UCLA Samueli Computer Science

CS32: Introduction to Computer Science

Discussion Week 5

Junheng Hao, Ramya Satish

Feb. 8, 2019

Announcement

UCLA Samueli Computer Science

- Homework 3 is due Tuesday, February 12.
  - Check all requirements especially for recursion!
  - In Problem 2, you are required to "Replace the incorrect implementations of these functions with correct ones that use recursion in a useful way; your solution must not use the keywords `while`, `for`, or `goto`. You must not use global variables or variables declared with the keyword `static`, and you must not modify the function parameter lists." (David Smallberg)

Outline

UCLA Samueli Computer Science

- Inheritance and polymorphism
- Recursion (as a problem solving technique)

Inheritance & Polymorphism

From last discussion

UCLA Samueli Computer Science

- **Inheritance**
  - Motivation & Definition: Deriving a class from another
  - Reuse, extension, specification (override)
  - Construction & Destruction
  - Override a member function

- **Polymorphism**
  - Virtual functions
  - Examples of polymorphism
  - Abstract base class

Inheritance

Motivation & Review

UCLA Samueli Computer Science

- The basis of all *Object Oriented Programming*. And you'll almost certainly get grilled on it! --- From: Nachenberg, Slides L6P3
- The process of deriving a new class using another class as base.
- Difference of *"is a"(class hierarchy)* and *"has a"(has member/properties)*

```
class Person {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
private:
  string m_sName;
  int m_nAge;
};
```

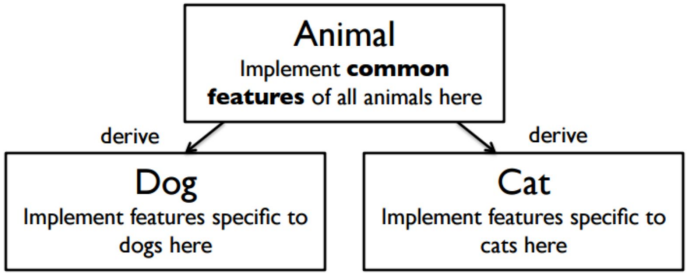
```
class Student {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getStudentID();
  void setStudentID();
  float getGPA();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  float m_GPA;
};
```

```
class Professor {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getProfID();
  void setProfID();
  bool getIsTenured();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  bool isTenured;
};
```

Inheritance

Example: Reuse and Extension

UCLA Samueli Computer Science

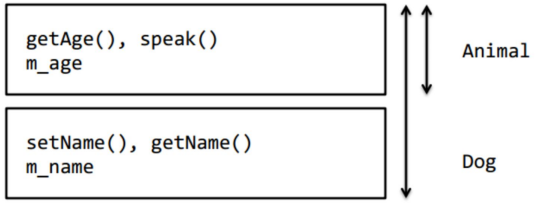



```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class




```
Dog d1;
d1.setName("puppy");
d1.getAge();
d1.speak();
```

```
Animal a1;
a1.speak();
a1.setName("abc");
```

Inheritance

Summary of Reuse and Extension

UCLA Samueli Computer Science

- Reuse
  - Every public method in the base class is automatically reused/exposed in the derived class (just as if it were defined).
  - Only public members in the base class are exposed/reused in the derived class(es)! Private members in the base class are hidden from the derived class(es)!
  - Special case for protected members.
- Extension
  - All public extensions may be used normally by the rest of your program.
  - Extended methods or data are unknown to your base class.

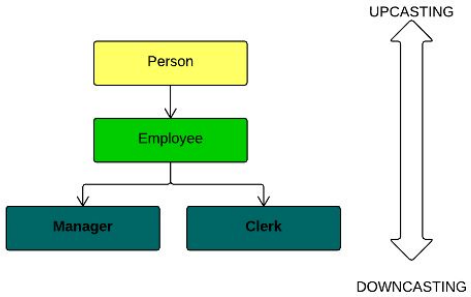
- What about overriding a member function from base classes?

Inheritance

Automatic conversion, Upcasting, Downcasting

UCLA Samueli Computer Science

- Upcasting: A derived class pointer (or reference) to be treated as base class pointer
- Downcasting: Converting base class pointer (or reference) to derived class pointer.



Inheritance

Specialization/Overriding member functions

UCLA Samueli Computer Science

- Overriding: same function name, return type and parameter list, defined again in derived classes and different from the base class.
- Different from overloading (same function name, different return type and/or different set of arguments)
- You can still call the member function of base classes, but it seems very rare.

