

Subclass hierarchies (C++ not'n)

Suppose you are developing an information system for a university: graduates, undergrads, profs, courses, ...

Abstract out commonalities: {Grad,Undergrad}=>Student,
{TA,Prof,Secretary}=>Employee, {Student,Employee}=>Person

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

OO Borgida'01

1

Constructing & Assigning

Constructors

For *base* class: .

```
PERSON::PERSON(str s,int a) {name=s;age=a;}
```

For *derived* class: .

```
STUDENT::STUDENT(str s,int a,int yr, int snum)
    : PERSON(s,a) { yearAdmitted=yr, std#=snum); }
```

Use:

DYNAMIC

```
PERSON* annPrs = new PERSON("ann",3);
STUDENT* joeStd = new STUDENT("joe",18,1992,12345678);
GRAD* aliGrd = new GRAD( .... );
```

Also possible: STATIC

```
PERSON p("ann",3,"");
STUDENT s("joe",18,1992,12345678);
```

BEWARE!!!

Person *x = joeStd; //preserves info that x has student because it stores only a pointer but

Person y= s; //truncates student to a person because of y was allocated only enough space for a person record.

Lesson: Stick to using pointers to objects in C++

OO Borgida'01

2

Inheritance and Over-riding

- Suppose we want to print personal information

```
class PERSON {  
    void print();  
}
```

`<--Declaration`
(in univ.h file)

plus

```
void PERSON::print()  
    {cout << name << age;}
```

`<-Implement'n`
(in univ.cc file)

Use:

```
annPrs->print(); // prints person info  
joeStd->print(); // also prints only person info because of inheritance
```

`<--Use`
(in test.cc)

- Suppose we want to print more info/different form in each subclass:

```
void STUDENT::print_all()  
    {cout<<"Student"<<stud#; PERSON::print(); }  
void GRAD::print_all()  
    { cout<<"Graduate student"<< dept;  
      STUDENT::print_all(); }
```

Use:

```
annStd->print_all(); //prints student info  
aliGrd->print_all(); //prints grad student info  
Student* s = aliGrd; s->print_all(); //Again only  
prints student info??!
```

Lesson: Choice of overloaded function based on **variable declaration**.

Subclass hierarchies Java

```
public class PERSON{  
    public string      name;  
    public int         age;  
    public PERSON(str s,int a)  
        {name=s; age=a;}  
    public void        print();      };  
  
public class STUDENT extends PERSON{  
    public int         std#;  
    ...  
    STUDENT (str s,int a,int yr, int snum)  
        {super(s,a); yearAdmitted=yr; std#=snum; };  
    public void        print_all()  
        {this.print(); "/" +std#+...};  
};  
public class GRAD extends STUDENT{  
    public DEPT*      dept;  
    ...  
    public void        print_all();  
};  
  
public class UNERGRAD extends STUDENT{  
    ...  
};
```

Java use:

```
PERSON p = new PERSON("ann",3);
STUDENT s = new STUDENT("joe",19,1992,12345678);
GRAD g = new GRAD(... );
p.print(); //personal info
s.print(); //personal info
s.print_all(); //student info
g.print_all(); //grad student info
s=g;
s.print_all(); // still GRAD.print_all called
```

This is sometimes called "*dynamic binding/dispatching*". Requires run-time determination of type!

Why useful, even in C++?

```
STUDENT* cs431 [25]; //enrolment for cs431
cs431[0] = new UNDEGRAD(...);
cs431[1] = new GRAD(...);
cs431[2] = new UNDERGRAD(...);
for(j=1; j<size; ++j){
    cs431[j]->print_all(); };
```

Still want exact information printed about each student.

Dynamic binding in C++

```
class PERSON{ ...
    public:
        virtual void print_all() =0;
    ...
class STUDENT: public PERSON{...
    virtual void print_all() //handles printing of student instances
    that are not in subclasses where print_all is over-written
    ...};
class GRAD: public STUDENT{ ...
    void print_all() // handles grads
```

- Virtual functions are the only ones that are dynamically bound.
- By having the =0; after the specification of a function, you indicate that there is no implementation for it on that class, so you cannot create instances of it: **Abstract class**, only good for inheritance/interfac spec. e.g., PERSON.
- An abstract function specifacaton (with a =0;) promises an implementation for every leaf subclass below it, though it may be inherited from intermediate class. e.g., no absolute requirement for GRAD::print_all, if STUDENT::print_all is present.
- If you omit printa_all()=0; from PERSON, it is no longer an abstract class, so you can create instance of it; also, as long as STUDENT::print_all is virtual, there is still dynamic dispatch; good for 'default' implementation.
- No need to repeat the keyword **virtual** after topmost occurence.

C++ , Java visibility etc.

- For members (data or function)
 - PRIVATE:** only visible inside that class' defs
 - PROTECTED:** only visible in that class and all its subclasses
 - PUBLIC:** visible to everyone

- For derivation/inheritance

```
class A : public B{ ...}
```

means that all public functions of B are also public on A
(forget about A : private B, etc. for now)

```
class STUDENT : public PERSON{
    protected:
        int    yearAdmitted;
        int    std#;
    public:
        void  print_all()          + access functions to std#,
            {cout <<"Student.... ";};
        ...      };
```

- In C++, giving function body in class declaration (like for print_all on STUDENT above) causes **inline** expansion; so **cannot** be used with **recursive** functions!
- C++ has no equivalent of Java **final**, to stop further refinement of a class.

Uses of subclass hierarchy

1. ensure consistent interfaces for subclasses (**PERSON**)
2. *subtype polymorphism*: anything applicable to a C (e.g., **PERSON::print**) applicable to objects of its subtype.
3. abbreviation: no need to repeat things
4. code reuse
5. automatic propagation of changes to subclasses

NOTE:

- these ideas are related
- some benefit user of class, some benefit programmer of class

Inheritance

"Means by which new components are constructed from old ones so that changes to the old ones affect the new ones (sometime)"

e.g. given a Deque, program a Queue

```
class Deque with append, remove, insertFront, removeRear;  
want class Queue with append, remove;
```

```
Queue subclassOf Deque: if you already have it, reuse;  
"Q: private D"
```

vs. given Queue, program a Dequeue

```
Deque ISA Queue: extend, subtype D:public Q
```

Inheritance is for the benefit of the *implementors!!!*

- interface inheritance: inherit constraints on what class instances can do;
e.g., container classes in OOPL like Smalltalk
- code & implementation inheritance;

Questions:

- can you block inheritance? (no **age** for some)
- can you refine inherited things? (**age** range)
- can you over-ride? (foreign **address**)