Object-Oriented Programming in C++

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

11/03/2016

Outline

Motivation

- What we know so far
- Structures and Functions
- What is Object-Oriented Programming?
- What is an object?

Classes

- Member variables and methods
- Lifetime of an Object
- Contructors
- Using objects
- Destructor

Building objects

- Composition
- Inheritance
- Polymorphism

Outline

Motivation

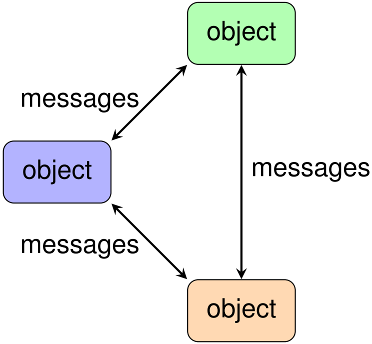
- so far our C/C++ code uses:
  - primitive data types and pointers
  - control flow constructs (loops, conditions)
  - functions
  - structures
- functions added the possibility to organize our code
- what we gain by this is simplicity by decomposing our large program into smaller, resuable components which are easier to understand
- structures finally gave us a way to define custom data types
- In the last few programs you have seen us define functions which work with the data stored in structures

Structures and Functions

```
// our data
struct Stack {
    int data[100];
    int head;
};

// methods working on data
void init( Stack & stack ) {
    stack.head = -1;
}

void push( Stack & stack , int value) {
    stack.data[++stack.head] = value;
}
```

What is Object-Oriented Programming?

- OOP started in 1960s (Languages: Simula, SmallTalk)
- Introduces objects as basic unit of computation
- Allows to extend type system
  - Usage: just like basic types (**`int`**, **`double`**, **`float`**, **`char`**, ...)
  - But with own structure and behavior



Static Languages (C++)

Types are known at compile time

Dynamic Languages (Python)

Types can be manipulated at runtime

Where are these “objects”?

- Objects exist in memory at runtime
- Just like objects of primitive types (integers, floating-point numbers)
  - We can interpret 4 bytes of data as integer number
  - We can interpret 8 bytes of data as floating-point number
- In C we have structs to create composite types containing multiple primitive types
- In C++ and other OOP languages this is further extended by associating behavior to a chunk of data through specifying methods to manipulate that data.

data	+	methods	=	object

TEMPLE UNIVERSITY

Object

State

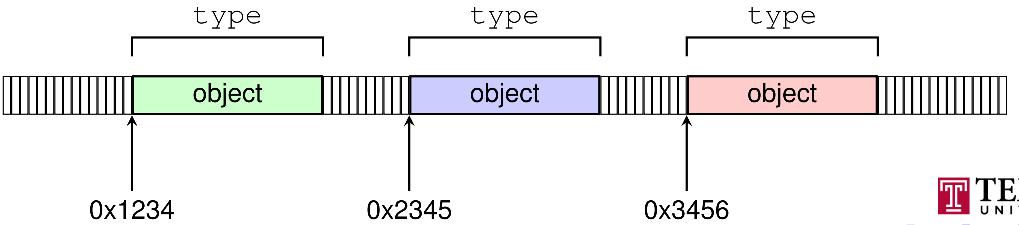
all properties of an object

Behavior

- How an object reacts to interactions, such as calling a certain method
- In OOP speak: *Response of an object when sending it messages*

Identity

Multiple objects can have the same state and behavior, but each one is a unique entitiy.



OOP in C/C++

- Object-Oriented Programming is just a concept
- an idea of how to think of your data and methods
- Not limited by a language
- It's just easier in some languages because they add features to support it
- You can do OOP in C using structures and functions
- The only downside in C is that it's more verbose

C++

- adds special language features for defining classes of objects
- it allows combining structures with methods working on them

Outline

C++ has **class**, C doesn't

Classes

```
class Vector2D {
public:
    double x;
    double y;

    double length() {
        return sqrt(x*x + y*y);
    }
};
```

Class

- like a **`struct`**, it defines a compound data type
- defines memory structure of objects
- how we can interact with objects
- how it reacts to interactions

Classes

```
class Vector2D {
public:
    double x;
    double y;

    double length() {
        return sqrt(x*x + y*y);
    }
};
```

Member Variables

- Variable in the scope of a class
- All objects of a class have their own memory and therefore **own copy** of each variable

Classes

```
class Vector2D {
public:
    double x;
    double y;

    double length() {
        return sqrt(x*x + y*y);
    }
};
```

Member Method

- A method which can be called for an object of the class
- It can access and modify the object state by manipulating member variables

`this` Pointer

Inside every member function you have access to the current object through a pointer called **`this`**. This is useful if you have local variables which have the same name as your member variables and you need to destinguish between them.

