Remainder of class

- Automatic Vectorizer project due on Saturday, December 17, 11:55pm.
- Final exam: Tuesday, December 20, in class, noon - 3:00pm. Closed books/notes; cummulative: more than 50% new material, i.e., material not covered by midterm exam; final counts 35% of your grade. **ANY CONFLICT**?
- Class evaluation cite open until Thursday, December 15 at https://sakai.rutgers.edu. Thanks!
- All sample solutions will be available on class web site by Friday, December 11.
- Homework problem set 6 will be posted by tomorrow. All sample solutions will be available by Wednesday, December 14.

Midterm Exam

Average: 70.8

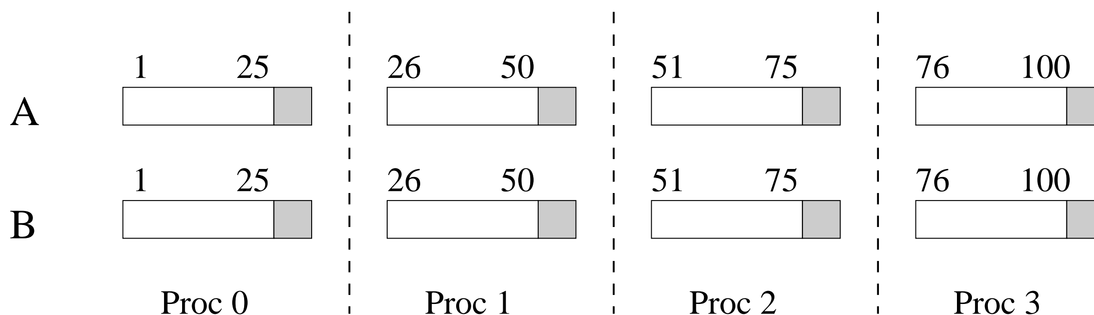
Median: 75.0

Grade Distribution:
0..9 = 0
9..10 = 0
11..20 = 1
21..30 = 0
31..40 = 0
41..50 = 4
51..60 = 1
61..70 = 3
71..80 = 4
81..00 = 4
91..100 =4

Compiling for Distributed–Memory Multiprocessors

Example

```
real A(100), B(100)
do i = 1, 99
  A(i) = A(i+1) + B(i+1)
enddo
```




```
real A(26), B(26) // 1 element overlap to the right
if my$proc == 3 then my$up=24 else my$up=25 endif
if my$proc > 0 then
  send( A(1), my$proc - 1) //send to left
  send( B(1), my$proc - 1) //send to left
endif
if my$proc < 3 then
  receive( A(26), my$proc + 1) //receive from right
  receive( B(26), my$proc + 1) //receive from right
endif
do i = 1, my$up
  A(i) = A(i+1) + B(i+1)
enddo
```

Historic Progression of Data Types

1. "User-defined" types
   - Can define arbitrary operations
   - Transparent: whole structure of type visible
   - Cannot control access to structure
2. Abstract data types
   - Encapsulation, information hiding
   - Opaque: hides data representation
   - Access restricted to well-defined interface functions
3. Object-orientation
   - Inheritance
   - Code re-use
   - Polymorphic behavior

Data Abstraction

Specification rather than implementation:

- Define *behavior* of data type (through interface functions)
- Hide *implementation* of data type
  $\Rightarrow$ hide details irrelevant to the use of the data type

User-defined Stack Type in C

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

typedef int bool;
typedef int elt;
#define MAX  20
#define EMPTY  -1

typedef struct {elt s[MAX];int top;} stack;

stack * create() {
 stack * newstack = (stack *) malloc(sizeof(stack));
 newstack->top = EMPTY;
 return (newstack);
}
void push(stack* stk,elt data)
 {stk->s[++stk->top]=data;}
void pop(stack* stk) {stk->top--;}
elt peek(stack* stk) {return (stk->s[stk->top]);}

