C++ Object-Oriented Prog.
Unit 4: Objects and Lists

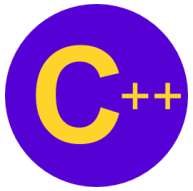
CHAPTER 13: OBJECTS AND CLASSES

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Overview of This Course



Chapter 13 Topics

- Review chapter 9
- Meaning of an Abstract Data Type
- Declaring and Using a class Data Type
- Using Separate Specification and Implementation Files
- Invoking class Member Functions in Client Code
- C++ class Constructors

Abstraction

Abstraction is the separation of the essential qualities of an object from the details of how it works or is composed

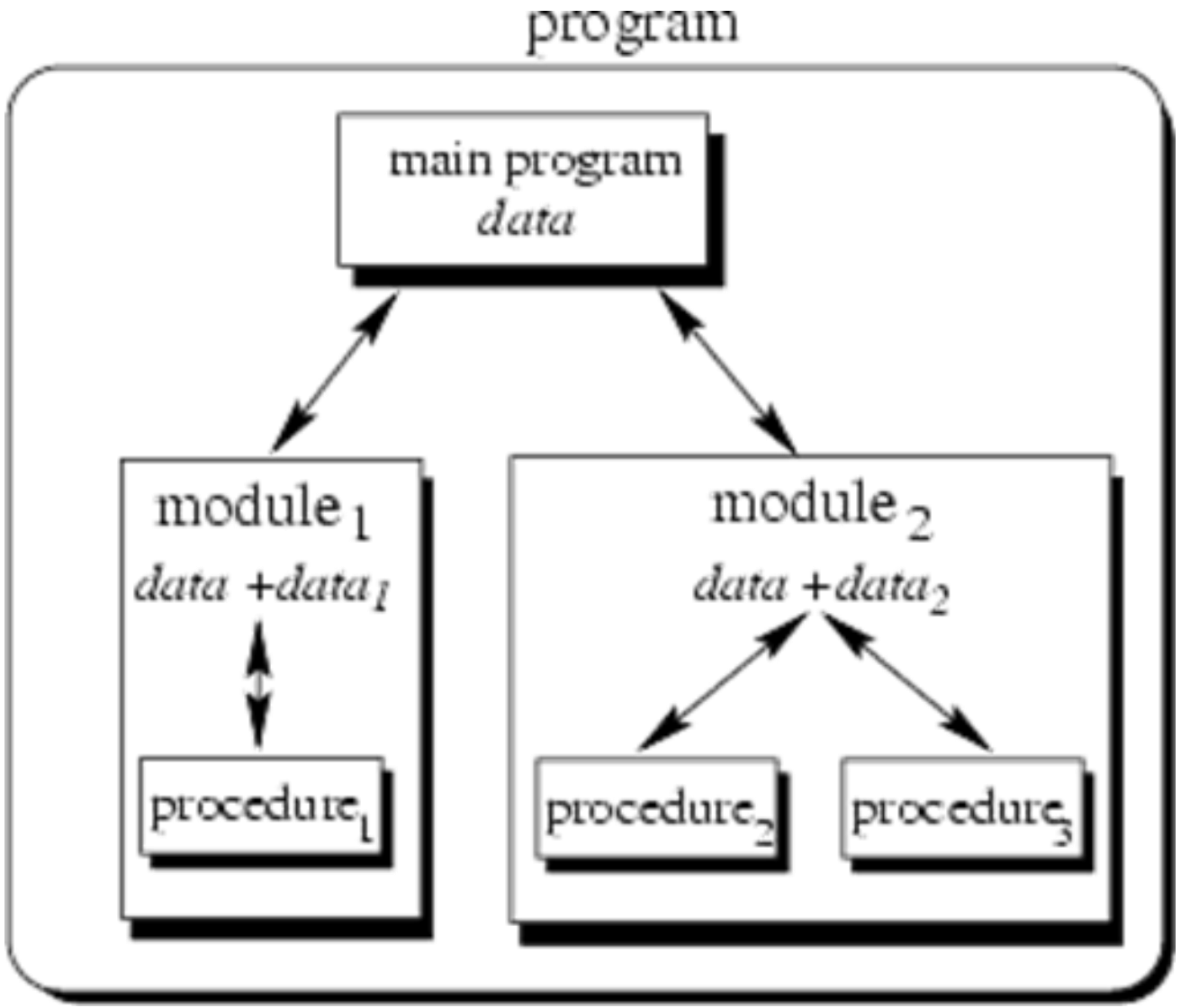
- Focuses on what, not how
- Necessary for managing large, complex software projects

Control Abstraction

Control abstraction separates the logical properties of an action from its implementation:

Search (list, item, length, where, found);

The function call depends on the function's specification (description), not its implementation (algorithm)

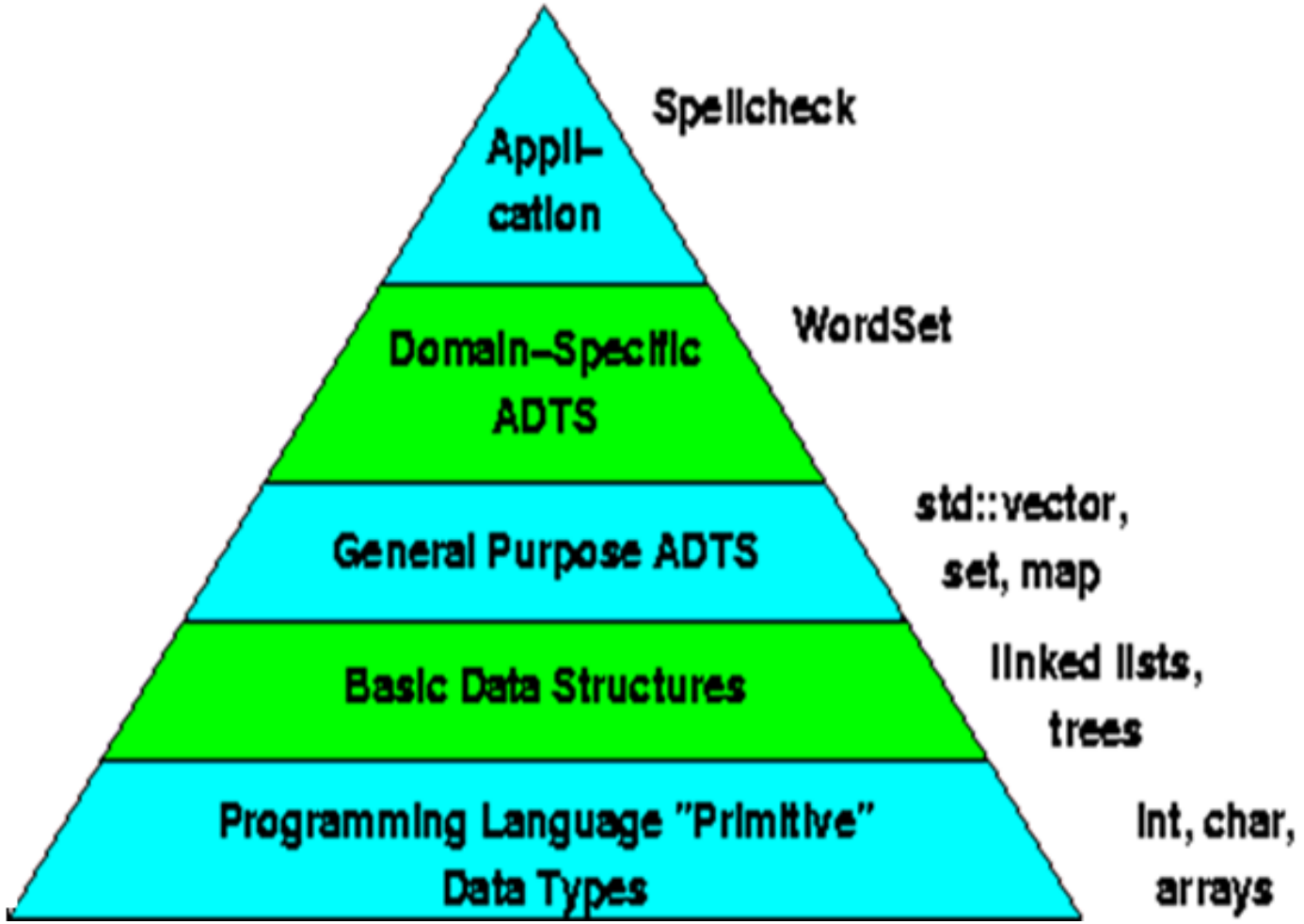

program
main program
data
module 1
data +data1
procedure1
module 2
data +data2
procedure2
procedure3

Data Abstraction

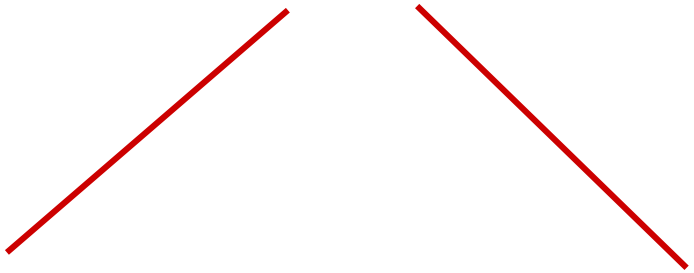
Data abstraction separates the logical properties of a data type from its implementation

LOGICAL PROPERTIES	IMPLEMENTATION
What are the possible values?	How can this be done in C++?
What operations will be needed?	How can data types be used?

Data Abstraction in C Language



Data Type



set of values (domain)

allowable operations on those values

FOR EXAMPLE, data type `int` has

domain	operations
-32768 . . . 32767	+, -, *, /, %, >>, <<

Possible implementation: object-oriented (class) or structured (struct, union, array, and etc)

Abstract Data Type (ADT)

An abstract data type is a data type whose properties (domain and operations) are

- specified (*what*) independently of
- any particular implementation (*how)*

For example . . .

ADT Specification Example

TYPE

Time

DOMAIN

Each Time value is a time in hours, minutes, and seconds.

OPERATIONS

- Set the time
- Print the time
- Increment by one second
- Compare 2 times for equality
- Determine if one time is “less than” another

Another ADT Specification

TYPE

ComplexNumber

DOMAIN

Each value is an ordered pair of real numbers (a, b) representing a + b*i*

Another ADT Specification, cont...

OPERATIONS

- Initialize the complex number
- Write the complex number
- Add
- Subtract
- Multiply
- Divide
- Determine the absolute value of a complex number

ADT Implementation

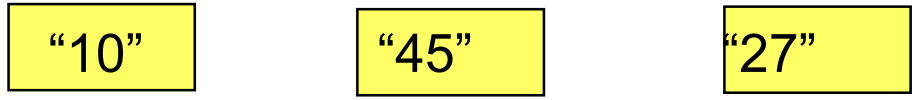
ADT implementation

- Choose a specific data representation for the abstract data using data types that already exist (built-in or programmer-defined)
- Write functions for each allowable operation

Several Possible Representations of ADT `Time`

3 int variables

3 strings

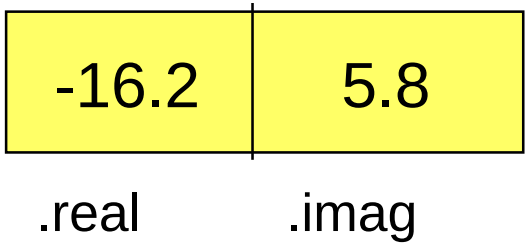


3-element int array

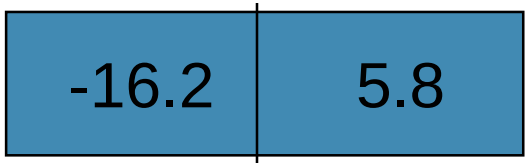
- Choice of representation depends on time, space, and algorithms needed to implement operations

Some Possible Representations of ADT `ComplexNumber`

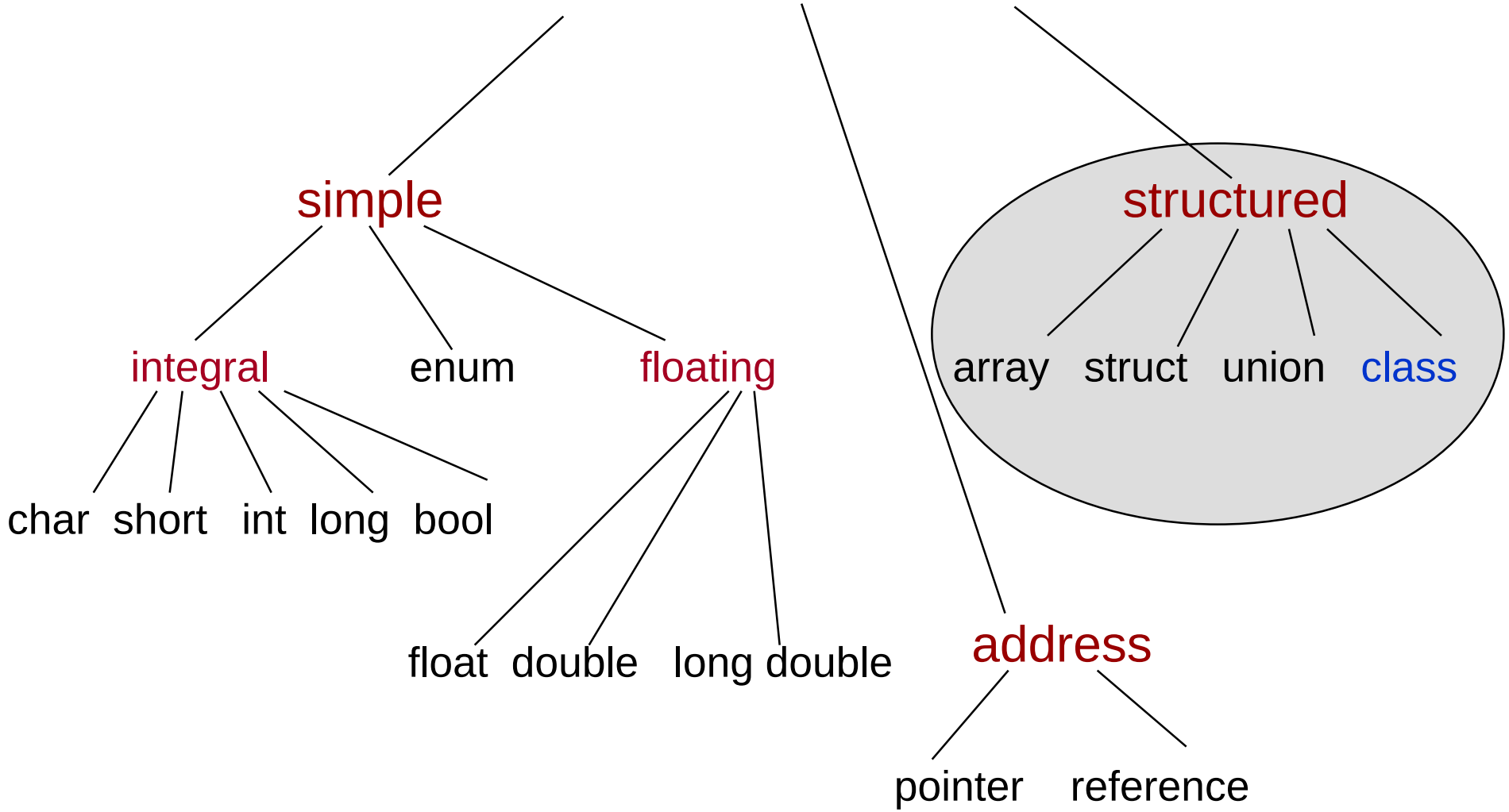
struct with 2 float members



2-element float array



C++ Data Types

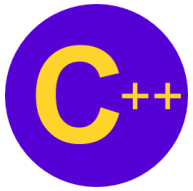

simple
integral
enum
floating
char
short
int
long
bool
float
double
long double
structured
array
struct
union
class
address
pointer
reference

