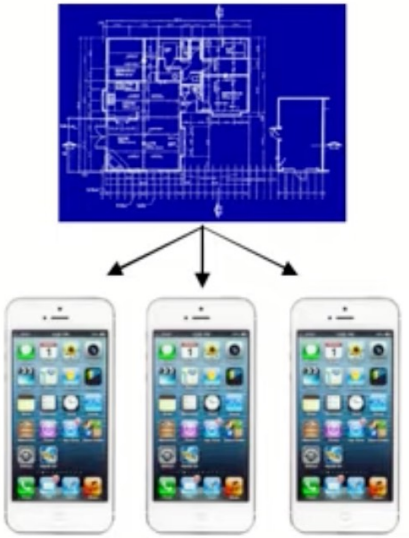
CS 106X, Lecture 14
Classes and Pointers

reading:

Programming Abstractions in C++, Chapter 6, 11

Classes and objects (6.1)

- **class**: A template for a new type of objects.
  - Allows us to add new types to the language.
  - Examples: Date, Student, BankAccount

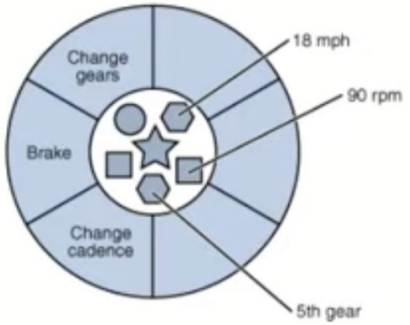


- **object**: Entity that combines **state** and **behavior**.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details.

object: mixture of state(vars) and behavior(methods)

Elements of a class

- **member variables**: State inside each object.
  - Also called "instance variables" or "fields"
  - Each object has a copy of each member.



- **member functions**: Behavior inside each object.
  - Also called "methods" object context: functions operate **inside** object
  - Each object has a copy of each method. methods: **encapsulation:保护变量不被篡改**
  - The method can interact with the data inside that object.

- **constructor**: Initializes new objects as they are created.
  - Sets the initial state of each new object.
  - Often accepts parameters for the initial state of the fields.

每个对象都有一个methods，vars的**copy**

The Classes Checklist

- ❑**Specify instance variables.** What information is inside this new variable type?
- ❑**Specify public methods**. What can this variable type do for others?
- ❑**Specify constructor(s).** How do you create a new variable of this type?

宏观着眼，现在h文件里面把check list的框架（vars，methods，constructor）搭好，再去.cpp里面实现功能

Interface vs. code

- C++ separates classes into two kinds of code files:
  - `.h`: A "header" file containing the interface (declarations).
  - `.cpp`: A "source" file containing definitions (method bodies).
    - class Foo => must write both `Foo.h` and `Foo.cpp`.

.h文件定义/声明（不具体实现），.cpp文件#include xx.h再对.h中的声明进行实现

- 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.

program will want to include your class

Class declaration (.h)

if not defined

```
#ifndef _classname_h
#define _classname_h

class ClassName {
public:
    ClassName(parameters);          // in ClassName.h
                                    // 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)
};

#endif
```

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

防止在不同文件导入.h文件的时候被重复调用，造成编译错误

分public和private section

别忘;

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

okay so here's what a class looks like

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;
}
```

methodName注意前面要有**ClassName::**

  - 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.

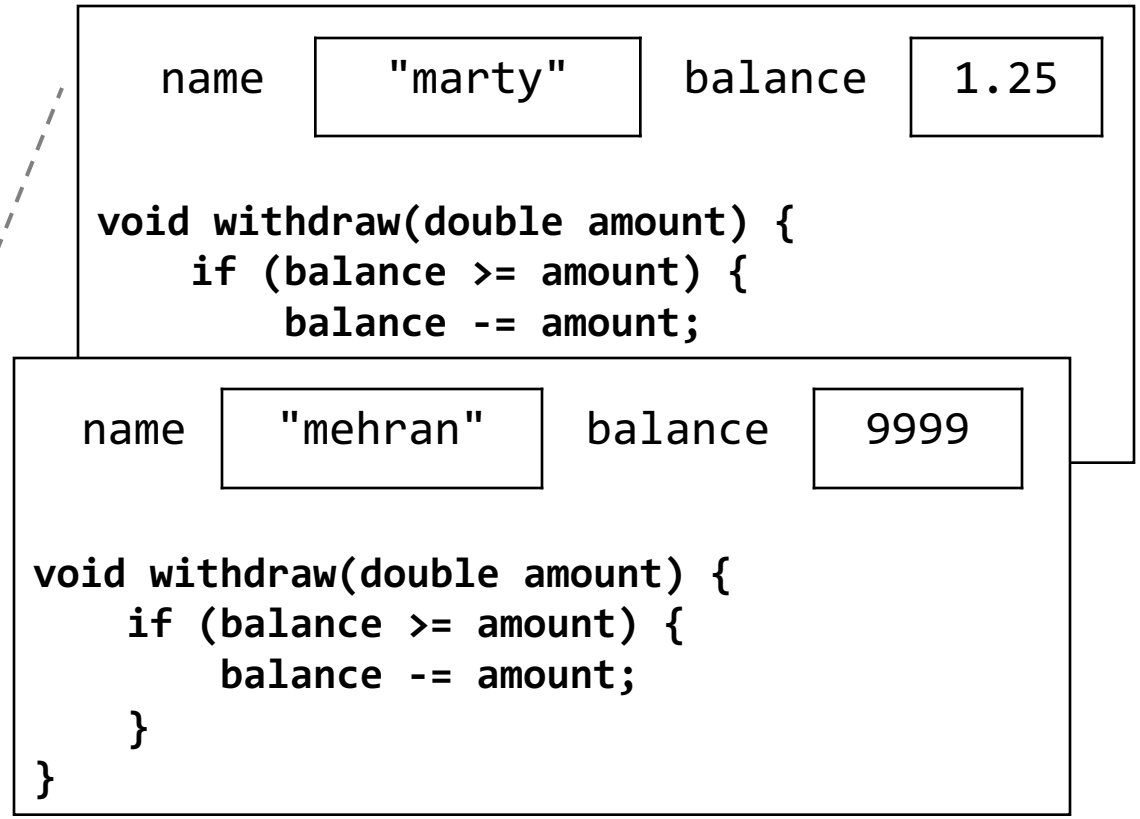
functions

Member func diagram

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