int main() { /**** using the stack ****/
stack *x;
    x = create();
    push(x,2); push(x,3);
    printf("%d\n", peek(x));
    return 0;
}
```

Problems with Data Types

- Implementation of the type can be seen
  (E.g., the array inside the stack)

- "Users" of the type can change its value arbitrarily
  E.g., `x->s[5] = 10;`
  - doesn't respect push/top access pattern
  - doesn't respect `elt` typedef

- "Users" of the data type can write operations that create inconsistent states
  (E.g., adding an entry without changing the top index)

- "Users" cannot extend the set of operations in a reliably safe fashion

Abstract Data Types (ADTs)

User *may only* manipulate objects of the type through use of provided functions *without* knowing internal representation

- Encapsulation: may only use provided functions
- Information hiding: cannot see internal representation

Advantages of Abstract Data Types

- Easier to use: as if only type names and function headers were visible
- Safety through access control
  - User can't make inconsistent states
  - User can't make assumptions about data representation
- Designer of ADT can modify implementation without affecting users
- Encourages modularity in programs, facilitating larger, more complex systems

Designing an Abstract Data Type

1. Specify interface
2. Identify and maintain invariants

Example: bounded stack

Interface:

- Stack of some kind of element `elt`
- `create` makes a new, empty stack
- `push` pushes new element on stack; cannot push onto full stack
- `pop` removes an element; cannot pop from empty stack
- `peek` returns the top element on stack
- `is_empty` determines if the stack is empty
- `is_full` determines if the stack is empty

Invariants

- peek(push(S, e)) = e
- pop(push(S, e)) = S
- is_empty(create())
- not is_empty(push(S, e))
- not is_full(pop(S))

Object-Oriented Programming (OOP)

- Similar to abstract data types
  - Allows users to build new types
  - Encapsulation
  - Information hiding
- Allows code sharing or reuse between related types: *inheritance*
- Theory of object-oriented programming is *not* finished
  - "First" object oriented language: Simula'67
  - Different languages work differently (compare C++ and Java)
  - Syntax can be complicated
  - Semantics may be ill-defined (especially consider multiple inheritance)

C++

Classes

- Describe abstract data types
- Encapsulate data and define operations on it

Class definitions

- Define *data members* (variables)
- Define *member functions* (or methods)
- Access restriction: `public` and `private`

Objects are instances of classes

ADT Stack in C++

```
// statically allocated stack ADT

#define  MAX  20 // default stack size

typedef int elt ;
typedef int boolean;

class stack{             // encapsulated data type
 private:
    elt s[MAX];             //
    int top;                // hidden data representation
    const int EMPTY = -1;   //

 public:
    stack() { top = EMPTY; }   // constructor => create()

    boolean isempty() { return (top == EMPTY); }

    boolean isfull() { return (top == MAX - 1); }

    void push(elt data)
         {  if (!isfull()) s[++top]=data;
            else cout<<" stack is full; cannot push\n";  }

    void pop()
         {  if (!isempty()) top--;
            else cout<<" stack is empty; cannot pop\n";  }

    elt peek()
        {  if (!isempty()) return s[top];
           else cout<<" stack is empty; cannot peek\n";  }
};
```

Constructors and Destructors

- Define a *constructor*, called to initialize objects (object instances) of the class (constructors may take arguments)
- May define a *destructor*, called to free heap memory used by objects
- Constructors and destructors for class `X`:
  constructor: `X(...)`
  destructor: `~X()`
- Constructor called implicitly when object is allocated (created)
- Destructor called implicitly when control leaves scope of object (end of object's lifetime).

ADT Stack in C++

```
// dynamically allocated stack ADT

typedef struct cell {
    elt info;
    struct cell* link; } CellType;

class stack{
 private:
   CellType * top;
 public:
   stack() {top=NULL;}

   ~stack() { while (top != NULL) pop(); }

   boolean isempty() { return (top == NULL); }

   boolean isfull() { return  0; }

   void push(elt data)
     { CellType* add = new CellType;
       add->info = data;
       add->link = top;
       top = add;  }

   void pop() { CellType* tmp;
                tmp = top;
                top = top->link;      // no error check
                delete tmp; }

   elt peek() { return (top->info); } // no error check
};
```

Functions (and Operators) in C++

- Function body can be defined outside class definition.
  Still need function interface (signature) declaration in class definition.
  (Somewhat similar idea: foo.h and foo.c file)
- Functions can have optional parameters.
- Functions and operators can be overloaded:
  - Have different implementations on different types
  - Must be distinguishable by type signature
  - Like `+` on `int` and `float`

ADT Stack in C++