LECTURE 2

Advanced Class Definition I

class Time Specification

```
//  Specification file (Time.h)
class  Time          // Declares a  class data type
{                    //  does not allocate memory
  public :           // Five public function members

   Time(int  hours , int  mins , int  secs);
   void  Set(int  hours , int  mins , int  secs);
    void  Increment ();
    bool  Equal (Time   otherTime)  const;
    bool  LessThan (Time   otherTime)  const;

  private :          // Three private data members

    int  hrs;
    int  mins;
    int  secs;
};
```



C++ `class` Type

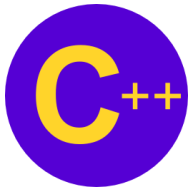
- Facilitates re-use of C++ code for an ADT
- Software that uses the class is called a client
- Variables of the class type are called class objects or class instances
- Client code uses class's public member functions to manipulate class objects

Client Code Using `Time`

```
#include   "time.h"   // Includes specification of the class
using  namespace  std;
int  main (){
    Time  currentTime; // Declares two objects of Time
    Time  endTime;
    bool  done  =  false;

    currentTime.Set (5, 30, 0);
    endTime.Set (18, 30, 0);
    while  (! done)
    { .  .  .

        currentTime.Increment ();
       if  (currentTime.Equal (endTime))
          done  =  true;
    };
}
```

class type Declaration

- The class declaration creates a data type and names the members of the class
- It does not allocate memory for any variables of that type!
- Client code still needs to declare class variables

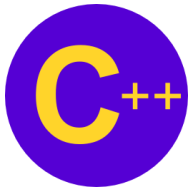


Remember …

Two kinds of class members:

1. data members and
2. function members

Class members are private by default



Remember as well . . .

- Data members are generally private
- Function members are generally declared public
- Private class members can be accessed only by the class member functions (and friend functions), not by client code

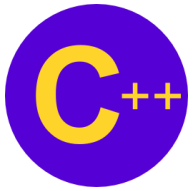
const function

A function is constant if const keyword is used in functional declaration

- A function becomes **const** when **const** keyword is used in function's declaration.
- The idea of **const** functions is not allow them to modify the object on which they are called.
- It is recommended practice to make as many functions **const** as possible so that accidental changes to objects are avoided.

Use of `const` with Member Functions

- When a member function does **not** modify the private data members:
- Use `const` in both the function prototype (in specification file) and the heading of the function definition (in implementation file)



Demo Program: const.cpp

Go Notepad++!

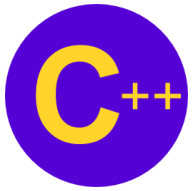
LECTURE 3

Advanced Class Definition II

Advanced Class Definition

- this pointer
- to_string and equal_to methods
- Overloading of standard operators.

this Keyword is a Pointer Not a Reference in C++



Use **this**->hrs = 3;

this->GetHours();

- When the language was first evolving, in early releases with real users, there were no references, only pointers.
- References were added when operator overloading was added, as it requires references to work consistently.

this Keyword is a Pointer Not a Reference in C++

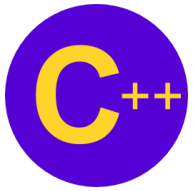
- One of the uses of this is for an object to get a pointer to itself. If it was a reference, we'd have to write **&this**.
- On the other hand, when we write an assignment operator we have to return ***this**, which would look simpler as return this. So if you had a blank slate, you could argue it either way. But C++ evolved gradually in response to feedback from a community of users (like most successful things).
- The value of backward compatibility totally overwhelms the minor advantages/disadvantages stemming from this being a reference or a pointer.

C++ does not always support to_string()

add this code into your .h/.hpp file (This is used for all primitive data types)

```
#include <sstream>
namespace st{
    template < typename T > std::string to_string( const T& n ){
        std::ostringstream stm ;
        stm << n ;
        return stm.str() ;
    }
}
```

C++ equals() function is always custom-defined



- If == is used, that will be checking the reference address (pointer).
- ==, >, <, >=, <= != operators can all be overridden in C++
- C++ used ==, equal_to and many other names to check the equality. A good habit is to use your own standard equal_to method.
- I would used equal_to and to_string in C++ language (As equals() and toString() in Java.

Overloading of ==

```
bool operator==(const Time &rhs) const {
    return hrs == rhs.hrs &&
           mins == rhs.mins &&
           secs == rhs.secs;
}
```

Operator Overloading

- Each C++ operator has a predefined meaning. Most of them are given additional meaning through the concept called **operator overloading**
- Main idea behind Operator overloading is to use C++ operators with the class objects.

Eg:

'+' can be used to add 2 ints, floats or doubles, the same **'+'** can be used to add 2 class objects, thereby **'+'** gets overloaded.

- When an operator is overloaded, its original meaning is not lost.

C++ provides a wide variety of operators to perform operations on various operands.

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-wise	&, |, ~, ^
Logical	&&, ||, !
Relational	>, <, ==, !=, <=, >=
Assignment or Initialization	=
Arithmetic Assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Shift	<<, >>, <<=, >>=
Unary	++, --
Subscripting	[]
Function Call	()
Dereferencing	->
Unary Sign Prefix	+, -
Allocate and Free	new, delete

Table 13.1: C++ overloadable operators

Demo Program: time.cpp

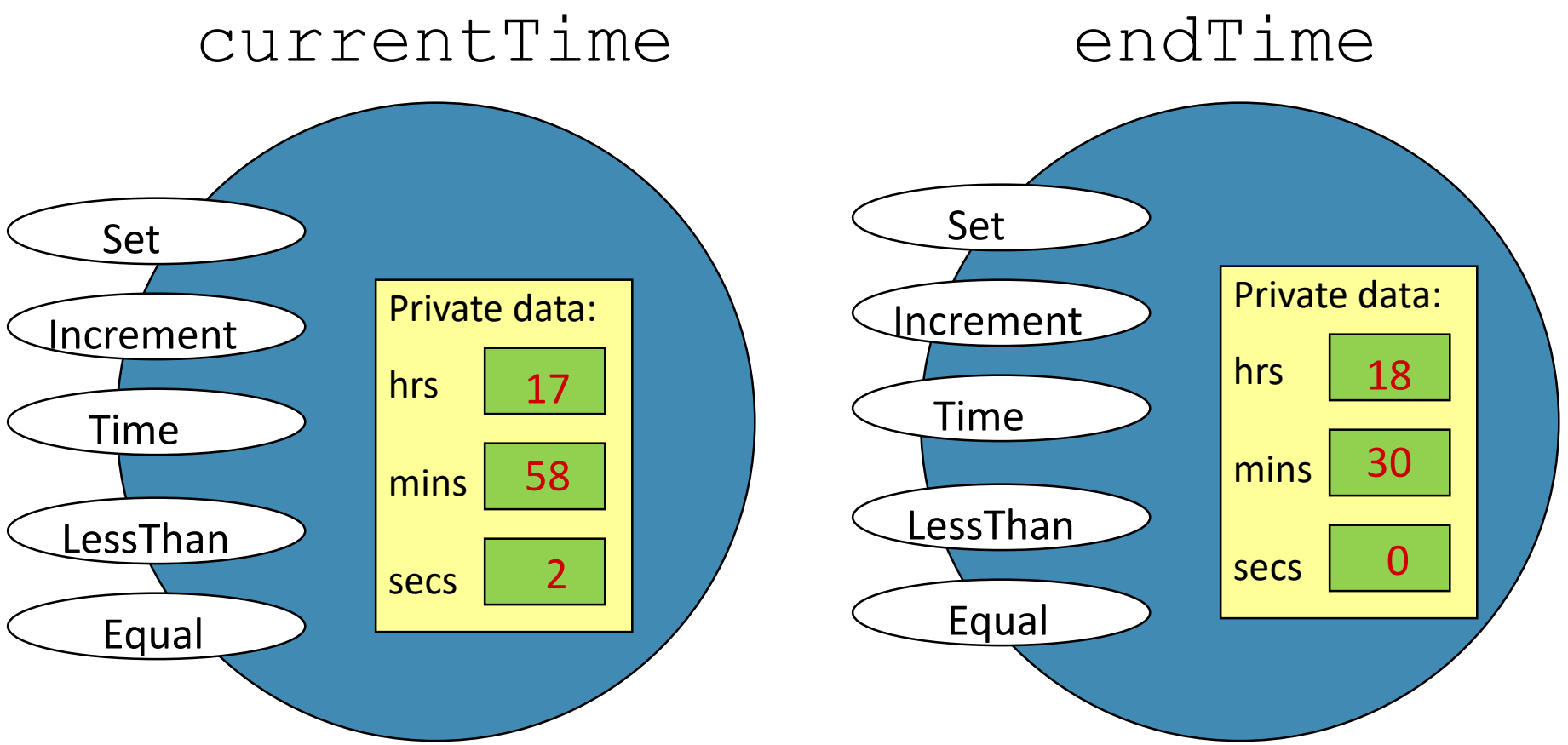
timeTester

Go Notepad++!!!

LECTURE 4

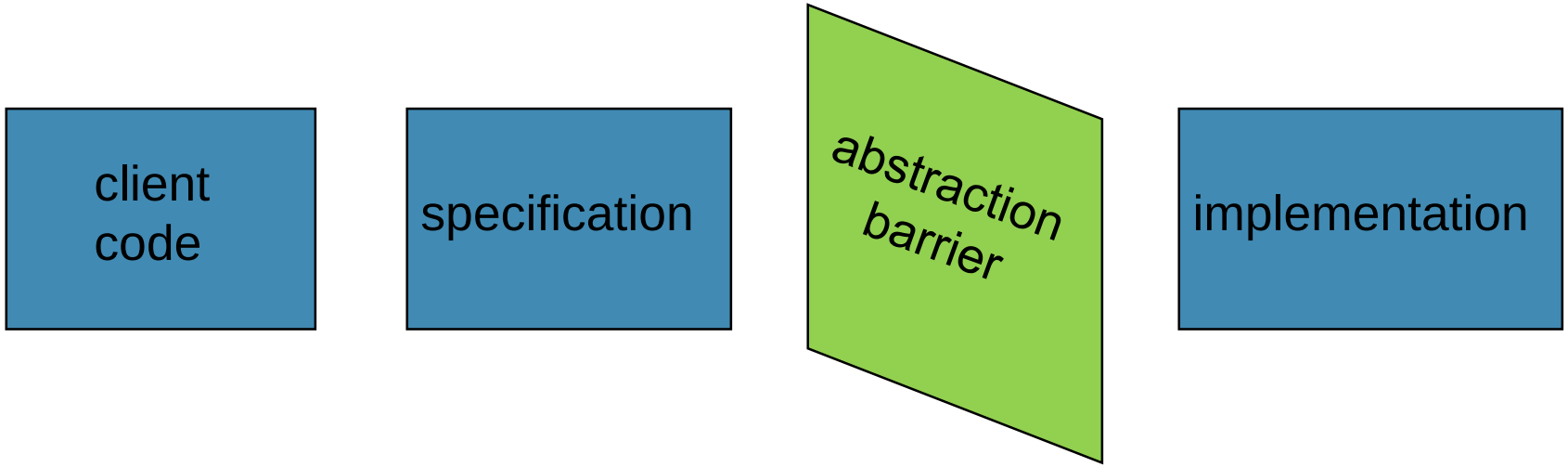
Aggregate Functions for Objects

Time Class Instance Diagrams



Information Hiding

- Information hiding - Class implementation details are hidden from the client's view
- Public functions of a class provide the interface between the client code and the class objects



Selection and Resolution

- C++ programs typically use several class types
- Different classes can have member functions with the same identifier, like Set()

Selection and Resolution

- Member selection operator is used to determine the object to whom member function Set() is applied

```
currentTime.Set(3, 3, 3);// Class Time
numberZ.Set(2, 5);          // Class ComplexNumber
```

- In the implementation file, the scope resolution operator is used in the heading before the function member's name to specify its class

```
void  Time::Set()   const
{          . . .
}
```

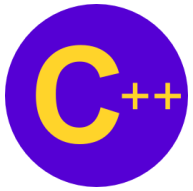
Aggregate class Operations

- Built-in operations valid on class objects are:
  - Member selection using dot (.) operator ,
  - Assignment to another class variable using (=),
  - Pass to a function as argument (by value or by reference),
  - Return as value of a function

  Other operations can be defined as class member functions

Aggregate class operations:

a.data, a=b, f(a) and return a



Separate Specification and Implementation

Specification File: .h,.hpp

Implementation File: .cpp

```
#define MAIN   // comment this line out if not __main__ function
#ifndef TIME_H
#define TIME_H
#include <iostream>
#include <string>
using  namespace  std;
/* to_string patch:
*/
#include <sstream>
namespace st{
   template < typename T > std::string to_string( const T& n ){
      std::ostringstream stm ;
      stm << n ;
      return stm.str() ;
   }
}
//  Specification file (Time.h)
class  Time   // Declares a  class data type
{       //  does not allocate memory
 public :      // Five public function members
   Time(int  h , int  m , int  s) ;
   void  Increment ();
   void  Set(int h, int m, int s);
   int   GetHours();
   int   GetMins();
   int   GetSecs();
   bool  Equal (Time   otherTime)  const;
   bool  LessThan (Time   otherTime)  const;
   string to_string();
   bool operator==(const Time &rhs) const { return hrs == rhs.hrs && mins == rhs.mins && secs == rhs.secs; }

 private :       // Three private data members
     int  hrs;
     int  mins;
     int  secs;
};
#endif
```

Definition of to_string()

Overloading of == operator

Specification File of a Module (class)

Put the specification of a class in a .h (.hpp) file.