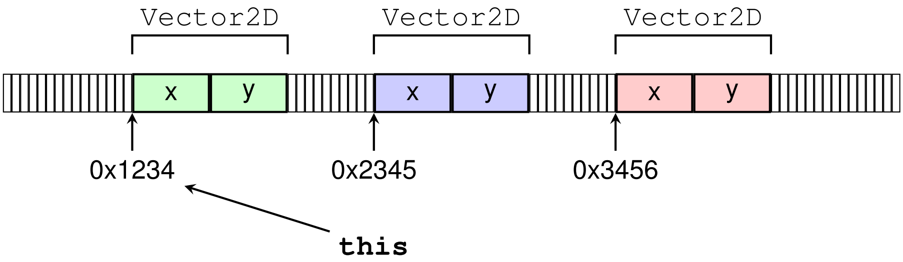
```
class Vector2D {
public:
  double x;
  double y;

  double length() {
    // this->x is the same as x
    return sqrt(this->x * this->x + this->y * this->y);
  }
};
```

`this` Pointer

```
Vector2D v1;
Vector2D v2
Vector2D v3;
```

```
v1.length()
v2.length()
v3.length()
```

If you call a member function on an object `v`, the **`this`** pointer inside that function will point to that particular object `v`

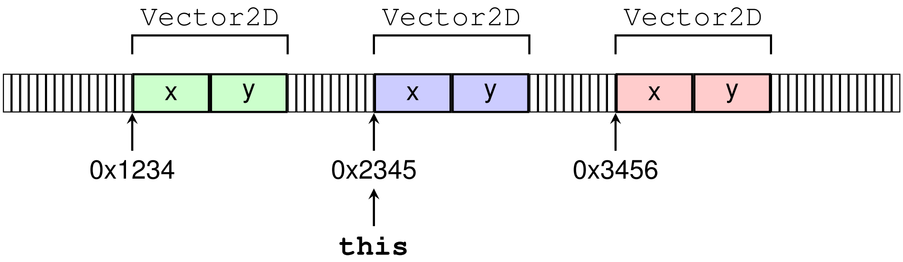


`this` Pointer

```
Vector2D v1;
Vector2D v2
Vector2D v3;
```

```
v1.length()
v2.length()
v3.length()
```

If you call a member function on an object `v`, the **`this`** pointer inside that function will point to that particular object `v`



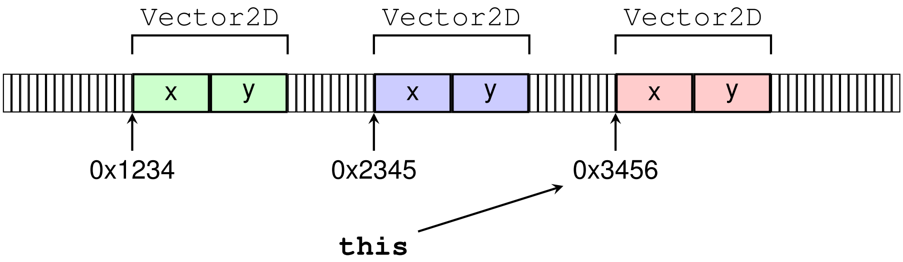
TEMPLE UNIVERSITY

`this` Pointer

```
Vector2D v1;
Vector2D v2
Vector2D v3;
```

```
v1.length()
v2.length()
v3.length()
```

If you call a member function on an object `v`, the **`this`** pointer inside that function will point to that particular object `v`



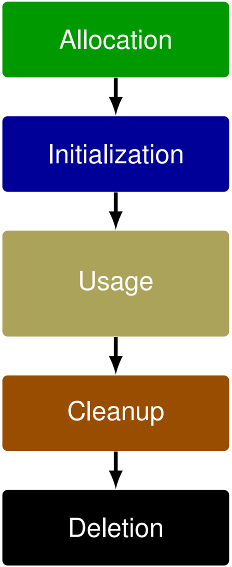
Classes

```
class Vector2D {
public:
    double x;
    double y;

    double length() {
        return sqrt(x*x + y*y);
    }
};
```

Public Interface

All variables and methods which can be used from an object when outside of the class code.

