

# Programming Abstractions

CS106B

Cynthia Lee

## Topics du Jour:

- **Make your own classes!**
  - › Needed for Boggle assignment!
  - › We are starting to see a little bit in MarbleBoard assignment as well

# Classes in C++

Making your own kinds of objects!

# Class examples

A calendar program might want to store information about dates, but C++ does not have a **Date** type.



A student registration system needs to store info about students, but C++ has no **Student** type.



A bank app might want to store information about users' accounts, but C++ has no **BankAccount** type.

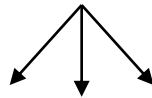


However, C++ does provide a feature for us to add new data types to the language: **classes**.

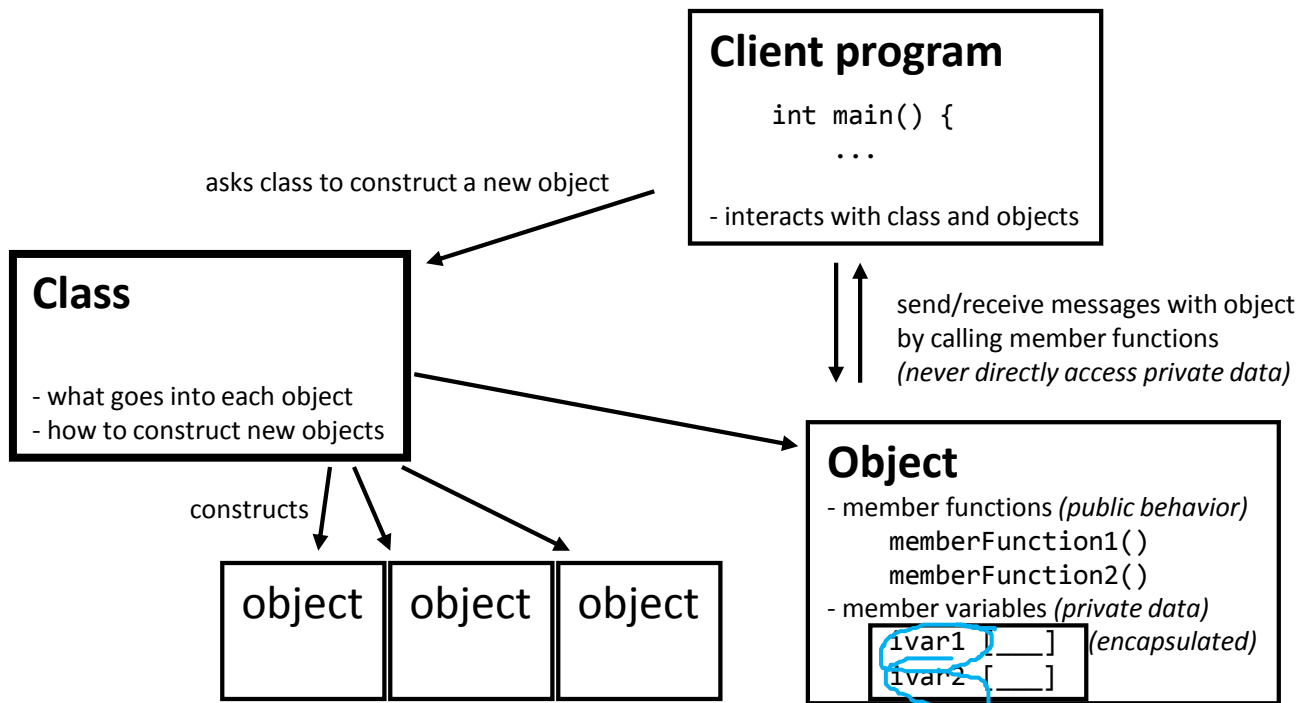
- Writing a class defines a new data type.

# Classes and objects (6.1)

- **class:** A program entity that represents a template for a new type of objects.
  - e.g. class Vector defines a new data type named Vector and allows you to declare objects of that type.
- **objects:** Entities that combines **state** and **behavior**.
  - **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.
  - **A new kind of abstraction:** Separation between concepts and details. Objects provide abstraction.