.h file should have information that other module needs to know (class interface):

1. Data Fields
2. Function Prototypes
3. User-defined Data types
4. In-line functions
5. struct, union
6. Overloading of functions
7. Generic Templates (complete Template)

Avoid redefinition when included

```
#ifndef XXX_H
#define XXX_H
/* put the definition here*/
#endif
```

```cpp
#include <iostream>
#include  "time.h"  // Includes specification of the class
#include <string>
using  namespace  std;
```

Implementation File of a Module (class)

Put the implementation of a class in a (.cpp) file.

1. main(): program entry point, used as tester function for this class
2. Implementation of member functions

```cpp
Time::Time (int  h=0 , int  m=0 , int  s=9) : hrs(h), mins(m), secs(s) {
    hrs = h;
    mins = m;
    secs = s;
}
void  Time::Increment (){
   secs++;
   mins += secs/60;
   secs  %= 60;
   hrs   += mins/60;
   mins %= 60;
   hrs   %=24;
}
int Time::GetHours(){ return hrs; }
int Time::GetMins(){ return mins; }
int Time::GetSecs(){return secs; }
void  Time::Set(int h, int m, int s) {
    hrs = h;
    mins = m;
    secs = s;
}
bool  Time::Equal (Time   otherTime) const {
   int h1 = this->GetHours();
   int h2 = otherTime.GetHours();
   int m1 = this->GetMins();
   int m2 = otherTime.GetMins();
   int s1 = this->GetSecs();
   int s2 = otherTime.GetSecs();
   if (h1 == h2 &&  m1 == m2 && s1  == s2) return true;
   return false;
}
bool  Time::LessThan (Time   otherTime) const{
   int h1 = this->GetHours();
   int h2 = otherTime.GetHours();
   int m1 = this->GetMins();
   int m2 = otherTime.GetMins();
   int s1 = this->GetSecs();
   int s2 = otherTime.GetSecs();
   if (h1 < h2) return true;
    else if (h1 == h2){
            if (m1 < m2) return true;
            else if (m1 == m2){
               if (s1 < s2) return true;
               else if (s1 == s2) return false;
                    else return false;  }
            else return false;}
    else return false;
}
```

const function

```cpp
string Time::to_string(){
   string rtn = "";
   rtn += "Time: " + st::to_string(this->GetHours()) + ":" + st::to_string(this->GetMins()) + ":" + st::to_string(this->GetSecs());
   return rtn;
}
#ifdef MAIN
int  main (){
   Time  currentTime; // Declares two objects of Time
   Time  endTime;
   bool  done  =  false;

   currentTime.Set (5, 30, 0);
   endTime.Set (5, 30, 5);
   while  (! done)
   {
      currentTime.Increment ();
      cout<<  currentTime.to_string() << endl;
      if  (currentTime.Equal (endTime))
         done  =  true;
      if (currentTime == endTime)    cout << "Program Finished" << endl;
   };
}
#endif
```

main() that can be disabled by commenting #define MAIN in .h file

Should be familiar …

- The member selection operator (.) selects either data members or function members. (->) for a pointer.
- Header files iostream and fstream declare the istream, ostream,and ifstream, ofstream I/O classes.

Should be familiar. . .

Both cin and cout are class objects and get and ignore are function members:

```
cin.get(someChar);
cin.ignore(100, '\n');
```

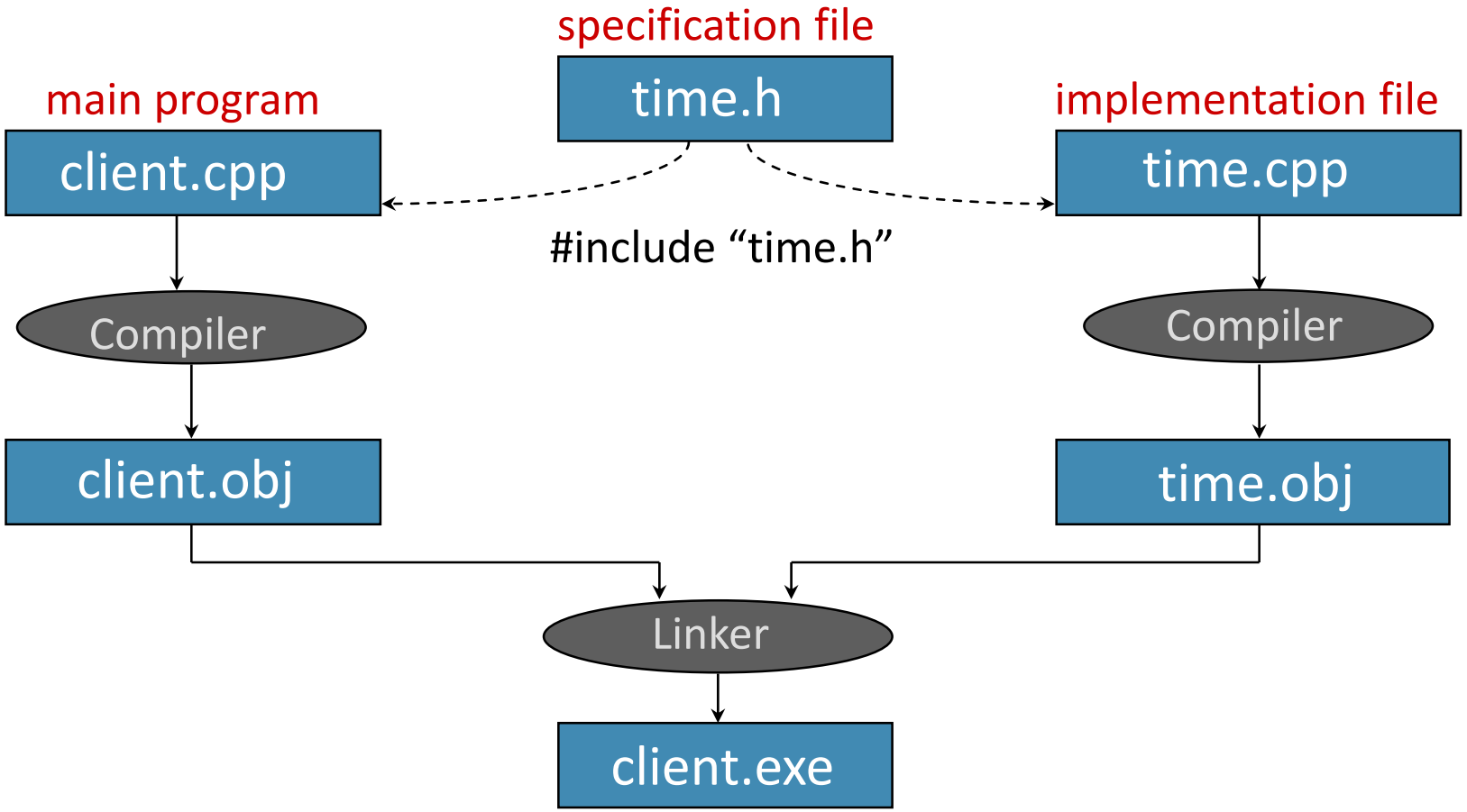
These statements declare myInfile as an instance of class ifstream and invoke function member open :

```
ifstream  myInfile;
myInfile.open("mydata.dat");
```

LECTURE 5

Each Class a Module

Separate Compilation and Linking of Files



Avoiding Multiple Inclusion of Header Files

- Often several program files use the same header file containing typedef statements, constants, or class type declarations
- But, it is a **compile-time error** to define the same identifier twice within the same namespace

Avoiding Multiple Inclusion of Header Files

This preprocessor directive syntax is used to avoid the compilation error that would otherwise occur from multiple uses of #include for the same header file

```
#ifndef   Preprocessor_Identifier
#define  Preprocessor_Identifier
        .
        .
        .
#endif
```

Example Using Preprocessor Directive `#ifndef`

```
// time .h
// Specification file
#ifndef   TIME_H
#define  TIME_H

class Time
{
 public:
  . . .

 private:
     . . .
};
#endif
```

For compilation the class declaration in

File time.h will be included only once

```
// time .cpp
// IMPLEMENTATION FILE
#include “time.h”
          . . .
```

```
// client.cpp
// Appointment program
#include “time.h”
   int   main (void)
   {
         . . .
   }
```

LECTURE 6

Constructors

Class Constructors

- A class constructor – a member function whose purpose is to initialize the private data members of a class object
- The name of a constructor is always the name of the class, and there is no return type for the constructor

Class Constructors

- A class may have several constructors with different parameter lists
- A constructor with no parameters is the default constructor
- A constructor is implicitly invoked when a class object is declared
- If there are parameters, their values are listed in parentheses in the declaration

Specification of `Time` Class Constructors

```
class  Time   // Time.h
{
public :      //  7 function members
   Time(int hours, int minutes, int seconds);
   void Increment();
   void Set(int hours, int minutes, int seconds) const;
   bool Equal(Time otherTime)  const;
   bool LessThan(Time otherTime)  const;
```

Specification of **Time** Class Constructors

```
// Parameterized constructor
   Time (int initHrs, int initMins, int initSecs);
   // Default constructor
   Time();
 private : //  3 data members
   int hrs;
   int mins;
   int secs;
};
```

Implementation of `Time` Default Constructor

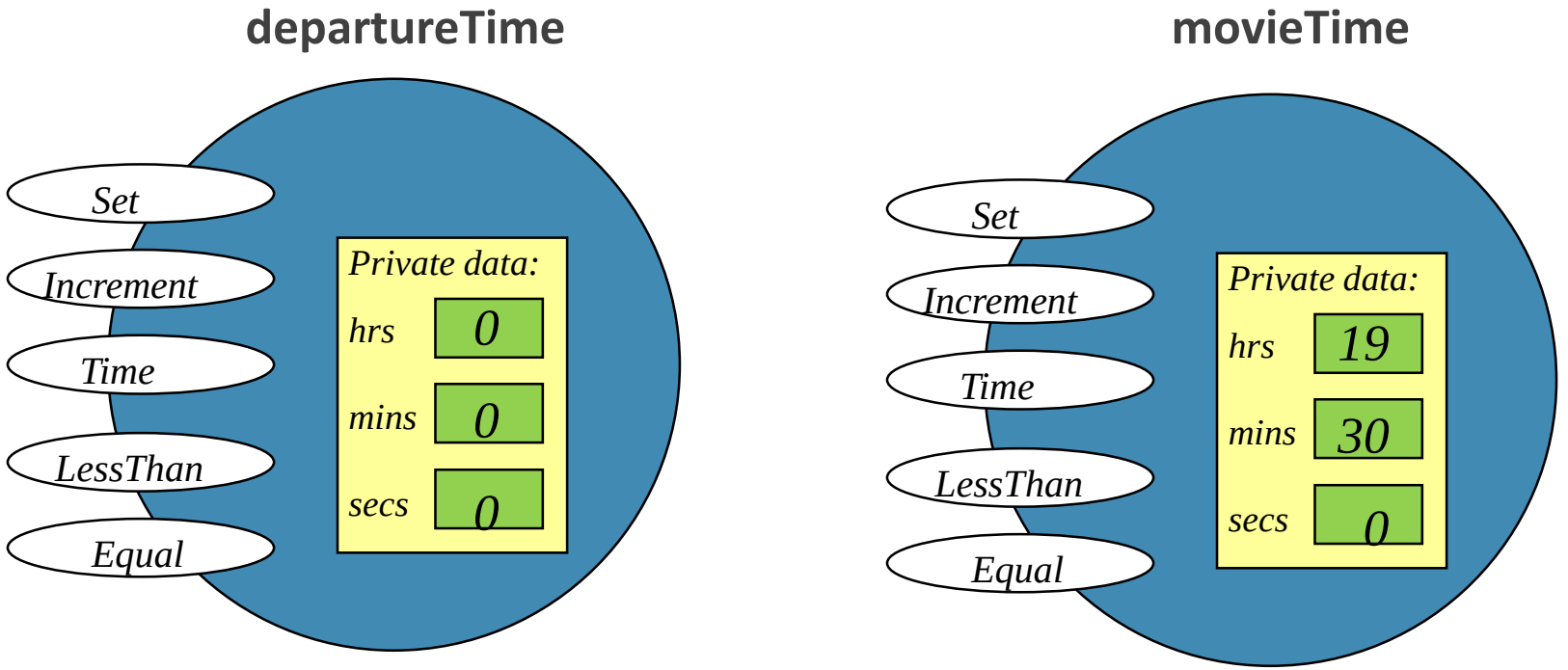
```
Time::Time ()
// Default  Constructor
// Postcondition:
//        hrs == 0 && mins == 0 &&  secs == 0
{
   hrs  = 0;
   mins = 0;
   secs = 0;
}
```

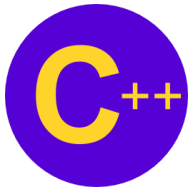
Parameterized Constructor

```
Time::Time( /* in */   int   initHrs,
            /* in */   int   initMins,
            /* in */   int   initSecs)
//  Constructor
//  Precondition:
//     0 <= initHrs <= 23 && 0 <= initMins <= 59
//     0 <= initSecs <= 59
//  Postcondition:
//                    hrs == initHrs && mins == initMins
//   && secs == initSecs
{
  hrs  =  initHrs;
  mins =  initMins;
  secs =  initSecs;
}
```

Automatic invocation of constructors occurs

```
Time departureTime;  // Default constructor invoked

Time movieTime (19, 30, 0); // Parameterized constructor
```





new MyClass() to allocate memory and create pointers
Myclass object to create struct-like objects. object is a reference

```
Myclass *object = new Myclass(); //object is on the heap
Myclass object; //object is on the stack


Myclass *object = new Myclass();  //declares pointer to class.
Myclass object;   //declares object of class Myclass

pointer_to_class->member;  // accessing class member using pointer to class
object.member;             //accessing class member using object of class
```

LECTURE 7

Header Files

What should be put into the Header Files?