Lifetime of an Object



Allocation

Allocate enough memory to store object data/state

Initialization

Set an initial object state

Usage

- Interact with objects through methods
- Access and modify object data

Cleanup

Make sure that any resources are freed before deletion

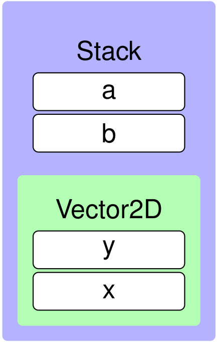
Deletion

Memory is freed, object ceases to exist

Allocation

Stack allocation

```
int main()
{
    int a = 30;
    int b = 50;
    Vector2D v;

    ...

    return 0;
}
```

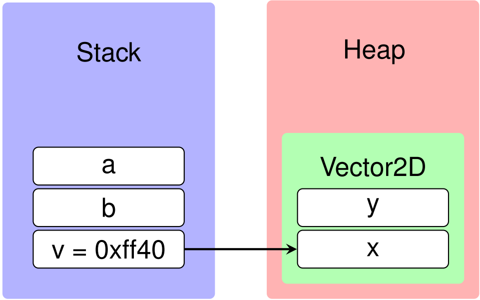


- variables of a **`class`** are just like a **`struct`**

Allocation

Dynamic/Heap allocation

```
int main()
{
   int a = 30;
   int b = 50;
   Vector2D* v = new Vector2D;

   ...

   return 0;
}
```



Initialization

```
class Vector2D {
public:
    double x;
    double y;

    Vector2D() {
        x = 0.0;
        y = 0.0;
    }

    Vector2D(double x0, double y0) {
        x = x0;
        y = y0;
    }
};
```

Constructors

- special methods which initialize an instance of a class
- multiple variants with different parameters possible
- initialize member variables

Default Contructor

- if there is no constructor at all, the compiler will create one with no parameters which does nothing

Constructor with initializers

```
class Vector2D {
public:
    double x;
    double y;

    Vector2D(double x0, double y0) : x(x0), y(y0)
    {
        // same as assigning x = x0 and y = y0
        // the only difference is that it comes before this
        // block of code
    }
};
```

Examples of using a constructor

Stack objects

```
Vector2D v1;
Vector2D v2();
Vector2D v3(10.0, 20.0);
```

Heap objects

```
Vector2D * v4 = new Vector2D;
Vector2D * v5 = new Vector2D();
Vector2D * v6 = new Vector2D(10.0, 20.0);
```

Usage

Stack Objects

```
{ // inside any block
    Vector2D v;

    // access members
    v.x = 10.0;
    v.y = 20.0;

    // call member functions
    double len = v.length();
} // end of scope -> deletion
```

Heap Objects

```
Vector2D * pv = new Vector2D();

// access members
pv->x = 10.0;
pv->y = 20.0;

// call member functions
double len = pv->length();

delete pv; // explicit deletion
```

TEMPLE UNIVERSITY

Cleanup

```
class Vector {
    double * data;
public:
    Vector(int dim) {
        data = new double[dim];
    }

    ~Vector() {
        delete [] data;
    }
};
```

- If the lifetime of variable on the stack ends or if an object is removed from the heap using **`delete`**, C++ calls a special method before cleaning up
- This method is called the **destructor** and has the name `~ClassName`

Responsibilities:

- Cleanup before destruction
- Free any acquired resources (file handles, heap memory)

Encapsulation

- classes allow us to bundle data with methods acting on that data
- our methods implement a specific behavior, which we present to the outside world
- other developers using our class can look at it as a **black box**
- they do not have to understand every detail of how it does its function, but only how to access it
- you can control what developers of your class can use with the **access modifiers** **`public`**, **`private`**, and **`protected`**

Access modifiers

`public`

Everyone outside of a class can access a data member or member function

`private`

Only code inside a class, so only member functions of that class, can access these data members and member functions.