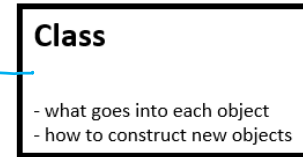
# Client, class, object



# Client, class, object: an example you already know!

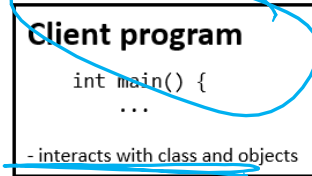
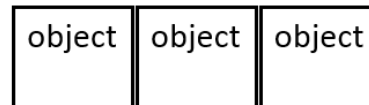
This was from the 3<sup>rd</sup> lecture:

```
int main() {  
    Grid<bool> board(8,8);  
    for (int i=0; i<board.numRows(); i++){  
        for (int j=0; j<board.numCols(); j++){  
            board[i][j] = false;  
        }  
    }  
    return 0;  
}
```

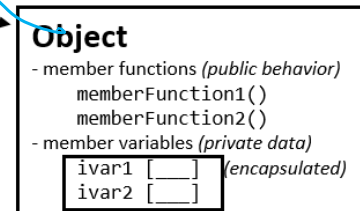


asks class to construct a new object

constructs



send/receive messages with object  
by calling member functions  
(never directly access private data)



# Elements of a class

**member variables**: State inside each object.

- Also called "instance variables" or "fields"
- Declared as private
- Each object created has a copy of each field.

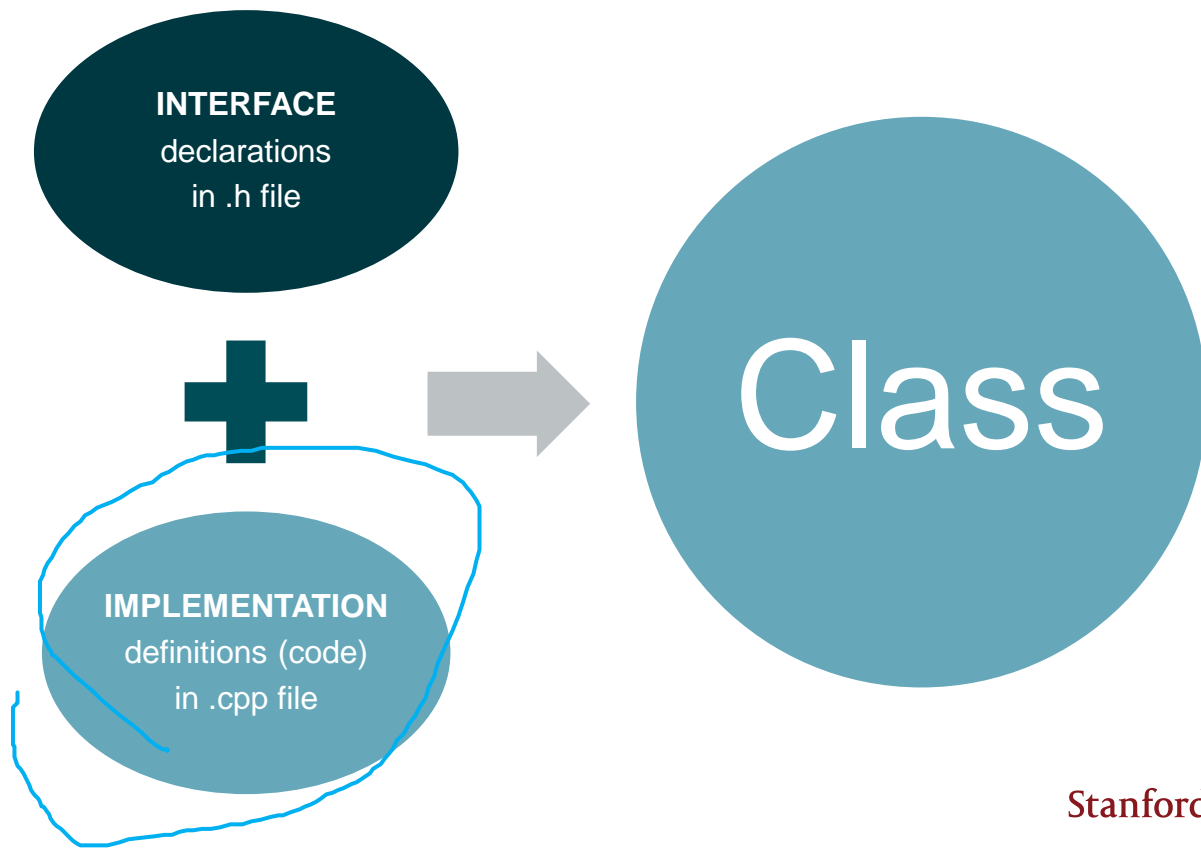
**member functions**: Behavior that executes inside each object.

- Also called "methods"
- Each object created has a copy of each method.
- The method can interact with the data inside that object.

**constructor**: Initializes new objects as they are created.

- Sets the initial state of each object as it is being created.
- Often accepts parameters for the initial state of the fields.