Header files (.h) are designed to provide the information that will be needed in multiple files.

Things like

- **class declarations**,
- **function prototypes**, and
- **enumerations**

typically go in header files. In a word, "definitions".

```
#ifndef DetectRegions_h
#define DetectRegions_h
#include "Plate.h"

using namespace std;

class DetectRegions{
    public:
        DetectRegions();
        string filename;
        void setFilename(string f);
        bool saveRegions;
        bool showSteps;
        vector<Plate> run(Mat input);
    private:
        vector<Plate> segment(Mat input);
        bool verifySizes(RotatedRect mr);
        cv::Mat histeq(Mat in);
};

#endif
```

What should be put into the Implementation files?

Code files (.cpp) are designed to provide the implementation information that only needs to be known in one file.

In general,

- **function bodies**, and
- **internal variables** that should/will never be accessed by other modules,

are what belong in .cpp files. In a word, "implementations".

How to determine where should the code be?

The simplest question to ask yourself to determine what belongs where is

"if I change this, will I have to change code in other files to make things compile again?"

If the answer is "yes" it probably belongs in the header file; if the answer is "no" it probably belongs in the code file.

Header Files Contents

In C++, there are some other things that could be put in the header because, they need, too, be shared:

- inline code
- templates
- constants (usually those you want to use inside switches...)

Move to the header EVERYTHING what need to be shared, including shared

Then, it means there could be sources in the headers?

- Yes. In fact, there are a lot of different things that could be inside a "header" (i.e. shared between sources).
  - Forward declarations
  - declarations/definition of functions/structs/classes/templates
  - implementation of inline and templated code
- It becomes complicated, and in some cases (circular dependencies between symbols), impossible to keep it in one header.

Headers could be broken down into three parts

This means that, in an extreme case, you could have:

- a forward declaration header
- a declaration/definition header
- an implementation header

Implementation File: (.cpp)

- an implementation source

```
// - - - - MyObject_forward.hpp - - - -
// This header is included by the code which need to know MyObject
// does exist, but nothing more.
template<typename T>
class MyObject ;
```

```cpp
// - - - - MyObject_declaration.hpp - - - -
// This header is included by the code which need to know how
// MyObject is defined, but nothing more.
#include <MyObject_forward.hpp>

template<typename T>
class MyObject
{
   public :
      MyObject() ;
   // Etc.
} ;

void doSomething() ;
```

```cpp
// - - - - MyObject_implementation.hpp - - - -
// This header is included by the code which need to see
// the implementation of the methods/functions of MyObject,
// but nothing more.
#include <MyObject_declaration.hpp>

template<typename T>
MyObject<T>::MyObject()
{
   doSomething() ;
}

// etc.
```

```cpp
// - - - - MyObject_source.cpp - - - -
// This source will have implementation that does not need to
// be shared, which, for templated code, usually means nothing...
#include <MyObject_implementation.hpp>

void doSomething()
{
   // etc.
} ;

// etc.
```

Other things

1. Constant from #define to **const**
2. #define macro to inline function
3. Remove extern
4. typdef
5. Enumerations
6. struct
7. classes

LECTURE 8

Initialization and Finalization (Constructor/Reference)

Three Types of Object Life-Cycle

- C
  - alloc() – use – free()
- C++
  - new() – constructor() – use – destructor()
- Java
  - new() – constructor() – use – [ignore / garbage collection]

Initialization and Finalization

The lifetime of an object is defined to be the interval during which it occupies space and can hold data

- Most object-oriented languages provide some sort of special mechanism to **initialize** an object automatically at the beginning of its lifetime
  - When written in the form of a subroutine, this mechanism is known as a **constructor**
  - A constructor does not allocate space
- A few languages provide a similar **destructor** mechanism to **finalize** an object automatically at the end of its lifetime

Initialization and Finalization

1. Choosing a constructor
2. References and values
   - If variables are references, then every object must be created explicitly - appropriate constructor is called
   - If variables are values, then object creation can happen implicitly as a result of elaboration
3. Execution order
   - When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
4. Garbage collection

Declarations and Constructors in C++

Temporary Objects (Reference)

Declarations and Constructors in C++:

```
foo   b;       // calls foo::foo()
```

If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor.

Declarations and Constructors with parameters in C++:

```
foo   b(10, 'x');       // calls foo::foo(int, char)
foo   c{10, 'x'};       // alternative syntax in C++11
```

Declarations and Constructors with Reference in C++:

```
foo   a;
…
foo   b(a);    // calls  foo::foo(foo&)
foo  c{a} ;    // alternative syntax
```

Copying of Objects:

```
foo   a;       // calls foo::foo()
…
foo   b = a;  // calls  foo::foo(foo&)
```

In recognition of this intent, a single-argument constructor in C++ is called a **copy constructor**. It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment. The effect is not the same as that of the similar code fragment.

Assignment:

```
foo   a, b;       // calls foo::foo() twice
…
b = a;       // calls  foo::operator=(foo&)
```

This is assignment not initialization.

Declarations and Constructors in C++

Temporary Objects (Reference)

In **C++**, the requirement that every object be constructed (and likewise destructed) applies not only to objects with names but also to temporary objects. The following, for example, entails a call to both the **string(const char*)** constructor ad the **~string()** destructor:

```
cout<< string("Hi, Mom").length();
```

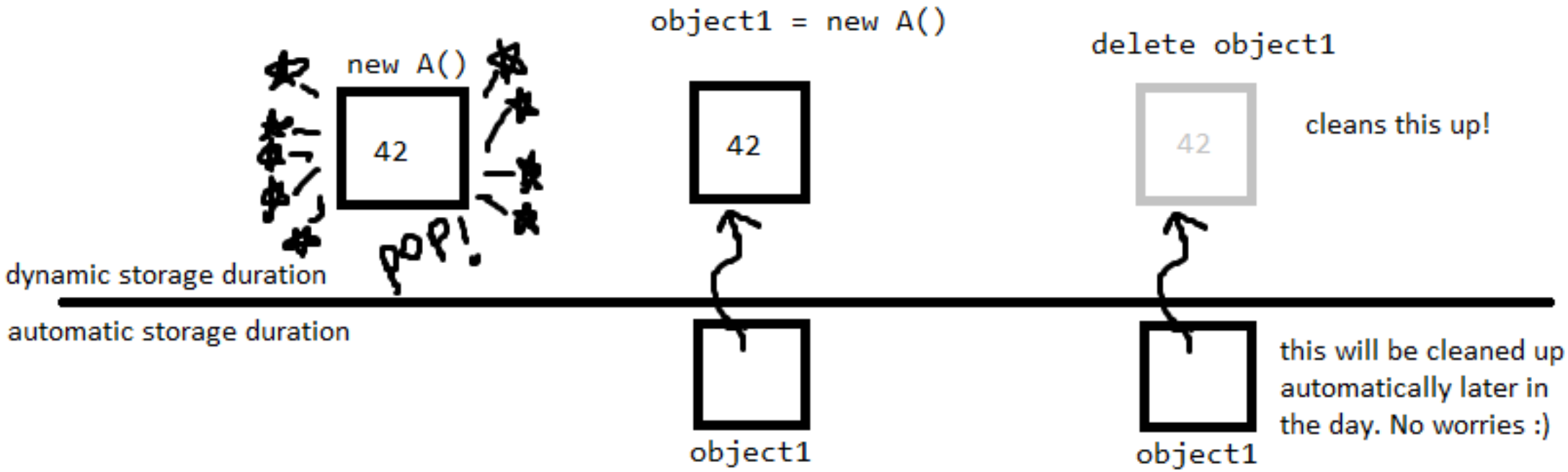
The destructor called at the end of the output statement: the temporary objects behaves as if its sop were just the line shown here.

The following entails not only two calls to the default string constructor and a call to string::operator(), but also constructor call to initialize the temporary object returned by operator() – the object whose length is then queried by the caller:

```
string a, b;
…
(a+b).length();
```

new Operator:

Object Created by memory allocation in heap.



Objects in C++

```
// 1. Naive C++ style
Object* p = new Object();
p->Use();
delete p;

// 2. More secure C++98 with RAII
std::auto_ptr<Object> ap(new Object());
ap->Use();

// 3. More secure C++11 with RAII
std::unique_ptr<Object> up(new Object());
up->Use();

// 4. New bullet-proof modern C++
auto up2 = std::make_unique<Object>();
up2->Use();
```

LECTURE 9

Object as return value

Return Value Optimization

- **f** is a function returning a value of class type **foo**.
- If instance of **foo** are too big to fit in a register, the compiler will arrange for **f**'s caller to pass an extra, hidden parameter that specifies the locations into which **f** should construct the return value. **(a mail box for return value)**
- If the return statement itself creates a temporary object -

```
return foo(args)
```

- **f**'s source looks more like this:

```
foo rtn;
…
return rtn;
```

Note: This option is known as Return value optimization.

Return Value Optimization

- In other programs the compiler may need to invoke a copy constructor after a function returns:

```
foo c;
…
c = foo(args);
```

- The location of **c** cannot be passed to the hidden parameter to unless the compiler is able to prove that **c**'s value will not be used during the call.
- **The bottom line:** returning an object from a function in C++ may entail zero, one or two invocations of return type's copy. Constructor, depending on whether the compiler is able to optimize either or both of the return statement and the subsequent use in the caller.

LECTURE 10

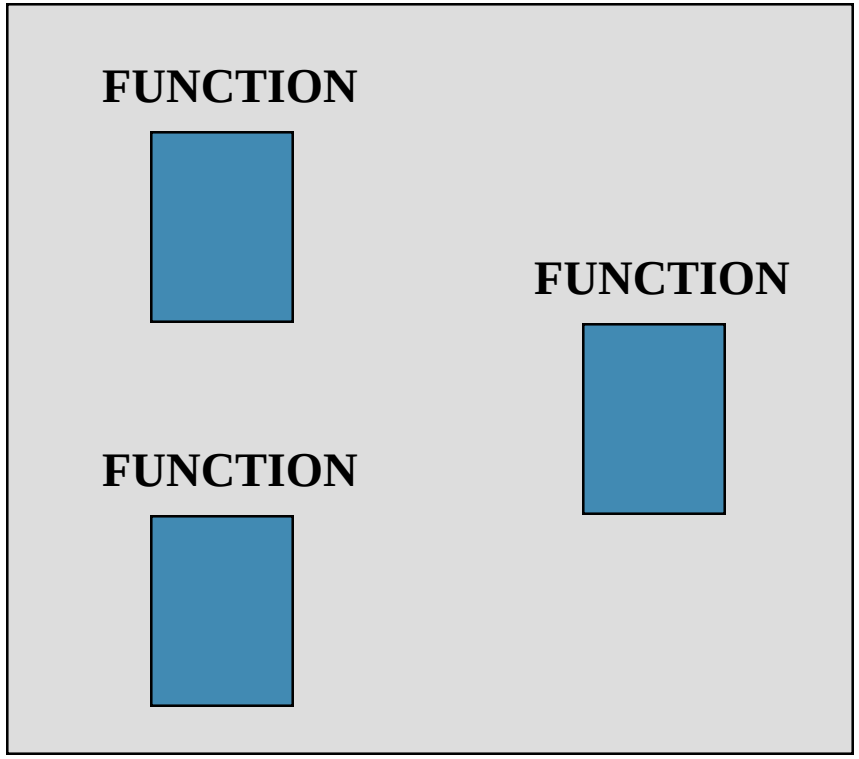
Inter-class Relationship

Inter-Class Relationship

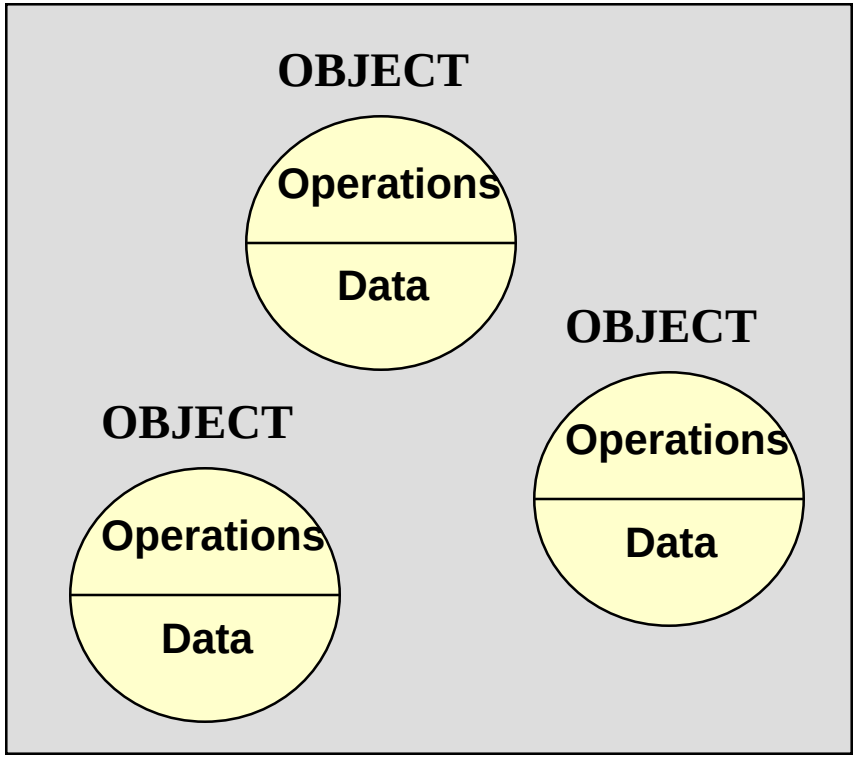
- Structured Programming vs. Object-Oriented Programming
- Using Inheritance to Create a New C++ class Type
- Using Composition (Containment) to Create a New C++ class Type
- Static vs. Dynamic Binding of Operations to Objects
- Virtual Member Functions

