CSci 3081W: Program Design and Development

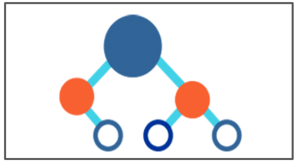
Lecture 03 - Classes

Brief Introduction

- My name is Frank Bender
- 2nd year masters student in Computer Science
- Research area is distributed systems for graphical applications
- Was a TA for this class for a few semesters
- Class is designed to be reflective of the industry
- Most influential class of my education

Roadmap for Today

Motivation for Design: Namespaces

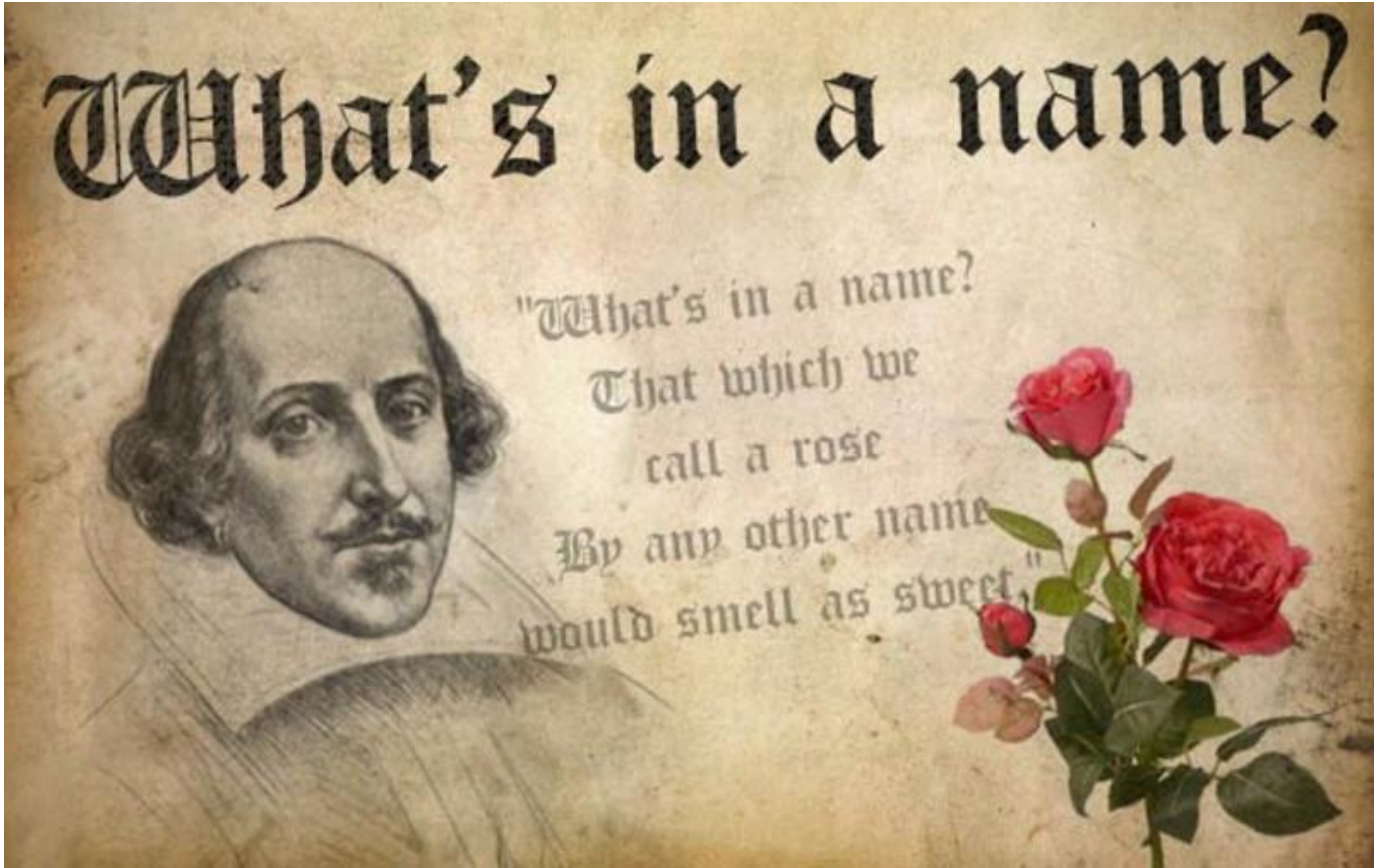


Design: Abstract Data Types

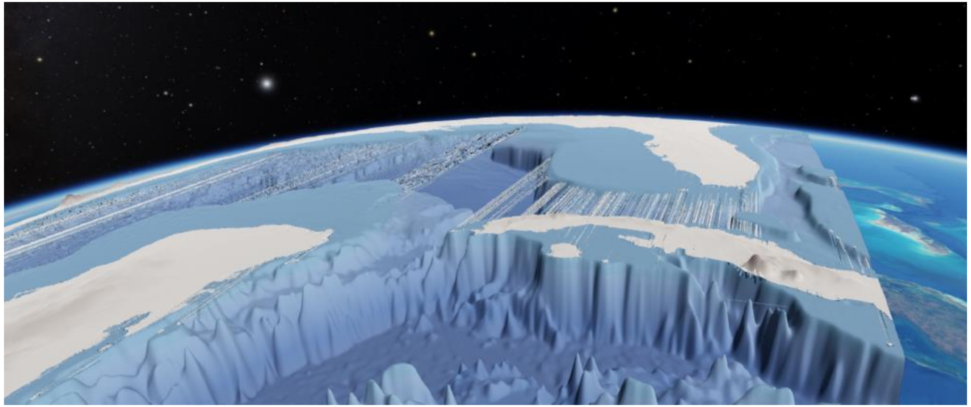
```
        < length; src_idx += 3, dst_idx += 4)
    uint8_t s0 = data[src_idx];
    uint8_t s1 = data[src_idx + 1];
    uint8_t s2 = data[src_idx + 2];

    dst[dst_idx + 0] = charset[(s0 & 0xfc) >> 2];
    dst[dst_idx + 1] = charset[((s0 & 0x03) << 4) | ((s1 & 0xf0) >> 4)];
    dst[dst_idx + 2] = charset[((s1 & 0x0f) << 2) | ((s2 & 0xc0) >> 6)];
    dst[dst_idx + 3] = charset[(s2 & 0x3f)];
}
if (src_idx < length)
{
    uint8_t s0 = data[src_idx
    uint8_t s1
```

