ADTs and Inheritance of Classes
Intro to Recursion

CS 16: Solving Problems with Computers I
Lecture #16

Ziad Matni
Dept. of Computer Science, UCSB

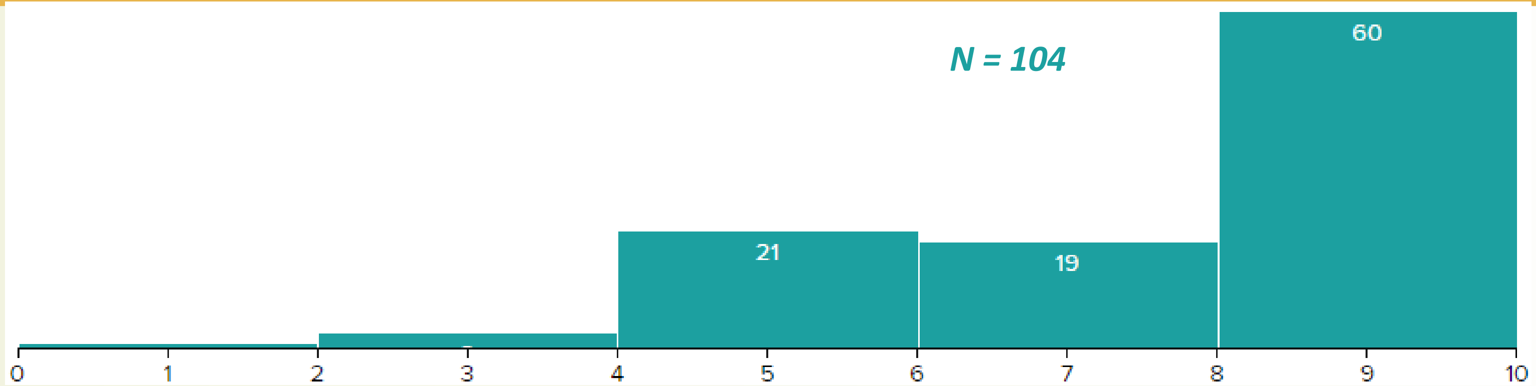
Administrative

- New labs (#8, #9) and new homework (#8, #9) are out!
  - Let's talk about them and their due dates…

- Quiz 8 is on Friday
  - Last quiz?!

- Final Exam!
  - Will be on Wed. Dec 16th
  - Starts at 9 AM
  - On Gradescope
  - Comprehensive (everything we've done all quarter)
  - I will give you a study guide by end of this week

Quiz 7



- Mean: **7.61/10**
- Median: **8/10**

- Was it tougher b/c it went faster than other tests?
  - It is on-par with actual in-person tests that we normally have

Questions 1-3

- Dec → Hex and Binary
  - Easiest (not the only) route is:
    - Dec to Hex using the /16 method
    - Convert each Hex into 4 bits

Example:	Then, 0x757 =
1879 / 16 = 117 R **7** 117 / 16 = 7 R **5** 7 / 16 = 0 R **7** So, it's **0x757**	**0111 0101 0111**

- Bin → Hex and Dec
  - Easiest (not the only) route is:
    - Bin to Hex by collect-4-bits
    - Convert Hex to Dec by Positional Notation

Example:	Then, 0x1CE81 =
1 1100 1110 1000 0001 = **0x1CE81**	$1 \times 16^4 + 12 \times 16^3 + 14 \times 16^2$ $+ 8 \times 16^1 + 1 \times 16^0$ = **118,401**

- Hex → Bin and Dec
  - Easiest (not the only) route is:
    - Hex to Bin
    - Convert Hex to Dec by Positional Notation

Example:	Then, 0xC033 =
0xC033 = **1100 0000 0011 0011**	$12 \times 16^3 + 0 \times 16^2$ $+ 3 \times 16^1 + 3 \times 16^0$ = **49,203**

Lecture Outline

- ADTs
- Intro to Inheritance
- Intro to Recursion

Demo file is called: **condTO.cpp**

But first… a Cool C++ shortcut…

The Conditional Ternary Operator (?)

