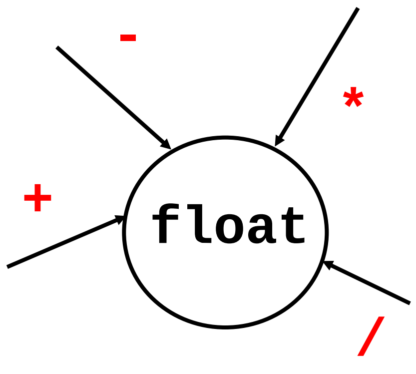
- Primitive Types as ADTs
  - A simple example
- Complex Number ADT
  - A detailed example to highlight the advantages of ADT
- Sphere ADT (own reading)
- All data structures covered later in the course are presented as ADTs
  - Specification: Essential operations
  - Implementation: Actual data structure and coding

ADT 1: Primitive Data Types

- Predefined data types are examples of ADT
  - E.g. int, float, double, char, bool
- Representation details are hidden to aid *portability*
  - E.g. float is usually implemented as

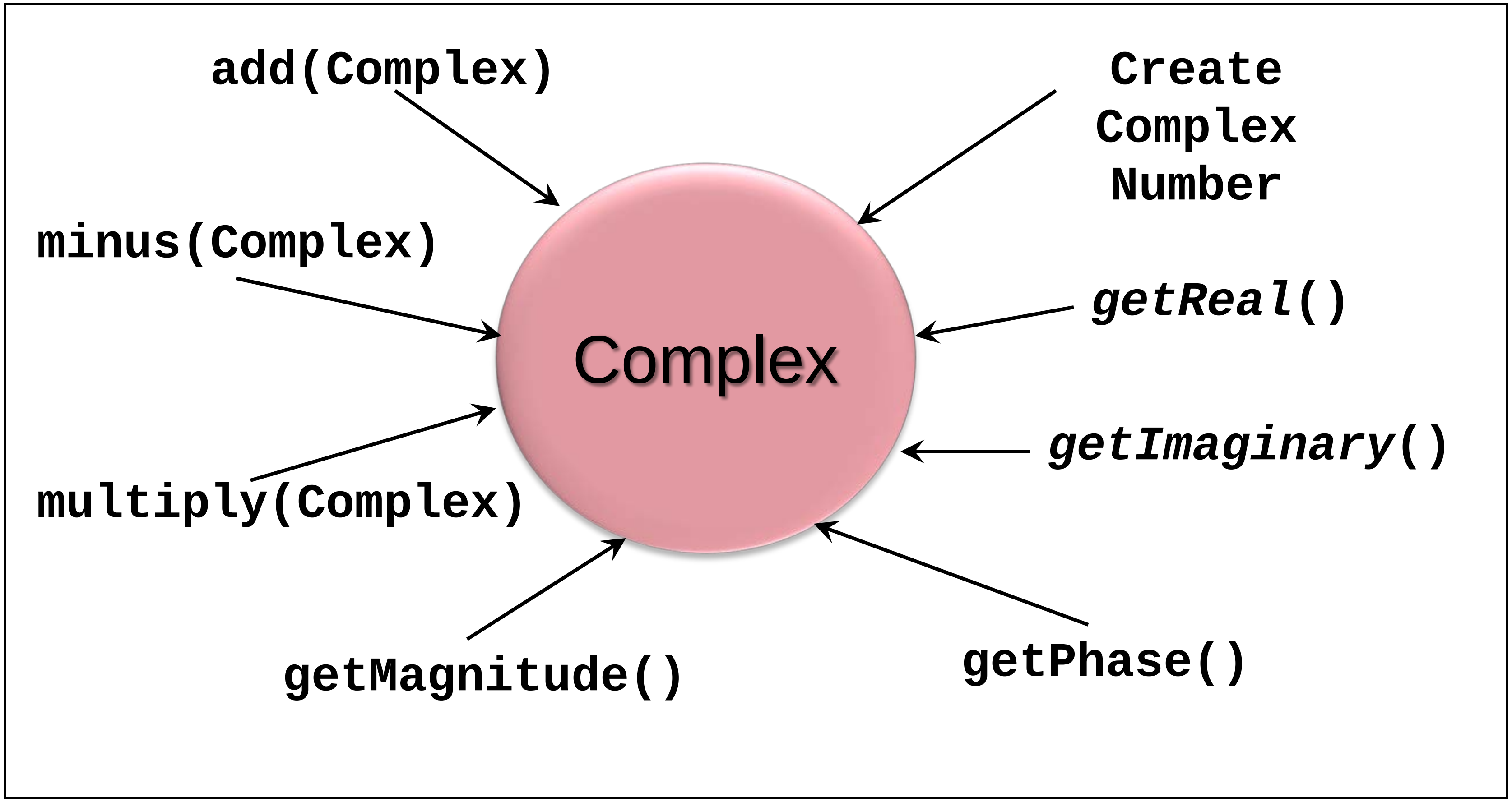