```
// client program
BankAccount marty;
BankAccount mehran;
...
marty.withdraw(5.00);

mehran.withdraw(99.00);
```



Preconditions

- **precondition**: Something your code *assumes is true* at the start of its execution.
  - Often documented as a comment on the function's header.
  - If violated, the class often **throws an exception**.

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

in the caller and so i want to tell them

The keyword this

- 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* to the current object

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.

Static data

- **static:** Shared by all objects of a class.
  - Opposite of regular member, which are duplicated in each object.
  - Useful when a class has some class-global shared state.

```
// BankAccount.h
class BankAccount {
    ...
private:
    static int ACCOUNT_ID = 1;
};
```

Static variables in a class: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables in a class **are shared by the objects**. There **can not be multiple copies** of same static variables for different objects. Also because of this reason static variables **can not be initialized using constructors.**

So, a static variable inside a class should be initialized explicitly by the user **using the class name and scope resolution operator outside the class**
e.g. int Classname::varName = 1
static元素在.cpp里面初始化

Class constants

- **class constant**: An unmodifiable static variable in the .h file.
  - Assign its value in the .cpp, outside of any method.
    - Don't write `static` when assigning the value in the .cpp.
  - For integral types, you can actually assign the variable in the .h file.

```
// BankAccount.h
class BankAccount {
    static const int BANK_ROUTING_NUM = 006029593;
    static const double INTEREST_RATE;
};

// BankAccount.cpp
// set the constant to store 3.25%
const double BankAccount::INTEREST_RATE = 0.0325;
```

Operator overloading (6.2)

- **operator overloading**: Redefining the behavior of a common operator in the C++ language.

  unary: `+ - ++ -- * & ! ~ new delete`

  binary: `+ - * / % += -= *= /= %= & | && || ^ == != < > <= >= = [] -> () ,`

- Syntax:

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

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

  – the ***parameters*** are the operands next to the operator;
  for example, `a + b` becomes `operator +(Foo a, Foo b)`

Op overload example

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

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

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

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;
}
```

  - `ostream` is a class that represents `cout`, file output streams, etc.

  - 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() << ": $"
        << fixed << setprecision(2)
        << ba.getBalance();
    return out;
}
```

Alternate syntax

- You can also declare operators inside the class.
  - The `this` object is implicitly the first parameter.
  - The internal operator can access the objects' private data.

```
// BankAccount.h
class BankAccount {
    bool operator ==(const BankAccount& ba2);
};
```

```
// BankAccount.cpp
bool BankAccount::operator ==(const BankAccount& ba2) {
    return name == ba2.name && balance == ba2.balance;
}
```

The friend keyword

- `friend` selectively allows puncturing object encapsulation:

```
// BankAccount.h
class BankAccount {
    ...
    friend ostream& operator <<(ostream& out,
                                const BankAccount& ba);
};
ostream& operator <<(ostream& out, const BankAccount& ba);
```

```
// BankAccount.cpp
ostream& operator <<(ostream& out, const BankAccount& ba) {
    out << ba.name << ": $"
        << fixed << setprecision(2)
        << ba.balance;
    return out;
}
```

okay so anyway i want to move

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

- **destructor**: Called when the object is deleted by the program.

  (when the object goes out of { } scope; opposite of a constructor)

  – Useful if your object needs to do anything important as it dies:
    - saving any temporary resources inside the object
    - freeing any dynamically allocated memory used by the object's members
    - ...

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

- **destructor**: Called when the object is deleted by the program.
  (when the object goes out of {} scope; opposite of a constructor)

  - Useful if your object needs to do anything important as it dies:
    - saving any temporary resources inside the object
    - freeing any dynamically allocated memory used by the object's members
    - ...

  - Does our `BankAccount` need a destructor? If so, what should it do?

about let's see

Template function (14.1-2)

```
template<typename T>
returntype name(parameters) {
    statements;
}
```

- **Template**: A function or class that accepts a *type parameter(s)*.
  - Allows you to avoid redundancy by writing a function that can accept many types of data.
  - Templates can appear on a single function, or on an entire class

Template func example

```
template<typename T>
T max(T a, T b) {
    if (a < b) { return b; }
    else       { return a; }
}
```

– The template is *instantiated* each time you use it with a new type.
  - The compiler actually generates a new version of the code each time.
  - The type you use must have an operator < to work in the above code.

```
int i    = max(17, 4);           // T = int
double d = max(3.1, 4.6);        // T = double
string s = max(string("hi"),     // T = string
               string("bye"));
```

Template class (14.1-2)

- **Template class**: A class that accepts a type parameter(s).
  - In the header and cpp files, mark each class/function as templated.
  - Replace occurrences of the previous type `int` with T in the code.

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName::name(parameters) {
    ...
}
```

Template .h and .cpp

- Because of an odd quirk with C++ templates, the separation between .h header and .cpp implementation must be reduced.
  - Either write all the bodies in the .h file (suggested),
  - Or `#include` the .cpp at the end of .h file to join them together.

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

#include "ClassName.cpp"
#endif   // _classname_h
```