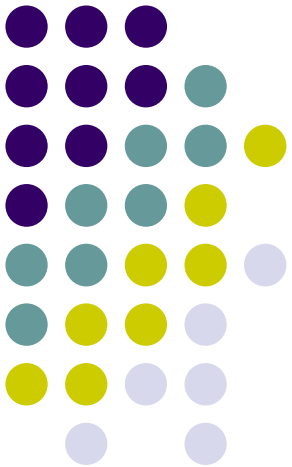
Advanced C++

January 15, 2013

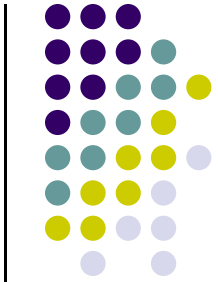
Mike Spertus

mike_spertus@symantec.com

YIM: spertus



Classes



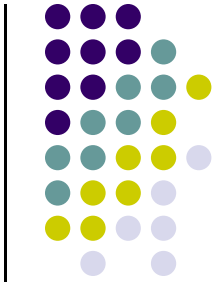
- Consider the following example (p. 61)

```
struct Student_info {
   string name;
   double midterm, final;
   vector<double> homework;
};  // Semicolon is required!
```

- Unlike C, no need for:

```
typedef struct Student_info Student_info;
```

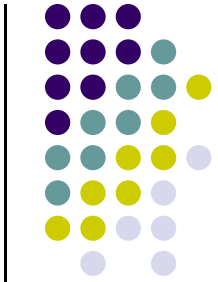
Hey, that's a `struct`, not a `class`!



- That's OK, the only difference between a struct and a class in C++ is different default visibility of members.
- The following is equivalent

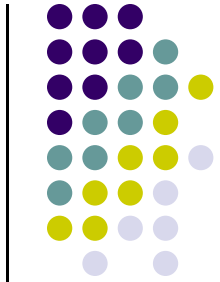
```
class Student_info {
public:
   string name;
   double midterm, final;
   vector<double> homework;
};
```

Visibility of members



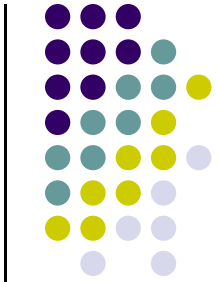
- public members are visible to everyone
- Protected members are visible to subclasses
- Private members are only visible to the class

Visibility (cont)



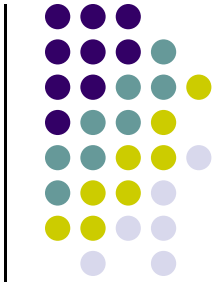
```
class A {
 void f() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // OK
  }public:
  int pub;
protected:
  int prot;
private:
  int priv;
};
class B : public A {
  void g() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // Error
  }
};

void h(A a)
{
    cout << a.pub; // OK
    cout << a.prot; // Error
    cout << a.priv; // Error
}
```

Let’s add methods (p. 157)



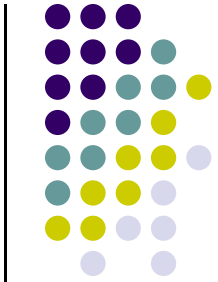
```
Struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  double grade() const;
};
// In .cpp file
double Student_info::grade() const
{
  return (midterm + final + median(homework))/3;
}
```

This is the same as the last slide



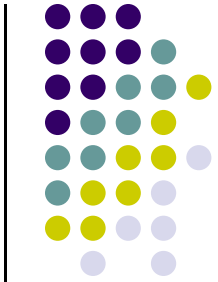
```
Struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  double grade() const {
    return (midterm + final + median(homework))/3;
  } // No semicolon!
}; // Semicolon!
```

Adding multiple grading strategies



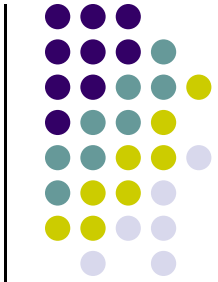
```
Struct Abstract_student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  // Don't define grading strategy here
  virtual double grade() const = 0;
};
```

Here are a couple of strategies



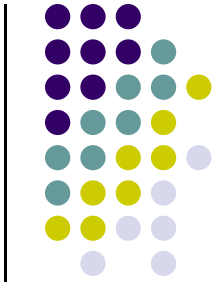
```
struct Balanced_grading: public Abstract_student_info {
  virtual double grade() const {
    return (midterm + final + median(homework))/3;
  }
};


struct Emphasize_final : public Abstract_student_info {
  double grade() const {
    return final; // Ignore the HW and the midterm
  }
};
```