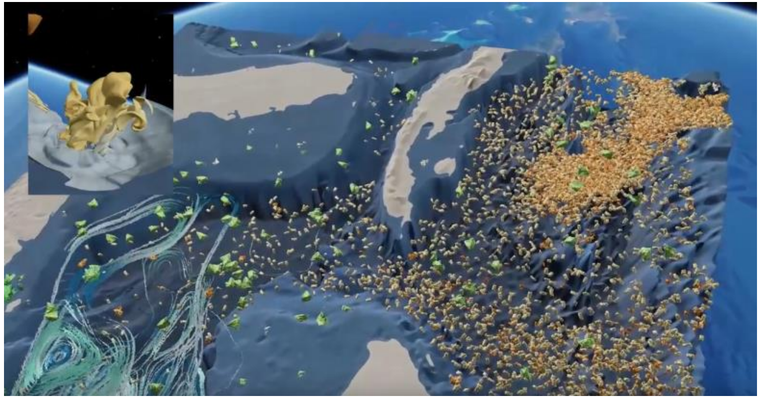
Development: C++ Classes

Motivating Example - Namespaces

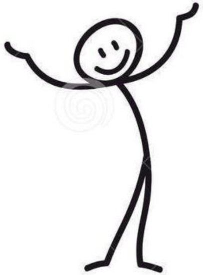


"The meeting of the Dans": building planetarium software





Daniel

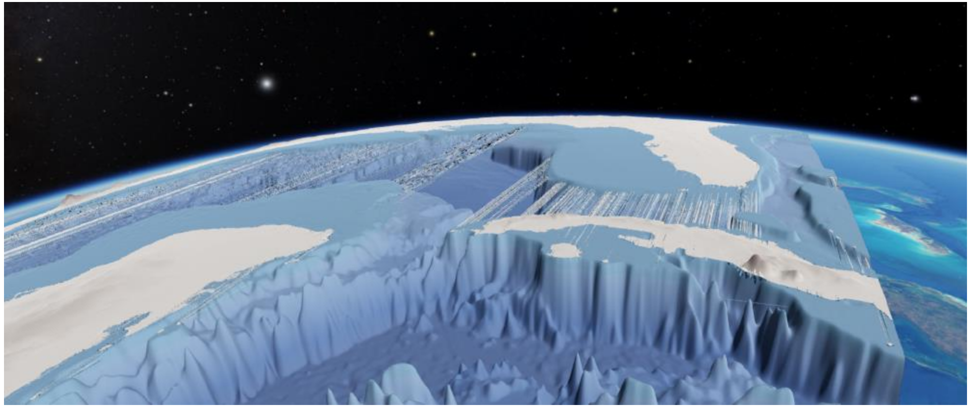


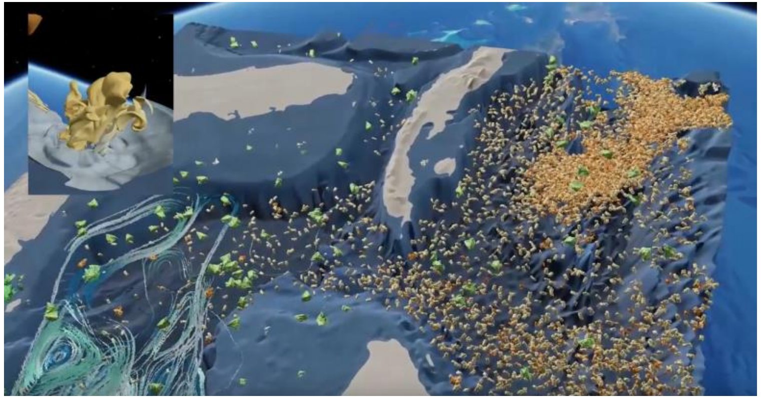
Daniel

Daniel

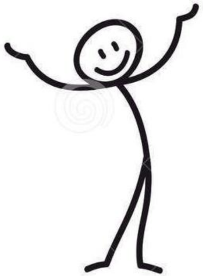
How do we distinguish between Dans?

“The meeting of the Dans”: building planetarium software





Daniel



Dr. Daniel Keefe

Daniel

Daniel Orban

Daniel

Daniel O. (not Orban)

We use a namespace!

- Last Name

The same is true for programming in any language.

Consider the **vec3** in each of the following libraries:

vmmlib A VECTOR AND MATRIX MATH LIBRARY

vec3

glm

vec3

Eigen

vec3

vec3

```
struct vec3 {
  float x, y, z;
};
```

Namespaces solve the problem of competing names.

Consider the **vec3** in each of the following libraries:

vmmlib

A VECTOR AND MATRIX MATH LIBRARY

vmml::vec3

glm

glm::vec3

Eigen

eigen::vec3

csci3081::vec3

```
struct vec3 {
  float x, y, z;
};
```

Namespaces solve the problem of competing names.

Consider the **vec3** in each of the following libraries:

vmmlib A VECTOR AND MATRIX MATH LIBRARY

vmml::vec3

glm

glm::vec3

Eigen

eigen::vec3

csci3081::vec3

Declaring a namespace:

```
namespace csci3081 {
        struct vec3 {
          float x, y, z;
        };
}
```

Using a namespace:

(a)

```
using namespace csci3081;
vec3 v;
cout << v.x << endl;
```

(b)

```
csci3081::vec3 v;
cout << v.x << endl;
glm::vec3 v2;
cout << v2[0] << endl;
```

Use the std namespace when using the C++ standard library.

Using the namespace means we do not need the scope operator ::.

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

int main()
{
                cout << "Hello World" << endl;
                return 0;
}
```

The scope operator :: allows us to use types without the using keyword.