- However, as a user, you do not need to know the above to use float variable in your program



The **float** ADT

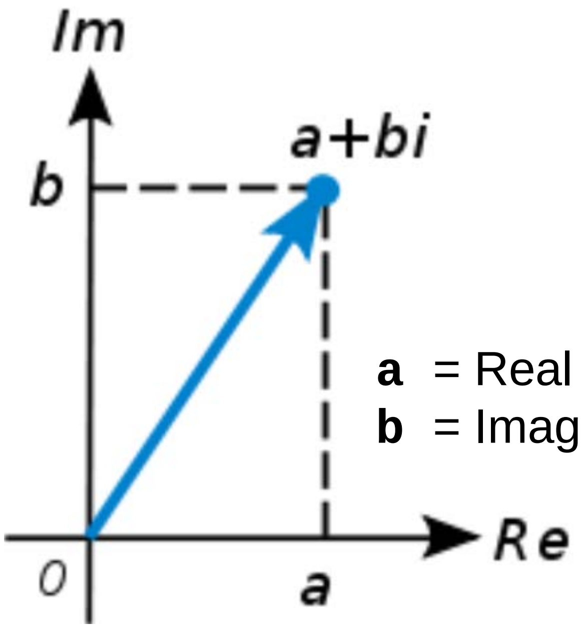
ADT 2: Complex Number



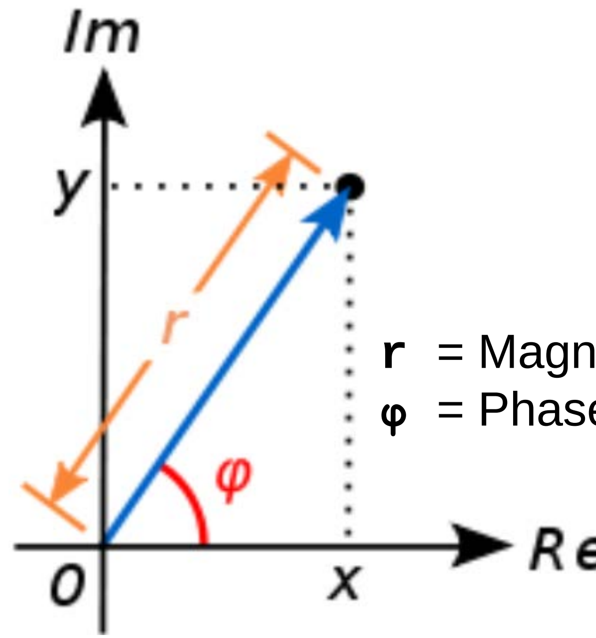
The **Complex** ADT

Complex Number: Representations

- Common representations of a complex number:



a = Real Part
b = Imaginary Part



r = Magnitude
φ = Phase (angle)

Rectangular Form

`( a + bi )`

Polar Form

`r( cos φ + isin φ )`

- Each form is easier to use in certain operations

Rectangular Form

- Complex numbers are added by separately adding the real and imaginary parts of the summands.