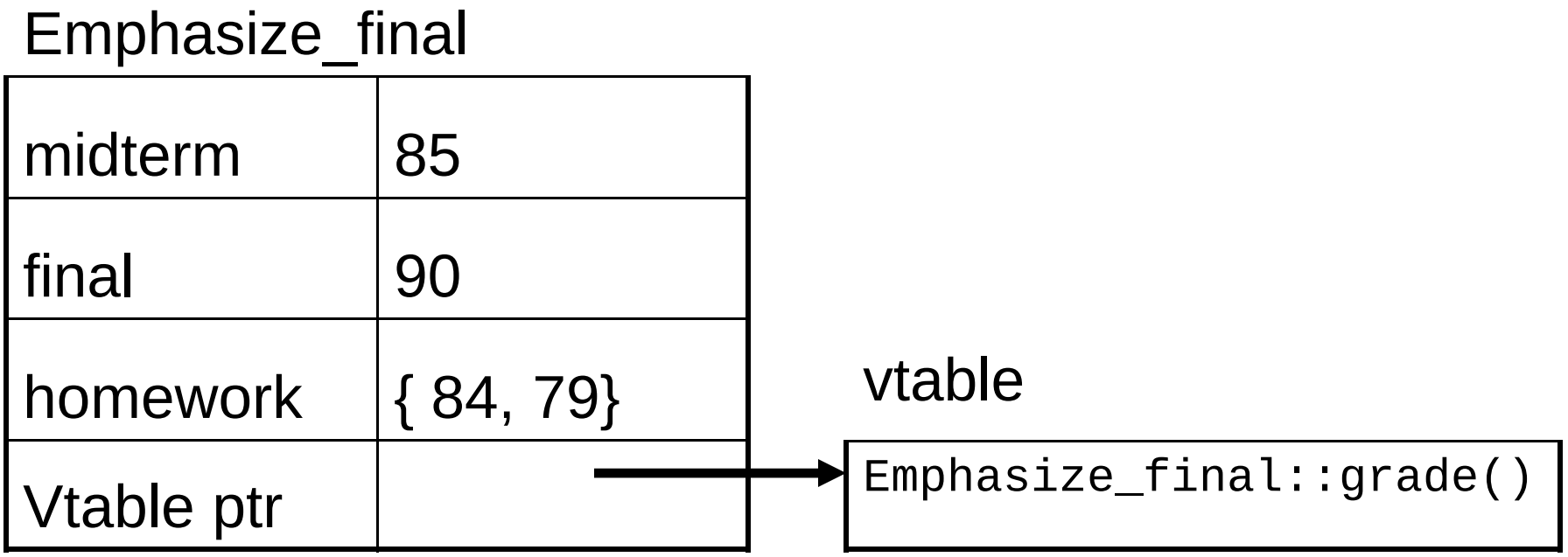
How do we choose which grading strategy to use?



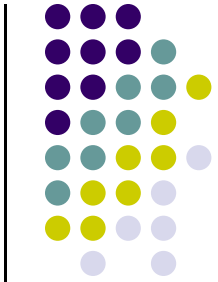
```
int main()
{
  Abstract_student_info *si = new Balanced_grading();
  si->read(cin);
  cout << “Grade is " << si->grade() << endl;
  delete si; // We’ll learn more about new
             // and delete later
  return 0;
}
```

Virtual function implementation



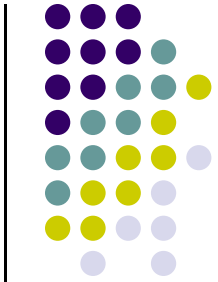


Static vs Dynamic types



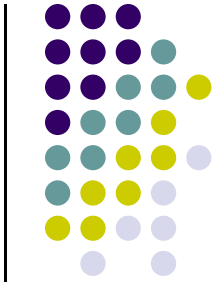
- The static type is the type of the expression
  - Known at compile time
- The dynamic type is the type of the actual object referred to by the expression
  - Known only at run-time
- Static and dynamic type only differ due to inheritance

