

6.088 Intro to C/C++

Day 4: Object-oriented programming in C++

Eunsuk Kang and Jean Yang

Today's topics

Why objects?

Object-oriented programming (OOP) in C++

- ▶ classes
- ▶ fields & methods
- ▶ objects
- ▶ representation invariant

Why objects?

At the end of the day...

computers just manipulate 0's and 1's



Figure by MIT OpenCourseWare.

But binary is hard (for humans) to work with

Towards a higher level of abstraction

?

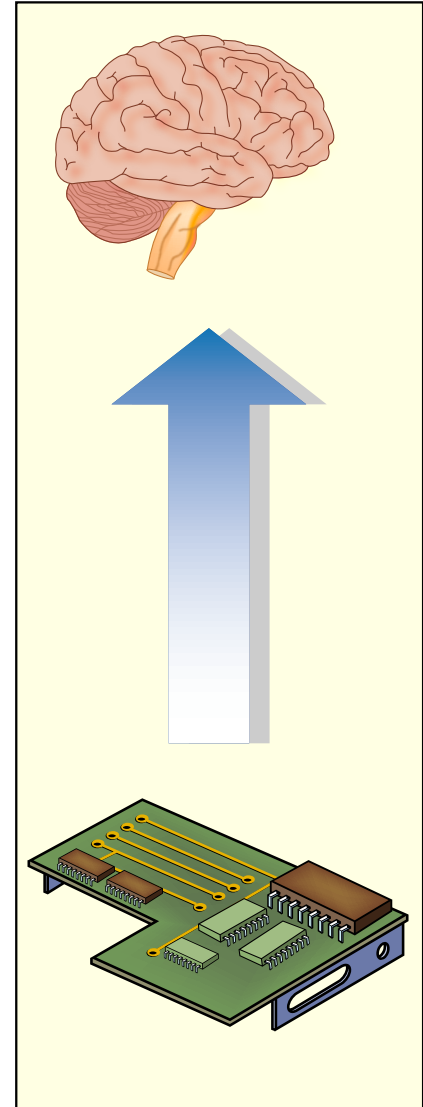
declarative languages (Haskell, ML, Prolog...)

OO languages (C++, Java, Python...)

procedural languages (C, Fortran, COBOL...)

assembly languages

binary code



There are always trade-offs

High-level languages

- ▶ simpler to write & understand
- ▶ more library support, data structures

Low-level languages

- ▶ closer to hardware
- ▶ more efficient (but also dangerous!)

What about C++?

What are objects?

Objects model elements of the **problem context**

Each object has:

- ▶ characteristics
- ▶ responsibilities (or required behaviors)

Example

Problem Design and build a computer hockey game

Object Hockey player

Characteristics Position, height, weight, salary, number of goals

Responsibilities Pass the puck, shoot, skate forward, skate backward, punch another player, etc.

Another example

Problem Computation modeling in biology

Write a program that simulates the growth of virus population in humans over time. Each virus cell reproduces itself at some time interval. Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus cells from their body. However, some of the cells are resistant to drugs and may survive.

Write a program that simulates the growth of virus population in humans over time. Each virus cell reproduces itself at some time interval. Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus cells from their body. However, some of the cells are resistant to drugs and may survive.

What are **objects**?

Characteristics?

Responsibilities?

Write a program that simulates the growth of **virus population in humans** over time. Each virus cell reproduces itself at some time interval. Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus cells from their body. However, some of the cells are **resistant to drugs** and may survive.

What are **objects**?

Characteristics?

Responsibilities?

Patient

characteristics

- ▶ virus population
- ▶ immunity to virus (%)

responsibilities

- ▶ take drugs

Virus

characteristics

- ▶ reproduction rate (%)
- ▶ resistance (%)

responsibilities

- ▶ reproduce
- ▶ survive

Questions

Why didn't we model an object named Doctor?
Surely, most hospitals have doctors, right?

Questions

Why didn't we model an object named Doctor?
Surely, most hospitals have doctors, right?

Doesn't every patient have an age? Gender? Illness?
Symptoms? Why didn't we model them as
characteristics?