```
Dog d1;
d1.Animal::speak();
```

- Consider how to apply `virtual` keyword in overriding member functions

```
void Animal::speak() const
{
  cout << "..." << endl;
}
```

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    void speak() const;
  private:
    string m_name;
};

void Dog::speak() const
{
  cout << "Woof!" << endl;
}
```

Inheritance

Construction

UCLA Samueli Computer Science

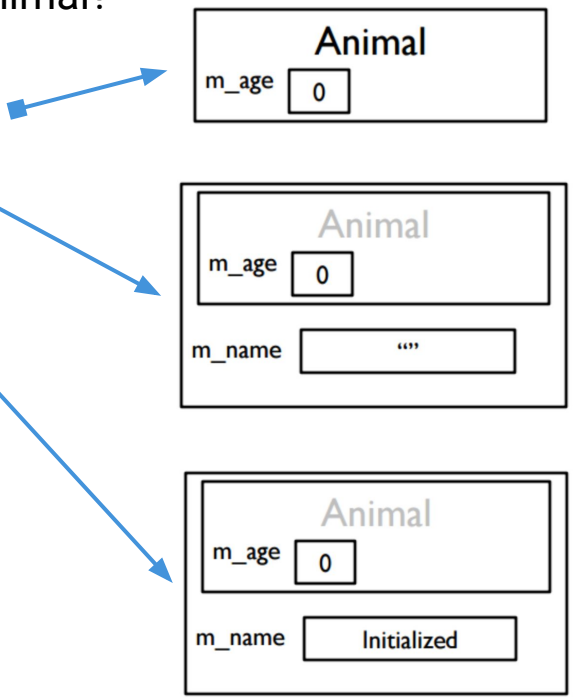
- How to construct a Dog, which is a derived class from Animal?
- Steps:
  - The base part of the class (Animal) is constructed.
  - The member variables of Dog are constructed.
  - The body of constructor (Dog) is executed.

```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class



Inheritance

Overload Constructor

UCLA Samueli Computer Science

- How to overload Dog's constructor to create `Dog::Dog(string initName, int initAge)` ?

Wrong:

```
Dog::Dog(string initName, int initAge)
:m_age(initAge), m_name(initname)
{}
```

Correct:

```
Dog::Dog(string initName, int initAge)
:Animal(initAge), m_name(initname)
{}

class Animal{
  public:
    Animal(init initAge);
    ...
}
```

Inheritance

Destruction

UCLA Samueli Computer Science

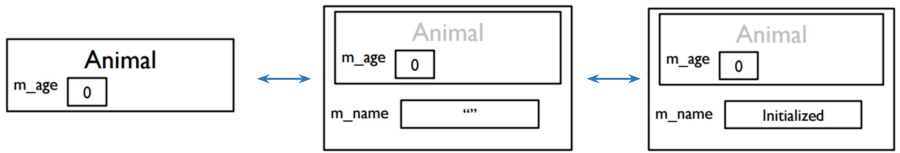
The order of destruction of a derived class: **Just reverse the order of construction.**

Order of construction:

1. Construct the base part, consulting the member initialization list (If not mentioned there, use base class's default constructor)
2. Construct the data members, consulting the member initialization list.(If not mentioned there, use member's default constructor if it's of a class type, else leave uninitialized.)
3. Execute the body of the constructor.

Order of destruction:

1. Execute the body of the destructor.
2. Destroy the data members (doing nothing for members of builtin types).
3. Destroy the base part.



Inheritance: Test Now!

Example: Worksheet 2 Question

UCLA Samueli Computer Science

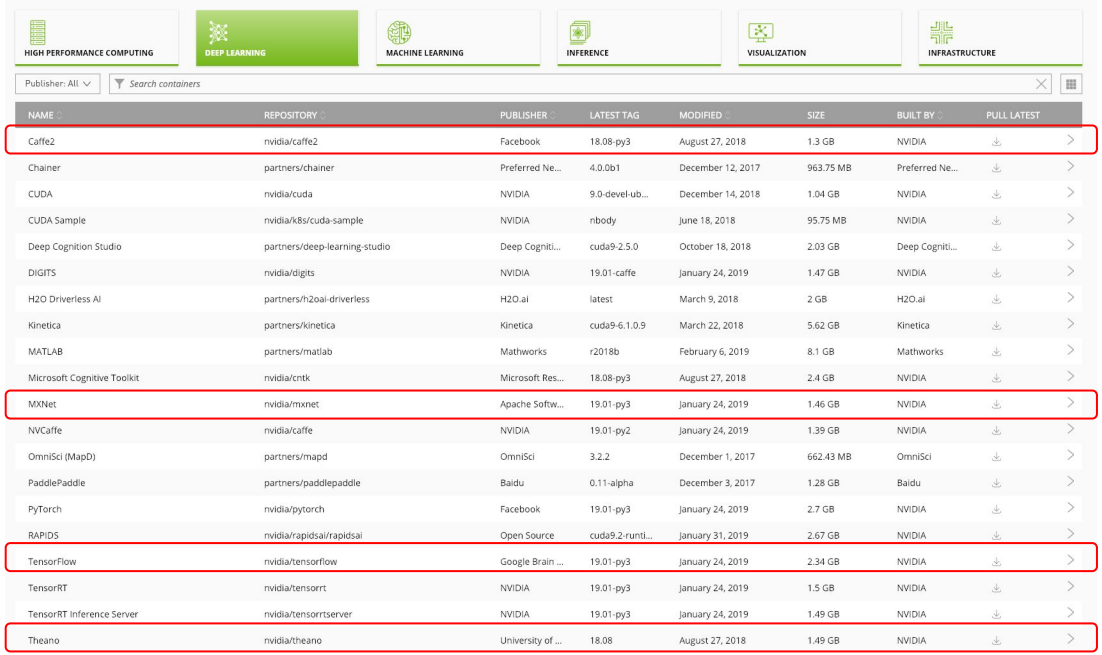
- Pay attention to:
  - Construction and destruction of derived classes and base classes
  - Difference of inherited class and data members

* Philosophy/Inheritance

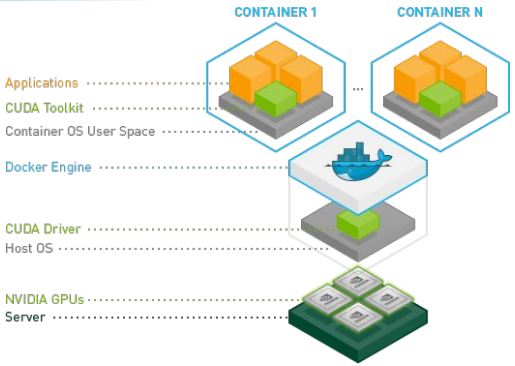
Another “not-so-related” example

UCLA Samueli Computer Science

- There are many examples and applications of “inheritance”.
- One example: Commonly-used Docker Images



NAME	REPOSITORY	PUBLISHER	LATEST TAG	MODIFIED	SIZE	BUILT BY	PULL LATEST
Caffe2	nvidia/caffe2	Facebook	18.08-py3	August 27, 2018	1.3 GB	NVIDIA	
Chainer	partners/chainer	Preferred Ne...	4.0.0b1	December 12, 2017	963.75 MB	Preferred Ne...	
CUDA	nvidia/cuda	NVIDIA	9.0-devel-ub...	December 14, 2018	1.04 GB	NVIDIA	
CUDA Sample	nvidia/k8s/cuda-sample	NVIDIA	nbody	June 18, 2018	95.75 MB	NVIDIA	
Deep Cognition Studio	partners/deep-learning-studio	Deep Cogniti...	cuda9-2.5.0	October 18, 2018	2.03 GB	Deep Cogniti...	
DIGITS	nvidia/digits	NVIDIA	19.01-caffe	January 24, 2019	1.47 GB	NVIDIA	
H2O Driverless AI	partners/h2oai-driverless	H2O.ai	latest	March 9, 2018	2 GB	H2O.ai	
Kinetica	partners/kinetica	Kinetica	cuda9-6.1.0.9	March 22, 2018	5.62 GB	Kinetica	
MATLAB	partners/matlab	Mathworks	r2018b	February 6, 2019	8.1 GB	Mathworks	
Microsoft Cognitive Toolkit	nvidia/cntk	Microsoft Res...	18.08-py3	August 27, 2018	2.4 GB	NVIDIA	
MXNet	nvidia/mxnet	Apache Softw...	19.01-py3	January 24, 2019	1.46 GB	NVIDIA	
NVCaffe	nvidia/caffe	NVIDIA	19.01-py2	January 24, 2019	1.39 GB	NVIDIA	
OmniSci (MapD)	partners/mapd	OmniSci	3.2.2	December 1, 2017	662.43 MB	OmniSci	
PaddlePaddle	partners/paddlepaddle	Baidu	0.11-alpha	December 3, 2017	1.28 GB	Baidu	
PyTorch	nvidia/pytorch	Facebook	19.01-py3	January 24, 2019	2.7 GB	NVIDIA	
RAPIDS	nvidia/rapidsai/rapidsai	Open Source	cuda9.2-runti...	January 31, 2019	2.67 GB	NVIDIA	
TensorFlow	nvidia/tensorflow	Google Brain ...	19.01-py3	January 24, 2019	2.34 GB	NVIDIA	
TensorRT	nvidia/tensorrt	NVIDIA	19.01-py3	January 24, 2019	1.5 GB	NVIDIA	
TensorRT Inference Server	nvidia/tensorrtserver	NVIDIA	19.01-py3	January 24, 2019	1.49 GB	NVIDIA	
Theano	nvidia/theano	University of ...	18.08	August 27, 2018	1.49 GB	NVIDIA	



Inheritance does not exactly just means base/derived class in C++ programming. It is everywhere.

Polymorphism

Motivation & Definition

UCLA Samueli Computer Science

- Polymorphism is how you make Inheritance truly useful.
- Think about example of dogs and animals. Once I define a function that accepts **`a`** (reference or pointer to **`a`**) `Animal`, not only can I pass `Animal` variables to that class, But I can also pass any variable that was derived from a `Animal`(such as `Dogs`)!

Polymorphism

Virtual Functions: Examples

UCLA Samueli Computer Science

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