```
#include <iostream>


int main()
{
                std::cout << "Hello World" <<
std::endl;
                return 0;
}
```

When would you use one or the other?

Design Principle: Naming is important even beyond namespaces.

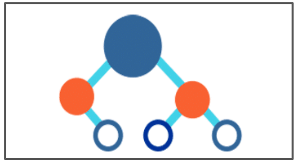
What happens if you don’t name things correctly?

- float temp1;
- class GenericItem {...};
- class ManagerOfThings {...};
- class SortAlgorithm {...};
- struct MultiStructuredTemplateBuildingPlan {...};
- a.execute();
- duck1.operation5();

We need to think about **design** questions in this class.

Roadmap for Today

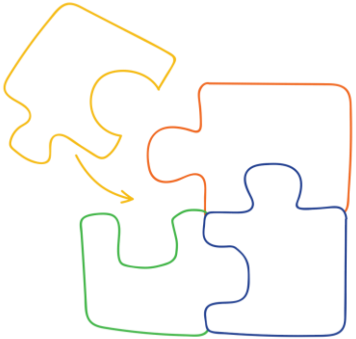
Motivation for Design: Namespaces



Design: Abstract Data Types

Development: C++ Classes

Design Exercise: What are some desirable characteristics for a good software design?



Design Exercise: What are some desirable characteristics for a good software design?

- **Minimal complexity**
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility**
- **Reusability**
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
    - McConnell (Code Complete - Ch. 5.2)

Object Oriented Design is one approach for meeting these criteria.

Design Exercise: What are some desirable characteristics for a good software design?

- **Minimal complexity** ← Clever solutions are not always the best
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility**
- **Reusability**
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
    - McConnell (Code Complete - Ch. 5.2)

Object Oriented Design is one approach for meeting these criteria.

Design Exercise: What are some desirable characteristics for a good software design?

- **Minimal complexity**
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility**
- **Reusability**
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
    - McConnell (Code Complete - Ch. 5.2)

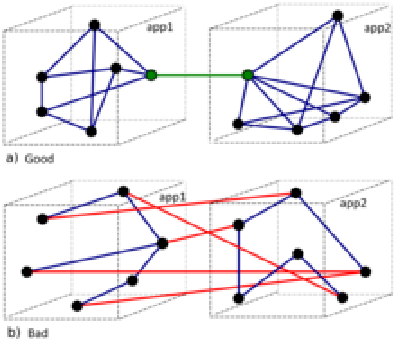
Django web development (in my experience - I’m probably doing something wrong)

Object Oriented Design is one approach for meeting these criteria.

Design Exercise: What are some desirable characteristics for a good software design?

- **Minimal complexity**
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility**