```
#include <iostream.h>
class stack{
 private:
    CellType * top;
 public:
    stack();
    ~stack();
    boolean isempty();
    boolean isfull();
    void push(elt data);
    void pop();
    elt peek();
};
stack::stack() { top=NULL; }
stack::~stack() { while (top != NULL) pop(); }

boolean stack::isempty() { return (top == NULL); }
boolean stack::isfull() { return  0; }

void stack::push(elt data)
     { CellType* add = new CellType;
       add->info = data;
       add->link = top;
       top = add;  }

void stack::pop() { CellType* tmp;
                    tmp = top;
                    top = top->link;     // no error check
                    delete tmp; }

elt stack::peek() { return (top->info); }// no error check
```

ADT Stack in C++ (Cont.)

```
int main() { /**** using the stack ****/
stack *x = new stack();
x->push(2);
x->push(3);
cout << x->peek());
return 0;
```

Efficiency in OO Code

Encapsulation and information hiding imply many function calls.

Function calls have high run-time overhead.

Efficient compilers *inline* calls where possible:

- Function code is expanded at point of call
- Like a macro

  $\Rightarrow$ larger, unreadable machine code

  $\Rightarrow$ faster machine code

C++ vs. Java (incomplete!)

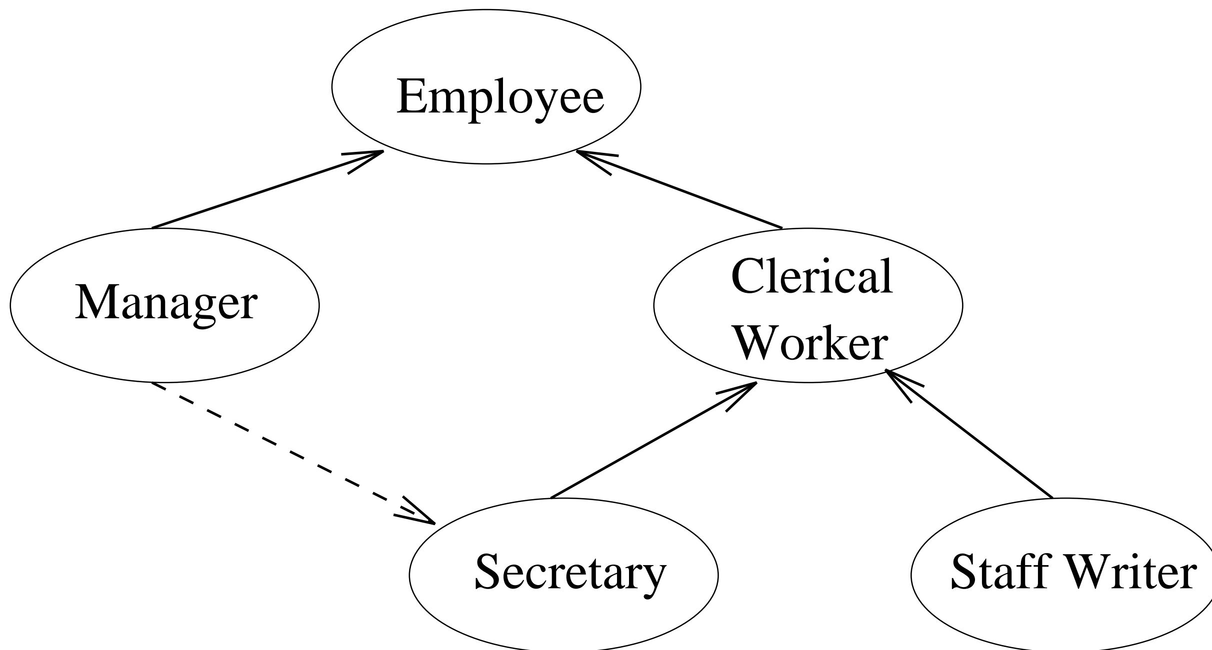
C++	Java
Pointers to objects	**References** to objects (more restricted than pointers)
Multiple inheritance	**Single** inheritance
Objects can be created **statically** and **dynamically**	All objects created **dynamically**
Explicit memory management	**Implicit** memory management (garbage collection)
Virtual functions dynamically bound	All functions dynamically bound
Operator overloading	No operator overloading
Allows global procedure definition e.g.: `main()`	All procedures and functions associated with a class

Relationships among entities

There are a number of important relationships among entities that are useful in object–oriented design, e.g.:

- **is–a** — An entity type T1 is in the *is–a* relationship to another entity type T2 if every entity of type T1 is a member of type T2. In the solution domain, this is represented as a relationship between classes implemented using *(public) inheritance* (T2 corresponds to *base class* or *superclass*, and T1 corresponds to *derived class* or *subclass*).
- **has–a** — An entity e_1 is in the *has–a* relationship with entity e_2, if e_2 is part of e_1 or e_1 uses e_2 for implementation. There are two "types" of the has–a relationship: class level (complete containment) or instance level (share instance via reference/pointer).
- **uses–a** — Occurs when one class instance takes another class instance as a parameter. For example, a manager might use a particular company facility. In this case, facility is not a manager, nor it is owned by manager.

Example: "Is–a" and "has–a" relationships



- - - - -> has a

———> is a

Inheritance in Object-Oriented Languages

Subtypes:

- A subtype S of type T:
  any operation that can apply to object t of type T
  can apply to object s of type S.
- Any object s of type S can be used in any context
  that an object of type T can.

Inheritance:

- Provides a means of subtyping and sharing code.
- Allows redefinition of operations.
- Terminology is base classes
  and derived classes.

C++ Derived Classes

- Inherit (data and function) members from base class.
- For now: We use **public** inheritance.
- May have its own constructors and/or destructors.
- Its own constructors/destructors may explicitly or implicitly call base class constructors/destructors.

Example: “Is–a” and “has–a” relationships

EmployeeExample.C:

```
#include <stream.h>

class Employee
{ public:
    int ID;
    Employee(int id) {ID = id;} };

class ClericalWorker : public Employee
{ public:
    int group;
    ClericalWorker(int id, int grp) : Employee(id) {group = grp;} };

class Secretary : public ClericalWorker
{ public:
    char *name;
    Secretary(int id, int grp) : ClericalWorker(id, grp) {} };

class StaffWriter : public ClericalWorker
{ public:
    char *name;
    StaffWriter(int id, int grp) : ClericalWorker(id, grp) {} };

class Manager : public Employee
{ public:
    int devision;
    Secretary *assistant;
    Manager(int id, char *nm) : Employee(id)
         {assistant = new Secretary(++id,1);
          assistant->name = nm;} };

main()
{
  Manager Bob(123, "Steve");

  cout << "Manager Bob (ID " << Bob.ID << ") works with "
       << Bob.assistant->name << " (ID " << Bob.assistant->ID << ")\n";
}
```

Inheritance and Constructors

============ classes.h ============

```
class A { public: A(); }
class B { public: B(); }
class C { public: C(); }
class D { public: D(); }

class X {
  private: A a; B b;
  public:  X(int x);
};
class Y : public X {
  private: C c; D d;
  public: Y(int y);
};
```

============ classes.C ============

```
#include "classes.h"
X::X(int x) { ... } // implement constructor
Y::Y(int y) : X(y+1) { ... } // implement constructor
```

============ main.C ============

```
#include "classes.h"
main() {
Y my_y(5); // Constructors called in this order
           // A() B() X(6) C() D() Y(5)
           // Destructors called in reverse order
}
```

Object Creation

```
class A { public: A(int){...} ... };
```

- static or stack allocation:

```
A a(5);
```

- dynamic or heap allocation:

```
A *a = new A(5);
```

- As in C, C++ provides explicit operations to delete storage.

```
delete a;
```

  This will call the object's destructor, or if no such destructor is specified, the object's default destructor.

Question: Why do you ever want to specify your own destructor?

Polymorphism in C++

Polymorphism: Same function works on different types.

C++ supports many flavors of polymorphism: coercion, overloading, subtyping, and parametric or generic (templates) polymorphism.

- Inheritance — permits new abstract data types to be derived from more general types. These types can inherit some or all the properties and behaviors (member data and functions) (→ *subtyping* – statically bound);
- Virtual functions — each derived class can implement the operations defined in the base class *differently*, while retaining a common class interfaces provided by the base class (→ *subtyping* – dynamically bound);
- Operator and function overloading — allow a function to be selected based on the type and number of its arguments (`signature`) (→ *ad–hoc polymorphism*);

Polymorphism in C++

- Templates — provide the ability to define polymorphic codes in which an aggregate type such as a list, can be instantiated for a variety of element types
  (→ *parametric polymorphism*).

Implementing subtyping