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

- Similarly, subtraction is defined by

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

- The multiplication of two complex numbers is defined by the following formula:

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$$

Polar Form

- Magnitude of complex number z = *a* + *i* *b*

$$r = |z| = \sqrt{a^2 + b^2}$$

$$\varphi = \begin{cases} \arctan(\frac{b}{a}) & \text{if a > 0} \\ \arctan(\frac{b}{a}) + \pi & \text{if a < 0 and b} \geq 0 \\ \arctan(\frac{b}{a}) - \pi & \text{if a < 0 and b < 0} \\ \frac{\pi}{2} & \text{if a = 0 and b > 0} \\ -\ \frac{\pi}{2} & \text{if a = 0 and b < 0} \\ \text{Indeterminate} & \text{if a 0 and b = 0} \end{cases}$$

Complex Number: Overview

- **Specification:**
  - Define the common expected operations for a complex number object
- **Implementation:**
  - Complex number can be implemented by at least two different internal representations
    - Keep the ***Rectangular form*** internally OR
    - Keep the ***Polar form*** internally
- Observes the ADT principle in action!

Complex Number: Specification

```
class Complex {
  public:

    double getReal();
    double getImaginary();

    double getMagnitude( );
    double getPhase( );

    void add( Complex& );
    void minus( Complex& );
    void multiply( Complex& );

    string toRectangularString();
    string toPolarFormString();
};
```

Operations to access information

Common arithmetic operations

Methods for printing

Complex.h

- At this point, implementation details are not important

User Program Example: Preliminary

```
//...header file not shown

int main () {
    Complex c1(...), c2(...);

    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    cout << c1.toPolarFormString() << endl;

    //...c2 can be printed in similar fashion

    cout << "add c2 to c1" << endl;
    c1.add( c2 );

    //print out c1 to check the addition
    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    return 0;
}
```

Depends on how constructors are defined

As a user, we can use the methods without worrying about the actual implementation!

testComplex.cpp

Complex Number – Version A

Rectangular Form Representation

`Complex`: Specification

```
class Complex
{
  private:
    double _real, _imag;

  public:
    Complex( double, double );

    double getReal( );
    double getImaginary( );

    double getMagnitude( );
    double getPhase( );

    void add( Complex& );
    void minus( Complex& );
    void multiply( Complex& );

    string toRectangularString( );
    string toPolarFormString( );
};
```

The real and imaginary part are kept as object attributes

Constructor defined:
Take in real and imaginary part as initial values

The same set of operations as defined in the specification. The only addition are the internal implementation details, i.e. attributes.

`ComplexRectangular.h`

Complex: Implementation

```
Complex::Complex( double real, double imag )
{
     _real = real;
     _imag = imag;
}

double Complex::getReal( )
{    return _real;   }

double Complex::getImaginary( )
{    return _imag;   }

double Complex::getMagnitude( )
{    return sqrt( _real*_real + _imag*_imag );  }

double Complex::getPhase( )
{   double radian;

    if (_real != 0 )
        radian = atan( _imag / _real );
    else if ( _imag > 0 ){
        radian = PI / 2;
    else
        radian = -PI / 2;
    return radian;
}
```

ComplexRectangular.cpp (part I)

Complex: Implementation

```
void Complex::add( Complex& otherC )
{
   _real = _real + otherC.getReal();
   _imag = _imag + otherC.getImaginary();

}

void Complex::minus( Complex& otherC )
{
   _real = _real – otherC.getReal();
   _imag = _imag – otherC.getImaginary();
}

void Complex::multiply ( Complex& otherC )
{
   double realNew, imagNew;

   realNew = _real * otherC.getReal() - _imag * otherC.getImaginary();
   imagNew  = _real * otherC.getImaginary() + _imag * otherC.getReal();

   _real = realNew;
   _imag = imagNew;

}
```

ComplexRectangular.cpp (part 2)

Complex: Implementation

```
string Complex::toRectangularString()
{
    ostringstream os;

    os << "(" << getReal() << ", " << getImaginary() << "i)";
    return os.str();
}

string Complex::toPolarFormString()
{
    double angle;
    ostringstream os;

    angle = getPhase();
    os << getMagnitude() << "( cos "  << angle;
    os <<  "+ i sin " << angle << ")";
    return os.str();
}
```

ComplexRectangular.cpp (part 3)

