UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 6

Yichao (Joey)

Feb. 14, 2019

Credit: Carey Nachenberg, Junheng Hao

Announcements

UCLA Samueli Computer Science

- Project 3 part 1 is due 11:00PM Thursday, February 20.
- Homework 4 is due 11:00 PM Tuesday, March 3.
- Midterm 2 is scheduled February 25.

Outline Today

UCLA Samueli Computer Science

- Inheritance and polymorphism
- Recursion
- Template and STL

Inheritance & Polymorphism

Motivation & Review

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;
};
```

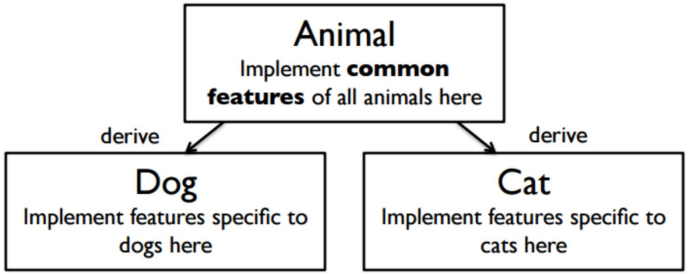
```
class Student {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getStudentID();
  void setStudentID();
  float getGPA();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  float m_GPA;
};
```

```
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

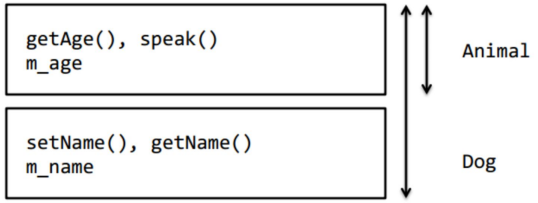



```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

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.

Inheritance

Summary of Reuse and Extension

UCLA Samueli Computer Science

When deriving a class from a public base class,

Public members of the base class become public members of the derived class

Protected members of the base class become protected members of the derived class.

A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

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.
- 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

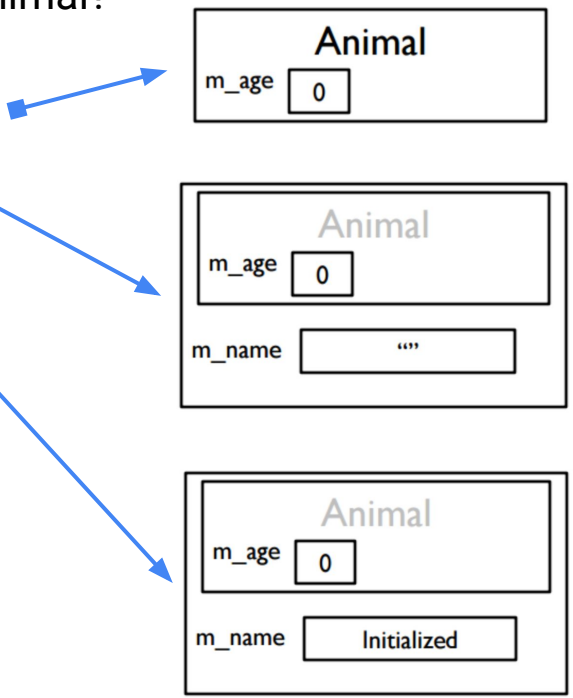
- How to construct a Dog, which is a derived class from Animal?
- Steps:
  - The base part of the class (Animal) is constructed.
  - The member variables of Dog are constructed.
  - The body of constructor (Dog) is executed.

```
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

Order of Construction and destruction

UCLA Samueli Computer Science

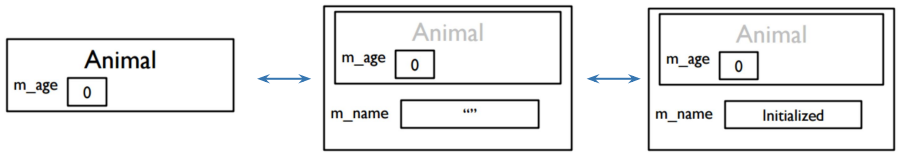
The order of destruction of a derived class: **Just reverse the order of construction.**

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.