# Interface vs. code



# Interface vs. code

In C++, when writing classes you must understand separation of:

- **interface:** Declarations of functions, classes, members, etc.
- **implementation:** Definitions of how the above are implemented.

C++ implements this separation using two kinds of code files:

- **.h:** A "header" file containing only interface (declarations).
- **.cpp:** A "source" file containing definitions.
  - › When you define a new class Foo, you write Foo.h and Foo.cpp.

The content of .h files is "#included" inside .cpp files.

- Makes them aware of declarations of code implemented elsewhere.
- At compilation, all definitions are *linked* together into an executable.

## Interface: Structure of a .h file

```
// classname.h  
if {  
#ifndef _classname_h  
#define _classname_h  
    class declaration;  
#endif  
3  
are
```

This is protection in case multiple .cpp files include this .h, so that its contents won't get declared twice

## Interface: A class declaration

```
class ClassName {                                // in ClassName.h  
public:  
    ClassName(parameters);                    // constructor  
  
    returnType name(parameters); // member functions  
    returnType name(parameters); // (behavior inside  
    returnType name(parameters); // each object)
```

```
private:  
    type name; // member variables  
    type name; // (data inside each object)
```

};

IMPORTANT: *must* put a semicolon at end of class declaration (argh)

## Class example (v1)

```
// Initial version of BankAccount.h.  
// Uses public member variables and no functions.  
// Not good style, but we will improve it.
```

```
#ifndef _bankaccount_h  
#define _bankaccount_h
```

```
class BankAccount {  
public:  
    string name;  
    double balance;  
};
```

```
// each BankAccount object  
// has a name and balance
```

```
#endif
```

## Using our objects

```
// v1 with public fields (bad)
```

```
BankAccount ba1;  
ba1.name = "Cynthia";  
ba1.balance = 1.25;
```

```
BankAccount ba2;  
ba2.name = "Mehran";  
ba2.balance = 9999.00;
```

ba1

name	= "Cynthia"
balance	= 1.25

ba2

name	= "Mehran"
balance	= 9999.00

Think of an object as a way of grouping multiple variables.

- Each object contains a name and balance field inside it.
- We can get/set them individually.
- Code that uses your objects is called *client* code.

# Member func. bodies

In *ClassName.cpp*, we write bodies (definitions) for the member functions that were declared in the *.h* file:

```
// ClassName.cpp  
#include "ClassName.h"  
  
// member function  
returnType ClassName: methodName(parameters) {  
    statements;  
}
```

- Member functions/constructors can refer to the object's fields.

*Exercise:* Write a `withdraw` member function to deduct money from a bank account's balance.

# The implicit parameter

## **implicit parameter:**

The object on which a member function is called.

- During the call `cynthia.withdraw(...)`, the object named `cynthia` is the implicit parameter.
- During the call `mehran.withdraw(...)`, the object named `mehran` is the implicit parameter.
- The member function can refer to that object's member variables.
  - › We say that it executes in the *context* of a particular object.
  - › The function can refer to the data of the object it was called on.
  - › It behaves as if each object has its own *copy* of the member functions.

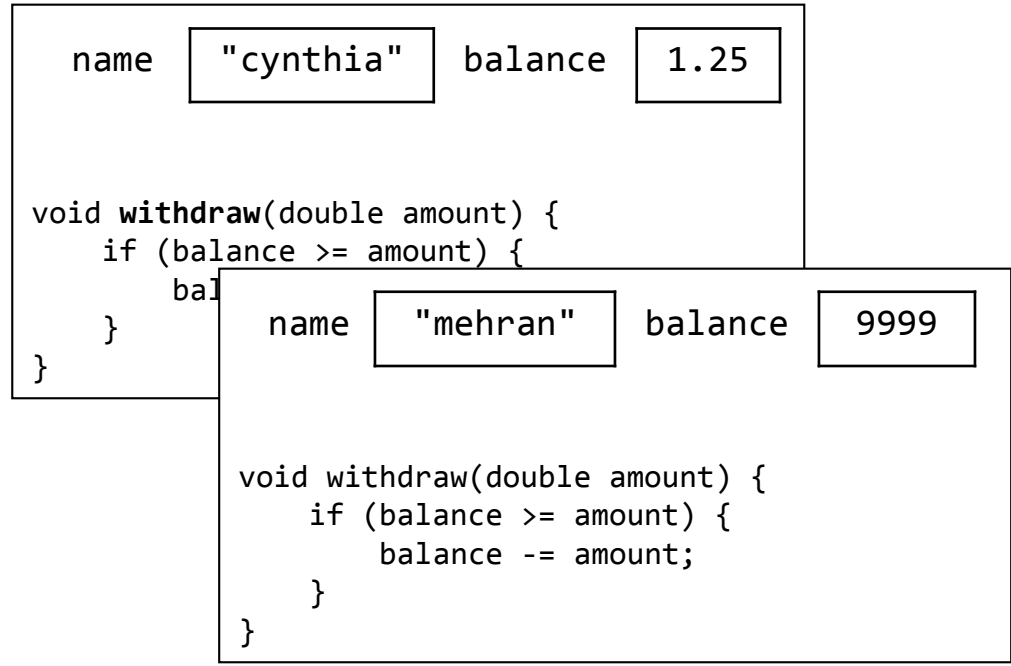
# Member func diagram

```
// BankAccount.cpp
```

```
void BankAccount::withdraw(double amount) {  
    if (balance >= amount) {  
        balance -= amount;  
    }  
}
```

```
// client program
```

```
BankAccount cynthia;  
BankAccount mehran;  
...  
cynthia.withdraw(5.00);  
  
mehran.withdraw(99.00);
```