User Program Example: **Version 2.0**

```
//...header file not shown

int main () {
    Complex c1(30, 10), c2(20, 20);

    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    cout << c1.toPolarFormString() << endl;

    //...c2 can be printed in similar fashion

    cout << "add c2 to c1" << endl;
    c1.add( c2 );

    //print out c1 to check the addition
    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    return 0;
}
```

The implementation details do not affect the behaviour of an ADT

testComplex.cpp

Complex Number – Version B

Polar Form Representation

`Complex`: Specification

```
class Complex
{
  private:
    double _mag, _phase;

  public:

    Complex( double, double );

    double getReal( );
    double getImaginary( );

    double getMagnitude( );
    double getPhase( );

    void add( Complex& );
    void minus( Complex& );
    void multiply( Complex& );

    string toRectangularString( );
    string toPolarFormString( );
};
```

The magnitude and phase from the complex plane origin are kept as object attributes

Implementation details should not affect the specified operations!

`ComplexPolar.h`

Complex: Implementation

```
Complex::Complex( double magnitude, double phase )
{
    _mag = magnitude;
    _phase = phase;
}

double Complex::getReal( )
{
    return _mag * cos( _phase );
}

double Complex::getImaginary( )
{
    return _mag * sin( _phase );
}

double Complex::getMagnitude( )
{
    return _mag;
}

double Complex::getPhase( )
{
    return _phase;
}
```

Note that the two parameters have different meaning compared to the **Complex** version

Since we keep only magnitude and phase as attributes, the real and imaginary parts need to be calculated

ComplexPolar.cpp (part I)

Complex: Implementation

```
void Complex::add( Complex& otherC)
{
    double real, imag;

    real = getReal() + otherC.getReal();
    imag = getImaginary() + otherC.getImaginary();

    _mag =  sqrt( real*real + imag*imag );
    if (real != 0 ){
        _phase = atan( imag / real );
    } else if ( imag > 0 ){
        _phase = PI / 2;
    } else {
        _phase = -PI / 2;
    }
}

void Complex::minus( Complex& otherC)
{
    double real, imag;

    real = getReal() – otherC.getReal();
    imag = getImaginary() – otherC.getImaginary();

    Convert back to polar form, similar to add() above
}
```

Convert to rectangular form for addition

Convert back to polar form

ComplexPolar.cpp (part 2)

Complex: Implementation

```
void Complex::multiply ( Complex& otherC)
{
    _mag *= otherC.getMagnitude();
    _phase += otherC.getPhase();
}

string Complex::toRectangularString()
{
    Code similar to rectangular form. Not Shown.
}

string Complex::toPolarFormString()
{
    Code similar to rectangular form. Not Shown.
}
```

Multiplication in Polar form is easy though!

ComplexPolar.cpp (part 3)

- At this point:
  - We have two **independent implementations** of complex number
  - They have different internal working, but support the same behaviour

User Program Example: **Version 3.0**

```
//...header file not shown

int main () {
    Complex c1(31.62, 0.322), c2(28.28, 0.785);

    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    cout << c1.toPolarFormString() << endl;

    //...c2 can be printed in similar fashion

    cout << "add c2 to c1" << endl;
    c1.add( c2 );

    //print out c1 to check the addition
    cout << "Complex number c1:\n";
    cout << c1.toRectangularString() << endl;
    return 0;
}
```