- Evaluates an expression, returning one value if that expression evaluates to true, or a different one if the expression evaluates as false.

- Syntax: `condition ? result1 : result2;`
- Equivalent to: `if (condition) result1 else result2;`

- Example:

```
cin >> num1 >> num2;
(num1 >= num2) ? cout << "Foo!" : cout << "Bar!"
```

If you enter: **2 10**, you see "Bar!" on std.out

If you enter: **21 3**, you see "Foo!" on std.out

Abstract Data Types

- A **data type** consists of one or more values together with a set of basic operations defined on the values
  - Take for example, the data type `int`
  - You know how it is used, but you don't know how the computer deals with it internally
    - Do you even have to?

- A data type is called an Abstract Data Type (**ADT**) if programmers using it **do not have access to the details** of *how* the values and operations are *implemented*
  - *i.e. They're abstract to the programmer*
  - *Think of driving a car vs. Knowing how the engine is designed…*

Classes To Produce ADTs

- We want our Class data types to be designed as ADTs!
  - ADTs are *concepts* – classes are code implementations

- Separate the specification of *how* the type is used by a programmer from the details of how the type is *implemented*
  - *So that the programmer (user) doesn't see the "insides" and doesn't need to!!*

<u>This means:</u>

- Make all member variables **private** members
- Basic operations a programmer needs should be **public** member functions

```cpp
class Person {
    private:
        int age;
    public:
    // 1. Constructor with no arguments (default constr.)
    Person() { age = 20; }
    // 2. Constructor with an argument
    Person(int a) {
        age = a;
    }
    int getAge() {
        return age;
    }
};
int main() {
    Person person1, person2(45);
    cout << "Person1 Age = " << person1.getAge() << endl;
    cout << "Person2 Age = " << person2.getAge() << endl;
    return 0;
}
```

ADT Interface

- The "**ADT interface**" tells us *how to use the ADT in a program*

- The interface consists of
  - The **public member functions**
  - The **comments** that explain how to use the functions

- The interface is "public facing" and should be *all* that is needed to know how to use the ADT in a program

```cpp
class Person {
    private:
        int age;
    public:
    // 1. Constructor with no arguments (default constr.)
    Person() { age = 20; }
    // 2. Constructor with an argument
    Person(int a) {
        age = a;
    }
    int getAge() {
        return age;
    }
};
int main() {
    Person person1, person2(45);
    cout << "Person1 Age = " << person1.getAge() << endl;
    cout << "Person2 Age = " << person2.getAge() << endl;
    return 0;
}
```

ADT Implementation

- The "**ADT implementation**" tells us *how the interface is realized in C++*

- The implementation consists of
  - The **private members** of the class
  - The definitions of public and private member functions

- The implementation is needed to *run* a program, but…

  …it's not needed to *write* the main part of a program

  (or any non-member functions, for that matter)

ADT Benefits

- Changing an ADT implementation does **NOT** require
  changing a program that *uses* the ADT
- ADTs make it **standard** (thus easier) to divide work among different programmers
  - One or more can write the ADT
  - One or more can write code that uses the ADT
- Writing/using ADTs breaks the larger programming task into smaller tasks
  - Makes the project easier to work with and easier to **debug**!
- Standards and conventions in programming come up all the time in CS

Interface Preservation

To preserve the interface of an ADT so that programs using it ***do not need*** to be changed:

1. Public member declarations *cannot be changed*
2. Public member definitions *can be changed*
3. Private member functions can be added, deleted, or changed

(go crazy!)

