Lecture 5
Abstract Data Type

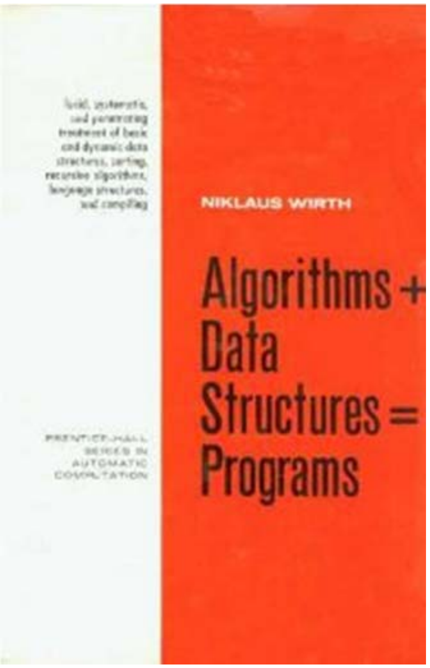
The Wall

Lecture Overview

- Abstraction in Programs
- Abstract Data Type
  - Definition
  - Benefits
- Abstract Data Type Examples

Abstraction

- The process of isolating implementation details and extracting only **essential property** from an entity



- Program = data + algorithms
- Hence, abstractions in a program:
  - **Data abstraction**
    - What operations are needed by the data
  - **Functional abstraction**
    - What is the purpose of a function (algorithm)

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

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 base on the specified operations
  - Do not care / need not know about the actual implementation
    - i.e. Different implementation do **not** affect the user

Abstraction as Wall : **Illustration**

```
int main() {
   int ans;
   ans = factorial(5);
   cout << ans << endl;

   return 0;
}
```

User of **`factorial()`**

```
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

- **`main()`** needs to know
  - **`factorial()`**'s purpose
  - Its parameters and return value
  - *Its limitations, $0 \le n \le 12$ for int*
- **`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()`**! →

Specification as Slit in the Wall

```
int main() {
    int ans;
    ans = factorial(5);
    cout << ans << endl;