Note that this version constructs with magnitude and phase

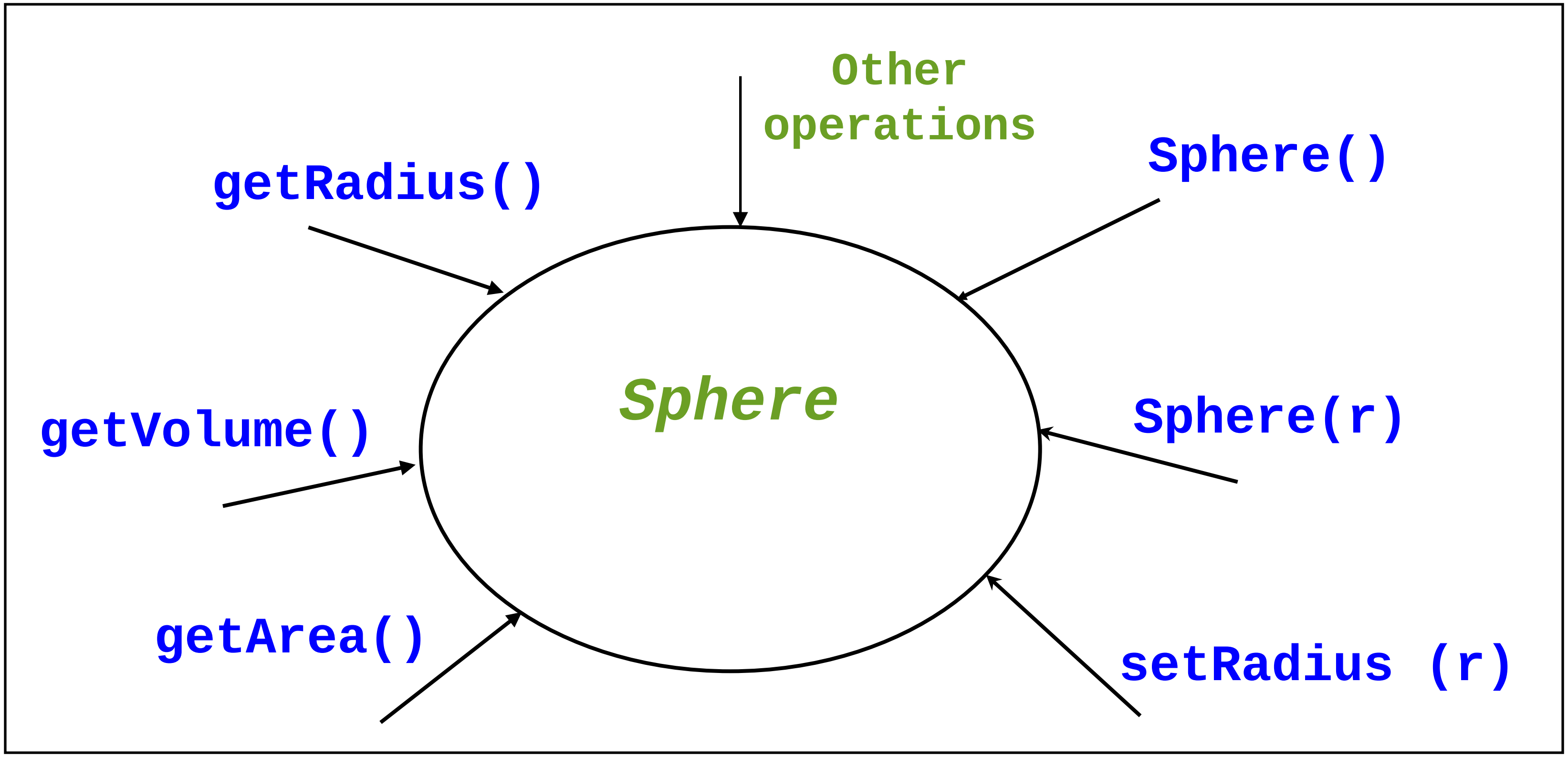
No change to code otherwise

testComplex.cpp

Complex Number: **Summary**

- This example highlights:
  - The separation of specification and implementation
  - A specification can have multiple implementations
- Why is this useful?
  1. We can try out different strategies in implementation without affecting the user
  2. We can use the best implementation in a certain situation
     - E.g. If multiplication is going to be the most common operations in a complex number program, we can choose to use the **`polar form`** implementation

ADT 3: Sphere



The **sphere** ADT

r: Radius, Floating Point Value

Sphere ADT: C++ Specification (1)