Construction & Destruction

Note: There is a difference between class composition and class inheritance.

```
#include <iostream>
#include <string>
using namespace std;

class A
{
public:
        A(){cout << "A()" << endl;}
        A(int x){cout << "A(" << x << ")" << endl; this->id = x;}
        ~A(){cout << "~A("<< this->id <<")" << endl;}
private:
        int id;
};

class B
{
public:
        B():a1(888),a2(444){cout << "B()" << endl;}
        ~B(){cout << "~B()" << endl;}
private:
        A a2;
        A a1;
};

int main()
{
        B b;
        return 0;
}
```

What if we change to **`class B: public A`** ?

Construction

One more test!

What is the output of

```
int main(){
    C c;
}
```

```
class A
{
  public:
    A() { cout << "A()" << endl; }
    A(int x) { cout << "A(" << x << ")" << endl; }
    ~A() { cout << "~A()" << endl; }
};

class B
{
  public:
    B() { cout << "B()" << endl; }
    B(int x) : m_a(x) { cout << "B(" << x << ")" << endl; }
    ~B() { cout << "~B()" << endl; }
  private:
    A m_a;
};

class C : public A
{
  public:
    C() : A(10), m_b2(5) { cout << "C()" << endl; }
    ~C() { cout << "~C()" << endl; }
  private:
    B m_b1;
    B m_b2;
};
```

```
A(10)
A()
B()
A(5)
B(5)
C()
~C()
~B()
~A()
~B()
~A()
~A()
```

*Philosophy/Inheritance

Another example

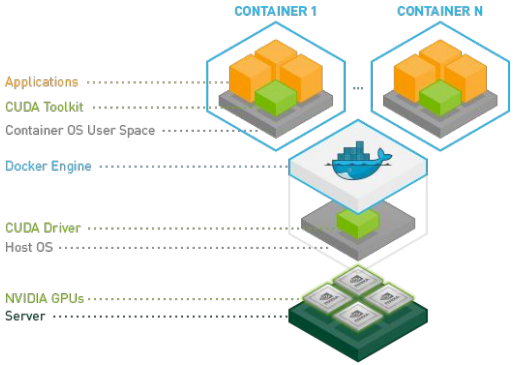
UCLA Samueli Computer Science

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

HIGH PERFORMANCE COMPUTING | DEEP LEARNING | MACHINE LEARNING | INFERENCE | VISUALIZATION | INFRASTRUCTURE

Publisher: All | Search containers

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.
- Once I define a function that accepts `Animal a` (reference or pointer to **a**),
  not only can I pass `Animal` variables to that class,
  but I can also pass any variable that was derived from `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 Circle: 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 referred** and **calls the right function.**

I will not forget to add `virtual` in front of my destructors when I use inheritance/polymorphism.

→ *What is the problem if not?*

```cpp
#include <iostream>

using namespace std;

class Base
{
public:
    Base(){
        cout << "Base Constructor Called\n";
    }
    ~Base(){
        cout << "Base Destructor called\n";
    }
};

class Derived1: public Base
{
public:
    Derived1(){
        cout << "Derived constructor called\n";
    }
    ~Derived1(){
        cout << "Derived destructor called\n";
    }
};

int main()
{
    Base *b = new Derived1();
    delete b;
}
```

```cpp
#include <iostream>

using namespace std;

class Base
{
public:
    Base(){
        cout << "Base Constructor Called\n";
    }
    virtual ~Base(){
        cout << "Base Destructor called\n";
    }
};

class Derived1: public Base
{
public:
    Derived1(){
        cout << "Derived constructor called\n";
    }
    ~Derived1(){
        cout << "Derived destructor called\n";
    }
};

int main()
{
    Base *b = new Derived1();
    delete b;
}
```

Virtual destructors are useful when you might potentially delete an instance of a derived class **through a pointer to base class**.

Since Base's destructor is not virtual and b is a Base* pointing to a Derived object, delete b has undefined behaviour.
Deleting through a base pointer that points to an object of another type requires a virtual destructor.

```
Base Constructor Called
Derived Constructor called
Derived destructor called
Base destructor called
```