- **Reusability**
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
  - McConnell (Code Complete - Ch. 5.2)

“Everyone belongs to everyone else.”
(Brave New World - Huxley)



Object Oriented Design is one approach for meeting these criteria.

Design Exercise: What are some desirable characteristics for a good software design?

- **Minimal complexity**
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility** ← Design Patterns
- **Reusability**
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
    - McConnell (Code Complete - Ch. 5.2)

Object Oriented Design is one approach for meeting these criteria.

Design Exercise: What are some desirable characteristics for a good software design?

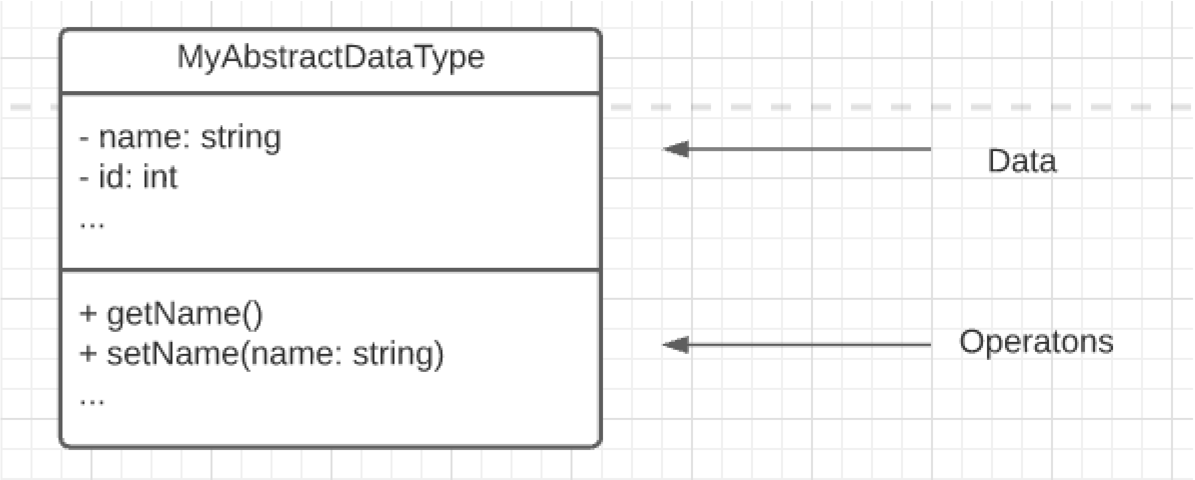
- **Minimal complexity**
- **Ease of maintenance**
- **Loose coupling**
- **Extensibility**
- **Reusability** ←
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification
  - McConnell (Code Complete - Ch. 5.2)

An Image Processing Library can be used inside many different types of applications.

Object Oriented Design is one approach for meeting these criteria.

Abstract data types (ADTs) are the foundation for object oriented programming.

UML - Unified Modeling Language



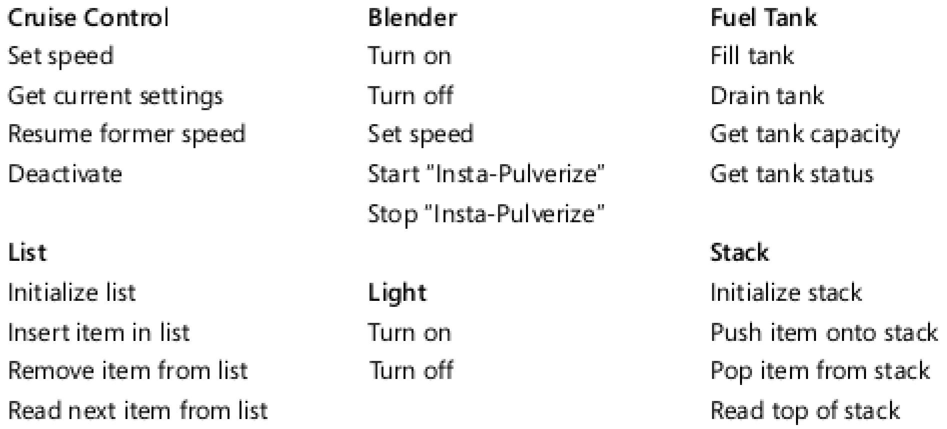
```
MyAbstractDataType
- name: string
- id: int
...
+ getName()
+ setName(name: string)
...
```

“An **abstract data type** is a collection of data and operations that work on the data”

- McConnell (Code Complete - Ch. 6.1)

Abstract data types (ADTs) are the foundation for object oriented programming.

Examples:



Cruise Control
Set speed
Get current settings
Resume former speed
Deactivate

List
Initialize list
Insert item in list
Remove item from list
Read next item from list

Blender
Turn on
Turn off
Set speed
Start "Insta-Pulverize"
Stop "Insta-Pulverize"

Light
Turn on
Turn off

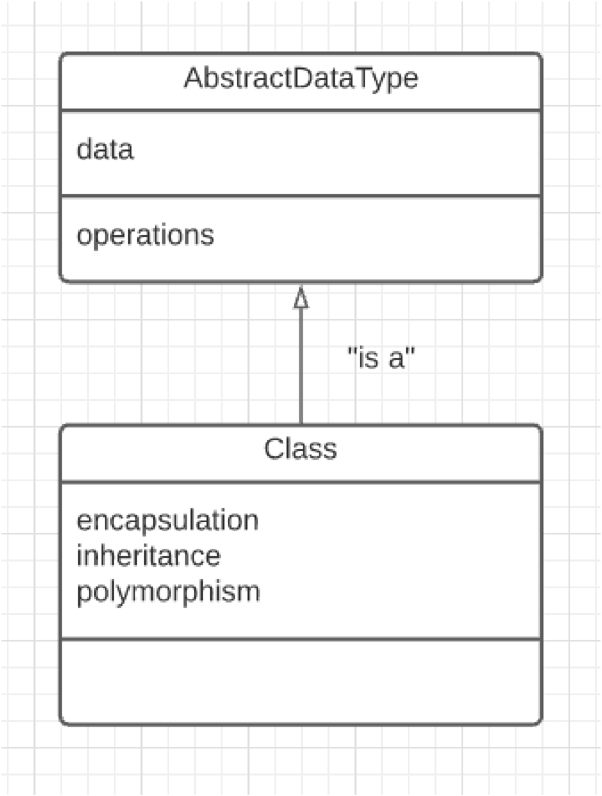
Fuel Tank
Fill tank
Drain tank
Get tank capacity
Get tank status

Stack
Initialize stack
Push item onto stack
Pop item from stack
Read top of stack

- McConnell (Code Complete - Ch. 6.1)

Notice that ADTs do not depend on a programming language.

So what are classes?