Tip:
Including Pre- and Post-condition in specification greatly facilitates the usage of the ADT

```
using namespace std;
const double PI = 3.14159;

class Sphere {
public:
   Sphere();
      // Default constructor: Creates a sphere and
      // initializes its radius to a default value.
      // Precondition: None.
      // Postcondition: A sphere of radius 1 created.
   Sphere(double initialRadius);
      // Constructor: Creates a sphere and initializes its radius.
      // Precondition: initialRadius is the desired radius.
      // Postcondition: A sphere of radius initialRadius created.
   void setRadius(double newRadius);
      // Sets (alters) the radius of an existing sphere.
      // Precondition: newRadius is the desired radius.
      // Postcondition: The sphere's radius is newRadius.
   double getRadius() const;
      // Determines a sphere's radius.
      // Precondition: None.
      // Postcondition: Returns the radius.
```

Sphere ADT: C++ Specification (2)

```
   double getDiameter() const;
       // Determines a sphere's diameter.
       // Precondition: None.
       // Postcondition: Returns the diameter.
   double getCircumference() const;
      // Determines a sphere's circumference.
      // Precondition: PI is a named constant.
      // Postcondition: Returns the circumference.
   double getArea() const;
      // Determines a sphere's surface area.
      // Precondition: PI is a named constant.
      // Postcondition: Returns the surface area.
   double getVolume() const;
      // Determines a sphere's volume.
      // Precondition: PI is a named constant.
      // Postcondition: Returns the volume.
   void displayStatistics() const;
      // Displays statistics of a sphere.
      // Precondition: None.
      // Postcondition: Displays the radius, diameter,
      // circumference, area, and volume.

private:
   double theRadius; // the sphere’s radius
}; // end class
```

Implement Sphere ADT (1)

```
#include "Sphere.h"                              // header file
#include <iostream>

Sphere::Sphere(): theRadius(1.0)
{ } // end default constructor

Sphere::Sphere(double initialRadius)
{
   setRadius (initialRadius);
} // end constructor

void Sphere::setRadius(double newRadius)
{
   if (newRadius > 0)
      theRadius = newRadius;
   else
      theRadius = 1.0;
} // end setRadius
```

Implement Sphere ADT (2)

```
double Sphere::getRadius() const
{
   return theRadius;
} // end getRadius

double Sphere::getDiameter() const
{
   return 2.0 * theRadius;
} // end getDiameter

double Sphere::getCircumference() const
{
   return PI * getDiameter();
} // end getCircumference

double Sphere::getArea() const
{
   return 4.0 * PI * theRadius * theRadius;
} // end getArea
```

Implement Sphere ADT (3)

```
double Sphere::getVolume() const {
   double radiusCubed = theRadius *
                         theRadius * theRadius;
   return (4.0 * PI * radiusCubed)/3.0;
} // end getVolume

void Sphere::displayStatistics() const {
   cout << "\nRadius = " << getRadius()
        << "\nDiameter = " << getDiameter()
        << "\nCircumference = " << getCircumference()
        << "\nArea = " << getArea()
        << "\nVolume = " << getVolume() << endl;
} // end displayStatistics
```

Sphere ADT: Sample User Program

```
// File: testSphere.cpp
#include <iostream>
#include "Sphere.h"

int main()
{
   Sphere unitSphere;          // radius is 1.0
   Sphere mySphere(5.0);       // radius is 5.0

   unitSphere.displayStatistics();

   mySphere.setRadius(4.2);    // resets radius to 4.2
   cout << mySphere.getDiameter() << endl;

   return 0;
} // end main
```

Colour Sphere: Extending Sphere ADT

- Inheritance can be used to easily extend existing ADT to support new ADT

- Example:
  - Colour sphere ADT : Sphere + colour

- The idea is similar to extending Bank Account class to get Saving Account class (Lecture 1a)

Colour Sphere ADT: C++ Specification

```
#include "Sphere.h"

enum Colour {RED, BLUE, GREEN, YELLOW};

class ColouredSphere : public Sphere
{
public:
   ColouredSphere(Colour);
   ColouredSphere(Colour, double);
   void setColour(Colour);
   Colour getColour();

private:
   Colour c;

}; //end ColouredSphere class
```

Implement Colour Sphere ADT

```
#include "ColouredSphere.h"

ColouredSphere::ColouredSphere (Colour initialColour)
       : Sphere(), c(initialColour)
{ } // end constructor

ColouredSphere::ColouredSphere (Colour initialColour,
                                double initialRadius)
       :Sphere(initialRadius),c(initialColour)
{ } // end constructor

void ColouredSphere::setColour (Colour newColour)
{
   c = newColour;
} // end setColour

Colour ColouredSphere::getColour ()
{
   return c;
} // end getColour
```

