# Lecture 4 – Data Structures

Inheritance and Class Hierarchies

1

## Lecture Outline

- *Inheritance* and how it facilitates code reuse
- How does C++ *find the "right" method* to execute?
  - (When more than one has the same name ...)
- Defining and using *abstract classes*
- Multiple inheritance:
- Sample *class hierarchy*: *drawable shapes*
- An *object factory* and how to use it
- Creating *namespaces*
  - Code *visibility*

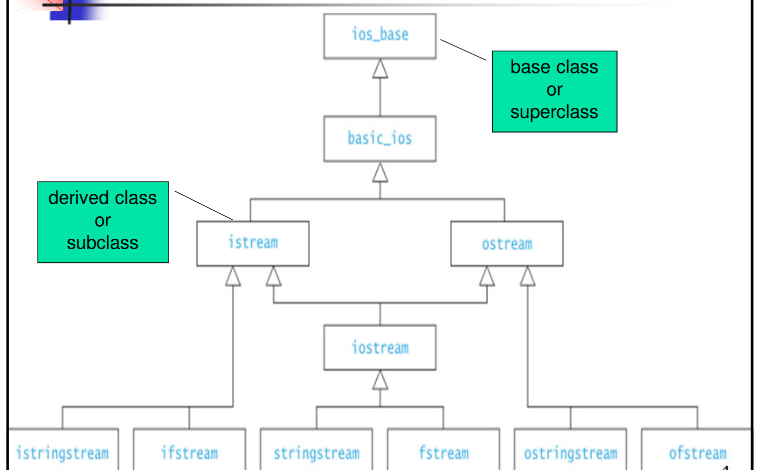
2

## Inheritance and Class Hierarchies

- Object-oriented programming (OOP) is popular because:
  - It enables *reuse* of previous code saved as *classes*
- *Inheritance* and hierarchical organization capture idea:
  - One thing is a *refinement* or *extension* of another

3

## Inheritance and Class Hierarchies (2)




4

## *Is-a* Versus *Has-a* Relationships

- Confusing *has-a* and *is-a* leads to misusing inheritance
- Model a *has-a* relationship with an *attribute* (variable)

```
class C { ... private: B part; ...}
```

- Model an *is-a* relationship with inheritance
  - If every `C` is-a `B` then model `C` as a *derived class* (also called *subclass)* of `B`
  - Show this: in `C` include `: public B`:

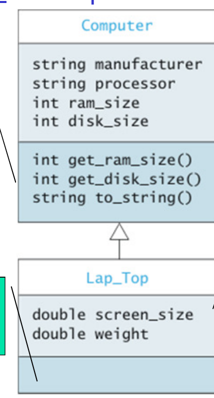
```
class C : public B { ... }
```

5

## A Superclass and a Subclass

- Consider two *classes*: `Computer` and `Laptop`
- A laptop is a *kind* of computer: therefore a *derived class*

methods of `Computer` *and all derived classes*




variables of `Computer` *and all derived classes*

*additional* variables for class `Laptop` (and its derived classes)

*additional* Methods for class `Laptop` (and its derived classes)

6

## Illustrating *Has-a* with `Computer`

```
class Computer {
  private:
    Memory mem;
  ...
};

class Memory {
  private:
    int size;
    int speed;
    std::string kind;
  ...
};
```

A `Computer` has only one `Memory`

But neither *is-a* the other

7

## Initializing Data Fields in a Derived Class

- What about data *fields of a base class?*
  - *Initialize* them by *invoking a base class constructor* with the appropriate parameters
- If the derived class constructor *skips calling the base class* ...
  - C++ automatically calls the *no-parameter* one
- ***Point:*** Insure base class fields initialized *before* derived class starts to initialize its part of the object

8

## Example of Initializing Derived Class Data

```
class Computer {
  private:
     string manufacturer;   ...
  public:
     Computer (const std::string& man, ...)
            : manufacturer(man), ...
      { ... }
}

class LapTop : public Computer {
  private:
     double weight; ...
  public: Laptop (std::string& man, ..., double wei, ...)
            :Computer(man, ...), weight(wei), ...
      { ... }
}
```