```
class cow
{ public:
    void speak() { cout << "MOOOO\n"; }
};
class calf : public cow
{ public:
    void speak() { cout << "moo\n"; }
};
// A function that takes a cow
// (and hence can take a calf) as an arg
void tip(cow *c) { c->speak(); }

main()
{
cow *c = new cow();
calf *d = new calf();

    c->speak();
    d->speak();
    tip(c);
    tip(d);
}
```

MOOOO moo MOOOO moo	OR	MOOOO moo MOOOO MOOOO

Implementing subtyping

For polymorphic dynamic subtyping behaviour, we want output

```
MOOOO
moo
MOOOO
moo
```

I.e., `tip(cow *c)`

- is a function on cows and all subclasses
- behaviour should depend on the argument object

How is it done?

Virtual Functions

- Used for **dynamic binding**: The actual function to call is determined at run time based on the actual type of an object
- Requires pointers, so that the known type (the type of the pointer) is more general than the actual type (the type of the object)
- C++ keyword `virtual`

Virtual Function Example

```
class cow    // Virtual function in superclass
{ public:
    virtual void speak() { cout << "MOOOO\n"; }
};
class calf : public cow
{ public:
    void speak() { cout << "moo\n"; }
};
void tip(cow *c) { c->speak(); }

main()
{
cow *c = new cow();
calf *d = new calf();
                     // OUTPUT:
    c->speak();      // MOOOO
    d->speak();      // moo
    tip(c);          // MOOOO
    tip(d);          // moo
}
```

Static vs Dynamic Binding

Static name binding (non-virtual):

- Resolve names at compile-time
- Can inline functions at compile-time

Dynamic name binding:

- Resolve names at run-time based on object type
- Can't inline functions
- Requires run-time name look-up
  ⇒ slower executables

⇒ area of optimization research for OOPL compilers

Things to note

- Constructors and Destructors are optional in C++. However, good programming practice should use at least a *constructor*, preferably a *destructor* as well as a *copy constructor* and *assignment operator* `operator=`. There are default versions, but they may not do what you want (e.g.: shallow, bitwise copy).
- function signatures in C++ consist of the type of the declared arguments, and do not include the return type. Example: `int foo()` and `void foo()` result in "ERROR: foo declared more than once".
- Constructors can be explicitly called;

Overloading vs. Overriding

Overriding and Overloading

- Subclassing with some methods being redefined with same signature is called **overriding**
  - **virtual functions**: Lookup (i.e., resolution or binding) happens at run-time based only on receiver's (actual referenced object's) run-time type
  - non-virtual function: lexical scoping, done at compile time.
- Subclassing with some methods being redefined with different signatures is called **overloading**
  - In Java, dynamic lookup establishes best match type signature(s) from a set of function signatures for actual arguments' and receiver's type (class)
  - In C++, function with a signature has to exist that matches the (compile -time, i.e., declared) arguments' and receiver's type exactly

Virtual Function and Pointers

```
#include<iostream.h>
class cow
{ public:
    virtual void speak() { cout << "MOOOO\n"; }
};
class calf : public cow
{ public:
    void speak() { cout << "moo\n"; }
};
// With cow instead of cow*, function
// treats calf argument as cow (static, or declared type)
void tip(cow c) { c.speak(); }

main()
{
cow c;
calf d;
                    // OUTPUT:
    c.speak();      // MOOOO
    d.speak();      // moo
    tip(c);         // MOOOO
    tip(d);         // MOOOO
}
```

More Virtual Functions

```
#include<iostream.h>
class cow
{ public:
    virtual void speak() { cout << "MOOOO\n"; } };
class blackCow : public cow
{ public:
    virtual void speak() { cout << "BOOO\n"; } };
class calf : public blackCow
{ public:
    void speakout() { cout << "moo\n"; } };

void tip(cow *c) { c->speak(); }

main()
{
cow *c = new cow;
blackCow *b = new blackCow;
calf *d = new calf;
                     // OUTPUT:
    c->speak();      // MOOOO
    b->speak();      // BOOO
    d->speakout();   // moo
    tip(c);          // MOOOO
    tip(b);          // BOOO
    tip(d);          // BOOO
}
// Just want to show that the search is dynamic along
// the class hierarchy
```

And More Virtual Functions

