C++ 2

- Review
- Example of a C++ class with reference variables
- Inheritance example
  - Use of base class constructor
  - Overloaded functions
  - Global functions
  - Operator overloading
- Object creation - static versus dynamic
- Virtual functions and their use
- Abstract classes
- Visibility

C++ Review

- Data abstraction
  - Control access to implementation
  - Specify interface
  - Identify and maintain invariants
- OOPLs
  - Encapsulation plus code reuse through inheritance
- Example: stacks in C versus stacks in C++

C++ Review

- Separation of member function specification from implementation
- Overloaded operators
- Constructors and destructors
  - Called implicitly
- Inheritance
  - Base class (parent), derived class (child)
- Use pointers to refer to dynamically created objects, not references
  - Have much the same usage rules as in C

Subtyping and Derived Classes

- A derived class inherits some behavior from its base class

```
class V : public A
```

- A subtype value can be used anywhere its supertype value can be used.
- If a public derived class inherits all members from its base class, without overriding any, then it has a subtyping relation with its base class.

UNIX diff (C++, Java)

- Pointer to objects
- Multiple inheritance
- Objects can be created statically or dynamically
- Virtual functions dynamically bound
- Operator overloading
- Class implementation can be defined separately from class specification
- C syntax
- Allows global procedure definition

- References to objects (more restricted than pointers)
- Single inheritance
- All objects created dynamically
- All functions dynamically bound
- No operator overloading
- Class implementation with class specification
- C-like syntax
- All procedures and functions associated with a class

Example

```
class X //this class to demonstrate automatic // calls
  constructor and destructor
{public:
     X() { printf("Constructor called.\n"); }
     ~X() { printf("Destructor called.\n"); }
     void f(int i=0)
      {  cout << "i is " << i << "\n"; }
} x2;   //this declares an object x of type X
main()
{   if (1) { X x1;  x1.f(); }
}
```

x1

x2

Output

```
Constructor called
Constructor called
i is 0
Destructor called
Destructor called.
```

C++ Class Example

```
#include <stdio.h>
#include <stream.h>

  class vector
{     int sz;
      int *v;
 public: vector(int);
       ~vector()  {delete v;}
       int size()  {return sz;}
       int& elem(int i){return v[i];}
       int& operator[](int);
};
```

class interface, missing some member implementations

constructor

destructor

member functions

overloaded subscript operator

C++ Class Example, cont.

```
void error(char *s)
{ cout << s << "\n";
  exit(1);
}
vector::vector(int i)
{ if (i <= 0) error("bad vector size");
  sz = i;
  v = new int[i];
}
int& vector::operator[](int i)
{ if (i < 0 || i >= sz) error("index out of bounds");
  return v[i];
}
```

procedure

constructor code

overloaded operator code

References in C++

- **References cannot be null; a reference designates a particular object**
- **Once a reference is given a value, it cannot be changed to point to a different object**
- **There is no explicit way to access the value at the memory address associated with a reference; you use it like you would use a variable.**
- **Used to simulate call by reference in C++ and to return values from functions**

Inheritance Example

Continues same C++ program

```
class vec : public vector
{ int high, low; //private members of class
 public: vec(int, int);
      int& elem(int i){return vector::elem(i - low);}
      int& operator[](int);
};
vec::vec(int i, int j) : vector(j - i + 1)
{  if (j < i) j = i;
  low = i;
  high = j;
}
int& vec::operator[](int i)
{ if (i < low || i > high)
      error("index out of bounds for vec");
  return elem(i);}
```

Explicit call to base class constructor with args

Inheritance Example, cont.

```
class newvec : public vector
{public:
  newvec(int s) : vector (s) {}
  newvec(newvec&);
  ~newvec() {}
  void operator=(newvec&);
  newvec operator+(newvec&);
};
//define a second constructor to create a newvec
//initialized to be a copy of another newvec
newvec::newvec(newvec& a) : vector(a.size())
{
  for (int i = 0; i < a.size(); i++)
     elem(i) = a.elem(i);
}
```

overloading

Inheritance Example, cont.

```
//define an assignment operator on newvecs
void newvec::operator=(newvec& a)
{ int i;
  if (size() != a.size())
      error("bad vector size for =");
  for (i = 0; i < size(); i++) elem(i) = a.elem(i);
}
newvec newvec::operator+(newvec& b)
{ int sz = size(); int i;
  if (sz != b.size()) error("bad vector sizes for +");
  newvec sum(sz);
  for (i = 0; i < sz; i++)
       sum.elem(i) = this->elem(i) + b.elem(i);
  return sum;
}
```

Inheritance Example, cont.

```
main()
{ int i;
  newvec v1(10);
  newvec v2(20);
  for (i = 0; i < 10; i++) v1[i] = i;
  for (i = 0; i < 20; i++) v2[i] = i-1;
  newvec v3(v1);
  newvec v4(v2);
  newvec v5 = v1 + v3;//shows overloaded = and +
  for (i = 0; i < 10; i++) cout << v5.elem(i) << " ";
  cout << "\n";
}
//run by typing: g++ bgrvec.cc followed by
//> a.out
//0 2 4 6 8 10 12 14 16 18
```