Basic OOP in C++

Classes

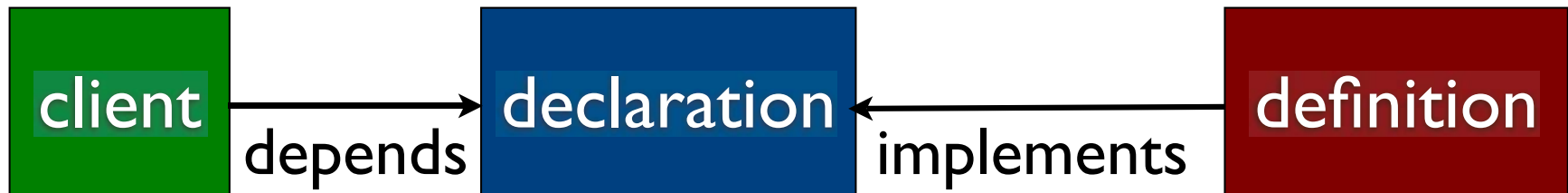
A class is like a cookie cutter; it defines the shape of objects

Objects are like cookies; they are **instances** of the class



Photograph courtesy of [Guillaume Brialon](#) on Flickr.

Classes: Declaration vs. definition



Declaration (.h files)

- ▶ list of functions & fields
- ▶ including functions that the class *promises* to its client
- ▶ it's like a “contract”

Definition (.cc files)

- ▶ implementation of functions

Class declaration

Class declaration

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

class name

field

```
class Virus {  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
};
```

constructors

method

don't forget the semi-colon!

Fields (characteristics)

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

Methods (responsibilities)

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

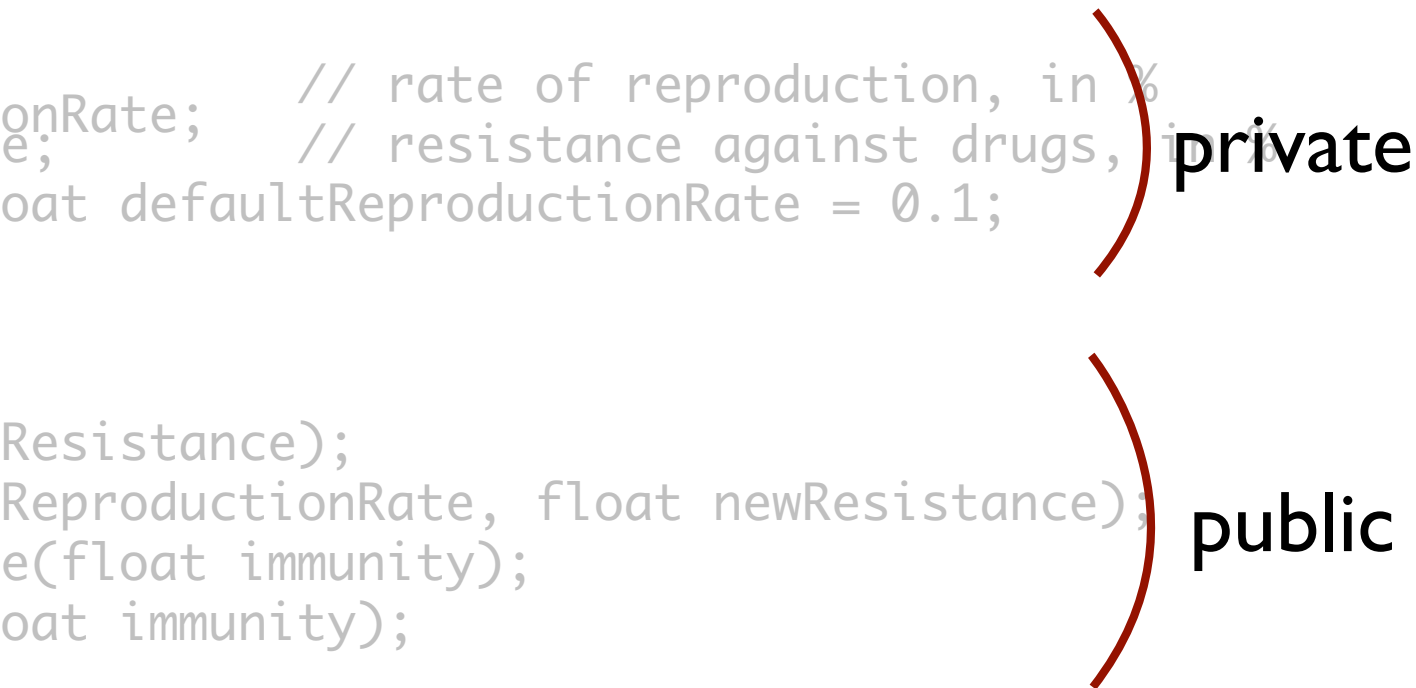
Constructors

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

Note the special syntax for constructor (no return type!)