```
#include<iostream.h>
class cow
{ public:
    void speak() { cout << "MOOOO\n"; } };
class blackCow : public cow
{ public:
    virtual void speak() { cout << "BOOO\n"; } };
class calf : public blackCow
{ public:
    void speak() { cout << "moo\n"; } };
void tip(cow *c) { c->speak(); }
void top(blackCow *b) { b->speak(); }

main()
{
cow *c = new cow;
blackCow *b = new blackCow;
calf *d = new calf;  // OUTPUT:
    c->speak();      // MOOOO
    b->speak();      // BOOO
    d->speak();      // moo
    tip(c);          // MOOOO
    tip(b);          // MOOOO
    tip(d);          // MOOOO
    top(b);          // BOOO
    top(d);          // moo
}
// Once the base class has a virtual function, the function will
// be considered virtual in all of its subclasses. Once virtual,
// always virtual from that point on in the hierarchy.
// Cannot switch back from virtual to non-virtual
```

More on Inheritance, Members, and Constructors

```
#include <iostream.h>
class cow
{ public:
    char name;  int num;
    virtual void speak()
      { cout << "MOOOO\n"; }
    char getname() { return name; }
    virtual int getnum() { return num; }
    cow() { name = 'C'; num = 1; }
};

class calf : public cow
{ public:
    int num;
    void speak() { cout << "moo\n"; }
    int getnum() { return num; }
    calf() { name = 'c'; num = 2; }
};


void tip(cow *c) { c->speak();
                   cout<<c->name<<"\n";
                   cout<<c->num<<"\n"; }
void top(calf *c) { c->speak();
                    cout<<c->name<<"\n";
                    cout<<c->num<<"\n"; }
void tup(cow *c) { c->speak();
                   cout<<c->getname()<<"\n";
                   cout<<c->getnum()<<"\n"; }
```

Example Continued...

```
main()
{
cow *c = new cow();
calf *d = new calf();

    c->speak();
    cout<<c->num<<"\n";
    d->speak();
    cout<<d->num<<"\n";
    tip(c);
//    top(c); illegal
    tup(c);
    tip(d);
    top(d);
    tup(d);
}
```

Example Continued...

OUTPUT:

MOOOO

1

moo

2

MOOOO

C

1

MOOOO

C

1

moo

c

1

moo

c

2

moo

c

2

Review: Virtual Functions

- Used for **dynamic binding**: The actual function to call is determined at run time based on the actual type of an object
- Requires pointers, so that the known type (the type of the pointer) is more general than the actual type (the type of the object)
- C++ keyword `virtual`
- Once a function is declared `virtual`, it will remain virtual for all derived classes.

Pure Virtual Functions and Abstract Classes

Pure virtual function:

- Virtual function with no function definition
  E.g., `virtual int f()=0;`

- Why?
  - When unable to define sensible version of function in base class
  - Ensures every derived class implements function

Pure Virtual Functions and Abstract Classes

Abstract class

- Class containing one or more pure virtual functions
- **Cannot** create objects of abstract class
- Can create pointers to abstract class
  $\Rightarrow$ still supports run-time polymorphic behaviour

Example:

```
#include <iostream.h>
class A {
public: virtual void Print() =0;
};
class B : public A {
public: void Print() { cout << " B \n"; }
};
class C : public A {
public: void Print() {cout << " C \n"; }
};
main() {
  // A a1;             -> ILLEGAL: compile-time error
  // A *a2 = new A();  -> ILLEGAL: compile-time error
  A *a;
  B *b = new B();
  C *c = new C();
  a = b; a->Print();
  a = c; a->Print();
}
```

Overloading and Templates

- Overloading — different implementations for different signatures
- Templates — parameterized types

C++ supports parametric polymorphism in the form of **templates**.

Example:

```
template<class T> class Stack {
  public:
    Stack(int n) {size=n; elements=new T[size];top=0;}
   ~Stack()      { ... }
    void push(T a) {top++; elements[top]=a;}
    T    pop()     {top--; return elements[top+1];}
  private:
    int top;
    int size;
    T * elements;
}
```

Stacks of type *int* or *char* can be instantiated as follows:

```
Stack<int>  s(99);
Stack<char> t(100);
```

Controlling Visibility (public inheritance)

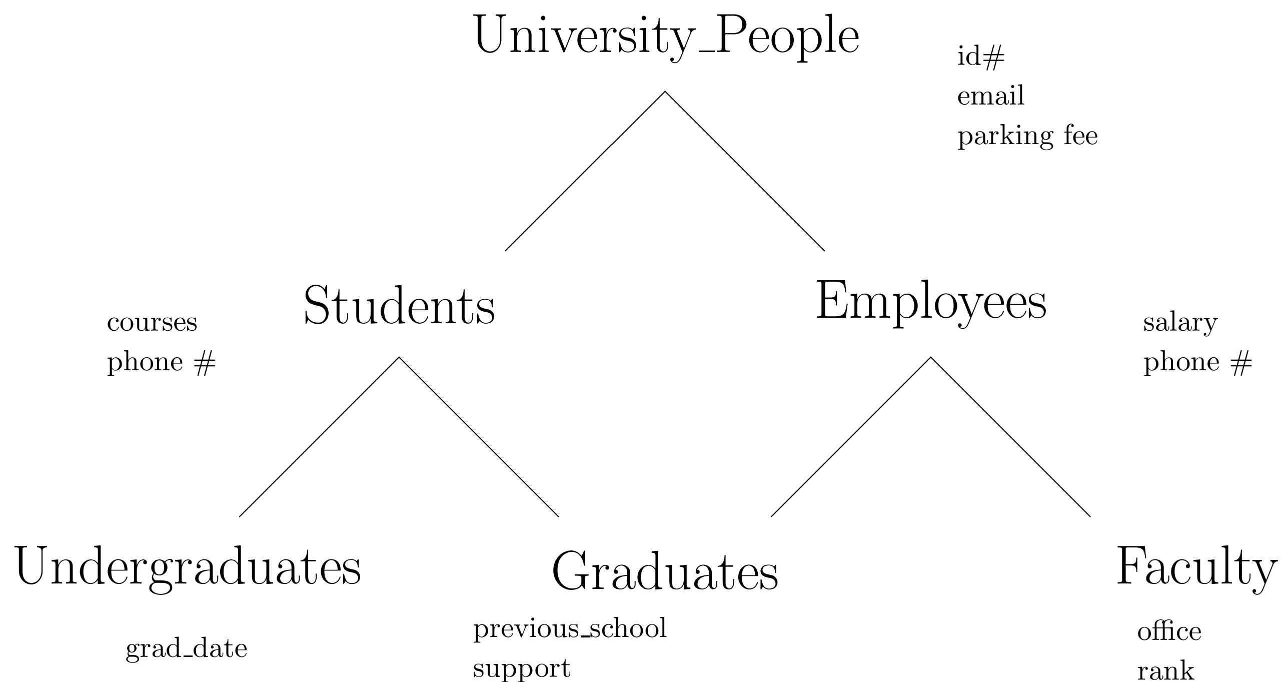
C++: Base classes can control inheritance:

- **private:** Only visible in class member functions (and friends)
- **protected:** Also visible in derived classes through inheritance
- **public:** Visible everywhere
- ⇒No specification**:** defaults to *private*

Multiple Inheritance

What if:

- a class is a subtype of several other classes?
- a class is like several other classes, i.e., exhibits behavior appropriate for several other classes?



Graduate students are both **students** and **employees**!

- More than one direct base class
- Object hierarchy a directed acyclic graph, not a tree

```
class Graduates: public Students, public Employees;
```

- Members with same name in multiple parents → need to disambiguate `Students::phone`)

The “this” pointer

Every member function is (implicitly) passed a pointer named “**this**.”

```
class lion {
private:
  char *name;
public:
  lion()  {
    name = get-string-from-user();
    cout << "Creating lion " << name;
    // Equivalent: cout << "Creating lion " << this->name;
    record(this);
    // Passing a pointer to the whole lion object.
    // Function record could keep track of
  } //     all lions created.
  ~lion()  {
    unrecord(this);
    // Passing a pointer to the whole lion object.
    // Function unrecord could keep track of
  } //     all lions destroyed.

};
```