    return 0;
}
```

User of `factorial( )`

Request of operation

`factorial(5)`

Result of operation

`5! = 120`

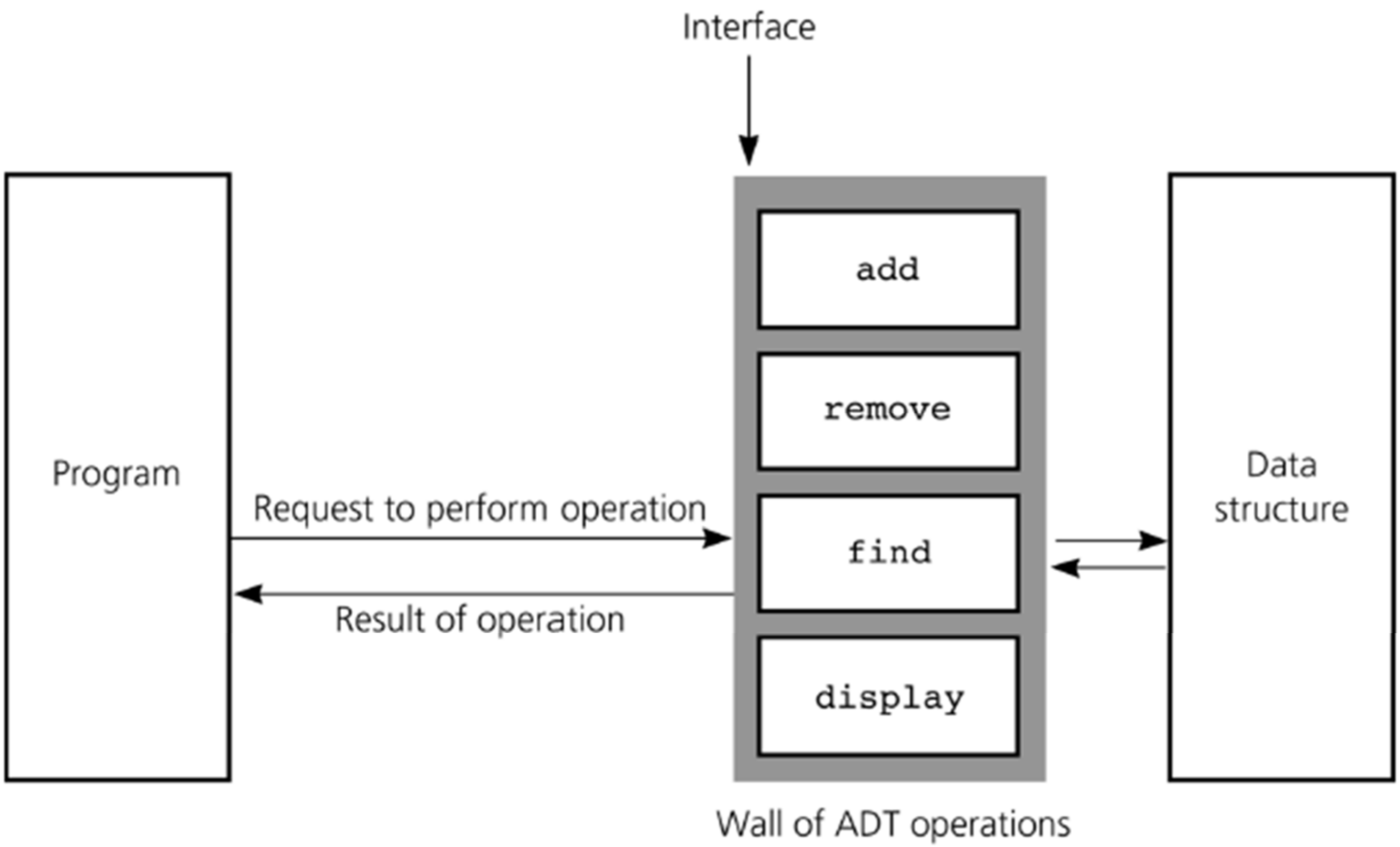
```
int factorial(int n) {
    ... ... ...
}
```

Implementation

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

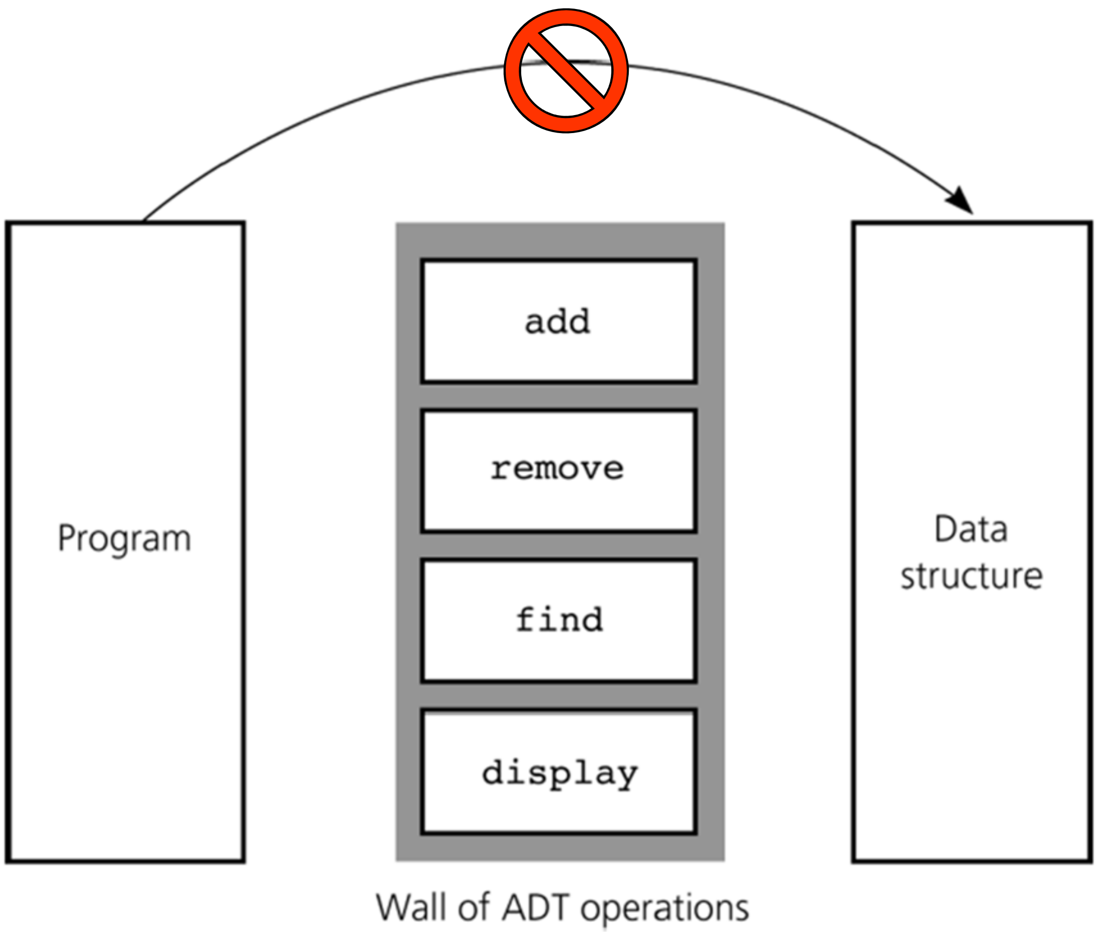
A wall of ADT operations

- ADT operations provides:
  - 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

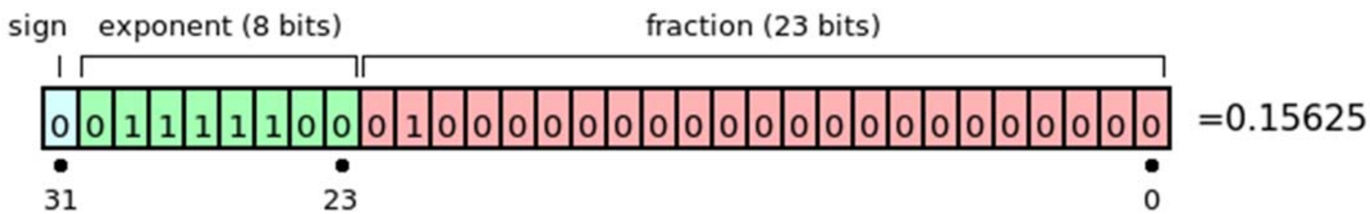
1. Primitive Types as ADTs
    - A simple example

2. Complex Number ADT
    - A detailed example to highlight the advantages of ADT

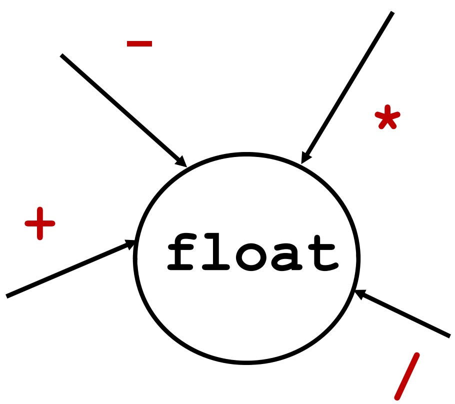