Access control: public vs. private

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance; // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```



private

public

private: can only be accessed inside the class

public: accessible by anyone

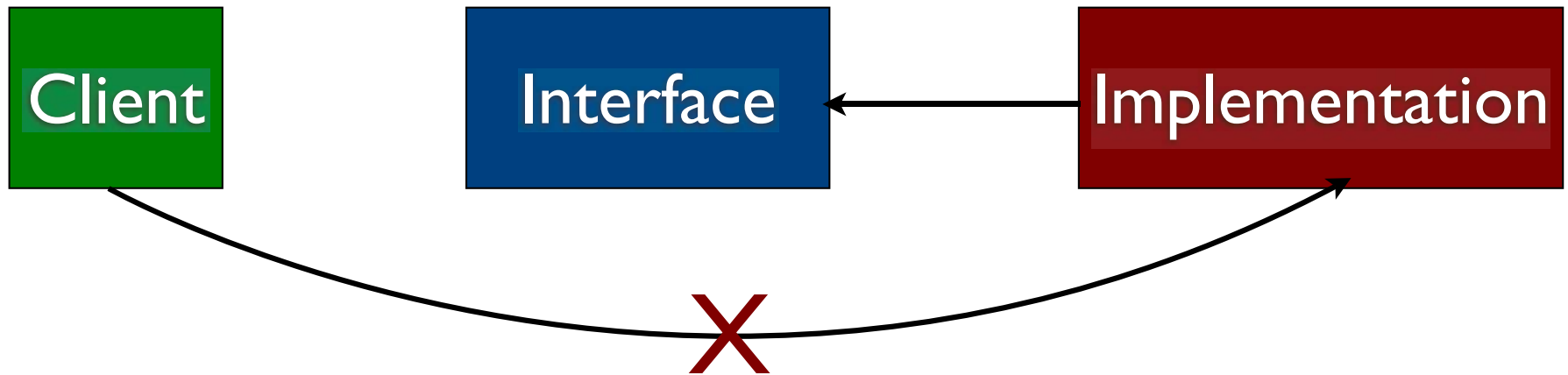
How do we decide private vs. public?



interface: parts of class that change infrequently
(e.g. virus must be able to reproduce)

implementation: parts that may change frequently
(e.g. representation of resistance inside virus)

Protect your private parts!



Why is this bad?

Access control: public vs. private

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance; // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

private

public

In general,

- ▶ keep member fields as private
- ▶ minimize the amount of public parts

Access control: constant fields

```
class Virus {  
  
    float reproductionRate; // rate of reproduction, in %  
    float resistance;       // resistance against drugs, in %  
    static const float defaultReproductionRate = 0.1;  
  
public:  
  
    Virus(float newResistance);  
    Virus(float newReproductionRate, float newResistance);  
    Virus* reproduce(float immunity);  
    bool survive(float immunity);  
  
};
```

Why make it a constant? Why not just declare it as a normal field?

Class definition

Class definition

```
#<stdlib.h>
#include "Virus.h"

Virus::Virus(float newResistance) {
    reproductionRate = defaultReproductionRate;
    resistance = newResistance;
}

Virus::Virus(float newReproductionRate, float newResistance) {
    reproductionRate = newReproductionRate;
    resistance = newResistance;
}

// If this virus cell reproduces,
// returns a new offspring with identical genetic info.
// Otherwise, returns NULL.
Virus* Virus::reproduce(float immunity) {

    float probab = (float) rand() / RAND_MAX; // generate number between 0 and 1

    // If the patient's immunity is too strong, it cannot reproduce
    if (immunity > probab)
        return NULL;

    // Does the virus reproduce this time?
    if (probab > reproductionRate)
        return NULL; // No!

    return new Virus(reproductionRate, resistance);
}

// Returns true if this virus cell survives, given the patient's immunity
bool Virus::survive(float immunity) {

    // If the patient's immunity is too strong, then this cell cannot survive
    if (immunity > resistance)
        return false;

    return true;
}

const float Virus::defaultReproductionRate;
```