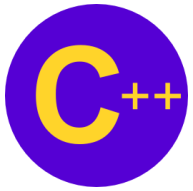
Two Programming Paradigms

Structural (Procedural) PROGRAM



Object-Oriented PROGRAM



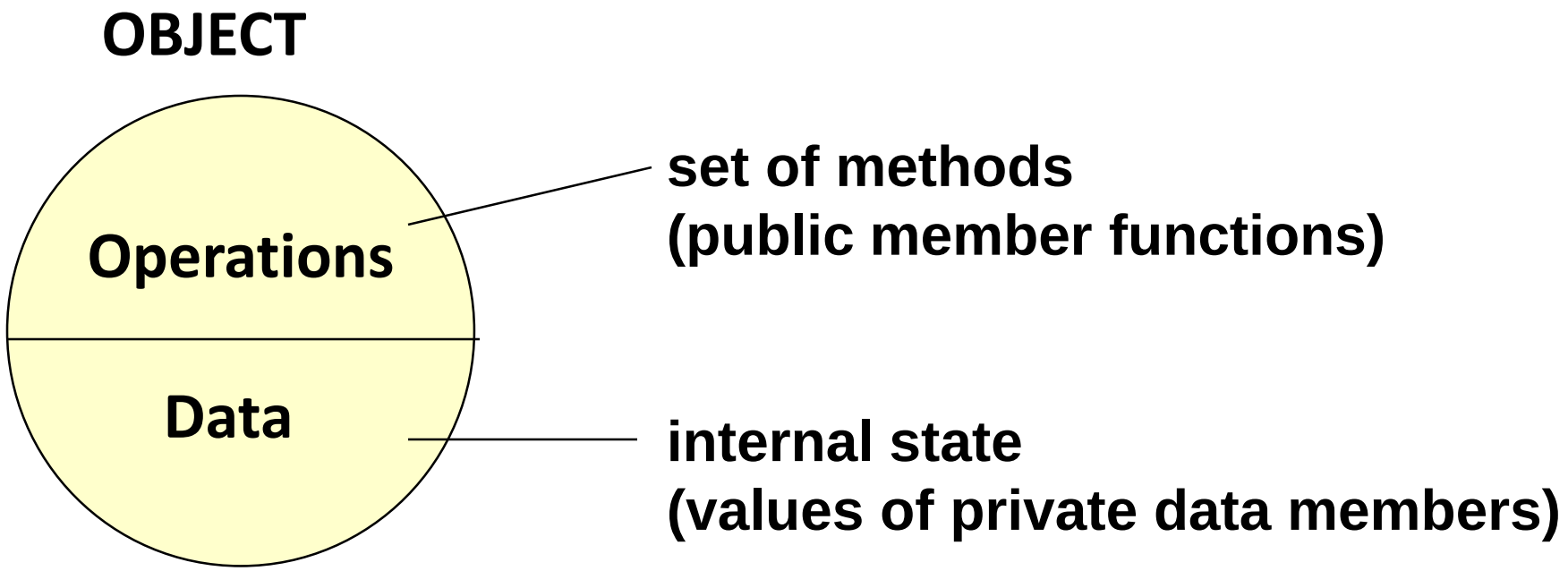


Object-Oriented Programming Language Features

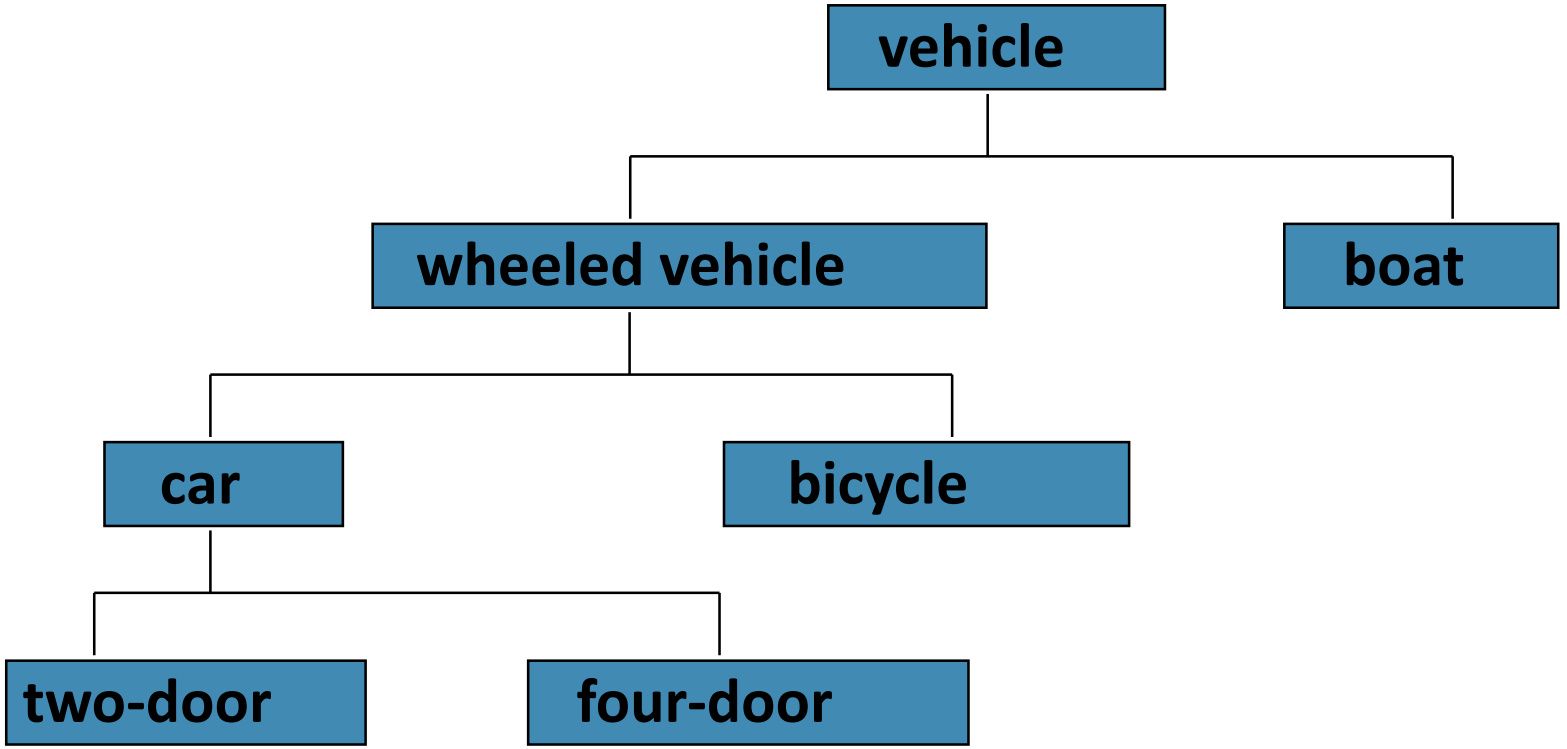
1. Data abstraction
2. Inheritance of properties
3. Dynamic binding of operations to objects

OOP Terms	C++ Equivalents
Object	Class object or class instance
Instance variable	Private data member
Method	Public member function
Message passing	Function call (to a public member function)

What is an object?



Inheritance Hierarchy Among Vehicles



Every car is a wheeled vehicle.

LECTURE 11

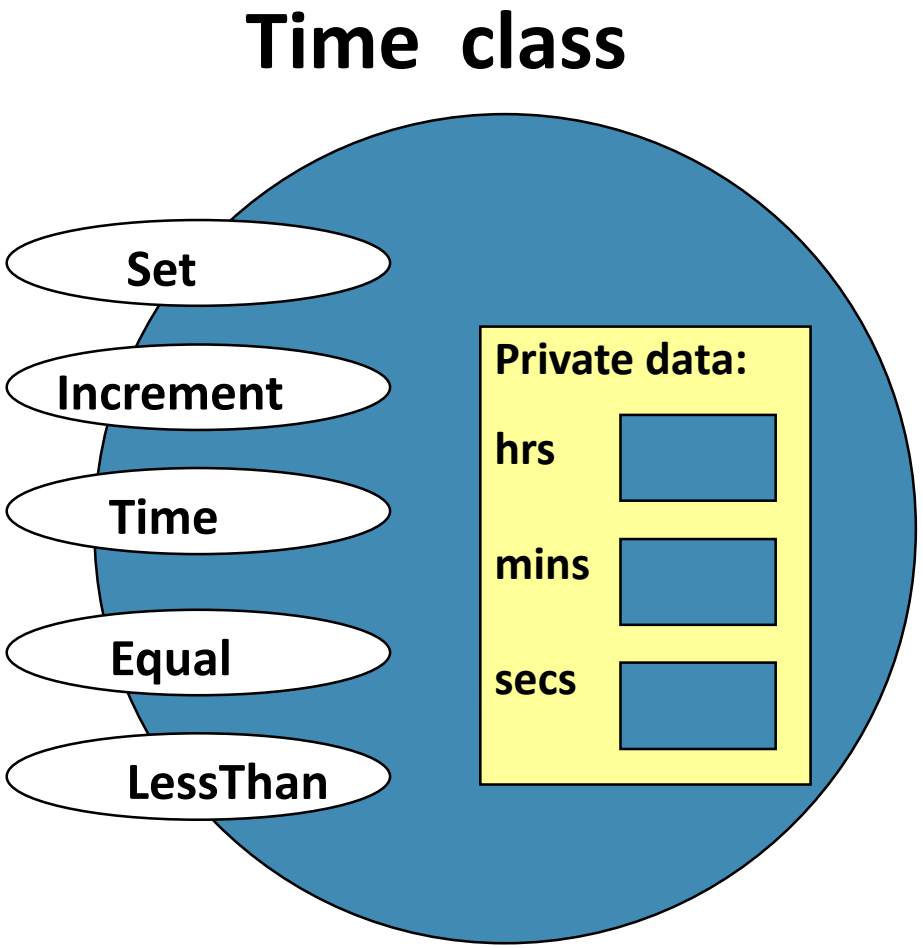
Inheritance

Inheritance

Inheritance is a mechanism by which one class acquires (inherits) the properties (both data and operations) of another class

- The class being inherited from is the **Base Class** (Superclass)
- The class that inherits is the **Derived Class** (Subclass)
- The derived class is specialized by adding properties specific to it

Class Interface Diagram



```
//#define MAIN   // comment this line out if not __main__ function
#ifndef TIME_H
#define TIME_H
#include <iostream>
#include <string>
using  namespace  std;
/* to_string patch:
*/
#include <sstream>
namespace st{
   template < typename T > std::string to_string( const T& n ){
      std::ostringstream stm ;
      stm << n ;
      return stm.str() ;
   }
}
//  Specification file (Time.h)
class  Time    // Declares a  class data type
{       //  does not allocate memory
 public :       // Five public function members
   Time(int  h , int  m , int  s) ;
   Time();
   void  Increment ();
   void  Set(int h, int m, int s);
   int   GetHours();
   int   GetMins();
   int   GetSecs();
   void   SetHours(int hours);
   void   SetMins(int minutes);
   void   SetSecs(int seconds);
   bool  Equal (Time   otherTime) ;
   bool  LessThan (Time   otherTime);
   string to_string();
   bool operator==(const Time &rhs) const { return hrs == rhs.hrs && mins == rhs.mins && secs == rhs.secs; }

 private :       // Three private data members
     int  hrs;
     int  mins;
     int  secs;
};
#endif
```