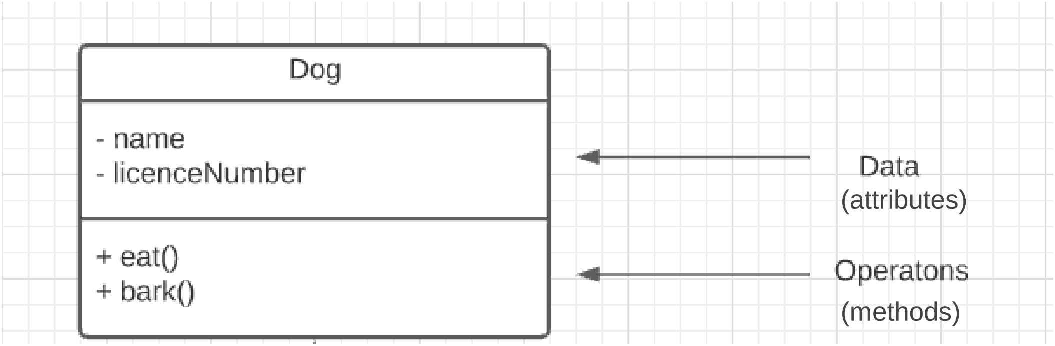
“One way of thinking of a class **is** as **a**n abstract data type plus inheritance and polymorphism.”

- McConnell (Code Complete - Ch. 6.1)

What is the difference between **Objects and Classes** in Object Oriented Programming?

Classes are type definitions

Objects are specific realizations / instances / items



Dog

- name
- licenceNumber

+ eat()
+ bark()

Data (attributes)

Operatons (methods)

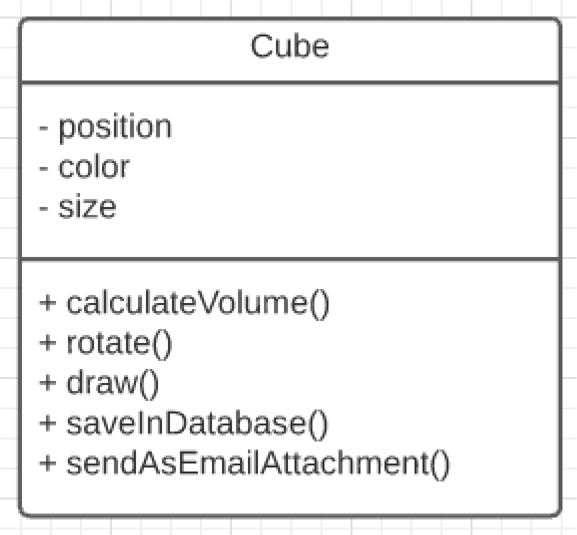
Fido

Sylvester

Rover

Bow & Wow

What is wrong with this Abstract Data Type (ADT)?

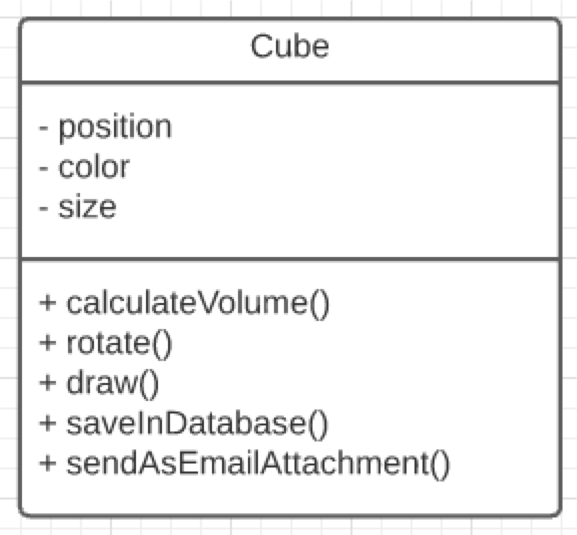


Cube

- position
- color
- size

+ calculateVolume()
+ rotate()
+ draw()
+ saveInDatabase()
+ sendAsEmailAttachment()

Design Principle: Low cohesion makes code hard to change and overly complex.



Cube

- position
- color
- size

+ calculateVolume()
+ rotate()
+ draw()
+ saveInDatabase()
+ sendAsEmailAttachment()

- draw()
  - UI code in a mathematical object
  - Specific graphics implementation.

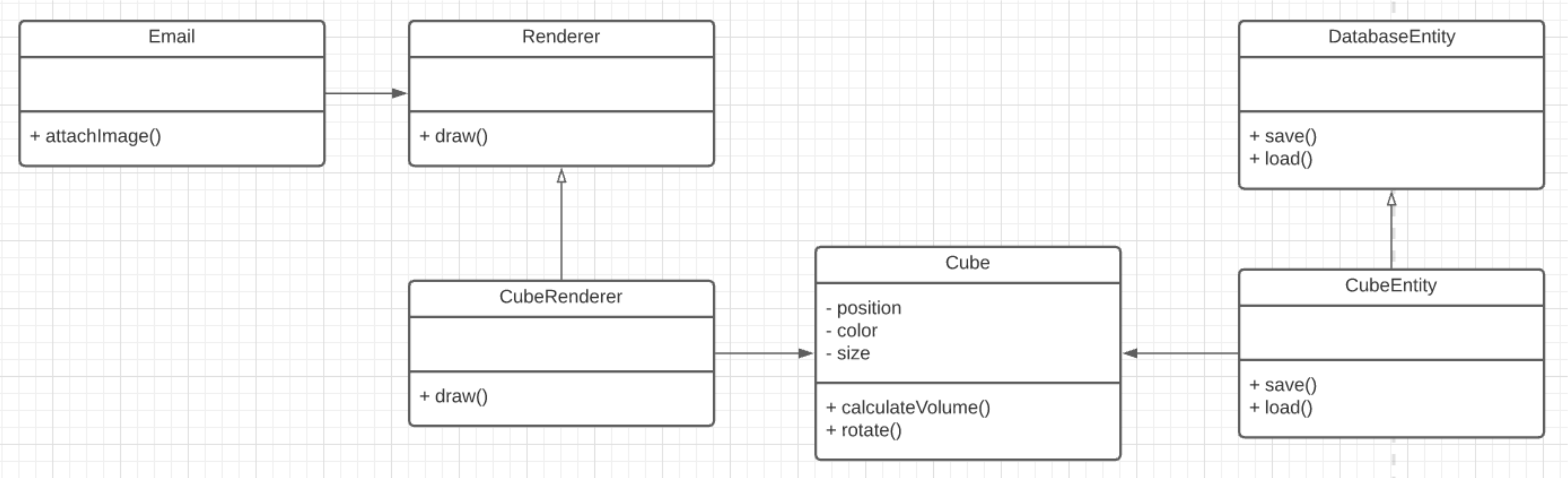
- saveInDatabase()
  - Complex database logic inside of cube

- sendAsEmailAttachment()
  - Need sender, recepiant, subject, and message
  - What if we wanted a different type of attachment?

“**Cohesion** refers to how closely all the routines in a class or all the code in a routine support a central purpose—how focused the class is.” - McConnel (Ch 5.3)

Design Principle: High cohesion makes code simpler, extensible, and reusable.

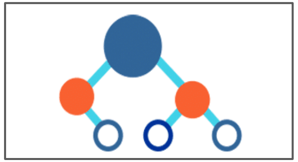
All operations must match the purpose of the class.



(i.e. utility methods are considered problematic).

Roadmap for Today

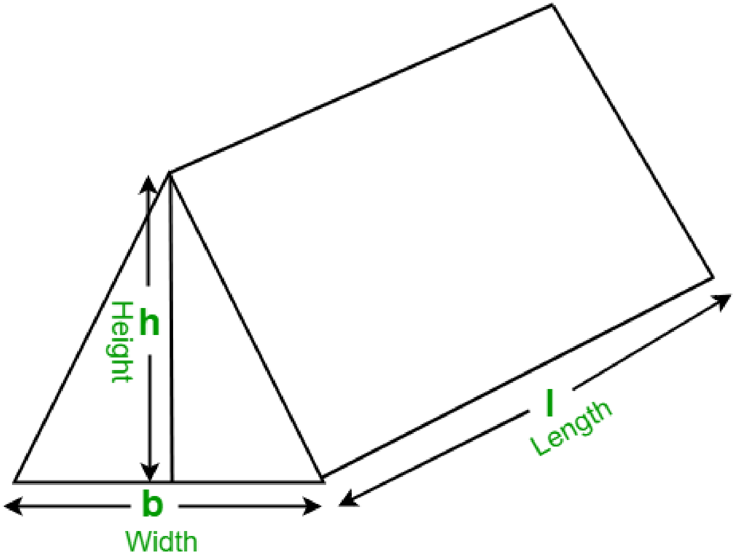
Motivation for Design: Namespaces

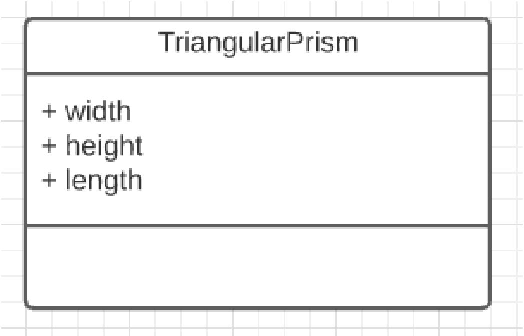


Design: Abstract Data Types

Development: C++ Classes

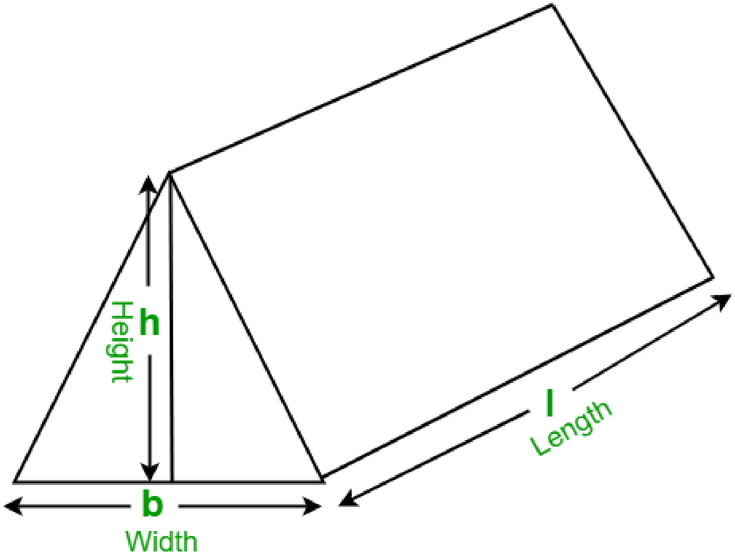
How can we represent a triangular prism ADT below in C/C++?

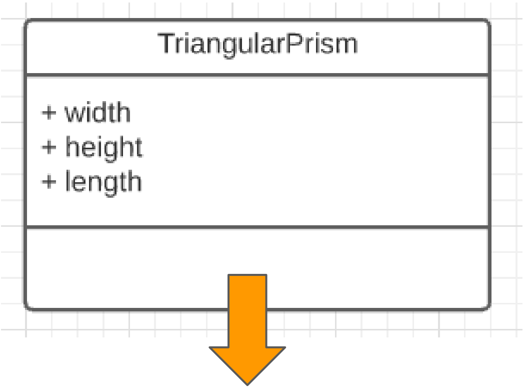



TriangularPrism

+ width
+ height
+ length

How can we represent a triangular prism ADT below in C/C++?



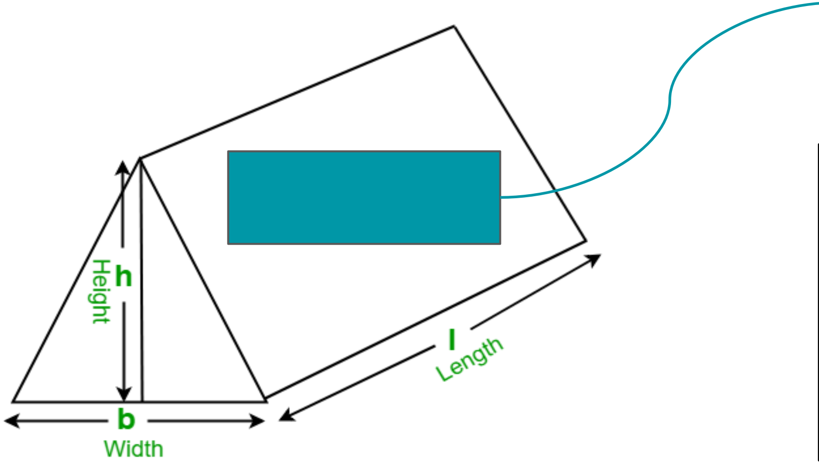


TriangularPrism

+ width
+ height
+ length