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;
}
```

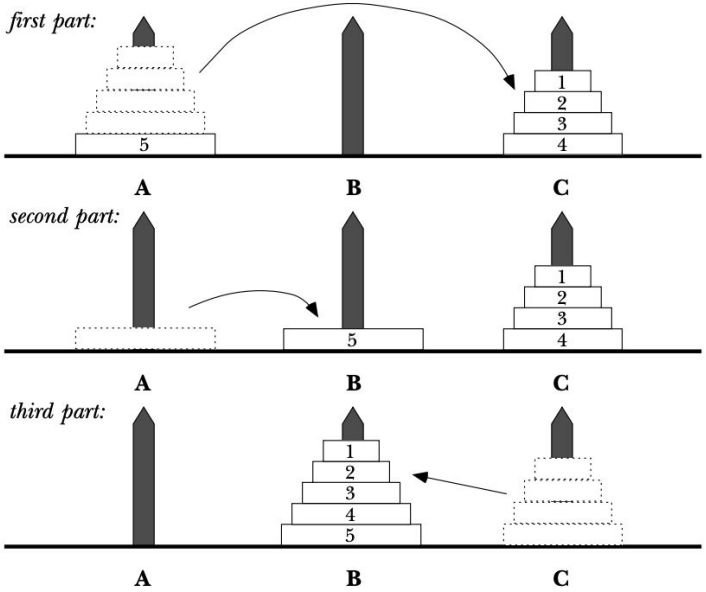
```
int factorial(int n)
{
    if (n <= 1)
        return 1;

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

Without explicit loops!

Recursion

Hanoi's story

UCLA Samueli Computer Science




```
➔  to hanoi :n
➔  hanoi :n-1
➔  movedisk :n
➔  hanoi :n-1
➔  end
```

```
➔  to hanoi :n-1
➔  hanoi :n-2
➔  movedisk :n
➔  hanoi :n-2
➔  end
```

```
➔  to hanoi :n-1
➔  hanoi :n-2
➔  movedisk :n
➔  hanoi :n-2
➔  end
```

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!

Recursion

Examples

UCLA Samueli Computer Science

- Problem 1: Given an integer array a and its length n, return whether the array contain any element that is smaller than 0.
- Problem 2: Given an integer array a and its length n, count the number of elements that are smaller than 0.

```
// a simple function with for loop
bool anyTrue(const double a[], int n)
{
  for (int k = 0; k < n; k++)
  {
    if (a[k] < 0)
      return true;
  }
  return false;
}
```

```
// try: without for loop
bool anyTrue(const double a[], int n)
{
  // recursion implementation
}
```

Recursion

Practice Examples

UCLA Samueli Computer Science

- Practice: Print out the permutations of a given vector (Difficulty: Hard).

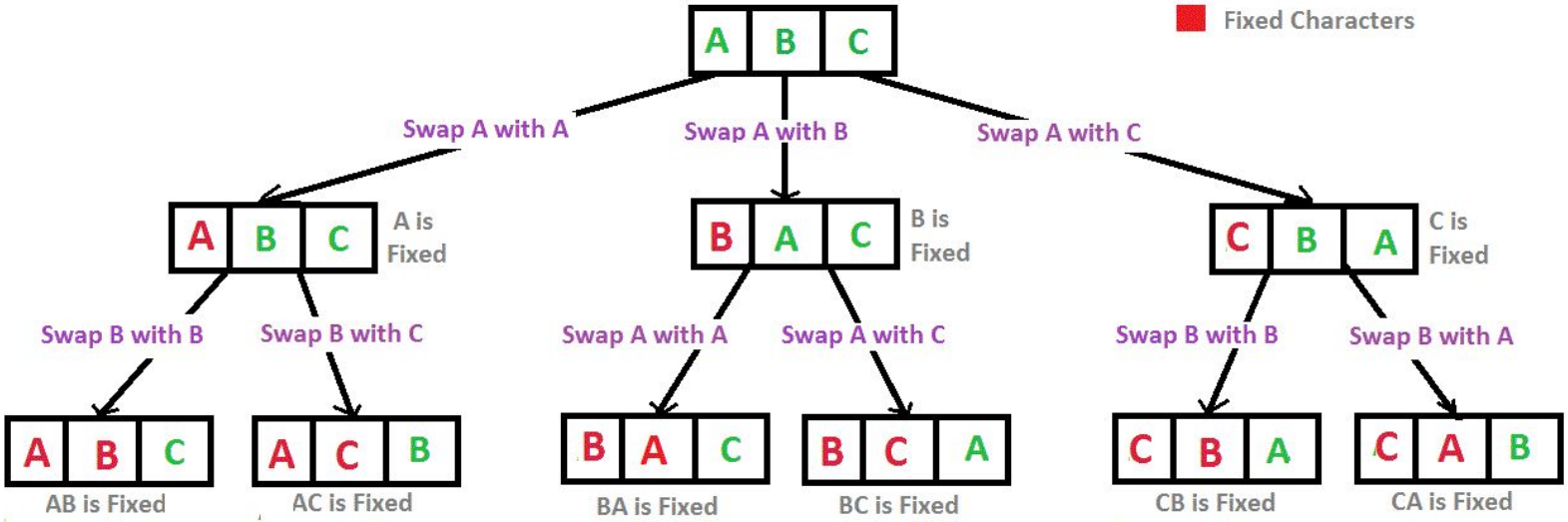
```
Input: [A,B,C]
Output: [A,B,C], [A,C,B], [B,A,C], [B,C,A], [C,A,B], [C,B,A]
Implement: void permutation(vector<string>& str, int n);
```