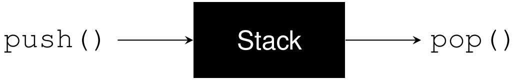
`protected`

Only code inside a class and its subclasses can access these data members and member functions. (we'll be covering subclasses in a bit)

Encapsulation

Example: Stack

- a stack has two methods
- `push` to add to it
- `pop` to remove the last pushed
- we do not care how it does it (arrays, linked-list, etc.)




```
class Stack {
private:
    int data[100];
    int head;

public:
    void push(int value);
    int pop();
};
```

Rationale behind access modifiers

- limiting access to class members is a design tool
- unlike Python's "we're all adults" philosophy, in C++ you can set up boundaries to guide other developers
- it prevents mistakes by not allowing to mess with implementation internals
- if you try to do something which wasn't intended, like accessing a private member, you get a compile error

Problems without encapsulation

```
class Circle {
public:
    double radius;
    double diameter;
};

Circle c;
c.radius = 10.0;
c.diameter = 30.0;
```

- it is easy to create an inconsistent internal state of an object if you have full access to all internal data
- in this example having a diameter of 30 and a radius of 10 doesn't make sense, but the class interface allows it

Hiding implementation details

```
class Circle {
    double radius;
public:
    double getRadius() {
        return radius;
    }

    double getDiameter() {
        return radius * 2.0;
    }

    void setRadius(double radius) {
        this->radius = radius;
    }

    void setDiameter(double diameter) {
        this->radius = diameter / 2.0;
    }
};
```

```
Circle c;
c.setRadius(10.0);
c.getDiameter(); // ->20

c.setDiameter(30.0);
c.getRadius(); // ->15
```

- by making one data member private and removing the diameter variable, we can ensure a consistent state
- accessing and modifying data is only done through methods, which allows managing the object state

Difference between **struct** and **class**

- You can use **struct** to create classes as well
- They can have constructors, destructors and members functions
- The only difference to **class** is its default access level

```cpp
struct A {
    int a;
    int b;
};

// is the same as
struct A {
public:
    int a;
    int b;
};
```

```cpp
class A {
    int a;
    int b;
};

// is the same as
class A {
private:
    int a;
    int b;
};
```

Separating declaration and definition of classes

Combined Declaration and Definition

```
class Vector2D {
public:
    double x;
    double y;

    Vector2D(double x0, double y0) {
        x = x0;
        y = y0;
    }

    double length() {
        return sqrt(x*x + y*y);
    }
};
```

Separating declaration and definition of classes

Declaration

```
class Vector2D {
public:
    double x;
    double y;

    Vector2D(double x0, double y0);

    double length();
};
```

- Usually put into its own header file
- e.g., `vector2d.h`

Separating declaration and definition of classes

Declaration

```
class Vector2D {
public:
    double x;
    double y;

    Vector2D(double x0, double y0);

    double length();
};
```

- Usually put into its own header file
- e.g., `vector2d.h`

Definition

```
#include "vector2d.h"

Vector2D::Vector2D(double x0,
                   double y0) {
    x = x0;
    y = y0;
}

double Vector2D::length() {
    return sqrt(x*x + y*y);
}
```

- Usually put into their own source file
- e.g., `vector2d.cpp`

TEMPLE UNIVERSITY

Example: Stack as a class

stack.h

```
class Stack {
    int * data;
    int size;
    int head;

    void grow();

public:
    Stack();
    ~Stack();

    void push(int value);
    int pop();
};
```

stack.cpp

```
#include "stack.h"
#include <string.h>

Stack::Stack() {
    data = new int[10];
    size = 10;
    head = -1;
}

Stack::~Stack() {
    delete [] data;
    data = NULL;
}

void Stack::grow() {
    int new_size = size * 2;
    int * new_data = new int[new_size];
    memcpy(new_data, data, size * sizeof(int));
    delete [] data;
    data = new_data;
    size = new_size;
}
```

```
void Stack::push(int value) {
    if(head+1 == size) {
        grow();
    }
    data[++head] = value;
}

int Stack::pop() {
    if(head < 0) return -1;
    return data[head--];
}
```

stack_test.cpp

```
#include "stack.h"
#include <stdio.h>

int main() {
    Stack s;

    for(int i = 0; i < 20; ++i) {
        s.push(i);
    }

    int value;
    while ((value = s.pop()) >= 0) {
        printf("pop: %d\n", value);
    }
    return 0;
}
```

Compilation

```
g++ -o stack_test stack_test.cpp stack.cpp
```

Live demo: access violation

Trying to access `s.data` in `main` will produce a compile error because it is a **`private`** member.