# Initializing objects

It's bad to take 3 lines to create a BankAccount and initialize it:

```
BankAccount ba;  
ba.name = "Cynthia";  
ba.balance = 1.25;           // tedious
```

We'd rather specify the fields' initial values at the start:

```
BankAccount ba("Cynthia", 1.25); // better
```

- We are able to do this with most types of objects in C++ and Java.
- You can achieve this functionality using a **constructor**.

# Constructors

```
ClassName::ClassName(parameters) {  
    statements to initialize the object;  
}
```

**constructor:** Initializes state of new objects as they are created.

- runs when the client declares a new object
- no return type is specified;  
it implicitly "returns" the new object being created
- If a class has no constructor, C++ gives it a *default constructor* with no parameters that does nothing.

# Constructor diagram

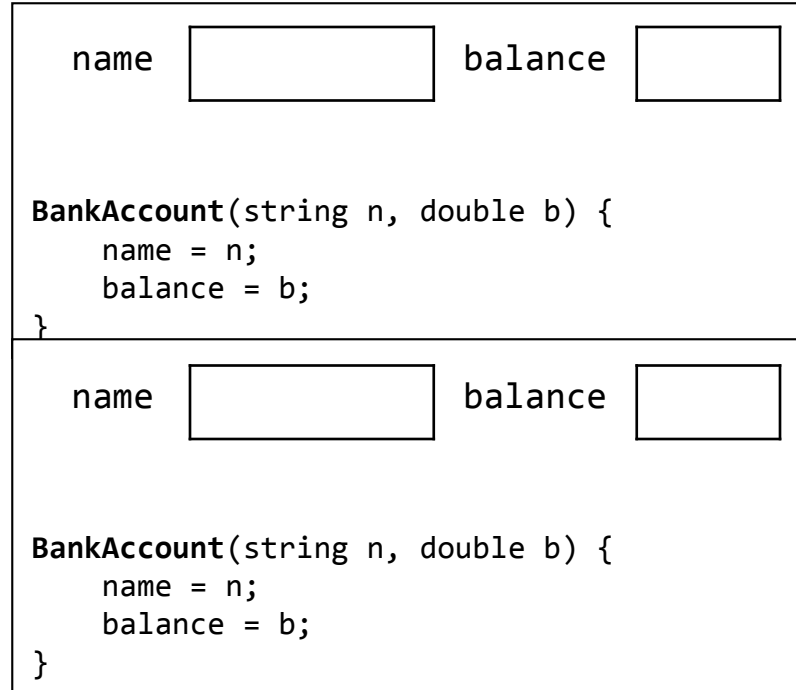
```
// BankAccount.cpp
```

```
BankAccount::BankAccount(string n, double b) {  
    name = n;  
    balance = b;  
}
```

```
// client program
```

```
BankAccount b1(  
    "Cynthia", 1.25);
```

```
BankAccount b2(  
    "Mehran", 9999);
```



# The keyword this

As in Java, C++ has a `this` keyword to refer to the current object.

- Syntax: `this->member`
- *Common usage:* In constructor, so parameter names can match the names of the object's member variables:

```
BankAccount::BankAccount(string name,  
                           double balance) {  
    this->name = name;  
    this->balance = balance;  
}
```

`this` uses `->` not `.` because it is a "pointer"; we'll discuss that later

# Preconditions

**precondition:** Something your code *assumes is true* at the start of its execution.

- Often documented as a comment on the function's header:

```
// Initializes a BankAccount with the given state.  
// Precondition: balance is non-negative  
BankAccount::BankAccount(string name, double balance) {  
    this->name = name;  
    this->balance = balance;  
}
```

- Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.
- What if we want to actually enforce the precondition?