- Note: What is the base case?

Recursion

Practice Examples

UCLA Samueli Computer Science



Recursion Tree for Permutations of String "ABC"

```cpp
#include <iostream>
using namespace std;

// Function to find all Permutations of a given string str[i..n-1]
// containing all distinct characters
void permutations(string str, int i, int n)
{
    // base condition
    if (i == n - 1)
    {
        cout << str << endl;
        return;
    }

    // process each character of the remaining string
    for (int j = i; j < n; j++)
    {
        // swap character at index i with current character
        swap(str[i], str[j]);        // STL swap() used

        // recur for string [i+1, n-1]
        permutations(str, i + 1, n);

        // backtrack (restore the string to its original state)
        swap(str[i], str[j]);
    }
}

// Find all Permutations of a string
int main()
{
    string str = "ABC";

    permutations(str, 0, str.length());

    return 0;
}
```

Recursion

Smallberg's notes

UCLA Samueli Computer Science

What I've emphasized all week is how to understand a recursive solution to a problem and know it's correct:

1. **Identify the base cases** (paths through the function that make no recursive calls) and recursive cases.
2. Come up with measure of the size of the problem for which the base case(s) provide a bottom, typically 0 or 1.
3. Verify that if the function is called with a problem of some size, **any recursive call it makes is to solve a problem of a strictly smaller size**. (Problem sizes should be nonnegative integers.) This proves termination, since a decreasing sequence of nonnegative integers must eventually hit bottom.
4. Now that we've proved termination, **verify that the base cases are handled correctly**.