Static type vs Dynamic type



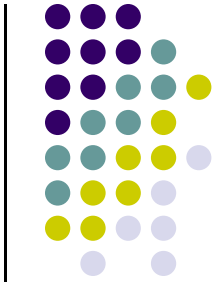
```
int i = 5; // S = int, D = int
Gorilla g; // S = Gorilla, D = Gorilla
Animal *a = new Gorilla // S = An, D = Gor
Animal a2 = *a; // Error!
```

Virtual vs. non-virtual method



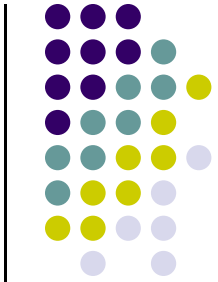
- A virtual method uses the dynamic type
- A non-virtual method uses the static type

Virtual vs. Non-virtual method



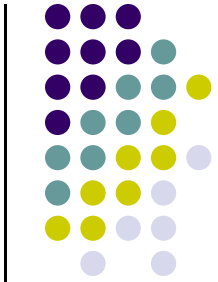
```
struct Animal {
  void f()
    { cout << "animal f"; }
  virtual void g()
    { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f()
    { cout << "gorilla f"; }
  void g()
    { cout << “gorilla g"; }
};
Animal *a = new Gorilla;
a->f();
a->g();
```

Virtual method implementation/performance



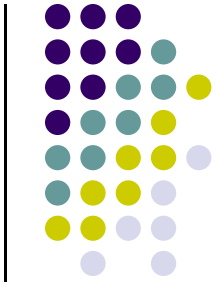
- Since the compiler doesn’t know what the type is at compile-time, a virtual function is called through a pointer to a table of functions that is stored in the object
- For a non-virtual method, the compiler knows what method will be called and calls it directly

Benchmark 1-Seem's right



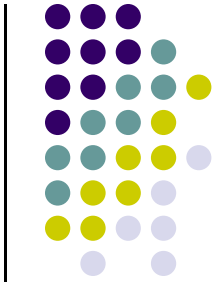
```
#include <iostream>
#include <math.h>
#include <boost/progress.hpp>
using namespace std;
class A {
public: // 15.83s if virtual 15.82 if not virtual
   virtual int f(double d, int i) {
       return static_cast<int>(d*log(d))*i;
   }
};
int main()
{
   boost::progress_timer t;
   A *a = new A();
   int ai = 0;
   for(int i = 0; i < 100000000; i++) {
           ai += a->f(i, 10);
   }
   cout << ai << endl;
}
```

Benchmark 2: Not so fast



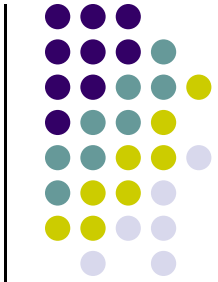
```
#include <iostream>
#include <cmath>
#include <boost/progress.hpp>
using namespace std;
class A {
public: // 15.36s if virtual 0.22 if not virtual
   virtual int f(double d, int i) {
       return static_cast<int>(d*log(d))*i;
   }
};
int main()
{
   boost::progress_timer t;
   A *a = new A();
   int ai = 0;
   for(int i = 0; i < 100000000; i++) {
           ai += a->f(10, i);
   }
   cout << ai << endl;
}
```

What happened?



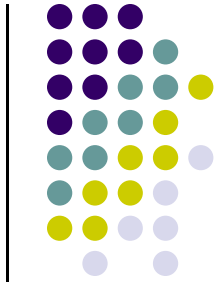
- The main performance cost of virtual functions is the loss of inlining and function level optimization
  - Not the overhead of the indirection
  - In the second benchmark, 10*log(10) only needed to be calculated once in the non-virtual case.
- Usually not significant, but this case is good to understand

Signatures of virtual methods



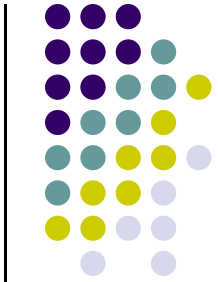
- Usually, when overriding a virtual method, the overriding method has the same signature as the method in the base class
- What happens if it is different?