Object Creation

Dynamic: `vector *v = new vector(10);`
`vec *w = new vec(10);`

Static: `vector a(10);`
`vec b(10);`

Differences - object created with a pointer in C++ stored on heap; otherwise, they are stored on stack

BE CAREFUL: `v = w;` `a = b;` **are legal but with different effects.** `v=w` **makes** `v` **point to a** `vec` **object;** `a=b` **truncates the** `vec` **object data fields which don't belong to** `vector` **to fit in the stack storage!**

Always create objects using pointers in C++

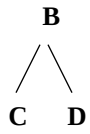
Virtual Functions

- **Dynamic binding only happens in C++ with functions declared to be virtual.**
  - **Define** `void vector::printv(){..}`
    `void vec::printv(){..}`
  - **Declare** `vector *v,vector *vv, vec *w`
  - **Initialize** `v,vv,w` **to point to objects of their declared types**
  - **Execute** `vv=w; v->printv();vv->printv();`
  - **Both calls will execute** `vector::printv()`
  - **Choice of function based on declared type of pointer**

Virtual Example


```
#include <stdio.h>
#include <stream.h> //example inspired by pohl book
class B {
public: virtual void print_i() {cout << 1 <<
            " inside B\n";}
};
class D : public B {
public: void print_i() { cout << 2 << " inside D\n";}
};
main()
{       B *pb = new B(); D *pd = new D();B *p;
        pb -> print_i();   //should print 1 inside B
        pd -> print_i();   //should print 2 inside D
        pb  = pd;
        pb -> print_i();   //should print 2 inside D
}
```

Abstract Classes




```
#include <stdio.h>
#include <stream.h> //example inspired by pohl book
class B {
public:  virtual void print_i() =0;
};

class D : public B {
public:  void print_i() { cout << 2 << " inside D\n";}
};
class C : public B {
public:
   void print_i() { cout << 3 << " inside C\n";}
};
```

Cannot create B objects because B is an abstract class; note missing Implementation for B::print_i()

Abstract Classes

```
main()
{ C *pc = new C(); D *pd = new D();B *pb;
  pc -> print_i();   //should print 3 inside C
  pd -> print_i();   //should print 2 inside D
  pb  = pd;
  pb -> print_i();   //should print 2 inside D
  pb = pc;
  pb -> print_i();  //should print 3 inside C
}
//40 scherzo!programs> ./a.out
//3 inside C
//2 inside D
//2 inside D
//3 inside C
```

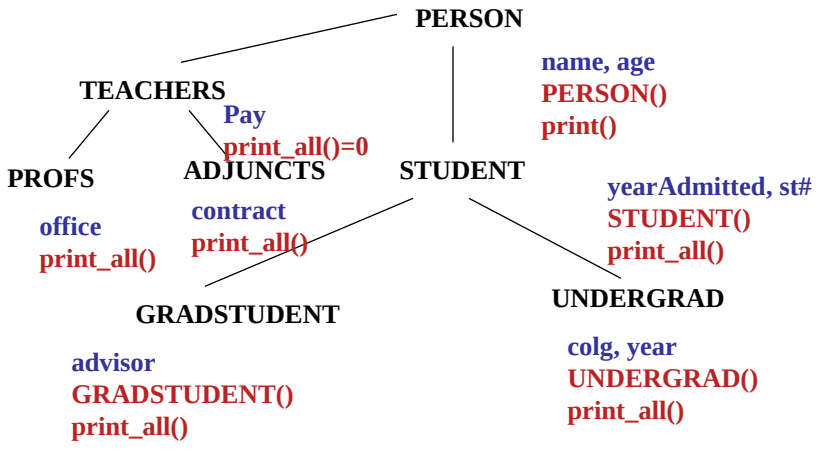
How to design classes?

Cf Prof Borgida

```
class PERSON{
str name;
int age;
PERSON(str n,int a);
void print();
};
class STUDENT: public PERSON{
int yearAdmitted;
int std#;
STUDENT(str,int,int,int)
void print_all();
};
```

```
class GRAD : public STUDENT{
PROF* advisor;
GRAD(...)
void print_all();
};
class UNDERGRAD:public
STUDENT{
COLLEGE* colg;
enum{1,2,3,4} year;
UNDERGRAD( ...)
void print_all() ;
};
```


Example



PROBLEM: grad students who teach courses!
(more later on this)

Visibility in C++

- For members and member functions
  - Private - only visible within that class
  - Protected - only visible within that class and its derived classes
  - Public - visible to everyone
- Always use public inheritance! Private inheritance exists but is difficult to use.

```
class D : public B
```


Inheritance in C++

- Use abstract class to create consistent interfaces for subclasses
- Subtype polymorphism if never redefine inherited functions
- Code sharing and reuse
- Automatic propagation of changes to subclasses
- C++ has no equivalent to Java *final* which prevents subclass extension
- Benefits to class designer and users

Discussion

- Assume we have a `Dequeue` class with `append(), remove(), insertFront(), removeRear()`
- And we want to define `Queue` as a subclass of `Dequeue`

```
Q: private DQ;
```

- Private inheritance allows use of `DQ` protected functions within the defn of class `Q`, but does not allow users of `Q` to apply `DQ` functions to `Q` objects.

- Contrast with: given a `Queue` class, extend it to a `Dequeue` subclass by adding `insertFront(), removeRear(),` `DQ : public Q`