- All data structures covered later in the course are presented as ADTs
    - Specification: Essential operations
    - Implementation: Actual data structure and coding

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

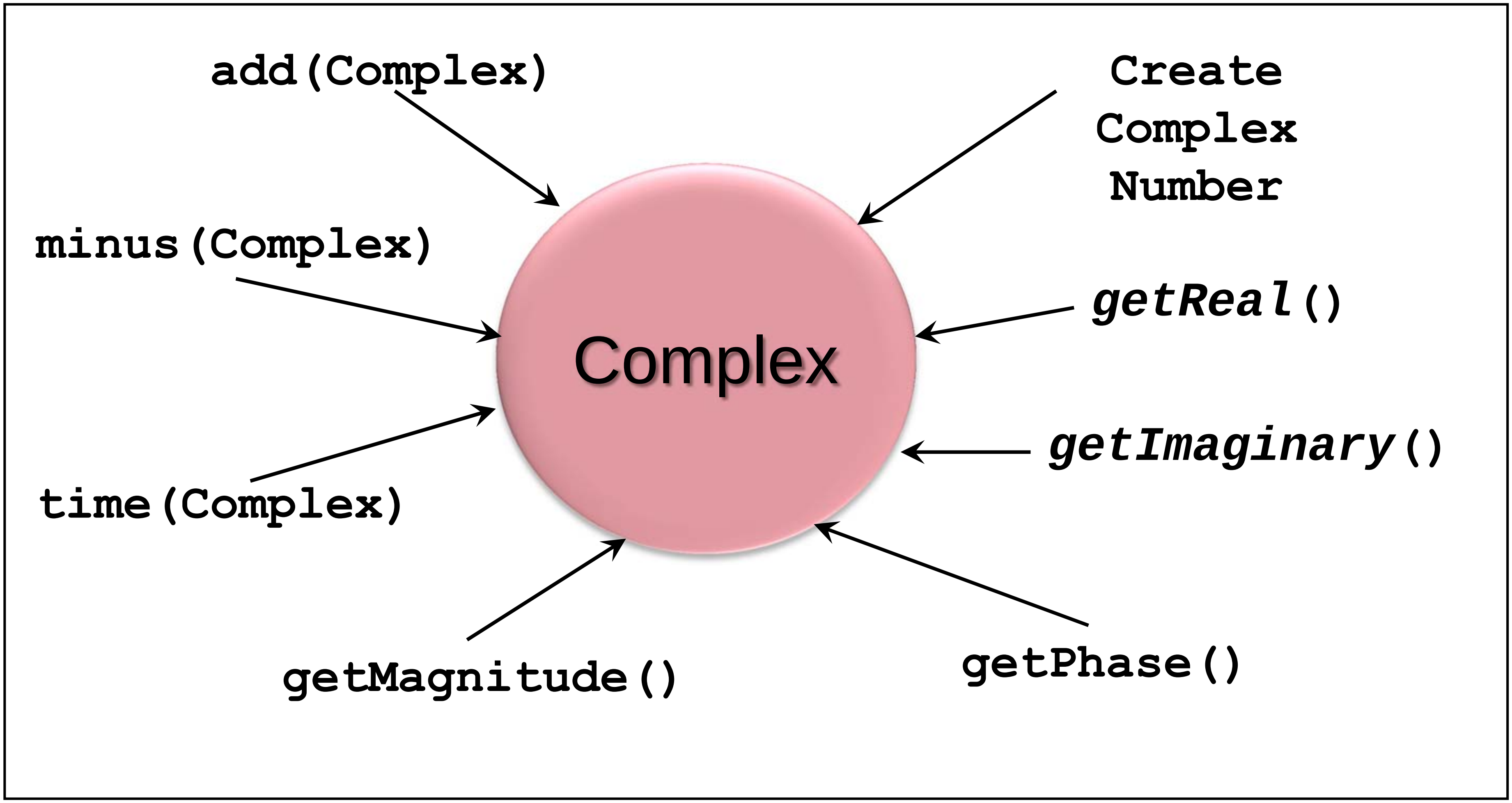


- However, as a user, you don't need to know the above to use float variable in your program



The `float` ADT

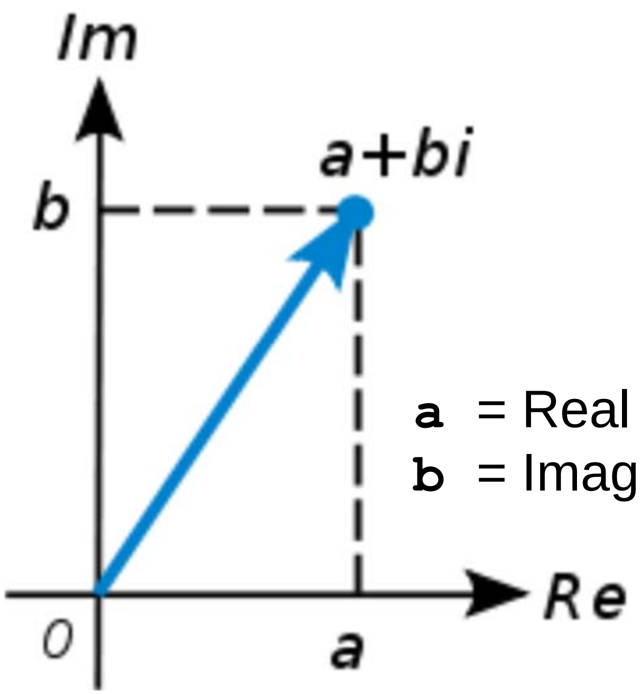
ADT 2 : Complex Number



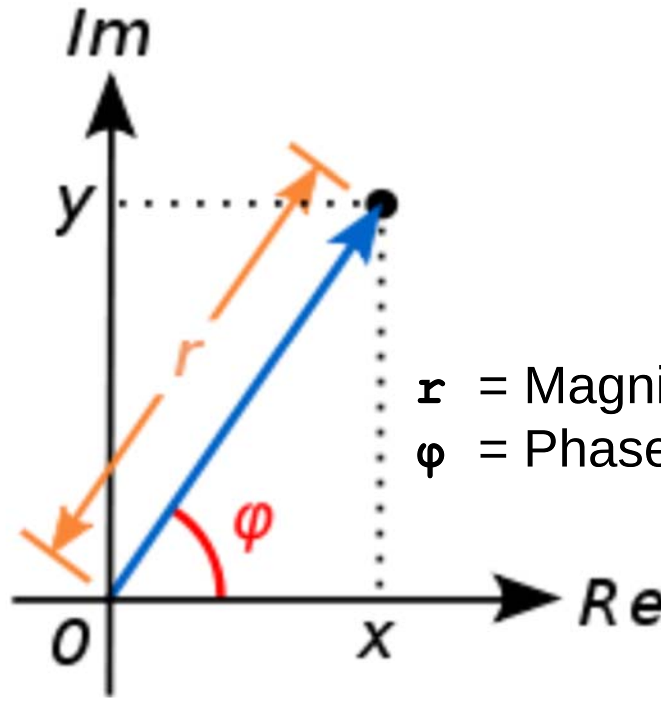
The **Complex** ADT

Complex Number: **Representations**

- Common representations of complex number:



a = Real Part
b = Imaginary Part



r = Magnitude
φ = Phase (angle)

Rectangular Form

(a + bi)

Polar Form

r(cos φ + i*sin φ)

- Each form is easier to use in certain operations

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: **Design**

- Complex number can be implemented as two classes:
    - Each utilize different internal representation

- A better alternative:
    - Let us define a **abstract base class** which captures the essential operations of a complex number
    - The super class is independent from the actual representation

- We can then utilize:
    - Inheritance and polymorphism to provide different actual implementations without affecting the user

Abstract Base Class: **ComplexBase**