Method hiding



```
struct A {
   virtual int f(double);
};
struct B : public A {
   int f(int); // Hides f(double) to protect from "fragile base class"
   void g() {
    f(7.5); // Calls f(int)
    A::f(7.5); // calls f(double)
   }
};
struct C : public A {
   using A::f;
   int f(int); // No longer hides f(double)!
   void g() {
    f(7.5); // Now calls f(double)
   }
};
```

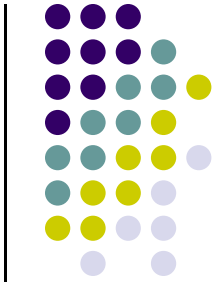
Constructors



- `new Student_info()` leaves `midterm, final` with nonsense values. (Use the original version. The one with the “pure virtual” method can’t be new’ed!)
- But not `homework`! We’ll understand that momentarily
- Fix as follows:

```
struct Student_info {
  Student_info() : midterm(0), final(0) {}
};
```

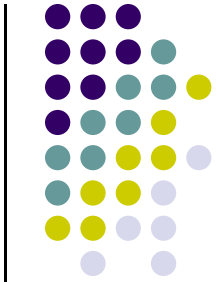
Destructors



- Consider

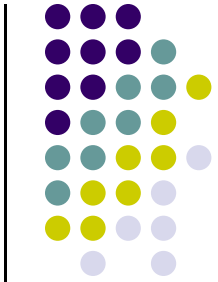
```
struct Use_grading_machine
 : public Abstract_student_info {
  Use_grading_machine() { // Constructor
    grading_machine = new Grading_machine();
  }
  // Use grading machine
  ~Use_grading_machine() {
    delete grading_machine;
  }
protected:
  Grading_machine *grading_machine;
};
```

Right idea, but not right



```
int main()
   {
     Abstract_student_info *si = new Use_grading_machine();
     si->read(cin);
     cout << “Grade is " << si->grade() << endl;
     delete si; // Calls Abstract_student_info’s destructor
     return 0;
   }
```

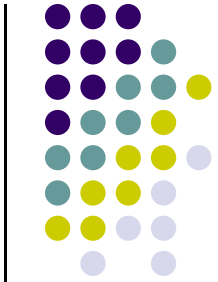
What we really need is for the destructor to be virtual!



```
Struct Abstract_student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;
  virtual ~Abstract_student_info() {}
  istream& read(istream&);
  // Don't define grading strategy here
  virtual double grade() const = 0;
};
```

- Best practice: Classes that are designed to be inherited from should have virtual destructors

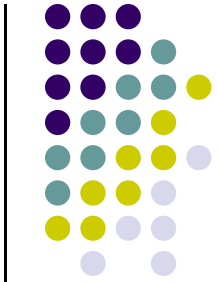
Order of construction



1. Base classes are constructed in the order they are declared
2. Members are constructed in the order they are declared
3. Total class constructor is called

Note that when a base class constructor is running, virtual functions for that base class are not yet overriden by their definitions in the total class. This can result in accidentally calling undefined virtual functions!

Order of Construction example

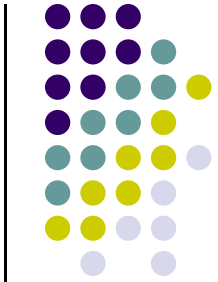


```
struct A { A() { cout << "Constructing A\n"; }};
struct B { B() { cout << "Constructing B\n"; }};
struct C : public B {
  C() { cout << "Constructing C\n"; }
};
struct D : public C {
  D() { cout << "Constructing D\n";
  A a;
};
int main() { D d; return 0}
```

prints

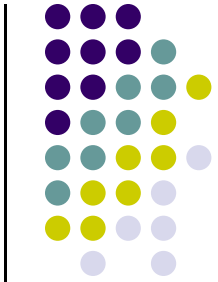
```
Constructing B
Constructing C
Constructing A
Constructing D
```

Order of destructors



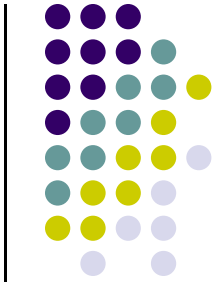
- Reverse of construction
- Similar comments about virtual functions

Implicitly generated constructors and destructors



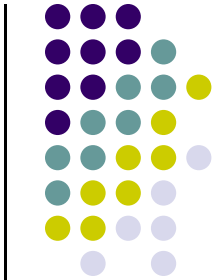
- If you don’t define a constructor or destructor, C++ will generate default ones with empty bodies.
- The default destructor is only virtual if a base class has already declared its destructor to be virtual

Review of when to inherit



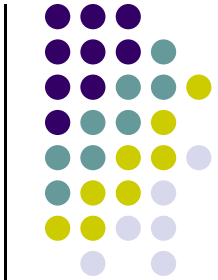
- Inherit to represent “is-a” relationship
  - A dachsund is a dog
- Use members to represent “has-a” relationship
  - A dachsund has a fur color
- Most languages don’t allow multiple inheritance
  - Shouldn’t an iostream, be both an istream and an ostream
  - In C++, it can be