Colour Sphere ADT: Sample User Program

```
#include "ColouredSphere.h"
#include <iostream>

using namespace std;
int main() {

    ColouredSphere ball(RED);

    ball.setRadius(5.0);
    cout << "The ball's diameter is ";
    cout << ball.getDiameter() << endl;

    ball.setColour(YELLOW);
    cout << "The ball's colour is ";
    cout << ball.getColour() << endl;

    return 0;
}
```

Output:

```
The ball’s diameter is 10.0
The ball’s colour is 3
```

Summary

- Abstraction is a powerful technique
  - Data Abstraction
  - Function Abstraction
- Abstract Data Type
  - External Behaviour
    - The specification
  - Internal Coding
    - The actual implementation

References

- Carrano, Data Abstraction and Problem Solving with C++, Chapter 3.
- Source: The two diagrams of complex number representation are taken from http://wikipedia.org

Appendix A
Modular Design in C++

Organizing header and implementation

Modular Design in C++

- A class file can be partitioned into 2 parts:
  - Header File (XXXX.h) : Contains only the class declaration with **no coding**
  - Implementation File (XXXX.cpp): Contains the implementation file
- Facilitates reuse
  - For example, a user program useComplex.cpp requires the usage of Complex methods
  - Only need to include the Complex.h
  - Compilation

    g++ useComplex.cpp Complex.cpp
- Recall Bank example in Lecture 1a

Modular Design: `BankAcct.h` Example

```
//BankAcct.h

class BankAcct {

private:
  int _acctNum;
  double _balance;

public:
  BankAcct( int );
  BankAcct( int, double );
  int withdraw( double );
  void deposit( double );
};
```

Just like function prototype, **only data type is important** for method parameter

Do not forget the ";" for each method prototype

- The above is a correct class header, which includes only declaration but no implementation

Modular Design: **BankAcct.cpp** Example

```
#include "BankAcct.h"
//BankAcct.cpp

BankAcct::BankAcct( int aNum )
{  _acctNum = aNum;
     _balance = 0;
}

BankAcct::BankAcct( int aNum, double amt )
{  ... Code not shown ... }

int BankAcct::withdraw( double amount )
{   if (_balance < amount)
          return 0;
      _balance -= amount;
      return 1;
}

void BankAcct::deposit( double amount )
{  ... Code not shown ... }
```

Include the header

"**BankAcct::**" tells the compiler that this method belongs to **BankAcct** class

"**BankAcct::**" should appear **after the return type** and **before the method name**

Actual implementation

Modular Design: User Program

- A user program only includes the header file that contains the class declaration
- Example:
  - TestBankAcct.cpp that plays with the BankAcct class

```
//User program: TestBankAcct.cpp

#include "BankAcct.h"

int main( )
{   BankAcct ba1( 1234, 300.50 );
    BankAcct ba2( 9999, 1001.40 );

    ba1.withdraw(100.00);
    ba2.withdraw(100.00);
}
```

Include the header

Compilation in sunfire

- Make sure all the files (header files, implementation files, user program) are under the same directory for simplicity
- Example command:

```
g++ -Wall TestBankAcct.cpp BankAcct.cpp
```

- Separate Compilation is also possible:

```
g++ -Wall –c BankAcct.cpp
```

g++ produces BankAcct.o if everything is ok

```
g++ -Wall TestBankAcct.cpp BankAcct.o
```

- If there is no change to the **`BankAcct`** implementation, there is no need to recompile **`BankAcct.cpp`** for future usage
- E.g.

```
g++ -Wall anotherProgram.cpp BankAcct.o
```

Appendix B
Exception

Make sure we are correct …

Exception: Motivation

- What to do when a *user* of the class/function makes an error?
  - Could be accidental/intentional
  - Is crashing the program (abort execution) the best way?
- A class/function can be used by many other programs
  - The error may need to be handled differently in each scenario
- C++ provides **exception** as a solution:
  - Indicates an error has occurred
  - Let the user program handle the error
  - Abort program execution if it is not handled