**Member Initialization List**

9

## `Protected` Visibility for Derived Class Data

- `private` data are *not accessible* to derived classes!
- `protected` data fields *accessible in derived classes*
- *Derived classes often written by others*, and
- Derived classes should *avoid relying on superclass details*
- ***So …*** in general, `private` is better

10

## Method *Overriding*

- If a *base class* declares a *member function with the same signature to be* ***`virtual`***
- And if *derived class has a member function with the same signature*
- Then that member function ***overrides*** the superclass method:

```
class A { ...
    public:
       virtual int M (float f, string& s) { bodyA }
}

public class B :public A { ...
    public:
       int M (float f, string& s) { bodyB }
}
```

- If we call `M` on an *instance of* `B` (or derived class of `B`), `bodyB` runs
- *In* `B` we can *access* `bodyA` with: `A::M(...)`
- The *derived* `M` must have *same return* type as *base* `M`

11

## Method *Overloading*

- **Method overloading:** *multiple* methods …
  - With the *same name*
  - But *different signatures*
  - In the *same class*
- *Constructors are often overloaded*
- Example:
  - `MyClass (int inputA, int inputB)`
  - `MyClass (float inputA, float inputB)`

12

## Example of Overloaded Constructors

```
class Lap_Top : public Computer {
  private:
     double weight; ...
  public:
     Lap_Top (const std::string& man,
             const std::string pro, ...,
             double wei, ...)
     :Computer(man, pro), weight(wei) ...
               { }
     Lap_Top (const std::string& man, ...,
                 double wei, ...)
     :  Computer(man, "Pentium"), weight(wei)
               {   }
}
```



## Polymorphism

- Pointer of *base class type* can point to *object of derived class type*
- *Polymorphism* means "many forms" or "many shapes"
- Polymorphism lets the C++ determine *at run time* which *method to invoke*
- *At compile time:*
  - C++ compiler *cannot determine* exact *type of the object*
  - But it *is* known at run time
- Compiler knows enough for safety: the *attributes* of the type
  - *Derived class guaranteed to obey*



## Interfaces vs Abstract Classes vs Concrete Classes

- An ***abstract class*** can have:
  - Abstract methods (no body)
  - Concrete methods (with body)
  - Data fields
- Unlike a concrete class, an *abstract class* ...
  - *Cannot be instantiated*
  - *Can declare abstract methods*
    - Which *must* be implemented in all *concrete* subclasses



## Abstract Classes

- Abstract classes cannot be instantiated
- An abstract class *can* have constructors!
  - *Purpose:* initialize data fields when a subclass object is created

## Example of an Abstract Class

```
class Food {
  public:
    const std::string name;
    double calories;
    double get_calories () {
      return calories;
  }
  protected:
    Food (const std::string the_name, double the_calories)
      : name(the_name), calories(the_calories)
            { }
  public:
    virtual double percent_protein() = 0;
    virtual double percent_fat() = 0;
    virtual double percentCarbs() = 0;
}
```

17

## Example of a Concrete Subclass

```
class Meat : public Food {
  private:
    const double prot_cal; ...;
  public:
    Meat (const std::string the_name, double the_prot_cal,
          double the_fat_cal, double the_carb_cal)
      : Food(name, the_prot_cal+the_fat_cal+the_carb_cal),
  proto_cal(the_proto_cal) ...
                {}
  public double percentProtein () {
    return 100.0 * (prot_cal / get_calories());
  }
  ...;
}
```

18

## Summary of Features of Actual Classes, Abstract Classes, and Interfaces

| Property | Actual Class | Abstract Class |
|---|---|---|
| Instances (objects) of this type can be created. | Yes | No |
| This can define instance variables and functions | Yes | Yes |
| This can define constants | Yes | Yes |
| The number of these a class can extend | Any number | Any number |
| This can extend another class | Yes | Yes |
| Can define abstract member functions | No | Yes |
| Pointers to this type can be created | Yes | Yes |
| References of this type can be declared | Yes | Yes |