Single inheritance

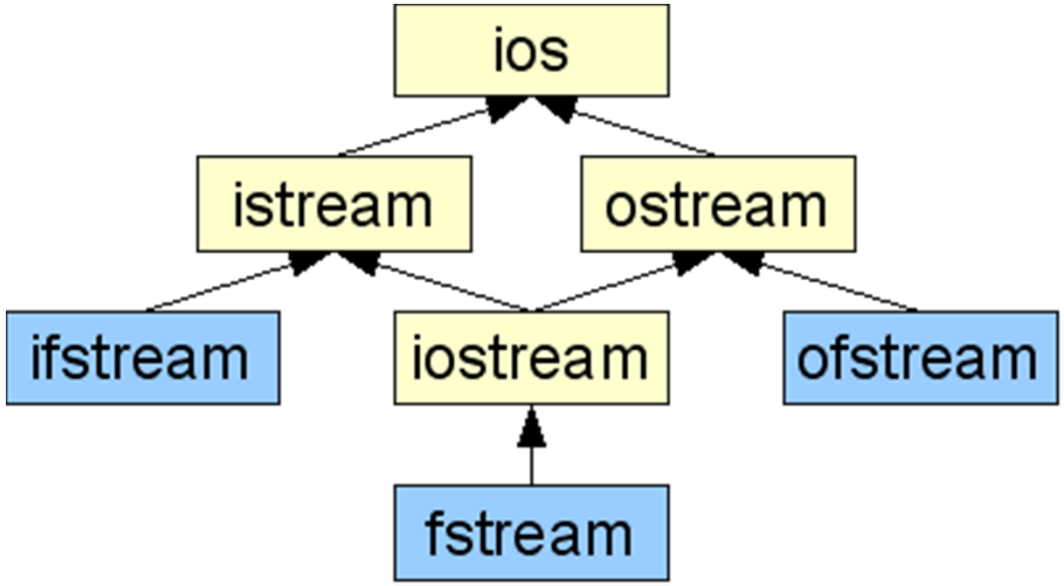


- class A : public B {…};
- class A : protected B {…};
- class A : private B {…};

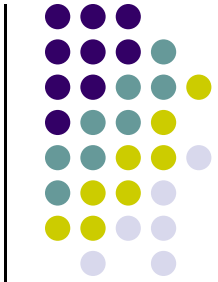
Multiple inheritance



- For a good discussion, see http://www.phpcompiler.org/doc/virtualinheritance.html



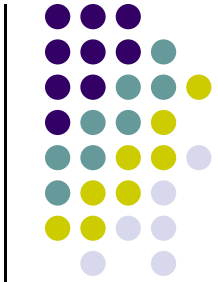
How to print a class



- Define an overloading of “<<” for your class

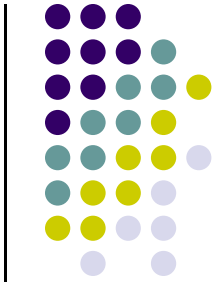
```
class MyClass {
…
int x, y;
};
ostream &
operator<<(ostream &os, const MyClass &mc)
{
  os << "MyClass(x = " << mc.x;
  os << ", y = " << mc.y << ")";
  return os;
}
```

Recommended reading



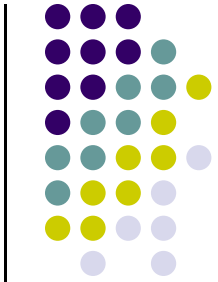
- All reading is from Koenig and Moo
- As of week 3, we have covered chapters 1-4, 5.1-5.3, 9, 13. If you need additional reinforcement, be sure to review those chapters.
- In week 4, we will be covering chapters 5-8, 14. You are encouraged to read those in advance

HW 2.1 (Part 1)



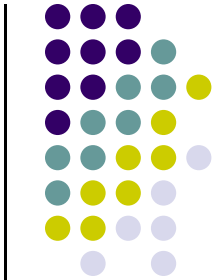
- In addition to std::copy, there is a similar function in the <algorithm> header called std::transform, which lets you apply a function to each element before copying. Look up or Google for std::transform to understand the precise usage.
- Write a program that puts several floats in a vector and then uses std::transform to produce a new vector with each element equal to the square of the corresponding element in the original vector and print the new vector (If you use ostream_iterator to print the new vector, you will likely get an extra comma at the end of the output. Don’t worry if that happens).