Exception: Indication and Handling

- Exceptions in C++ consist of two components:
  - Exception indication
    - Code detects an error and informs the user program
    - Also known as **throw exception** or **raise exception**
  - Exception handling
    - User program receives the exception and handles it appropriately
    - If the exception is not handled, the program execution is aborted
    - Known as **exception catching** in C++
- Analogy:
  - Exception is just like baseball, the function/class throws it, the user program catches it
  - Uncaught baseball will hit you in the face ☺ (i.e. crashing the program)

Exception Indication: Syntax

- When an error is detected, an exception can be thrown with a *throw* statement.
- A *throw* statement can appear anywhere within a program
- Syntax:

```
throw ExceptionObject;
```

- The **`ExceptionObject`** can be any data type:
  - integer
  - string
  - any object
  - etc.
- The **`ExceptionObject`** should contain useful information about the exception

Exception Indication: Example

```
double divide(double a, double b) throw (string)
{
    if ( b == 0 )
        throw string("b is zero!!");

    return a / b;
}
```

Indicate that this function **may throw** a **string** as exception. Not compulsory.

Actual exception raising

- Good Practice:
  - Indicate what type of exception(s) a function/method may throw at the end of function/method header
  - Not compulsory

Exception Handling: Syntax

- The code that deals with an exception is said *to catch* (handle) the exception
- Syntax:

```
try {
        statement(s)
} catch ( ExceptionObject ) {
        statement(s)
}
```

- Terminology:
  - `try { ... }` is known as *try block*
  - `catch { ... }` is known as *catch block*
- A try block is followed by one or more catch blocks

Exception Handling: Example

```
int main()
{  double a, b;
   cin >> a >> b;
   try {
       cout << divide(a,b) << endl;
   } catch (string excpt){
       cout << excpt << endl;
   }
}
```

Indicate the data type of the expected exception

- The above program handles the exception by printing out the error message
  - Other handling is possible depending on the program’s need
- Other possibilities:
  - Try to get another set of input again
  - Stop the program
  - Assume **b** to be 1 and proceed to do division
  - etc.

Exception: Execution Flow

```
double divide(double a, double b) throw (string)
{
    cout << "Before checking" << endl;
    if ( b == 0 )
        throw string("b is zero!!");
    cout << "After checking" << endl;
    return a / b;
}
```

```
int main()
{  double a, b;
   cin >> a >> b;
   try {
      cout << "Before Divide()" << endl;
      cout << divide(a,b) << endl;
      cout << "After Divide()" << endl;
   } catch (string excpt){
      cout << "In Catch Block" << endl;
      cout << excpt << endl;
   }
   cout << "After try-catch" << endl;
}
```

User Input : 3.5 4.5

Output:
Before Divide()
Before checking
After checking
0.777778
After Divide()
After try-catch

User Input : 3.5 0

Output:
Before Divide()
Before checking
In Catch Block
b is zero!!
After try-catch

Exception: User Defined Exceptions

- Simple data type like *string* is usually adequate as exception object:
  - For simple program with few exception source/type

- For complex program with many exception types:
  - Design and organize specialized exception classes OR
  - Make use of the standard exception types defined in standard library

User Defined Exception

- Exception class:
  - Just like a normal C++ class
  - Main abilities:
    - Store information about the exception
    - Retrieve exception information

```
class SimpleException
{
 private:
    string _errorMsg;

 public:
  SimpleException ( string message )
  {   _errorMsg = message; }

  string getMessage () const
  {  return _errorMsg; }
};
```

Note:
This simple exception class will be used in subsequent lectures

User Defined Exception: Example

```
double divide(double a, double b)
                            throw (SimpleException)
{

    if ( b == 0 )
        throw SimpleException( "Divide by zero!" );

    return a / b;
}
```

```
int main()
{  double a, b;
   cin >> a >> b;
   try {
       cout << divide(a,b) << endl;
   } catch (SimpleException excpt){
       cout << excpt.getMessage() << endl;
   }
}
```