```
struct TriangularPrism {
  float height;
  float width;
  float length;
};
```

What if we wanted to add a color?

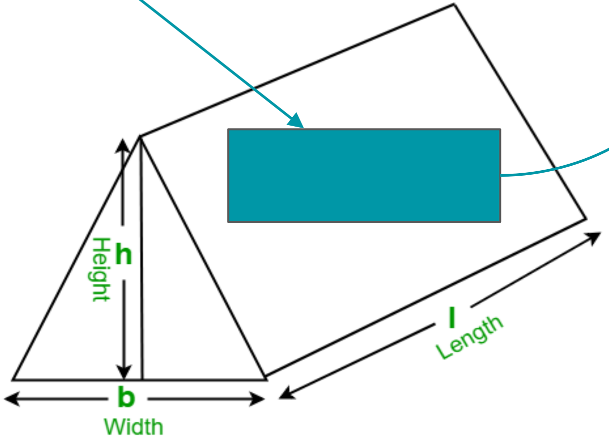


- Height - float
- Width - float
- Length - float
- Color - ???

```
struct TriangularPrism {
  float height;
  float width;
  float length;
  ??? color;
};
```

What if we wanted to add a color?

```
struct RGBColor {
  int red;
  int green;
  int blue;
};
```



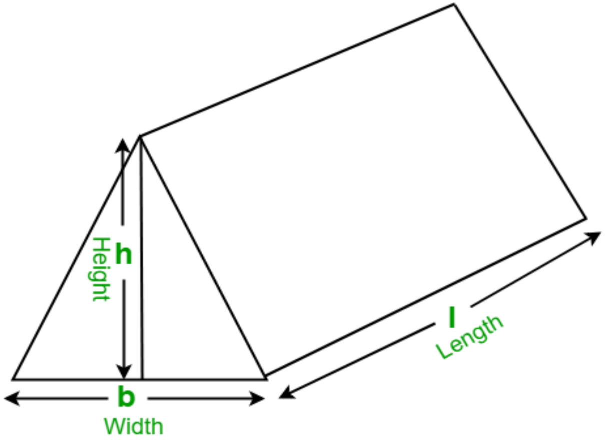
- Height - float
- Width - float
- Length - float
- Color - RGBColor

```
struct TriangularPrism {
  float height;
  float width;
  float length;
  RGBColor color;
};
```

Structs allow us to build self-contained complex data structures.

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

struct TrianglularPrism {
                float width;
                float height;
                float length;
};

int main()
{
                TrianglularPrism prism;
                prism.width = 1.0;
                prism.height = 2.0;
                prism.length = 4.0;

                cout << "Prism: " << prism.width << "
" << prism.height << " " << prism.length << endl;
                return 0;
}
```

- Declaration
- Usage



Poll: What are the differences between a **struct** and **class** in C++?

```
struct TriangularPrism {
      float height;
      float width;
      float length;
};
```

```
class TriangularPrism {
      float height;
      float width;
      float length;
};
```

The major difference between C and C++ is **Object Oriented Programming**.

	Encapsulation	**Inheritance**	**Polymorphism**
C	No	No	No
C++	Yes	Yes	Yes

We will talk about Encapsulation today.

Encapsulation allows us to control access to variables.

```
struct TrianglularPrism {
public:
                float width;
private:
                float height;
                float length;

public:
                float volume() {
                                return 0.5*width, height, length;
                }
};

int main() {
                TrianglularPrism prism1;
                prism1.width = 20;
                TrianglularPrism prism2;
                prism1.width = 10;
                prism1.height = 30;

                cout << (prism1.volume() - prism2.volume()) <<
endl;
                return 0;
}
```

What is wrong with the following code?

Encapsulation allows us to control access to variables.

```
struct TrianglularPrism {
public:
                float width;
private:
                float height;
                float length;

public:
                float volume() {
                                return 0.5*width, height, length;
                }
};

int main() {
                TrianglularPrism prism1;
                prism1.width = 20;
                TrianglularPrism prism2;
                prism1.width = 10;
                prism1.height = 30;

                cout << (prism1.volume() - prism2.volume()) << 
endl;
                return 0;
}
```

Fail

We can only access attributes and actions that have been declared public.

Why in the world would we want this?