19

## Operations Determined by Type of Pointer Variable

- Pointer can point to object whose type is a *derived class* of the variable's declared type
- Type of the *variable* determines what operations are legal
- C++ is *strongly typed*

```
Computer* a_computer = new Lap_Top( ... );
```

  - Compiler always verifies that variable's type includes the class of every expression assigned to the variable

20

## Casting in a Class Hierarchy

- *Dynamic Casting* obtains a pointer of different, but *matching,* type
- Dynamic Casting *does not change* the object!
  - `Lap_Top* my_lap_top = dynamic_cast<Lap_Top*>(a_computer);`
- *Downcast:*
  - Cast *baseclass* type to *derived* type
  - Checks *at run time* to make sure it's ok
  - If not ok, returns a NULL pointer.



## Polymorphism and Type Tests (2)

- Polymorphic code style is more *extensible*
  - Works *automatically* with new subclasses
- Polymorphic code is more *efficient*
  - System does *one indirect branch* vs *many tests*
- ***So …*** uses of **`dynamic_cast`** are *suspect*

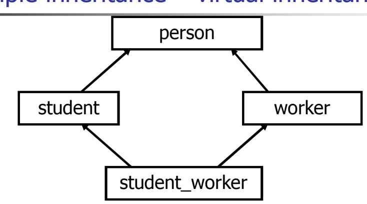


## Multiple Inheritance

- *Multiple inheritance:* the ability to *extend* more than one class
- Multiple inheritance …
  - Is difficult to implement efficiently
  - Can lead to *ambiguity*: if *two parents implement the same method,* which to use?
  - C++ allows multiple inheritance
    - But, it must be used with caution and detailed knowledge of the base classes.
  - Derived class will have two subobjects of the common base.
    - *class student : virtual public person { }*
    - *Virtual inheritance – no duplicate in the derived class. Methods must work the same way in the class hierarchy.*



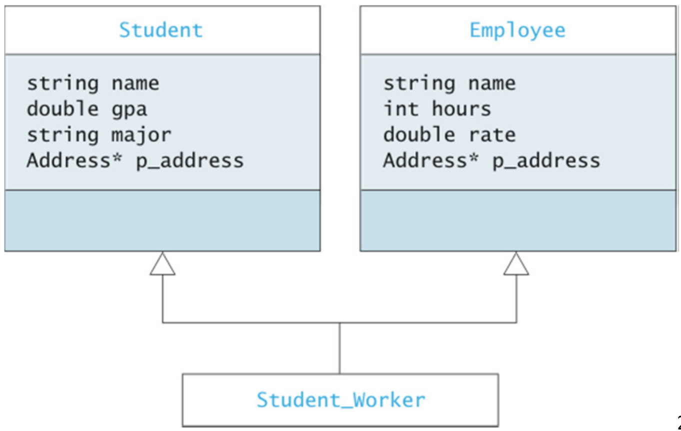
## Multiple inheritance – virtual inheritance




- Constructor execution order:
  - Base classes initialized in declaration order
  - Members initialized in declaration order
  - The body of the constructor.
- Virtual base classes are constructed before any of their derived classes and before any non-virtual base classes.

## Multiple Inheritance

- A class can extend two or more classes




25

## Namespaces

- A C++ ***namespace*** is a group of *related declarations*
- The C++ *standard library* is in namespace ***std***.
- Other libraries are defined in their own namespace.
  - Classes we define for the class that are similar to those in the standard library can be placed into namespace MS
  - This avoids conflicts between classes we create and those in the standard library

26

## The default namespace

- There is a *default namespace*
  - It contains *files that have no namespace declared*
- Default namespace ok for small projects
  - Namespaces good for larger groups of classes

27

## Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in C++
- **`private`**: Good for members that should be invisible even in subclasses
- **`protected`**: Good for visibility to extenders of classes
- **`public`**: Good for visibility to all

28

## Visibility Supports Encapsulation (2)

- Encapsulation provides insulation against change
- Greater visibility means less encapsulation

- ***So:*** *use minimum visibility* possible for getting the job done!