```
class ComplexBase {
  public:

    virtual double getReal() = 0;
    virtual double getImaginary() = 0;

    virtual double getMagnitude() = 0;
    virtual double getPhase() = 0;

    virtual void add(ComplexBase*) = 0;
    virtual void minus(ComplexBase*) = 0;
    virtual void time(ComplexBase*) = 0;

    virtual string toRectangularString() = 0;
    virtual string toPolarFormString() = 0;
};
```

"Pure" specifier

All methods in this class are **pure virtual methods**

ComplexBase.h

- **ComplexBase** is a "placeholder" class
  - Specifies all necessary operations but with no actual implementation

User Program Example: **Preliminary**

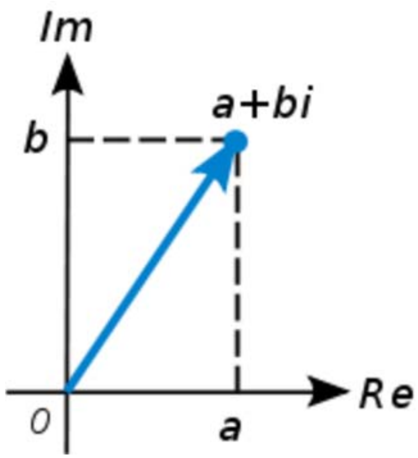
```
//...header file not shown

int main() {
    ComplexBase *c1, *c2;

    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;
}
```

To be replaced by actual implementations of the `ComplexBase` class

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

ComplexTest.cpp

Complex Number – Version A



Rectangular Form Representation

ComplexRectangular: Specification

```
class ComplexRectangular : public ComplexBase {
  private:
    double _real, _imag;

  public:
    ComplexRectangular(double, double);

    virtual double getReal();
    virtual double getImaginary();

    virtual double getMagnitude();
    virtual double getPhase();

    virtual void add(ComplexBase*);
    virtual void minus(ComplexBase*);
    virtual void time(ComplexBase*);

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

The real and imaginary part are kept as object attributes

Methods in this class do not have the **pure specifier** ➔ we will give actual implementation

ComplexRectangular.h

ComplexRectangular: Implementation

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

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

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

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

double ComplexRectangular::getPhase() {
    double radian;

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

Comments are removed and indentation are adjusted to fit the code in the slide.

ComplexRectangular.cpp (part I)

ComplexRectangular: Implementation

```
void ComplexRectangular::add(ComplexBase* complexPtr) {
   _real = _real + complexPtr->getReal();
   _imag = _imag + complexPtr->getImaginary();
}

void ComplexRectangular::minus(ComplexBase* complexPtr) {
   _real = _real - complexPtr->getReal();
   _imag = _imag - complexPtr->getImaginary();
}

void ComplexRectangular::time(ComplexBase* complexPtr) {
   double realNew, imagNew;

   realNew = _real * complexPtr->getReal() +
             _imag * complexPtr->getImaginary();
   imagNew  = _real * complexPtr->getImaginary() +
              _imag * complexPtr->getReal();

   _real = realNew;
   _imag = imagNew;
}
```

ComplexRectangular.cpp (part 2)

ComplexRectangular: Implementation

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

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

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

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

`ComplexRectangular.cpp` (part 3)

- Check your understanding:
  - Why does the arithmetic methods take **ComplexBase*** instead of **ComplexRectangular***?
  - Why do we use **complexPtr->getReal()** instead of **complexPtr->_real**?