`class Time` Specification

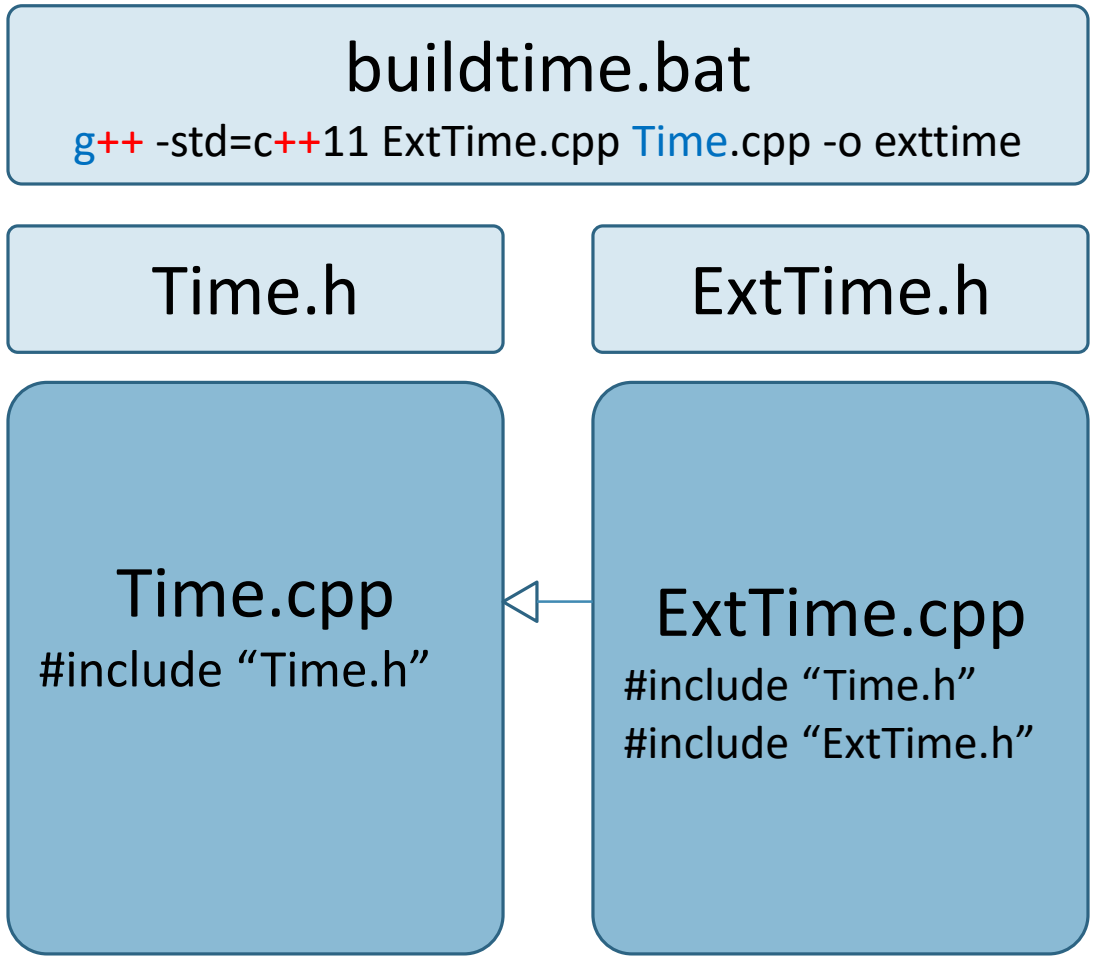
1. Add Default Constructor (must-have) and its implementation.
2. Accessor/Mutator (Get/Set) (must-have) because of private date fields.
3. to_string should be defined at top-level class (base-class)
4. Main disabled by undefine the MAIN constant

`class Time` Implementation

1. main() disabled. Setters added. Default Constructor Defined.

```
#include <iostream>
#include "time.h" // Includes specification of the class
#include <string>
using namespace std;
Time::Time (int h=0 , int m=0 , int s=9) : hrs(h), mins(m), secs(s) {
    hrs = h;
    mins = m;
    secs = s;
}
Time::Time(){
}
void Time::Increment (){
  secs++;
  mins += secs/60;
  secs %= 60;
  hrs  += mins/60;
  mins %= 60;
  hrs  %=24;
}
int Time::GetHours(){ return hrs; }
int Time::GetMins(){ return mins; }
int Time::GetSecs(){return secs; }
void Time::SetHours(int hours){
  hrs = hours;
}
void Time::SetMins(int minutes){
  mins = minutes;
}
void Time::SetSecs(int seconds){
  secs = seconds;
}
void Time::Set(int h, int m, int s) {
    hrs = h;
    mins = m;
    secs = s;
}
bool Time::Equal (Time  otherTime) {
  int h1 = GetHours();
  int h2 = otherTime.GetHours();
  int m1 = GetMins();
  int m2 = otherTime.GetMins();
  int s1 = GetSecs();
  int s2 = otherTime.GetSecs();
  if (h1 == h2 && m1 == m2 && s1 == s2) return true;
  return false;
}
```

```
bool Time::LessThan (Time  otherTime) {
  int h1 = GetHours();
  int h2 = otherTime.GetHours();
  int m1 = GetMins();
  int m2 = otherTime.GetMins();
  int s1 = GetSecs();
  int s2 = otherTime.GetSecs();
  if (h1 < h2) return true;
   else if (h1 == h2){
          if (m1 < m2) return true;
          else if (m1 == m2){
            if (s1 < s2) return true;
            else if (s1 == s2) return false;
                 else return false; }
          else return false; }
   else return false;
}
string Time::to_string(){
  string rtn = "";
  rtn += "Time: " + st::to_string(this->GetHours()) + ":" + st::to_string(this->GetMins()) + ":" + st::to_string(this->GetSecs());
  return rtn;
}
#ifdef MAIN
int main (){
  Time currentTime; // Declares two objects of Time
  Time endTime;
  bool done = false;

  currentTime.Set (5, 30, 0);
  endTime.Set (5, 30, 5);
  while (! done)
  {
    currentTime.Increment ();
    cout<< currentTime.to_string() << endl;
    if (currentTime.Equal (endTime))
      done = true;
    if (currentTime == endTime)   cout << "Program Finished" << endl;
  };
}
#endif
```



Note: ExtTime inherits Time.

```
1   #define MAIN   // comment this line out if not __main__ function
2   #ifndef EXTTIME_H
3   #define EXTTIME_H
4   #include <iostream>
5   #include <string>
6   #include  "time.h"
7   enum ZoneType{EST, CST, MST, PST, EDT, CDT, MDT, PDT};
8   //  Specification file (Time.h)
9   class  ExtTime: public Time   // Declares a  class data type
10  {         //  does not allocate memory
11    public :        // Six public function members
12      ExtTime(int hours, int minutes, int seconds, ZoneType  timeZone) ;
13      ExtTime();
14      void Set(int hours, int minutes, int seconds, ZoneType  timeZone);
15      string GetTimeZoneString();
16      ZoneType GetTimeZone();
17      void SetZone(ZoneType z);
18      string to_string();
19    private :
20      ZoneType  zone; // Additional data member
21  };
22  #endif
```

class ExtTime (Spec)

1. Define enum type ZoneType in .h file.
2. New Constructor which will call Time()
3. Setter (Mutator) and Accessor for this class.
4. Overridden to_string() method.
5. ExtTime extends Time class

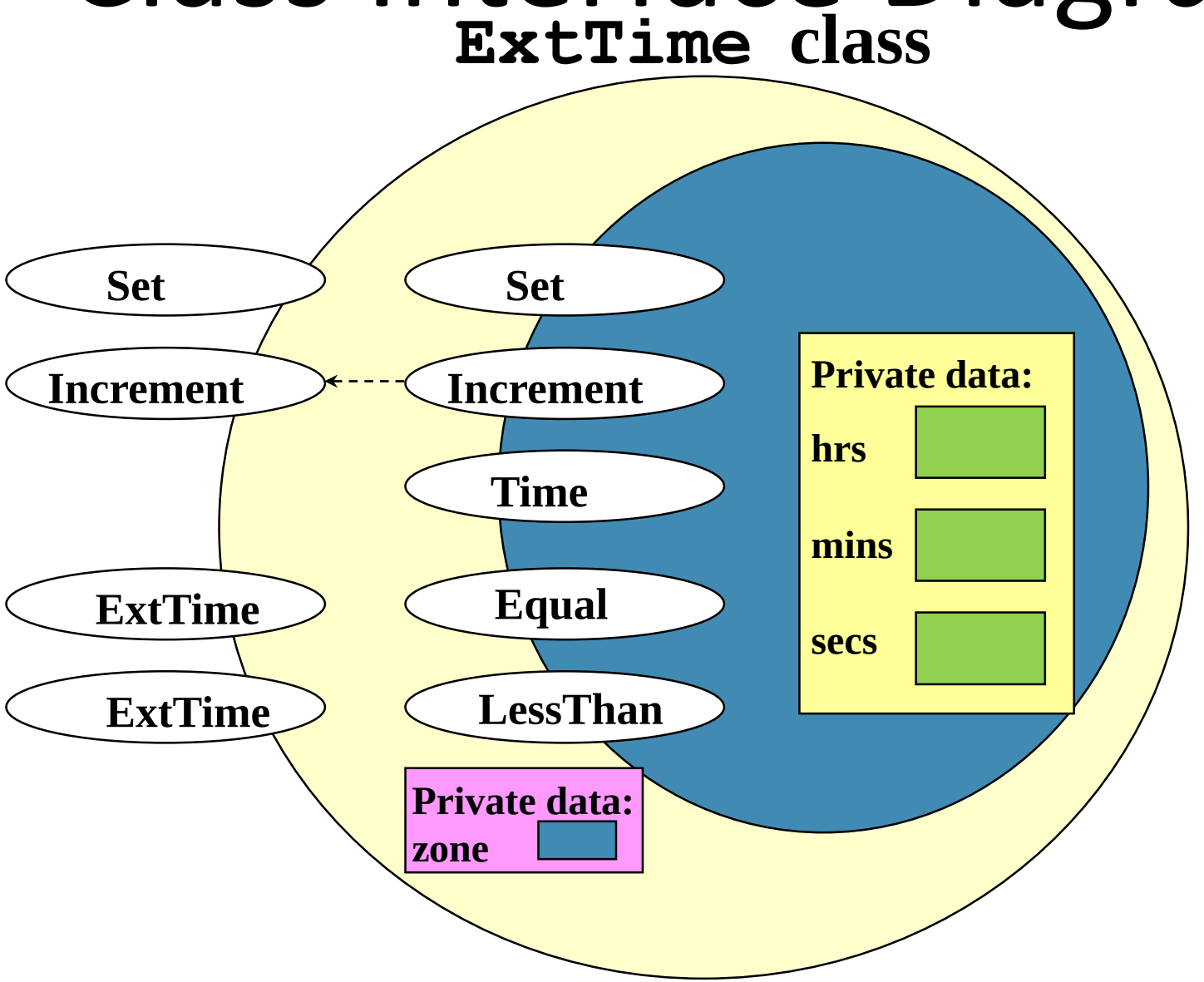
```cpp
#include <iostream>
#include  "exttime.h"  // Includes specification of the class
#include "time.h"
#include <string>
using  namespace  std;
ZoneType ExtTime::GetTimeZone(){
   return zone;
}
string ExtTime:: GetTimeZoneString(){
   string rtn="";
   switch (zone){
      case EST: rtn="EST";   break;
      case CST: rtn="CST";   break;
      case MST: rtn="MST";  break;
      case PST: rtn="PST";   break;
      case EDT: rtn="EDT";  break;
      case CDT: rtn="CDT";   break;
      case MDT: rtn="MDT";  break;
      case PDT: rtn="PDT";   break;
   }
   return rtn;
}
void ExtTime::SetZone(ZoneType z){
    zone = z;
}
ExtTime::ExtTime(int hours, int minutes, int seconds, ZoneType  timeZone) {
   SetHours(hours);
   SetMins(minutes);
   SetSecs(seconds);
   SetZone(timeZone);
}
ExtTime::ExtTime (){
}
```

```
34  void ExtTime::Set(int hours, int minutes, int seconds, ZoneType  timeZone){
35      SetHours(hours);
36      SetMins(minutes);
37      SetSecs(seconds);
38      SetZone(timeZone);
39  }
40  string ExtTime::to_string(){
41       string rtn = "";
42       rtn += "Time: " + st::to_string(GetHours()) + ":" + st::to_string(GetMins()) + ":" + st::to_string(GetSecs())+"-"+GetTimeZoneString();
43      return rtn;
44  }
45  #ifdef MAIN
46  int  main (){
47      ExtTime  currentTime; // Declares two objects of Time
48      ExtTime  endTime;
49      bool  done  =  false;
50
51      currentTime.Set (5, 30, 0, PDT);
52      endTime.Set (5, 30, 5, PDT);
53      while  (! done)
54      {
55          currentTime.Increment ();
56          cout<<  currentTime.to_string() << endl;
57          if  (currentTime.Equal (endTime))
58              done  =  true;
59          if (currentTime == endTime)   cout << "Program Finished" << endl;
60      };
61  }
62  #endif
```

class ExtTime:public Time

- Says class Time is a public base class of the derived class ExtTime
- As a result, all public members of Time (except constructors) are also public members of ExtTime
- In this example, new constructors are provided, new data member zone is added, and member functions Set and Write are overridden

Class Interface Diagram



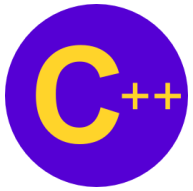
Accessibility

Inheritance does not imply accessibility.

- The derived class inherits all the members of its base class, both public and private
- The derived class (its member functions) cannot access the private members of the base class

Client Code Using `ExtTime`

```
#include  "exttime.h"
          .
          .
          .
ExtTime   thisTime ( 8, 35, 0, PST);
ExtTime   thatTime; // Default constructor called

string s1 = thatTime.to_string(); // Outputs 00:00:00  EST
cout << s1 <<  endl;

thatTime.Set (16, 49, 23, CDT);
string s2 =thatTime.to_string(); // Outputs 16:49:23  CDT
cout << s2 << endl;

thisTime.Increment();
thisTime.Increment();
string s3 = thisTime.to_string(); // Outputs 08:35:02  PST
cout << s3 << endl;
```



Constructor Rules for Derived Classes

- At run time, the base class constructor is implicitly called first, before the body of the derived class's constructor executes
- If the base class constructor requires parameters, they must be passed by the derived class's constructor

Execution Sequences of Constructors

Statements	Sequence Of Execution	Remarks
Class II : public I	I() – Base class constructor II() – Derived class constructor	Single inheritance
Class III: public I, II	I() – Base class constructor II() – Base class constructor III() – Derived class constructor	Multiple inheritance
Class III: public I, virtual II	II() – Virtual class constructor I() – Base class constructor III() – Derived class constructor	Multiple inheritance
class III: public II	I() – First base class constructor II() – Second base class constructor III() – Derived class constructor	Multilevel inheritance

Implementation of `ExtTime` Default Constructor

```
ExtTime::ExtTime ( )
// Default  Constructor
// Postcondition:
//         hrs == 0   &&   mins == 0   &&  secs == 0
//      (via an implicit call to base class default
//       constructor)
//      &&  zone == PDT
{
    zone  =  PDT;
}
```

Implementation of Another `ExtTime` Class Constructor

```
ExtTime::ExtTime(int initHrs, int initMins,
                 int initSecs, ZoneType initZone)
           : Time (initHrs, initMins, initSecs)
// Constructor initializer
// Pre: 0 <= initHrs <= 23 && 0 <= initMins <= 59
//         0 <= initSecs <= 59 && initZone is assigned
// Post: zone == initZone && Time set by base
//       class constructor
{
    zone  = initZone;
}
```

Implementation of `ExtTime::Set` function

```
void  ExtTime::Set  (int initHrs,int initMins,
                     int initSecs,ZoneType initZone)
// Pre: 0 <= initHrs <= 23 && 0 <= initMins <= 59
//           0 <= initSecs <= 59 && initZone is assigned
// Post: zone == timeZone  && Time set by base
//       class function
{
    Time::Set (initHrs, initMins, initSecs);
    zone  = initZone;
}
```

Implementation of `ExtTime::to_string()` Function

```
string ExtTime::to_string(){
   string rtn = "";
   rtn += "Time: " + st::to_string(GetHours()) + ":" +
  st::to_string(GetMins()) + ":" +
  st::to_string(GetSecs()) + "-" + GetTimeZoneString();
  return rtn;
}
```

Responsibilities

- Responsibilities are operations implemented as C++ functions
- Action responsibilities are operations that perform an action
- Knowledge responsibilities are operations that return the state of private data variables

What responsibilities are Missing?