```
class Square: public Shape {
  public:
    Square(int side){ m_side=side; }
    virtual double getArea()
    { return (m_side*m_side); }
    ...
  private:
    int m_side;
};
```

```
class Square: public Shape {
  public:
    Circle(int rad){ m_rad=rad; }
    virtual double getArea()
    { return (3.14*m_rad*m_rad);}
    ...
  private:
    int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
  cout << "Cost is: $";
  cout << x.getArea()*3.25;
}
int main() {
  Square s(5);
  Circle c(10);
  PrintPrice(s);
  PrintPrice(c);
}
```

When you use the `virtual` keyword, C++ figures out what class is being referenced and calls the right function.

Polymorphism works with pointers too.

I will not forget to add `virtual` in front of my destructors when I use inheritance/polymorphism. → *What is the problem if not?*

Polymorphism

Pure Virtual Functions & Abstract Base Class

UCLA Samueli Computer Science

- Sometimes we have no idea what to implement in base functions. For example, without knowing what the animal is, it is difficult to implement the `speak()` function.
- Solution: Pure virtual functions
- Note:
  - Declare pure virtual functions in the base class. (=0!)
  - Considered as dummy function.
  - The derived class **MUST** implement all the pure virtual functions of its base class.
- If a class has at least one pure virtual function, it is called *abstract base class*.

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

Never actually used!

```
class Animal
{
  public:
    Animal();
    virtual ~Animal();
    int getAge() const;
    virtual void speak() const = 0;
  private:
    int m_age;
};
```

Polymorphism

Cheatsheet from Carey's slides

UCLA Samueli Computer Science

You can't access private members of the base class from the derived class:

```
// BAD!
class Base
{
public:
...


private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
   {
      v = q;  // ERROR!
   }


   void foo()
   {
      v = 10; // ERROR!
   }
};
```

```
// GOOD!
class Base
{
public:
   Base(int x)
     { v = x; }
   void setV(int x)
     { v = x; }
   ...
private:
   int v;
};

class Derived: public Base
{
public:

   Derived(int q)
     : Base(q)   // GOOD!
   {
     ...
   }

   void foo()
   {
      setV(10);  // GOOD!
   }
};
```

Always make sure to add a virtual destructor to your base class:

```
// BAD!
class Base
{
public:
  ~Base() { ... } // BAD!
  ...
};

class Derived: public Base
{
  ...
};
```

```
// GOOD!
class Base
{
public:
  virtual ~Base() { ... } // GOOD!
  ...
};

class Derived: public Base
{
  ...
};
```

```
class Person
{
public:
  virtual void talk(string &s) { ... }
};

class Professor: public Person
{
public:
    void talk(std::string &s)
    {
        cout << "I profess the following: ";
        Person::talk(s); // uses Person's talk
    }
};
```

Don't forget to use virtual to define methods in your base class, if you expect to re-define them in your derived class(es)

To call a base-class method that has been re-defined in a derived class, use the base:: prefix!

So long as you define your BASE version of a function with virtual, all derived versions of the function will automatically be virtual too (even without the virtual keyword)!

Polymorphism

Cheatsheet from Carey's slides (Cont'd)

UCLA Samueli Computer Science