Header inclusion

```
#include <stdlib.h>
#include "Virus.h"
```

```
Virus::Virus(float newResistance) {
    reproductionRate = defaultReproductionRate;
    resistance = newResistance;
}
```

```
Virus::Virus(float newReproductionRate, float newResistance) {
    reproductionRate = newReproductionRate;
    resistance = newResistance;
}
```


Constructor definition

```
#include <stdlib.h>
#include "Virus.h"

Virus::Virus(float newResistance) {
    reproductionRate = defaultReproductionRate;
    resistance = newResistance;
}

Virus::Virus(float newReproductionRate, float newResistance) {
    reproductionRate = newReproductionRate;
    resistance = newResistance;
}
```

Remember to initialize all fields inside constructors!

Constructor definition

Can also do:

```
Virus::Virus(float newReproductionRate, float newResistance) {  
    reproductionRate = newReproductionRate;  
    resistance = newResistance;  
}
```

```
Virus::Virus(float newReproductionRate, float newResistance) :  
    reproductionRate(newReproductionRate), resistance(newResistance)  
{  
}
```

Method definition

```
// Returns true if this virus cell survives,  
// given the patient's immunity  
bool Virus::survive(float immunity) {  
  
    // If the patient's immunity is too strong,  
    // then this cell cannot survive  
    if (immunity > resistance)  
        return false;  
  
    return true;  
}
```

Working with objects

Patient class declaration

```
#include "Virus.h"

#define MAX_VIRUS_POP 1000

class Patient {

    Virus* virusPop[MAX_VIRUS_POP];
    int numVirusCells;
    float immunity;           // degree of immunity, in %

public:

    Patient(float initImmunity, int initNumViruses);
    ~Patient();
    void takeDrug();
    bool simulateStep();

};
```

Patient class declaration

```
#include "Virus.h"
#define MAX_VIRUS_POP 1000
class Patient {
    Virus* virusPop[MAX_VIRUS_POP];
    int numVirusCells;
    float immunity; // degree of immunity, in %
public:
    Patient(float initImmunity, int initNumViruses);
    ~Patient();
    void takeDrug();
    bool simulateStep();
};
```

Array of pointers to objects

Constructor

Destructor

Static object allocation

```
class Patient {
    ...
    public:
        Patient(float initImmunity, int initNumViruses);
    ...
};

int main() {

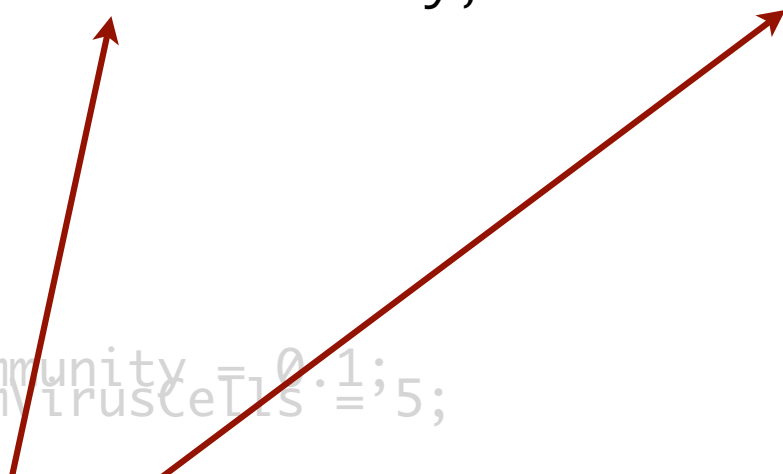
    float initImmunity = 0.1;
    int initNumVirusCells = 5;

    Patient p(0.1, 5);

    p.takeDrug();
}
```

Calling the constructor

```
class Patient {  
    ...  
    public:  
        Patient(float initImmunity, int initNumViruses);  
    ...  
};  
  
int main() {  
  
    float initImmunity = 0.1;  
    int initNumViruses = 5;  
  
    Patient p(0.1, 5);  
    p.takeDrug();  
}
```



The diagram illustrates the call to the constructor. Two red arrows originate from the arguments '0.1' and '5' in the constructor call 'Patient p(0.1, 5);' within the main function. One arrow points to the parameter 'float initImmunity' in the constructor signature 'Patient(float initImmunity, int initNumViruses);' inside the class definition. The other arrow points to the parameter 'int initNumViruses' in the same signature.