- The Time class needs
  - int Hours()
  - int Minutes()
  - int Seconds()
- The ExtTime class needs
  - ZoneType zone()

Demo Program: exttime.cpp

buildtime, buildclient (with client code)

Go Notepad++!!!

Note:
buildtime.bat (self-testing using ExtTime.cpp's main() code.
Need to turn on the MAIN in ExtTime.h
Buildclient.bat (with client code): Need to turn off the MAIN in ExcTime.h

LECTURE 12

Visibility

Visibility Rules

- Public and Private parts of an object declaration/definition
- 2 reasons to put things in the declaration
  - so programmers can get at them
  - so the compiler can understand them
- At the very least the compiler needs to know the size of an object, even though the programmer isn't allowed to get at many or most of the fields (members) that contribute to that size
  - That's why private fields have to be in declaration

C++

- C++ distinguishes among
  - public class members
    - accessible to anybody
  - protected class members
    - accessible to members of this or derived classes
  - private
    - accessible just to members of this class
- A C++ structure (*struct*) is simply a class whose members are public by default
- C++ base classes can also be public, private, or protected

C++Inheritance and Visibility

- Example:

```
class circle : public shape { ...
```

anybody can convert (assign) a circle* into a shape*

```
class circle : protected shape { ...
```

only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

```
class circle : private shape { ...
```

only members and friends of circle can convert (assign) a circle* into a shape*

Class

Visibility

- With the introduction of inheritance, object-oriented languages must supplement the scope rules of module-based languages to cover additional issues.
- For example,
  - should private members of a base class be visible to methods of a derived class?
  - Should public members of a base class always be public members of a derived class (i.e., be visible to users of the derived class)?
  - How much control should a base class exercise over the visibility of its members in derived classes?

Default Hiding:

In C++, the definition of class queue can specify that its base (list) class is to be private:

```
class queue : private list {
public:
    using list::empty;
    using list::head;
    // but NOT using list::append
    void enqueue(gp_list_node* new_node);
    gp_list_node* dequeue();
};
```

Note:
Sharing is an exception.

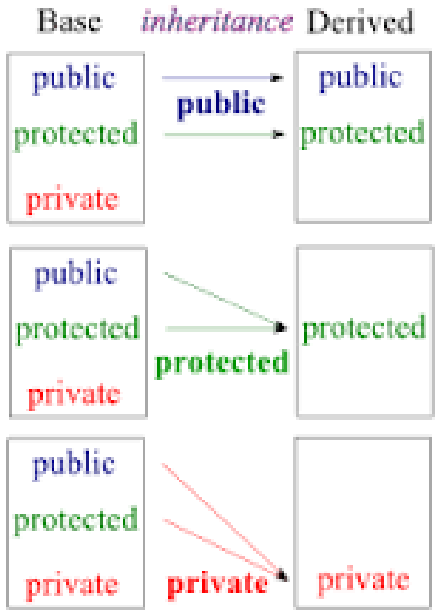
Class

Visibility

Inhibiting by delete:

```
class queue : public list {
  …
  void append(list_node * new_node) = delete;
}
```

Note:
Hiding is exceptional in this case.



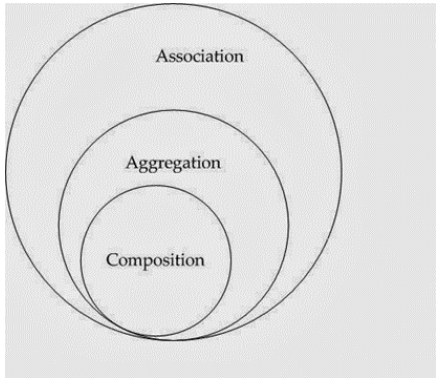
C++ Visibility

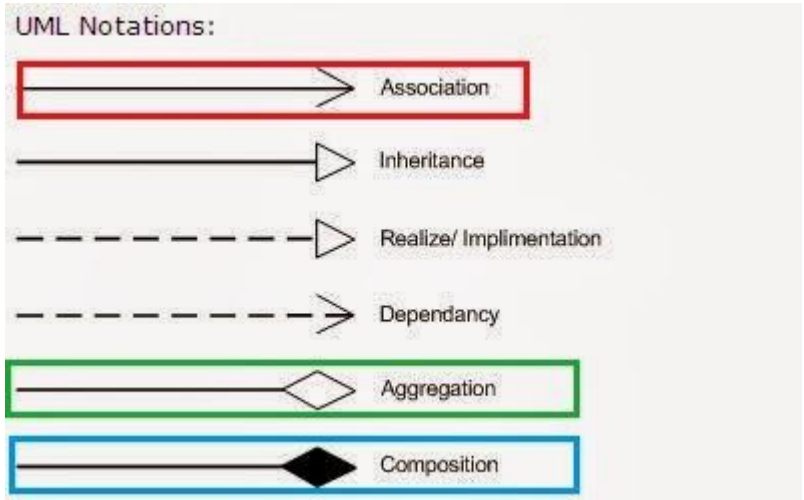
Base Class Visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

LECTURE 13

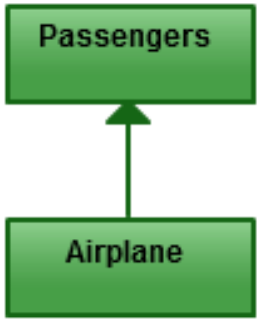
Composition has-A Relationship

Types of Class-To-Class Relationships

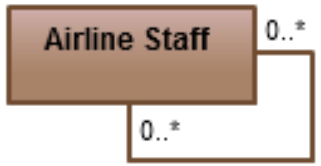




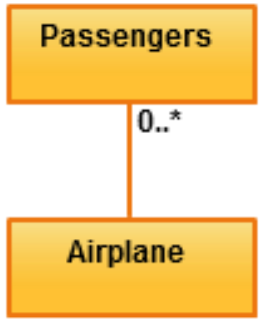
Association



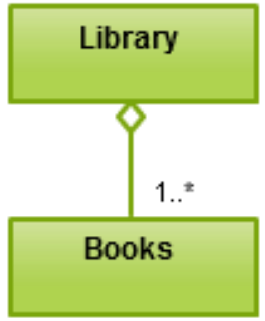
Directed Asscoation



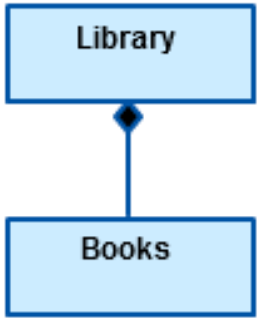
Reflexive Assciation



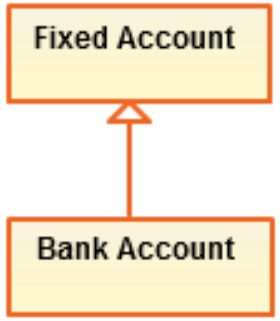
Multiplicity



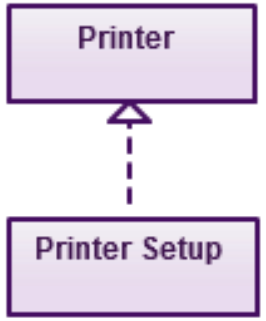
Aggregation



Composition



Inheritance



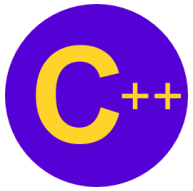
Realization

Composition (or Containment)

Composition (containment) is a mechanism by which the internal data (the state) of one class includes an object of another class

An Entry Object

```
#include "Time.h"
#include "Name.h"
#include <string>

class Entry{
  public:
    string NameStr() const;
    // Returns a string made up of first name and last name
    string TimeStr() const;
    // Returns a string made up of hour, colon, minutes
    Entry();      // Default constructor
    Entry(……)     // Parameterized constructor
  private:
    Name name;
    Time time;
}
```



Demo Program: Entry.cpp

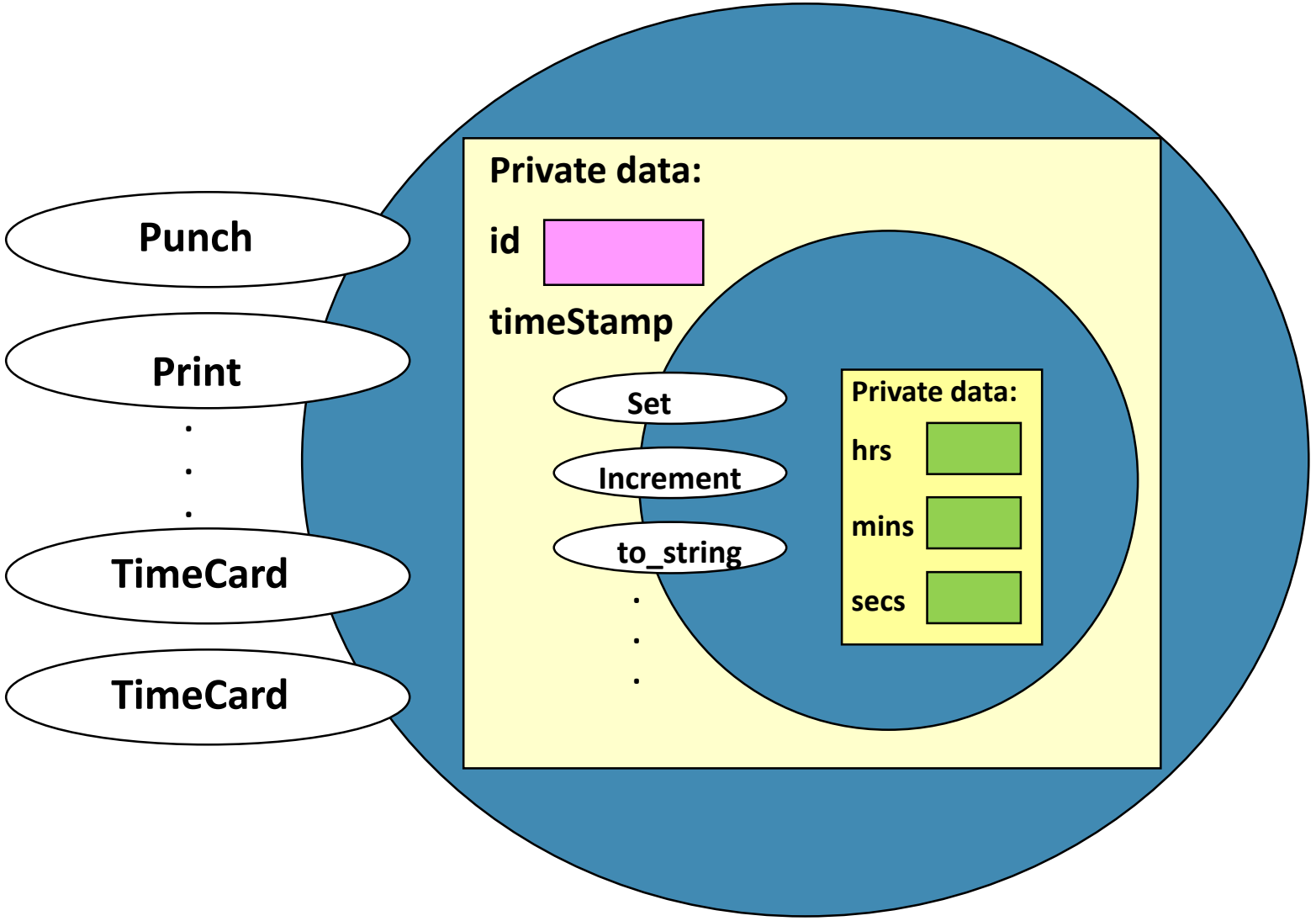
Go Notepad++!!!

A TimeCard object has a Time object

```
#include "time.h"

class TimeCard {
 public:
      void  Punch ( /* in */  int   hours,
                    /* in */  int   minutes,
                    /* in */  int   seconds ) ;
      void  Print ( ) const ;

      TimeCard  ( long   idNum, int initHrs,

                  int  initMins,  int initSecs ) ;
      TimeCard ( ) ;
  private:
      long    id ;
      Time   timeStamp ;
  } ;
```

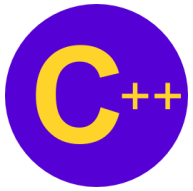
TimeCard Class

TimeCard has a Time object



Implementation of `TimeCard` Class Constructor

```
TimeCard :: TimeCard  ( /* in */   long   idNum,
                        /* in */   int    initHrs,
                        /* in */   int    initMins,
                        /* in */   int    initSecs )

   : timeStamp (initHrs, initMins, initSecs)    // constructor initializer

// Precondition:  0 <= initHrs <= 23   &&   0 <= initMins <= 59
//                                          0 <= initSecs <= 59   &&   initNum is assigned
// Postcondition:
//                              id == idNum  &&  timeStamp set by its constructor
{
        id  =  idNum ;
}
```



Demo Program: TimeCard.cpp

Go Notepad++!!!