Live demo: modifying implementation

The stack implementation can be easily changed without touching the main program as long as the class interface is unchanged.

Outline

Composition

```
class Point {
public:
    int x;
    int y;

    Point(int x, int y);
};

class Rectangle {
public:
    Point top_left;
    Point bottom_right;

    Rectangle(int x1, int y1,
              int x2, int y2);
};
```

- as with **structs**, we can build more complex classes by composing them out of other types

Composition: Initialization of member variables with a constructor

- Q: how do we initialize member variables with a constructor?

Composition: Initialization of member variables with a constructor

- Q: how do we initialize member variables with a constructor?
- A: ⇒ using initializers in constructor

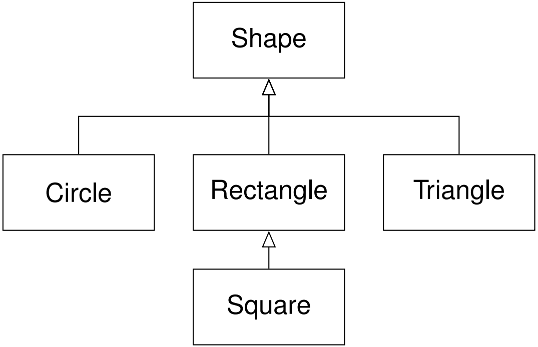
```
Rectangle(int x1, int y1, int x2, int y2) :
    top_left(x1, y1),
    bottom_right(x2, y2)
{
}
```

Type hierarchies and Inheritance

- objects of a class can inherit state and behavior of another class and make adjustments
- The class from which a class inherits is called **base class**
- We call a class which inherits from a base class a **derived** class

Usage:

- Extend classes with new functionality
- Make minor modifications
- Extract common functionality



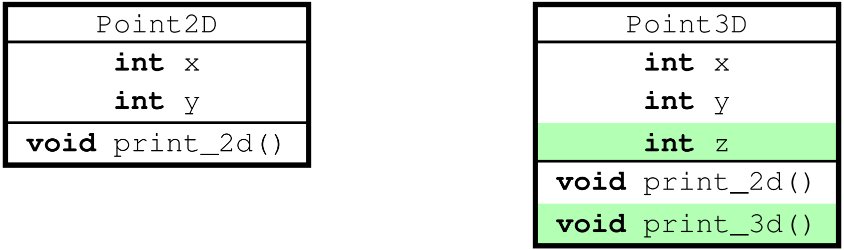
Inheritance

```
class Point2D {
public:
    double x;
    double y;

    void print_2d();
};

class Point3D : public Point2D {
public:
    double z;

    void print_3d();
}
```

- class `Point3D` inherits all data members and methods of class `Point2D` and makes them **`public`** accessible.
- **`protected`** inheritance only allows derived classes and the class itself access to the base class members
- **`private`** inheritance only allows the new class to access the base class members

Inheritance

What you can use after inheritance:



Inheritance: Calling the base class constructor

```
class Point2D {
public:
    double x;
    double y;

    Point2D(double x, double y) :
        x(x),
        y(y)
    {
    }
};
```

Inheritance: Calling the base class constructor

```
class Point2D {
public:
    double x;
    double y;

    Point2D(double x, double y) :
        x(x),
        y(y)
    {
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    Point3D(double x, double y,
            double z) :
        Point2D(x,y),
        z(z)
    {
    }
};
```

Using derived objects

Standard usage

```
Point2D p2d;
p2d.x = 11;
p2d.y = 22;
p2d.print_2d();

Point3D p3d;
p3d.x = 11;
p3d.y = 22;
p3d.z = 33;
p3d.print_2d();
p3d.print_3d();
```

Using derived objects

Standard usage

```
Point2D p2d;
p2d.x = 11;
p2d.y = 22;
p2d.print_2d();

Point3D p3d;
p3d.x = 11;
p3d.y = 22;
p3d.z = 33;
p3d.print_2d();
p3d.print_3d();
```

NEW: compatible base pointers

```
// Pointers are compatible!
Point2D * p = &p3d;

// use Point3D object
// like Point2D with
// base pointer type
p->x = 11;
p->y = 22;
p->print_2d();
```