Deleting statically allocated objects

```
class Patient {
    ...
public:
    Patient(float initImmunity, int initNumViruses);
    ...
};

int main() {

    float initImmunity = 0.1;
    int initNumVirusCells = 5;

    Patient p(0.1, 5);

    p.takeDrug();
}
```

Automatically destroyed at the end of scope

Objects on heap

To allocate an object on heap:

▶ use “new” keyword (analogous to “malloc”)

To deallocate:

▶ use “delete” keyword (analogous to “free”)

```
Patient* p = new Patient(0.1, 5);  
...  
delete p;
```

Dynamic object creation: Example

```
Patient::Patient(float initImmunity, int initNumVirusCells) {  
    float resistance;  
  
    immunity = initImmunity;  
  
    for (int i = 0; i < initNumVirusCells; i++) {  
        //randomly generate resistance, between 0.0 and 1.0  
        resistance = (float) rand()/RAND_MAX;  
  
        virusPop[i] = new Virus(resistance);  
    }  
  
    numVirusCells = initNumVirusCells;  
}
```

Using dynamically allocated objects

```
bool Patient::simulateStep() {
    Virus* virus;
    bool survived = false;
    ...

    for (int i = 0; i < numVirusCells; i++){
        virus = virusPop[i];

        survived = virus->survive(immunity);

        if (survived) {
            ...
        } else {
            ...
        }
        ...
    }
}
```

What happens during destruction?

The destructor is automatically called

```
Patient::~~Patient(){  
    for (int i = 0; i < numVirusCells; i++){  
        delete virusPop[i];  
    }  
}
```

```
Patient* p = new Patient(0.1, 5);  
...  
delete p;
```

But why didn't we have a destructor for Virus?

Representation invariant

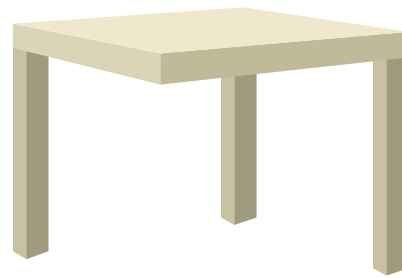
Representation invariant

Statements about characteristics of objects

- ▶ defines what it means for an object to be valid
- ▶ e.g. “Every IKEA coffee table must have four legs”



Valid



Invalid

Figures by MIT OpenCourseWare.

Example

```
class Patient {  
  
    Virus* virusPop[MAX_VIRUS_POP];  
    int numVirusCells;  
    float immunity;           // degree of immunity, in %  
  
public:  
  
    Patient(float initImmunity, int initNumViruses);  
    ~Patient();  
    void takeDrug();  
    bool simulateStep();  
  
};
```

What are the representation invariants for Patient?

Rep. invariant violation

```
void Patient::takeDrug(){  
    immunity = immunity + 0.1;  
}
```

What's wrong with this method?

Preserving rep. invariant

```
bool Patient::checkRep() {  
    return (immunity >= 0.0) && (immunity < 1.0) &&  
        (numVirusCells >= 0) &&  
        (numVirusCells < MAX_VIRUS_POP);  
}
```

checkRep

- ▶ returns true if and only if the rep. invariants hold true
- ▶ call checkRep at the **beginning** and **end** of every public method
- ▶ call checkRep at the **end** of constructors

Preserving rep. invariant

```
#include <cassert>

class Patient {
    ...
    bool checkRep();
public:
    ...
};

void Patient::takeDrug() {
    assert(checkRep());
    ...
    assert(checkRep());
}

Patient::Patient(float initImmunity, int initNumViruses) {
    ...
    assert(checkRep());
}
```

Preserving rep. invariant

Will calling `checkRep()` slow down my program?

Yes, but you can take them out once you are confident about your code.

Until next time...

Homework #4 (due 11:59 PM Monday)

- ▶ implementing BSTs using classes

Next lecture

- ▶ inheritance & polymorphism

- ▶ templates

References

Thinking in C++ (B. Eckel) **Free online edition!**

Essential C++ (S. Lippman)

Effective C++ (S. Meyers)

C++ Programming Language (B. Stroustrup)

Design Patterns (Gamma, Helm, Johnson, Vlissides)

Object-Oriented Analysis and Design with Applications (G. Booch, et. al)

MIT OpenCourseWare
<http://ocw.mit.edu>

6.088 Introduction to C Memory Management and C++ Object-Oriented Programming
January IAP 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.