```
class SomeBaseClass
{
public:
   virtual void aVirtualFunc() { cout << "I'm virtual"; } // #1
   void notVirtualFunc() { cout << "I'm not"; }           // #2
   void tricky()                                          // #3
   {
       aVirtualFunc();                                    // ***
       notVirtualFunc();
    }
};

class SomeDerivedClass: public SomeBaseClass
{
public:
   void aVirtualFunc()   { cout << "Also virtual!"; }     // #4
   void notVirtuaFuncl() { cout << "Still not"; }         // #5
};

int main()
{
   SomeDerivedClass d;
   SomeBaseClass  *b = &d;  // base ptr points to derived obj

   // Example #1
   cout << b->aVirtualFunc();      // calls function #4

   // Example #2
   cout << b->notVirtualFunc(); // calls function #2

   // Example #3
   b->tricky();       // calls func #3 which calls #4 then #2
}
```

Example #1: When you use a BASE pointer to access a DERIVED object, AND you call a VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the DERIVED version of the function.

Example #2: When you use a BASE pointer to access a DERIVED object, AND you call a NON-VIRTUAL function defined in both the BASE and the DERIVED classes, your code will call the BASE version of the function.

Example #3: When you use a BASE pointer to access a DERIVED object, all function calls to VIRTUAL functions (***) will be directed to the derived object's version, even if the function (tricky) calling the virtual function is NOT VIRTUAL itself.

Recursion

Basics

UCLA Samueli Computer Science

- Function-writing technique where the functions refers to itself.
- Let's talk about the factorial example again!
  - Similar to mathematical induction → Prove `k=1` is valid and prove `k=n` is valid when `k=n-1` is valid.
  - Base cases are important and need to be carefully considered.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}
```

Without explicit loops!

Recursion

Pattern: How to write a recursive function

UCLA Samueli Computer Science

- Step 1: Find the base case(s).
  - What are the trivial cases? Eg. empty string, empty array, single-item subarray.
  - When should the recursion stop?
- Step 2: Decompose the problem.
  - Take tail recursion as example.

> ➔ Take the first (or last) of the ***n*** items of information
> ➔ Make a recursive call to the rest of **(*n*-1)** items. The recursive call will give you the correct results.
> ➔ Given this result and the information you have on the first (or last item) conclude about current ***n*** items.

- Step 3: Just solve it! *(Well, easier said than done~)*

Recursion

Practice Examples

UCLA Samueli Computer Science

- Practice: Print out the permutations of a given vector.

```
Input: [1,2,3]
Output: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]
Implement: void permutation(vector<int>& nums,  int start);
```

- Note: Some data structures are easy to implement recursive technique: arrays, trees (will be discussed later).

UCLA Samueli Computer Science

Thank you!

Q & A

Group Exercise

UCLA Samueli Computer Science

Exercise problems from Worksheet 1 & 2 (see “LA worksheet” tab in CS32 website).

Topics today:

- Linked List (Worksheet 1, Question 5-12)
- Stack and queue (Worksheet 2, Question X)
- Inheritance (Worksheet 2, Question X)

Answers will be posted after discussions.

Answer 5

UCLA Samueli Computer Science

```
bool isSolvable(int x, int y, int c) {
  if (c == 0)
    return true;
  if (c < 0)
    return false;

  return isSolvable(x, y, c - x) ||
isSolvable(x, y, c - y);
}
```

Answer 12

UCLA Samueli Computer Science

```
string lcs(string s1, string s2) {
    if (s1.empty()  ||  s2.empty()) //
base case: either empty
        return "";
    // split the strings into head and
tail for simplicity
    char s1_head = s1[0];
    string s1_tail = s1.substr(1);
    char s2_head = s2[0];
    string s2_tail = s2.substr(1);

    // if heads are equal, use the head
and
    // recursively find rest of common
subsequence
    if (s1_head == s2_head)
        return s1_head + lcs(s1_tail,
s2_tail);
    // heads different, so check for
common subsequences not
    // including one of the heads
    string if_behead_s1 = lcs(s1_tail,
s2);
    string if_behead_s2 = lcs(s1,
s2_tail);

    // return the longer of the
subsequences we found
    if (if_behead_s1.length() >=
if_behead_s2.length())
        return if_behead_s1;
    else
        return if_behead_s2;
}
```

Answer 13

UCLA Samueli Computer Science

```
Node* merge(Node* l1, Node* l2)
     {
     // base cases: if a list is empty,
return the other list
          if (l1 == nullptr)
               return l2;
          if (l2 == nullptr)
               return l1;

     // determine which head should be the
head of the merged list
     // then set the next pointer to the
head of the recursive calls
          Node* head;
          if (l1->val < l2->val) {
               head = l1;
               head->next = merge(l1->next,
l2);
          }
          else {
               head = l2;
               head->next = merge(l1,
l2->next);
          }

     // return the head of the merged list
          return head;
     }
```