What could go wrong here without encapsulation?

```
struct BankAccount {
               int accNum;
               float balance;
};

struct Bank {
               BankAccount accounts[50];
               float totalAmount;

               BankAccount& getAccount(int id) {
               return accounts[id];
       }


               void deposit(BankAccount& account, float amount)
{
                              accounts[account.accNum].balance
+= amount;
                              totalAmount += amount
               }

               float withdraw(BankAccount& account, amount) {
                              accounts[account.accNum].balance
-= amount;
                              totalAmount -= amount;
                              return amount
```

```
int main() {
               Bank bank;
               BankAccount& acc = bank.getAccount(10);
               bank.deposit(acc, 20);
               bank.withdraw(acc, 10);
               return 0;
}
```

What could go wrong here without encapsulation?

```
struct BankAccount {
                int accNum;
                float balance;
};

struct Bank {
                BankAccount accounts[50];
                float totalAmount;

                BankAccount& getAccount(int id) {
                return accounts[id];
        }


                void deposit(BankAccount& account, float amount)
{
                                accounts[account.accNum].balance
+= amount;
                                totalAmount += amount
                }

                float withdraw(BankAccount& account, amount) {
                                accounts[account.accNum].balance
-= amount;
                                totalAmount -= amount;
                                return amount
```

```
int main() {
                Bank bank;
                BankAccount& acc = bank.getAccount(10);
                bank.deposit(acc, 20);
                bank.withdraw(acc, 10);

                // We can give banks money
                bank.totalAmount = 100000.0;

                // We can change our account number
                acc.accNum = 10;

                // We can give ourselves money
                acc.balance += 100;

                // We can create a new account with an overflow
                Bank.accounts[500].balance = 50000000000.0;

                return 0;
}
```

What could go wrong here without encapsulation?

```
struct BankAccount {
               int accNum;
               float balance;
};

struct Bank {
               BankAccount accounts[50];
               float totalAmount;

               BankAccount& getAccount(int id) {
               return accounts[id];
        }

               void deposit(BankAccount& account, float amount)
{
                              accounts[account.accNum].balance
+= amount;
                              totalAmount += amount
               }

               float withdraw(BankAccount& account, amount) {
                              accounts[account.accNum].balance
-= amount;
                              totalAmount -= amount;
                              return amount
```

```
int main() {
               Bank bank;
               BankAccount& acc = bank.getAccount(10);
               bank.deposit(acc, 20);
               bank.withdraw(acc, 10);

               // We can give banks money
               bank.totalAmount = 100000.0;

               // We can change our account number
               acc.accNum = 10;

               // We can give ourselves money
               acc.balance += 100;

               // We can create a new account with an overflow
               Bank.accounts[500].balance = 50000000000.0;

               return 0;
}
```

Encapsulation is the future!

Enter center stage: **Classes**

```
struct BankAccount {
        int accNum;
        float balance;

public:
        void deposit(float amount) {
                balance += amount;
        }

        float withdraw(amount) {
                balance -= amount;
                return amount
        }
};
```

```
class BankAccount {
        int accNum;
        float balance;

public:
        void deposit(float amount) {
                balance += amount;
        }

        float withdraw(amount) {
                balance -= amount;
                return amount
        }
};
```

What is the difference between a **struct** and a **class**?

The only difference between a struct and a class is classes are **private** by default and structs are **public** by default.

```
struct BankAccount {
// public:  (by default)
                int accNum;
                float balance;

public:
                void deposit(float amount) {
                                balance += amount;
                }

                float withdraw(amount) {
                                balance -= amount;
                                return amount
                }
};
```

```
class BankAccount {
// private:  (by default)
                int accNum;
                float balance;

public:
                void deposit(float amount) {
                                balance += amount;
                }

                float withdraw(amount) {
                                balance -= amount;
                                return amount
                }
};
```

When should we use a struct versus a class?

Convention: (no hard set rule here)

Simple objects
(want easy access to variables)

Complex objects / everything else
(want to control variables and logic)

```
struct BankAccount {
            int accNum;
            float balance;
};
```

```
class BankAccount {
private:
            int accNum;
            float balance;

public:
            void deposit(float amount) {
                        balance += amount;
            }

            float withdraw(BankAccount& account, amount) {
                        balance -= amount;
                        return amount
            }
};
```

When should we use a struct versus a class?

Convention: (no hard set rule here)

Simple objects
(want easy access to variables)

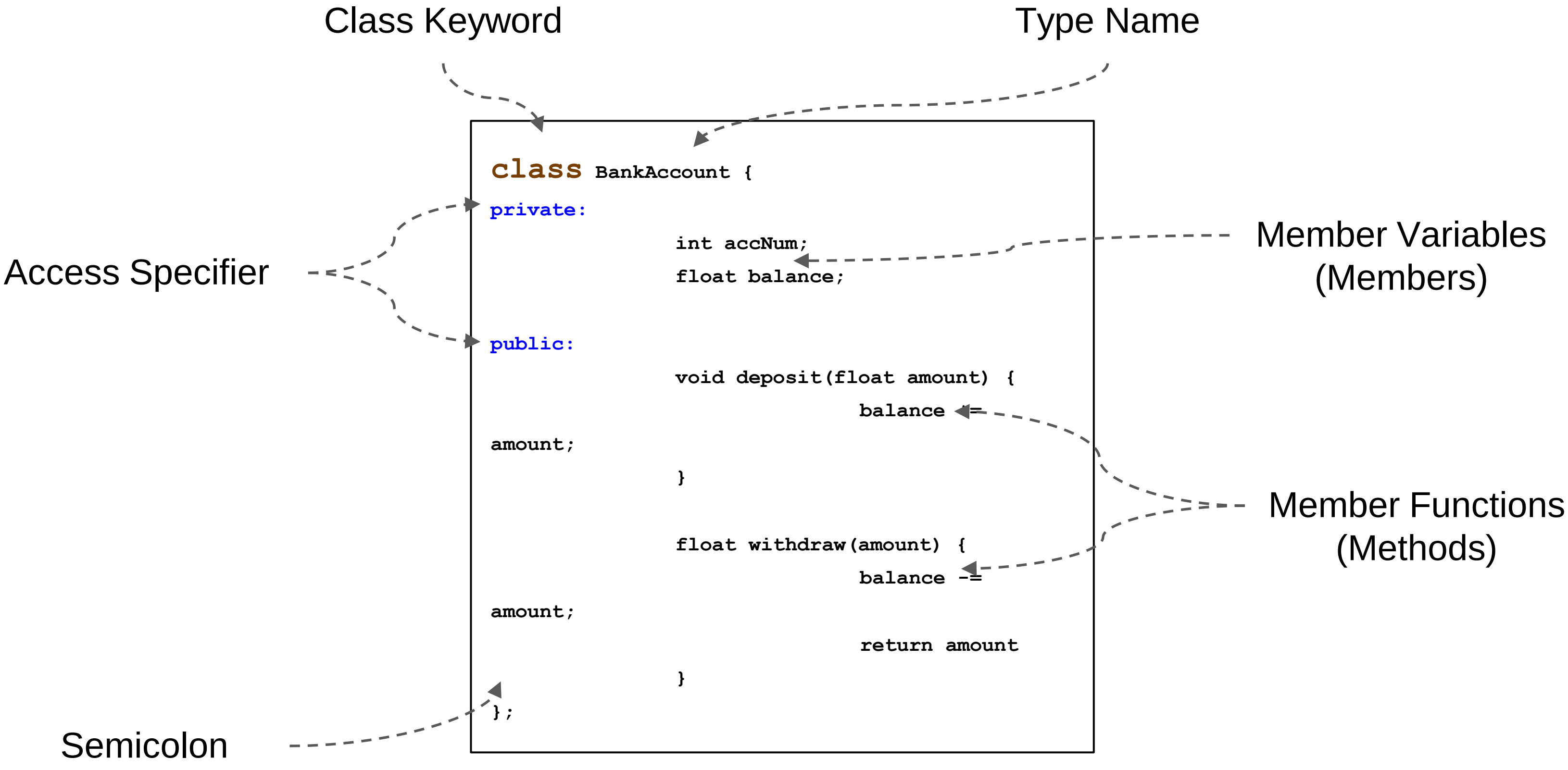
```
struct BankAccount {
                int accNum;
                float balance;
};
```