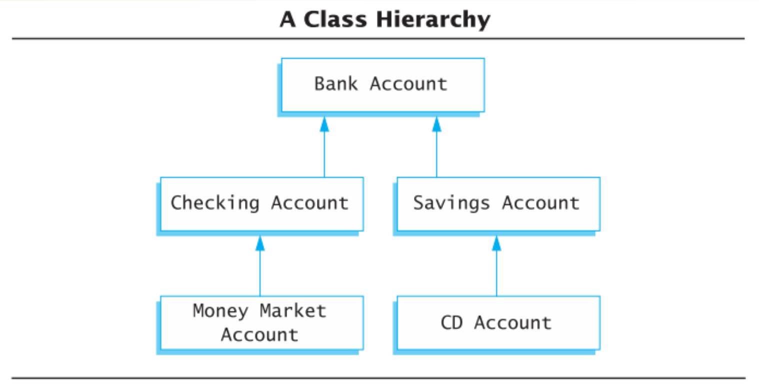
Inheritance

- **Inheritance** refers to ***derived classes***
  - Derived classes are classes that are obtained from *another class* by adding features
  - A derived class *inherits* the **member functions and variables** from its **parent class** without having to re-write them

- Example:
  - **cin** belongs to the class of **all input streams**, but not the class of input-file streams
  - I/O file streams (ifstream, ofstream) are actually derived classes from general input stream class
    - Have added features/member functions, like .open() and .close()

Inheritance Example

- Natural hierarchy of bank accounts
- Most general type of bank account: "Bank Account": it stores a balance (!)
- A *Checking Account* "IS A" "Bank Account" that also allows customers to write checks
- A *Savings Account* "IS A" "Bank Account" without checks but that provides higher interest rates



Accounts are more specific as we go down the hierarchy

Each box in the diagram above can be a class

Inheritance Relationships

- The more specific class is called a **derived** or **child** class
- The more general class is called the **base**, **super**, or **parent** class
- If class B is derived from class A then we can say:
  - Class B is a derived class of class A
  - Class B is a child of class A
  - Class A is the parent of class B
  - Class B inherits the member functions and variables of class A

Defining Derived Classes

- Give the class name as usual, but add a **colon** and then the name of the base class

```
class SavingsAccount : public BankAccount
{
   …
}
```

child

parent

- Objects of type **SavingsAccount** can access member functions defined in **SavingsAccount** or from **BankAccount**

Example

Demo file is called:
`inheritenceExample.cpp`

```
// Create a class called Vehicle - this will be our PARENT class
class Vehicle {
    public:
        Vehicle();
        void set_name(string n);
        void set_color(string c);
        void set_no_of_wheels(int nofw);
        void print();
    private:
        string name;
        string color;
        int number_of_wheels;
        bool check_name_isnt_blank();
        bool check_color_isnt_blank();
        bool check_no_of_wheels_is_positive();
};
```

```
// Create a class called Bicycle - this will be a CHILD class to Vehicle
class Bicycle:public Vehicle {
    public:
        Bicycle(bool b);
        void set_speed_bike_status(bool sbs);
    private:
        bool is_it_speed_bike;
};
```

Example

```
// Define the Vehicle Class Member Functions
Vehicle::Vehicle() {
    name="";
    color="";
    number_of_wheels=0;
}
void Vehicle::set_name(string n) {
    name = n;
}
void Vehicle::set_color(string c) {
    color = c;
}
void Vehicle::set_no_of_wheels(int nofw) {
    number_of_wheels = nofw;
}
void Vehicle::print() {
    cout << "Name::::::::::::: " << name << endl;
    cout << "Color:::::::::::: " << color << endl;
    cout << "Number of Wheels: " << number_of_wheels << endl;
}
```

Example

```
// These are the Vehicle class private member functions
// Just placeholders (stubs) for the purposes of this demo
bool Vehicle::check_name_isnt_blank() {
    return true;
}
bool Vehicle::check_color_isnt_blank() {
    return true;
}
bool Vehicle::check_no_of_wheels_is_positive() {
    return true;
}

// Define the Bicycle Class Member Functions
Bicycle::Bicycle(bool b) {
    set_no_of_wheels(2);
    is_it_speed_bike = b;
}
void Bicycle::set_speed_bike_status(bool sbs) {
    is_it_speed_bike = sbs;
}
```