Everything that worked with `Point2D` still works with objects of `Point3D`

Using derived objects

Standard usage

```
Point2D p2d;
p2d.x = 11;
p2d.y = 22;
p2d.print_2d();

Point3D p3d;
p3d.x = 11;
p3d.y = 22;
p3d.z = 33;
p3d.print_2d();
p3d.print_3d();
```

NEW: compatible base **references**

```
// References are compatible!
Point2D & p = p3d;

// use Point3D object
// like Point2D with
// base pointer type
p.x = 11;
p.y = 22;
p.print_2d();
```

Everything that worked with `Point2D` still works with objects of `Point3D`

```cpp
class Point2D {
public:
    double x;
    double y;

    void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```cpp
class Point3D : public Point2D {
public:
    double z;

    void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

```cpp
Point2D a;
a.x = a.y = a.z = 0.0;

Point3D b;
b.x = b.y = b.z = 1.0;

a.print();
b.print();
```

- Q: What is the output?

```
class Point2D {
public:
    double x;
    double y;

    void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

```
Point2D a;
a.x = a.y = a.z = 0.0;

Point3D b;
b.x = b.y = b.z = 1.0;

a.print();
b.print();
```

- Q: What is the output?

```
0.0, 0.0
1.0, 1.0, 1.0
```

```
class Point2D {
public:
    double x;
    double y;

    void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

```
Point2D * a = new Point2D;
a->x = a->y = a->z = 0.0;

Point2D * b = new Point3D;
b->x = b->y = b->z = 1.0;

a->print();
b->print();
```

- Q: What is the output?

```cpp
class Point2D {
public:
    double x;
    double y;

    void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```cpp
class Point3D : public Point2D {
public:
    double z;

    void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

```cpp
Point2D * a = new Point2D;
a->x = a->y = a->z = 0.0;

Point2D * b = new Point3D;
b->x = b->y = b->z = 1.0;

a->print();
b->print();
```

- Q: What is the output?

```
0.0, 0.0
1.0, 1.0
```

In both cases the compiler treats the objects as `Point2D`. **The behavior of `Point3D` is lost!**

Polymorphism

```
class Point2D {
public:
    double x;
    double y;

    virtual void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    virtual void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

- We need a mechanism to ensure object behavior stays the same even if we use a base class pointer or reference
- **Polymorphism** allows us to modify the behavior inherited from base classes and replacing their implementation with new methods
- Polymorphic methods must be declared as **`virtual`**

```
class Point2D {
public:
    double x;
    double y;

    virtual void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    virtual void print() {
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

```
Point2D * a = new Point2D;
a->x = a->y = a->z = 0.0;

Point2D * b = new Point3D;
b->x = b->y = b->z = 1.0;

a->print();
b->print();
```

```
0.0, 0.0
1.0, 1.0 1.0
```

TEMPLE UNIVERSITY

Usage of Polymorphism

```
class Shape {
    virtual double area() { ... };
};

class Rect : public Shape {
...
    virtual double area() { ... }
}

class Circle : public Shape {
...
    virtual double area() { ... }
}
```

```
Shape ** shapes = new Shape*[10];
shapes[0] = new Rect;
shapes[1] = new Circle;
...
double total = 0.0;

for(int i = 0; i < 10; i++) {
    total += shape[i]->area();
}
```

- Polymorphism allows you to use base class pointers and references to implement general algorithms and data structures which work with any derived type
- At runtime a mechanism called *dynamic dispatch* determines the type of an object and executes the correct method for that type

Abstract classes

```
class Shape {
    virtual double area() = 0;
};

class Rect : public Shape {
    double width;
    double height;
public:
    ...
    virtual double area() {
        return width * height;
    }
};
```

- Virtual functions without implementation are called **pure-virtual functions**
- Classes containing pure-virtual functions are called **abstract classes**. You can not create objects from them
- Behavior must be defined in derived classes
- but abstract base class pointer is compatible with all derived pointer types

```
Shape * s = new Rect;
s->area();
```

TEMPLE UNIVERSITY

Calling the base class implementation

```
class Point2D {
public:
    double x;
    double y;

    virtual void print() {
        printf("%f,%f\n", x, y);
    }
};
```

```
class Point3D : public Point2D {
public:
    double z;

    virtual void print() {
        Point2D::print();
        printf("%f,%f,%f\n", x, y, z);
    }
};
```

- the base class implementation of a member function is called by fully qualifying its name
- `Base::function_name(...)`

Example: Drawing Shapes