# Throwing exceptions

`throw expression;`

Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.

```
// Initializes a BankAccount with the given state.  
// Precondition: balance is non-negative  
BankAccount::BankAccount(string name, double balance) {  
    if (balance < 0) {  
        throw "Illegal negative balance";  
    }  
    this->name = name;  
    this->balance = balance;  
}
```

Why would anyone ever *want* a program to crash?

# Private data

**private:**

*type name;*

**encapsulation:** Hiding implementation details of an object from its clients.

- Encapsulation provides *abstraction*.
  - › separates external view (behavior) from internal view (state)
- Encapsulation protects the integrity of an object's data.

A class's data members should be declared *private*.

- No code outside the class can access or change it.

## Accessor functions

We can provide methods to get and/or set a data field's value:

```
// "read-only" access to the balance ("accessor")
double BankAccount::getBalance() {
    return balance;
}

// Allows clients to change the field ("mutator")
void BankAccount::setName(string newName) {
    name = newName;
}
```

- Client code will look like this:

```
cout << ba.getName() << ":$" << ba.getBalance() << endl;
ba.setName("Cynthia");
```

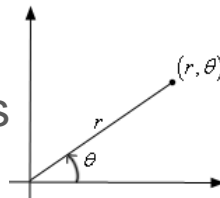
# Encapsulation benefits

Provides abstraction between an object and its clients.

Protects an object from unwanted access by clients.

Allows you to change the class implementation.

- Point could be rewritten to use polar coordinates (radius  $r$ , angle  $\theta$ ), but with the same methods.



Allows you to constrain objects' state (**invariants**).

- Example: Don't allow a BankAccount with a negative balance.

# Extra topics

Operators and const

# Operator overloading

Making your own kinds of objects work with operators!

## Operator overloading (6.2)

C++ allows you to *overload*, or redefine, the behavior of many common operators in the language:

- unary: + - ++ -- \* & ! ~ new delete
- binary: + - \* / % += -= \*= /= %= & | && || ^  
== != < > <= >= = [] -> ( ) ,

*Cynthia! 5.00;*

Overuse of operator overloading can lead to confusing code.

- *Rule of Thumb:* Don't abuse this feature. Don't define an overloaded operator unless its meaning and behavior are completely obvious.

## Op overload syntax

Declare your operator in a .h file, implement it in a .cpp file.

```
returnType operator op(parameters);           // .h
```

```
returnType operator op(parameters) {           // .cpp  
    statements;  
};
```

- where ***op*** is some operator like +, ==, <<, etc.
- the ***parameters*** are the operands next to the operator;  
for example, a + b becomes operator +(Foo a, Foo b)

Overloaded operators can also be declared inside a class (not shown here)

## Op overload example

```
// BankAccount.h
class BankAccount {
    ...
};

bool operator==(BankAccount& ba1, BankAccount& ba2);
bool operator!=(BankAccount& ba1, BankAccount& ba2);
```

---

```
// BankAccount.cpp
bool operator==(BankAccount& ba1, BankAccount& ba2) {
    return ba1.getName() == ba2.getName()
        && ba1.getBalance() == ba2.getBalance();
}

bool operator!=(BankAccount& ba1, BankAccount& ba2) {
    return !(ba1 == ba2);    // calls operator ==
}
```

## Make objects printable

To make it easy to print your object to cout, overload the << operator between an ostream and your type:

```
ostream& operator << (ostream& out, Type& name) {  
    statements;  
    return out;  
}
```

- The operator returns a reference to the stream so it can be chained.

› `cout << a << b << c` is really `((cout << a) << b) << c`

- › Technically cout is being returned by each << operation.

## << overload example

```
// BankAccount.h
class BankAccount {
    ...
};

ostream& operator <<(ostream& out, BankAccount& ba);
```

---

```
// BankAccount.cpp
ostream& operator <<(ostream& out, BankAccount& ba) {
    out << ba.getName() << ": $"
        << setprecision(2) << ba.getBalance();
    return out;
}
```

# Classes and const

# The keyword `const`

C++ `const` keyword indicates that a value cannot change.

```
const int x = 4;           // x will always be 4
```

a **const reference parameter** can't be modified by the function:

```
void foo(const BankAccount& ba) {    // won't change ba
```

- › Any attempts to modify `d` inside `foo`'s code won't compile.

a **const member function** can't change the object's state:

```
class BankAccount { ...  
    double getBalance() const;    // won't change account
```

- › On a `const` reference, you can only call `const` member functions.