Example

```
int main()
{
    // Define the class objects
    Vehicle semi;
    Bicycle bikey(true);

    // Set parameters (that are private member variables) using
    // the public member functions of Vehicle
    semi.set_name("18 Wheeler");
    semi.set_color("Red");
    semi.set_no_of_wheels(18);

    // Same thing, BUT NOTICE THAT:
    // .set_name() and .set_color() are Vehicle class member functions
    //          passed on to the Bicycle object BY INHERITENCE!
    // .set_speed_bike_status() is only a Bicycle member function,
    //          i.e. can only be used by Bicycle objects
    bikey.set_name("Chuck");
    bikey.set_color("Blue");
    bikey.set_speed_bike_status(false);

    // Use the Vehichle member function .print() to print out the
    // values of private member variables in semi and bikey objects
    cout << "The Vehicle, semi, has the following features:\n";
    semi.print();
    cout << endl;
    cout << "The Bicycle, bikey, has the following features:\n";
    bikey.print();

    return 0;
}
```

Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion

A child couldn't sleep,
so her mother told a story about a little frog,
who couldn't sleep,
so the frog's mother told a story about a little bear,
who couldn't sleep,
so bear's mother told a story about a little weasel
...who fell asleep.
...and the little bear fell asleep;
...and the little frog fell asleep;
...and the child fell asleep.

Recursive Functions

- **Recursive: (adj.) Repeating unto itself**
- **A recursive function contains a *call* to itself**

- When breaking a task into subtasks,
  it may be that the subtask
  is a *smaller example*
  of the same task

Example: The Factorial Function

Recall: x! = 1 * 2 * 3 … * x

You could code this out as either:

- A loop:

```
(for k=1; k < x; k++) { factorial *= k; }
```

- Or a recursion/repetition:

```
factorial(x)  = x * factorial(x-1)
              = x * (x-1) * factorial (x-2)
              = etc…
until you get to factorial(1) (then what?!?)
```

Example: Recursive Formulas

- Recall from Math, that you can create a recursive formula from a sequence

Example:

- Consider the arithmetic sequence:

5, 10, 15, 20, 25, 30, …

- I note that I can write each number in the sequence as:

$\mathbf{a_n = a_{n-1} + 5}$ *(**n** being the position)*

For example: $\mathbf{a_4} = a_3 + 5$

$= (a_2 + 5) + 5$

$= ((a_1 + 5) + 5) + 5$ ← *At this point, I need to designate a_1 as 5 since it's the starting value*

$= (5 + 5 + 5 + 5) = \mathbf{20}$

The Base Case

$a_n = a_{n-1} + 5$

- If we assume that we start the sequence at n = 1… (an arbitrary value)

  … then we could devise an algorithm for **a(n)** like this:

1. If n = 1, then **return 5** to **a(n)** ← The **BASE** case
2. Otherwise, **return a(n-1) + 5** ← The **RECURSION** (i.e. the function calling itself with a diff. argument)

- I'll ***need to know*** what that base case is, otherwise I risk not ending my recursion (or not making sense of it)

Demo file is called:
simpleRecursion.cpp

Coding It

```
int Series(int n) {
// The BASE case:
   if (n <= 1) {        // why <= and not == ??
      return 5;
   }                    // why is there no “else” statement??

// The RECURSION:
   return Series(n – 1) + 5;
}
```

YOUR TO-DOs

- Start on Lab #8 and Homework #8
  - Those are due next Monday, like usual

- After the next pre-recorded video, start on Lab #9 and Homework #9
  - Those are due by Friday *next* week (last day of the quarter).
  - **NO LATE SUBMISSIONS ALLOWED FOR THIS ONE!!**

- Take advantage of office hours this week!!
- Look for a practice exam for the final towards the end of this week.

- Take Quiz #8 on Friday!

</LECTURE>