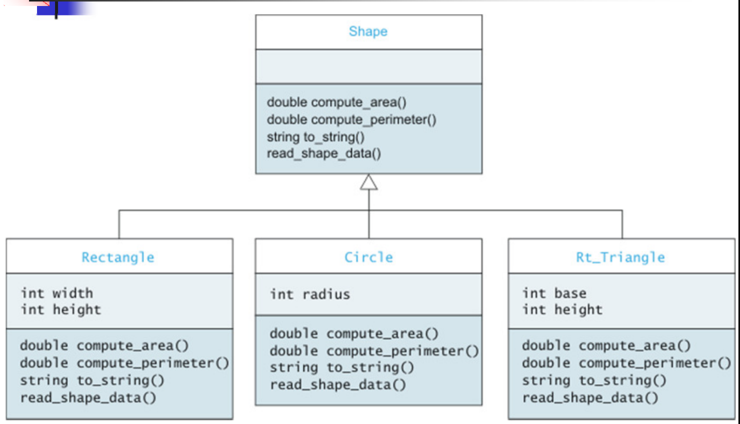
29

## Visibility Supports Encapsulation (3)

| Visibility | Applied to Class Members |
|---|---|
| `private` | Visible only within this class. |
| `protected` | Visible to classes that extend this class |
| `public` | Visible to all classes and functions |

30

## A Shape Class Hierarchy




31

## A Shape Class Hierarchy (2)

| Data Field | Attribute |
|---|---|
| `int width` | Width of a rectangle |
| `int height` | Height of a rectangle |
| **Function** | **Behavior** |
| `double compute_area() const` | Computes the rectangle area (width × height) |
| `double compute_perimeter() const` | Computes the rectangle perimeter (2 × width + 2 × height) |
| `void read_shape_data()` | Reads the width and height |
| `string to_string() const` | Returns a string representing the state |

32

## Object Factories

- ***Object factory:*** *function that creates instances of other classes*
- Object factories are *useful when*:
  - The necessary *parameters are not known* or must be derived via computation
  - The appropriate *implementation should be selected at run time* as the result of some computation

33

## Example Object Factory

```
Shape* get_shape() {
  char fig_type;
  cout << "Enter C for circle\n"
       << "Enter R for rectangle\n"
       << "Enter T for right triangle\n";
  cin >> fig_type;
  switch (fig_type) {
     case 'c': return new Circle();
     case 'r': return new Rectangle();
     case 't': return new Rt_Triangle();
     default: return NULL;
  }
}
```

34

## Separating Interface from Implementation

**Syntax : Class Header File *(public interface)***
// Declaration of the class (including the function prototypes)
// type of the file: header file ( .h)

```
#ifndef  ClassName_H
#define ClassName_H
// class declaration
class ClassName
{
   public :
   ....
   private :
   ....
};
#endif
```

35

## Member Function Definitions

**Syntax: *(implementation of the class)***

// type of the file: source file ( .cpp)

```
#include <C++ system header files>
#include "programmer-defined header files"
// constructor implementation
className : : className ( . . . ) : . . .
{ . . . }

// member functions implementation
returnType className : : memberFunction ( . . . )
{ . . . }
```

36

## Syntax : Driver Program

```
// Driver to test the class
// type of the file: source file ( .cpp)

#include <C++ system header files>
#include "programmer-defined header files"
returnType main ( )
{
   ...
}
```

37

## Time Class

```
// Example: time1.h
// Declaration of the Time class

#ifndef TIME1_H
#define TIME1_H
// Time abstract data type definition
class Time {
public:
   Time();                    // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();      // print military time format
   void printStandard();      // print standard time format
private:
   int hour;    // 0 - 23
   int minute;  // 0 - 59
   int second;  // 0 - 59
};
#endif
```

38

## When Constructors and Destructors are Called

- **Constructors** and **destructors** are called automatically. Generally, **destructor** calls are made in the *reverse order of the constructor calls.*
- *Constructors* are called for *objects* defined in *global* scope *before any other function* (including **main**) in that file begins execution. The corresponding *destructors* are called when **main** terminates or the **exit** function is called.