User Program Example: **Version 2.0**

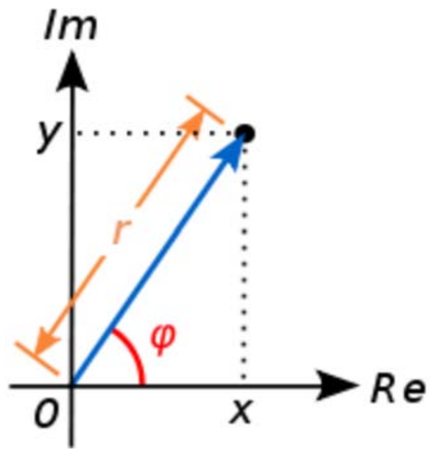
```
//...header file not shown

int main() {
    ComplexBase *c1, *c2;

    c1 = new ComplexRectangular(30, 10);
    c2 = new ComplexRectangular(20, 20);

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

    cout << "Complex number c2:\n";
    cout << c2->toRectangularString() << endl;

    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;
}
```

Subclass Substitution
c1, **c2** can point to **ComplexRectangular** objects

The implementation details doesn't affect the behavior of an ADT

ComplexTest.cpp

Complex Number – Version B



Polar Form Representation

ComplexPolar: Specification

```
class ComplexPolar : public ComplexBase {
  private:
    double _mag, _phase;

  public:
    ComplexPolar(double, double);

    virtual double getReal();
    virtual double getImaginary();

    virtual double getMagnitude();
    virtual double getPhase();

    virtual void add(ComplexBase*);
    virtual void minus(ComplexBase*);
    virtual void time(ComplexBase*);

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

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

ComplexPolar.h

ComplexPolar: Implementation

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

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

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

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

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

Note that the two parameters have different meaning compared to the `ComplexRectangular` verison

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

`ComplexPolar.cpp` (part I)

ComplexPolar: Implementation

```
void ComplexPolar::add(ComplexBase* complexPtr) {
    double real, imag;

    real = getReal() + complexPtr->getReal();
    imag = getImaginary() + complexPtr->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 ComplexPolar::minus(ComplexBase* complexPtr) {
    double real, imag;

    real = getReal() - complexPtr->getReal();
    imag = getImaginary() - complexPtr->getImaginary();
}
```

Convert to rectangular form for addition

Convert back to polar form

Convert back to polar form, similar to *add*() above

ComplexPolar.cpp (part 2)

ComplexPolar: Implementation

```
void ComplexPolar::time(ComplexBase* complexPtr) {
    _mag *= complexPtr->getMagnitude();
    _phase += complexPtr->getPhase();
}

string ComplexPolar::toRectangularString() {

}
```

Code similar to ComplexRectangular. Not Shown.

```
string ComplexPolar::toPolarFormString() {

}
```

Code similar to ComplexRectangular. 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 behavior

User Program Example: **Version 3.0**

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

int main() {
    ComplexBase *c1, *c2;

    c1 = new ComplexPolar(31.62, 0.322);
    c2 = new ComplexPolar(28.28, 0.785);

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

    cout << "Complex number c2:\n";
    cout << c2->toRectangularString() << endl;

    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 `ComplexPolar` constructs with magnitude and phase

No change to code otherwise

`testComplex.cpp`

User Program Example: **Version 4.0**

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

int main() {
    ComplexBase *c1, *c2;

    c1 = new ComplexRectangular(30, 10);
    c2 = new ComplexPolar(28.28, 0.785);

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

    cout << "Complex number c2:\n";
    cout << c2->toRectangularString() << endl;

    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 `c1` and `c2` need not be the same implementation!

Can you figure out how `c1` and `c2` can interoperate?

`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

Summary

- Abstraction is a powerful technique
  - Data Abstraction
  - Function Abstraction

- Abstract Data Type
  - External Behavior
    - The specification
  - Internal Coding
    - The actual implementation

References

- [Carrano]
  - 4th / 5th Edition, Chapter 3
- [Koffman & Wolfgang]
  - Chapter 1.4
- Source:
  - The two diagrams of complex number representation are taken from http://wikipedia.org