Classes protect the developer from potentially doing something silly.

Complex objects / everything else
(want to control variables and logic)

```
class BankAccount {
                int accNum;
                float balance;

public:
                void deposit(float amount) {
                                balance += amount;
                }

                float withdraw(BankAccount& account, amount) {
                                balance -= amount;
                                return amount
                }
};
```

Classes and objects are the way we will work going forward. Classes are defined as follows:



Class Keyword

Type Name

Access Specifier

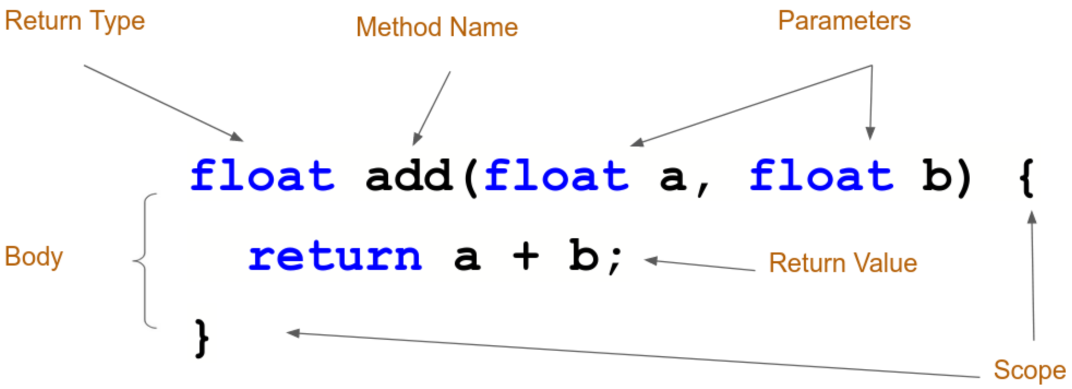
Member Variables
(Members)

Member Functions
(Methods)

Semicolon

```
class BankAccount {
private:
                int accNum;
                float balance;

public:
                void deposit(float amount) {
                                balance +=
amount;
                }

                float withdraw(amount) {
                                balance -=
amount;
                                return amount
                }
};
```

User Defined Functions (& Class Methods) allow programs to reuse calculations.



```
float add(float a, float b) {
  return a + b;
}
```

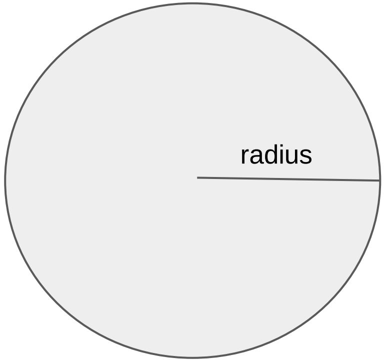
We can create classes and use their members / methods with the “.” operator.

```
class BankAccount {
private:
               int accNum;
               float balance;

public:
        void getBalance() {
                              return balance;
               }
        void setBalance(float amount) {
                              balance = amount;
               }

               void deposit(float amount) {
                              balance +=
amount;
               }

               float withdraw(amount) {
                              balance -=
amount;
                              return amount
               }
};
```

```
int main() {
               // Create new account
               BankAccount myAccount;

               // set initial balance
               myAccount.setBalance(100.0);

               // withdraw and deposit
               myAccount.withdraw(20.0);
               myAccount.withdraw(20.0);
               myAccount.deposit(10.0);
               myAccount.withdraw(20.0);

               // output final balance
               cout << myAccount.getBalance() << endl;

               // Cannot use the following
               // cout << myAccount.balance << endl;

               return 0;
}
```

In-Class Exercise: Create a simple circle class using C++.

Write a class called Circle that has methods to calculate the area of the circle, the diameter and the circumference. Also provide the necessary getters/setters that are needed.



Feel free to use knowledge from previous classes.
(e.g. constructors, getters and setters, etc...).

Advanced Concepts

- Alignment

  How are member variables in a class ordered?

Advanced Concepts

- Alignment

  How are member variables in a class ordered?

- Name mangling

  We heard about namespaces, but what about name mangling?

Advanced Concepts

- Alignment

  How are member variables in a class ordered?

- Name mangling

  We heard about namespaces, but what about name mangling?

- Memory Management

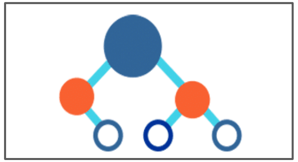
  What does the “new” keyword do?

C vs. C++ Revisted

- C++: Circle* circle = new Circle(10, 10, 16, “red”)
- C: Circle* circle = (Circle*)&malloc(sizeof(Circle))
  - Circle->x = 10
  - Circle->y = 10
  - Circle->radius = 16
  - Circle->color = “red”

- C++: delete circle
- C: free(&circle)

Summary



Design Principles

- Naming is important.
- Classes are Abstract Data Types with inheritance and polymorphism.
- Low cohesion makes code hard to change and overly complex.
- High cohesion makes code simpler, extensible, and reusable.

Development

- C++ Namespaces
- C++ Class Basics
- Encapsulation