HW 2.1 (Part 2)



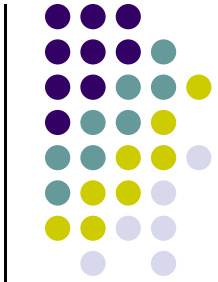
- We will extend the program in part 1 to calculate and print the distance of a vector from the origin.
- There is also a function in the <numeric> header called `accumulate` that can be used to add up all the items in a collection.
  - (Googling for "accumulate numeric example" gives some good examples of using `accumulate`. We're interested in the 3 argument version of `accumulate`).
- After squaring each element of the vector as in part 1, use accumulate to add up the squares and then take the square root. (That this is the distance from the origin is the famous Pythagorean theorem, which works in any number of dimensions).

HW 2.1 (Extra credit part 3)

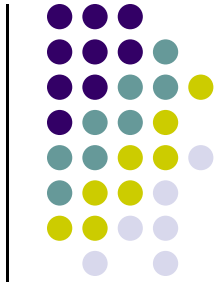


- There is also a four argument version of `accumulate` that can combine `transform` and `accumulate` in a single step. Use this to provide a more direct solution to Part 2 of this problem
  - In real life, you'd probably use the `inner_product` function in the `<numeric>` header, but don't do that for this exercise.

HW2.2

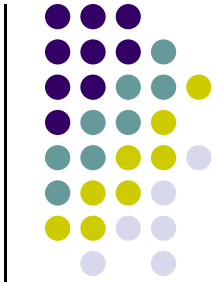


- One of the above slides referred to a function median, that takes the median of a vector of doubles.
- Part 1. Write the median function using the sort function in the algorithm header.
- Part 2. Write the median function using the partial_sort function in the algorithm header. Do you think this is more efficient? Why? (You can give an intuitive answer without precise mathematical analysis)
- Part 3. Write the median function using the nth_element function header. Do you think this is even more efficient? Why?
- Part 4. Write a template function that can find the median of a vector of any appropriate type.
  - Although we haven't discussed writing our own template functions yet, looking at the definition of `min` in slide 5 from last week should give you all you need.
  - You can use any of the underlying algorithms from parts 1 to 3 above
- Extra credit. If there are an even number of elements, use the average of the middle 2 element



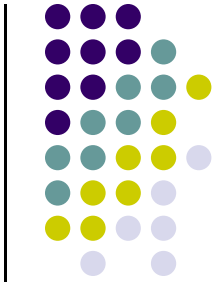
- Do one of the next two problems for full credit
- Do both for extra credit

HW 2.3



- Rewrite the pascal.cpp file in chalk (or your own) to be class-oriented

HW 2.4



- Write a program that takes a sequence of paths, and produces a tree diagram of the files. E.g., let's say that your program is named "tree_print". You'd expect to use this together with find and xargs, e.g.,

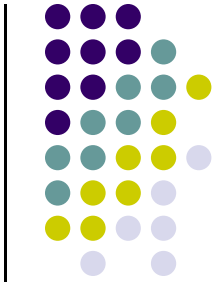
```
find . | xargs tree_print
```

  should produce a tree oriented print out. For example, suppose I execute the line above in a directory foo, which contained file bar, and directory baz with file bag. You'd expect the output

```
.
  bar
  baz
    bag
```

- If you want to (and your system's find command supplies the necessary info), you can put a "/" on the end of directory names
- Your program should be written in a class-oriented fashion.
- "find" is standard on Unix. You can get a copy of the "find" command that runs on Windows at http://gnuwin32.sourceforge.net/packages/findutils.htm. (Make sure you download the prerequisites too).
- You may find string::find_first_of and string::find_last_of worth looking at. Alternatively, Boost::Tokenizer could help.

HW 2.5—Extra Credit



- Your goal in this problem is to call a method that doesn't exist!
- Recall that a pure virtual method is not given a body:
  struct A { … virtual void f() = 0; };
- Normally, in any class you instantiate the pure virtual method f is overridden, so you don't call a non-existent method
  - In fact, it's illegal to "new" a class with pure virtual methods
- Your task is to write a program that calls A::f() even though it has no body.
  - What happens when it tries to run A::f()? (Obviously, something bad)
- See the note on slide 26 for a hint