Recursion

Practice Examples: Merge sort and Quick sort

UCLA Samueli Computer Science

Merge sort

1. Find the middle point to divide the array into two halves:

```
middle m = (l+r)/2
```

2. Call mergeSort for first half:

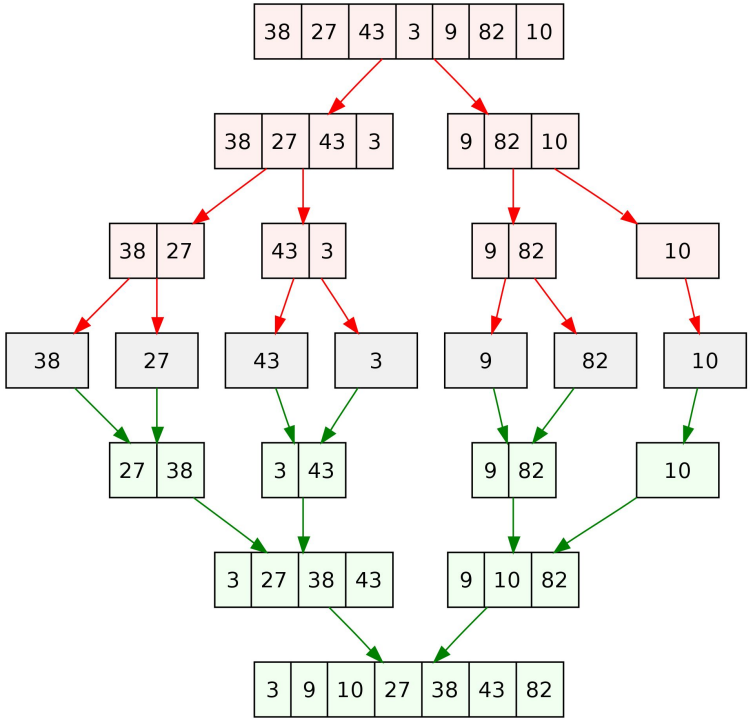
```
mergeSort(arr, l, m)
```

3. Call mergeSort for second half:

```
mergeSort(arr, m+1, r)
```

4 .Merge the two halves sorted in step 2 and 3:

```
merge(arr, l, m, r)
```



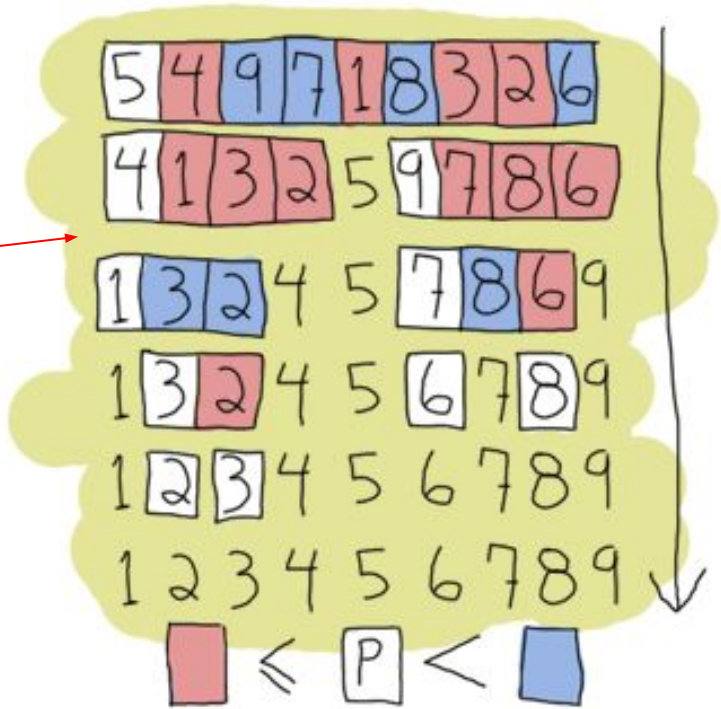
Recursion

Practice Examples: Merge sort and Quick sort

UCLA Samueli Computer Science

Quick Sort

Two recursion calls!



Template

Motivation: More generic class

- Think about the `Pair` class. The class should not work only with integers. That is we want a **"generic" `Pair` class**.
- `Pair<int> p1; Pair<char> p2;`

```
class Pair {
    public:
       Pair();
       Pair(int firstValue,
            int secondValue);
       void setFirst(int newValue);
       void setSecond(int newValue);
       int getFirst() const;
       int getSecond() const;
    private:
       int m_first;
       int m_second;
};
```

```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

Template

Multi-type template

- What if we need pair with different types?
- Change your template class: `Pair<int, string> p1;`

```
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```

```
template<typename T, U>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             U secondValue);
        void setFirst(T newValue);
        void setSecond(U newValue);
        T getFirst() const;
        U getSecond() const;
    private:
        T m_first;
        U m_second;
};
```

Template

Change member functions in template classes

- Member function should also be edited in template class as well.

```
void Pair::setFirst(int newValue)
{
  M_first = newValue;
}
```

```
template<typename T>
void Pair<T>::setFirst(T newValue)
{
  M_first = newValue;
}
```

Template

Template Specialization

- What if we want a template class with certain data type to **have its own exclusive behaviors**? For example, in `Pair` class we only allow `Pair<char>` has `uppercase()` and `lowercase()` function but not for `Pair<int>`.

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

```
Pair<int> p1;
Pair<char> p2;

p1.uppercase(); //error
p2.uppercase(); //correct
```

Template

Const references as parameters

- When you are not changing the values of the parameters, make them `const` references to avoid potential computational cost. (Pass by value for ADTs are slow.)

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

Template

Some notes

- Generic comparisons:
  - `bool operator>=(const ItemType& a, const ItemType& b)`
- Use the template data type (e.g. `T`) to define the type of at least one formal parameter.
- Add the prefix `template <typename T>` before the class definition itself and before each function definition outside the class.
- Place the postfix `<T>` Between the class name and the :: in all function definition.