LECTURE 14

Binding

Order in Which Constructors are Executed

Given a class X,

- if X is a derived class its **base class** constructor is executed first
- next, constructors for member objects (if any) are executed (using their own default constructors if none is specified)
- finally, the body of X's constructor is executed

In C++ . . .

When the type of a formal parameter is a parent class, the argument used can be

the same type as the formal parameter

or,

any descendant class type

Static Binding

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter(s)
- When pass-by-value is used, static binding occurs

Static Binding Is Based on Formal Parameter Type

```
void  Print ( /* in */ Time someTime)
{
     cout  <<  "Time is ";
     string s = someTime.to_string( );
     cout  << s <<  endl;
}
```

CLIENT CODE

```
Time startTime(8, 30, 0);
ExtTime endTime(10,45,0,CST);
Print ( startTime);
Print ( endTime);
```

OUTPUT

```
Time is 08:30:00
Time is 10:45:00
```

Dynamic Binding

- Dynamic binding is the run-time determination of which function to call for a particular object of a descendant class based on the type of the argument
- Feclaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding

Virtual Member Function

```
// Specification file ( "time.h")

class  Time
{

public:

   .  .  .

   virtual void to_string() const;
     // Forces dynamic binding, = 0 for abstract method

   .  .  .

private:

   int  hrs;
   int  mins;
   int  secs;
};
```

Dynamic binding requires pass-by-reference

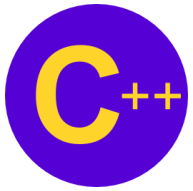
```
void Print   ( /* in */ Time& someTime)
{
     cout  <<  "Time is  ";
     string s = someTime.to_string( );
     cout  << s << endl;
}
```

CLIENT CODE

```
Time startTime ( 8, 30, 0);
ExtTime endTime (10,45,0,CST);
Print ( startTime);
Print ( endTime);
```

OUTPUT

```
Time is 08:30:00
Time is 10:45:00 CST
```



Polymorphic operation

An operation that has multiple meanings depending on the type of the object to which it is bound at run time, e.g., the `to_string` function

Using virtual functions in C++

- Dynamic binding requires pass-by-reference when passing a class object to a function
- In the declaration for a virtual function, the word virtual appears only in the base class
- If a base class declares a virtual function, it must implement that function, even if the body is empty
- A derived class is not required to re-implement a virtual function; if it does not, the base class version is used

Slicing Problem

When using pass by value

- A copy of the argument is sent to the parameter
- Only common data members are copied when an object of the child class is passed to an object of the parent class
- A child class is often “larger” than its parent

ExtTime		Time
(hrs, mins, secs, zone)	=>	(hrs, mins, secs)

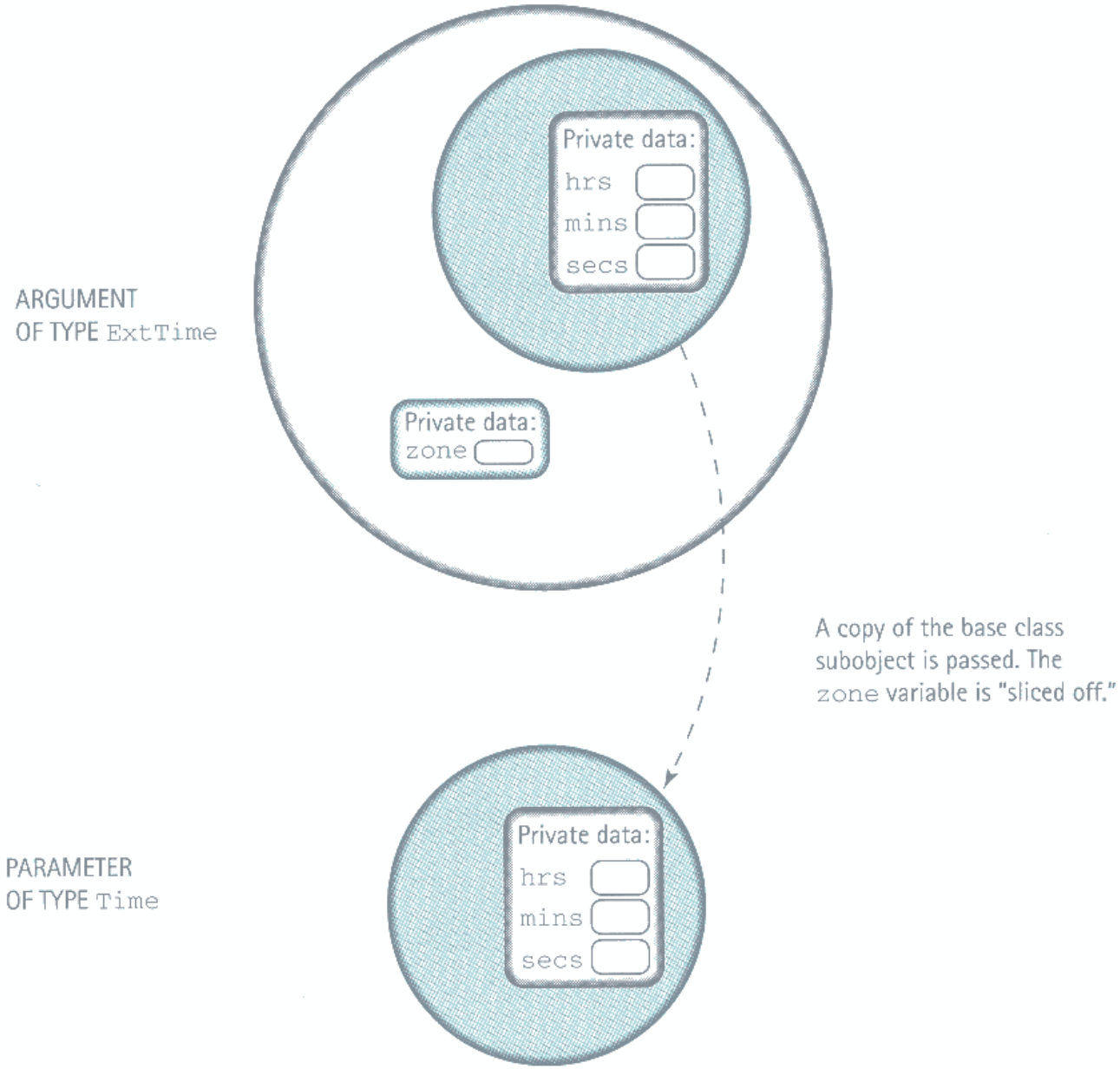


Figure 14–6 *The Slicing Problem Resulting from Passing by Value*

Slicing Problem

Pass by reference

Slicing problem also occurs with assignment:

```
parentClassObject =
 childClassObject
```

- No slicing with passing by reference:

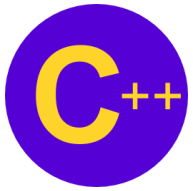
```
void Print( /* in */ Time&
 someTime )
```

- However, still the `to_string` function of the parent class is used in the call within the `Print` function:

```
string s = someTime.to_string();
```

LECTURE 15

Conclusion on Object and Class



Object-Oriented Design

- Identify the Objects and Operations
- Determine the relationship among objects
- Design and Implement the driver

Object-Oriented Design

Application (Problem) Domain:

The aspect of the world that we are modeling

- **Checking accounts**
- **Bank tellers**
- **Spreadsheet rows (columns)**
- **Robot arms (legs)**

Solution Domain:

The computer program that solves the real-life problem

- **Counters**
- **Lists**
- **Menus**
- **Windows**

Identify the Objects and Operations

The problem:

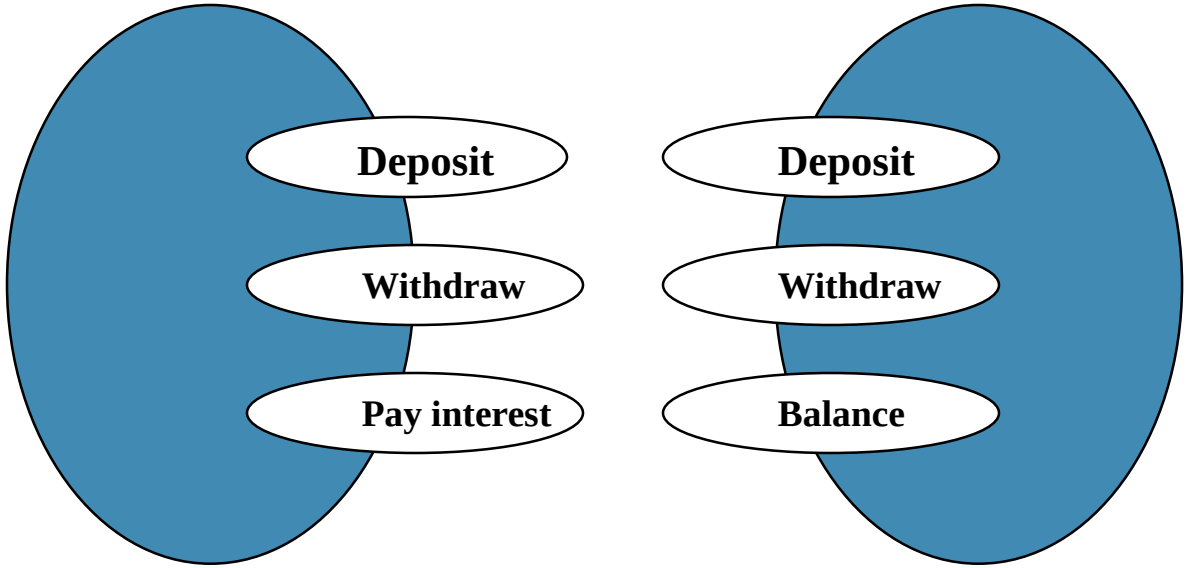
… The program must handle a customer's savings account. The customer is allowed to deposit funds into the account and withdraw funds from the account, and the bank must pay interest on a quarterly basis. …

Key nouns	Key phrases
Savings account Customer	Deposit funds Withdraw funds Pay interest

Object Table

Savings account

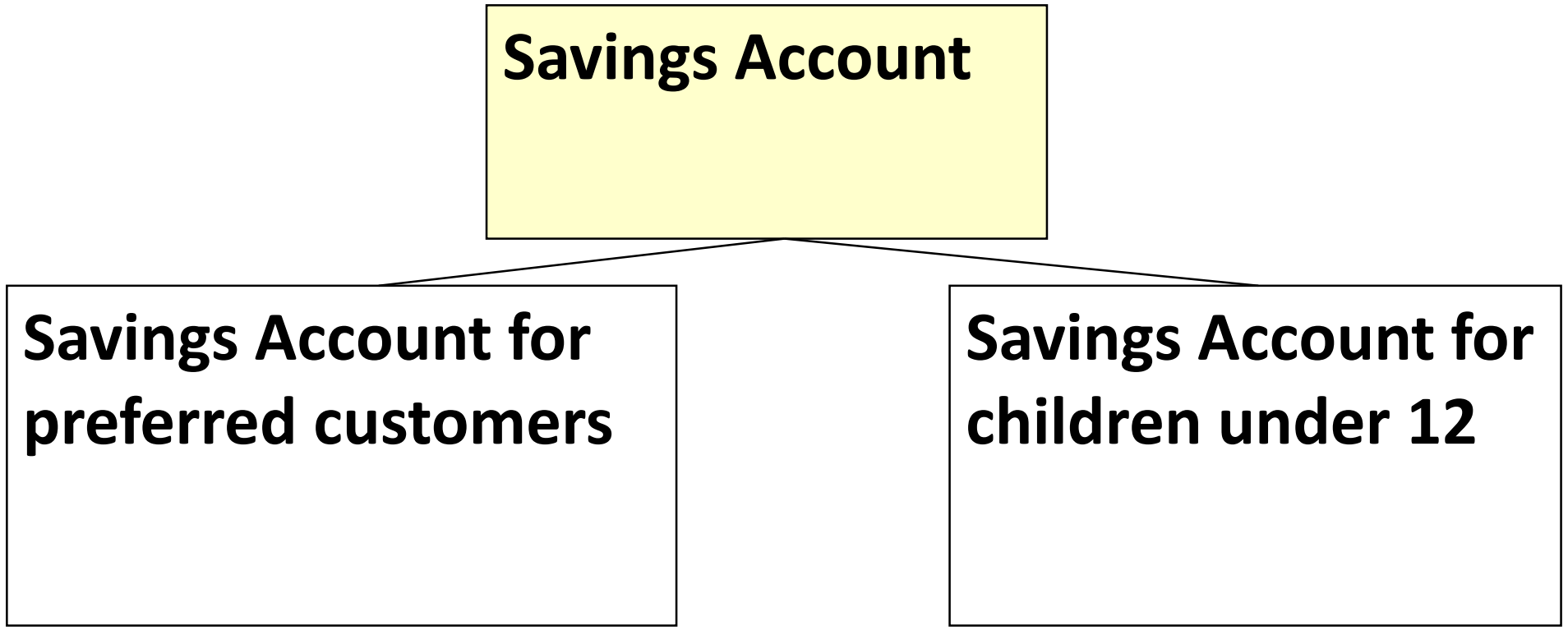
Customer



Note: Object table may change. New objects may be added

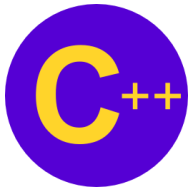
Determine the Relationships among Objects

Using inheritance or composition to build objects hierarchy



Implementation of the Design

- Choose a suitable data representation
  - Built-in data type
  - Existing ADT
  - Create a new ADT
- Create algorithms for the abstract operations
  - Top-down design is often the best technique to create algorithms for the function tasks



Design the Driver

- The driver is the glue that puts the objects together
- In C++, the driver becomes the `main` function
- The driver has very little to do but process user commands or input some data and then delegate tasks to various objects.

Case Study

- Beginning of the Appointment Calendar Program
- Class Entry composed of a Name class and a Time class
- Current Time class represents active time (seconds with Increment)
- Need a static Time class for time in the appointment calendar

Is inheritance appropriate?