39

## When Constructors and Destructors are Called cont'd

- *Constructors* are called for **automatic** local objects when execution reaches the *point* where the objects are *defined*. The corresponding *destructors* are called when the *objects leave scope* ( i.e., the *block* in which they are defined is exited). *Constructors* and *destructors* for **automatic** objects are called each time the objects *enter and leave scope*.
- *Constructors* are called for ***static local** objects* only once when execution *first* reaches the point where the *objects are defined*. Corresponding *destructors* are called when **main** terminates or the **exit** function is called.

40

## Example

```
// Example create.h
// Definition of class CreateAndDestroy.
#ifndef CREATE_H
#define CREATE_H
class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};
#endif
```

41

## Implementation File

```
// Fig. 6.9: create.cpp
// Member function definitions for class CreateAndDestroy

#include <iostream>
#include "create.h"
CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}
CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

42

## Driver Program

```
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
#include "create.h"
void create( void );                          // prototype
CreateAndDestroy first( 1 );                  // global object
 cout << "  (global created before main)" << endl;
int main()
{
  CreateAndDestroy second( 2 );               // local object
  cout << "  (local automatic in main)" << endl;
  static CreateAndDestroy third( 3 );         // local object
  cout << "  (local static in main)" << endl;
  create();                         // call function to create objects
  CreateAndDestroy fourth( 4 );               // local object
  cout << "  (local automatic in main)" << endl;
  return 0;
}
```

43

## Driver Program cont'd

```
// Function to create objects

void create( void )
{
  CreateAndDestroy fifth( 5 );
  cout << "  (local automatic in create)" << endl;
  static CreateAndDestroy sixth( 6 );
  cout << "  (local static in create)" << endl;
  CreateAndDestroy seventh( 7 );
  cout << "  (local automatic in create)" << endl;
}
```

44

## Output

| | | | |
|---|---|---|---|
| Object | 1 | constructor | (global created before main) |
| Object | 2 | constructor | (local automatic in main) |
| Object | 3 | constructor | (local static in main) |
| Object | 5 | constructor | (local automatic in create) |
| Object | 6 | constuctor | (local static in create) |
| Object | 7 | constructor | (local automatic in create) |
| Object | 7 | destructor | |
| Object | 5 | destructor | |
| Object | 4 | constructor | (local automatic in main) |
| Object | 4 | destructor | |
| Object | 2 | destructor | |
| Object | 6 | destructor | |
| Object | 3 | destructor | |
| Object | 1 | destructor | |

45

## Discussions

- Function **main** declares three objects.
- Objects **second** and **fourth** are *local automatic* objects, and object **third** is a *static local* object.
- The *constructor* for each of these objects are called when execution reaches the point where each object is declared.
- The *destructors* for objects **fourth** and **second** are called in that order when the end of **main** is reached.
- Because object **third** is *static*, it exists until program termination. The *destructor* for object **third** is called before the *destructor* for **first**, but after all other objects are destroyed.

46

## Discussions cont'd

- Function **create** declares three objects – **fifth** and **seventh** are *local automatic* objects, and **sixth** is a *static local* object.
- The **destructors** for objects **seventh** and **fifth** are called in that order when the end of **create** is reached.
- Because **sixth** is *static*, it exists until program termination. The **destructor** of **sixth** is called before the **destructors** for **third** and **first**, but after all other objects are destroyed.

47