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void);
  private:
    T m_a;
};
```

```
template <typename T>
void Foo<T>::setVal(T a)
{
  m_a = a;
}
template <typename T>
void Foo<T>::printVal(void)
{
  cout << m_a << "\n";
}
```

STL: Standard Template Library

Easy and efficient implementation

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues…

STL: Standard Template Library

Notes on `vector` and `list`

- You may only use brackets to access existing items in `vector`. **Keep the current size vector in mind** especially after `push_back()` and `pop_back()`.
- **You cannot access list element by brackets.**
- Choose between vector and list:
  - `vectors` are based on **dynamic arrays** placed in contiguous storage. Fast on access but slow on insertion/deletion.
  - `lists` are the opposite (**linked list**). It offers fast insertion/deletion, but slow access to middle elements.

STL: Standard Template Library

Notes on `size` and `capacity`

- Question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

```
size: ?
capacity: ?

size: ?
capacity: ?

max size: ?
```

STL: Standard Template Library

Notes on `size` and `capacity`

- Question: Size and capacity of a vector?

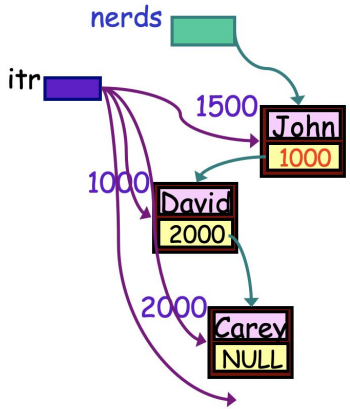
```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;

  return 0;
}
```

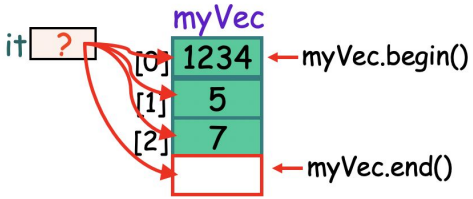
```
→ On my computer:
size:1
capacity:1
size:101
capacity:128
```

STL: Standard Template Library

Implementation example: Iterators

- STL Iterators: Use `.begin()` and `.end()`
  - `.begin()` : return an iterator that points to the first element.
  - `.end()`: return an iterator that points to the ***past-the-last*** element.
- A container as a `const` reference cannot use regular iterator but **need to use `const` iterator**. Example: `list<string>::const_iterator it;`
- Examples






```
void main()
{
  vector<int>   myVec;
  myVec.push_back(1234);
  myVec.push_back(5);
  myVec.push_back(7);
  vector<int>::iterator it;
  it = myVec.begin();
  while (  it != myVec.end() ){
    cout << (*it);
    it++;
  }
}
```

STL: Standard Template Library

Warning: using iterators for changing vector

- It could be dangerous to **use iterator to traverse a vector when we have performed insertion/deletion**.
- Safe solution: **Reinitialize iterators** of a vector whenever its size has been changed.

```
// Guess what is the output?
int main ()
{
  vector<int> v{1,2};
  v.push_back(3);
  v.push_back(4);
  v.push_back(5);
  vector<int>::iterator b = v.begin();
  vector<int>::iterator e = v.end();
  for (int i = 6; i < 100; i++) { v.push_back(i); }
  while (b != e) {
    cout << *b++ << endl;
  }
}
```

Standard Template Library

How to use STL? No need to recite all of them!

- Remember the basic provided libraries (such as size, etc)
- Check http://www.cplusplus.com/reference/stl/ for more details if needed.

STL: Standard Template Library

Some more topics

- More STL examples, such as `map, set, etc.`
- More STL algorithms, such as `find(), sort(), etc.`

STL

Iterator invalidation

UCLA Samueli Computer Science

Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		
Associative containers	`set` `multiset` `map` `multimap`	Yes		Yes, except erased element(s)		
Unordered associative containers	`unordered_set` `unordered_multiset`	No	Yes	N/A		Insertion caused rehash
	`unordered_map` `unordered_multimap`	Yes		Yes, except erased element(s)		No rehash

STL Table list

Member function table

UCLA Samueli Computer Science

		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Header		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	data	data	data														
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity								bucket_count	bucket_count	bucket_count	bucket_count			
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	insert_or_assign								insert_or_assign				insert_or_assign				
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	try_emplace								try_emplace				try_emplace				
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	pop
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
	merge				merge	merge	merge	merge	merge	merge	merge	merge	merge	merge			
	extract						extract	extract	extract	extract	extract	extract	extract	extract			
List operations	splice				splice_after	splice											
	remove				remove	remove											
	remove_if				remove_if	remove_if											
	reverse				reverse	reverse											
	unique				unique	unique											
	sort				sort	sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	contains						contains	contains	contains	contains	contains	contains	contains	contains			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

Group Exercises: Worksheet

UCLA Samueli Computer Science

- Exercise problems from **Worksheet #** (see “LA worksheet” tab in CS32 website). Answers will be posted next week.
- Questions for today: 1, 2, 3, 5, (6 if we have time :))