H.O. #7
Fall 2015
Gary Chan

C++ Classes

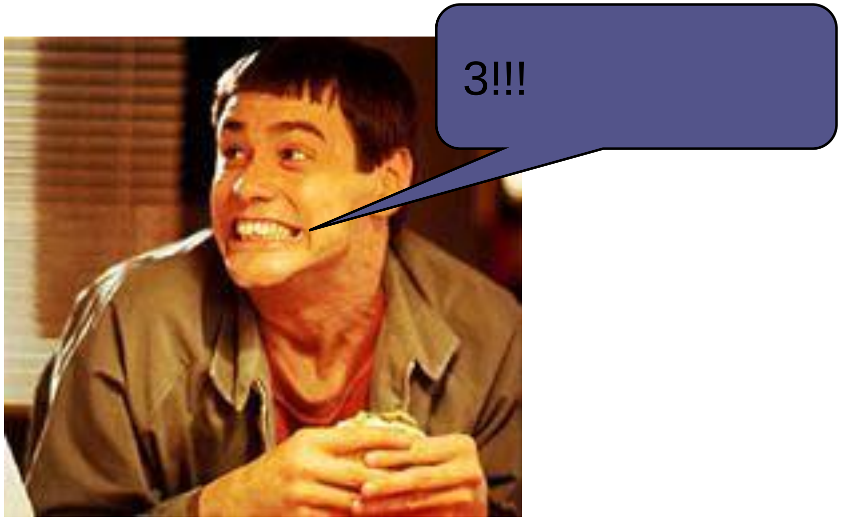
N:1-4; D:1,3,9,10

Outline

- Procedural vs. Object-Oriented Programming
- Basic OOP
  - Private and public data and member functions
  - Accessor and mutators
  - Constructors and initializer
  - Separate compilation and conditional compilation directives to avoid redundant declarations
  - Constant member functions
  - Operator overloading and `friend`
  - Destructors
- Other issues
  - Composition: Objects as members of classes
  - Using `this` pointer
  - Static class members

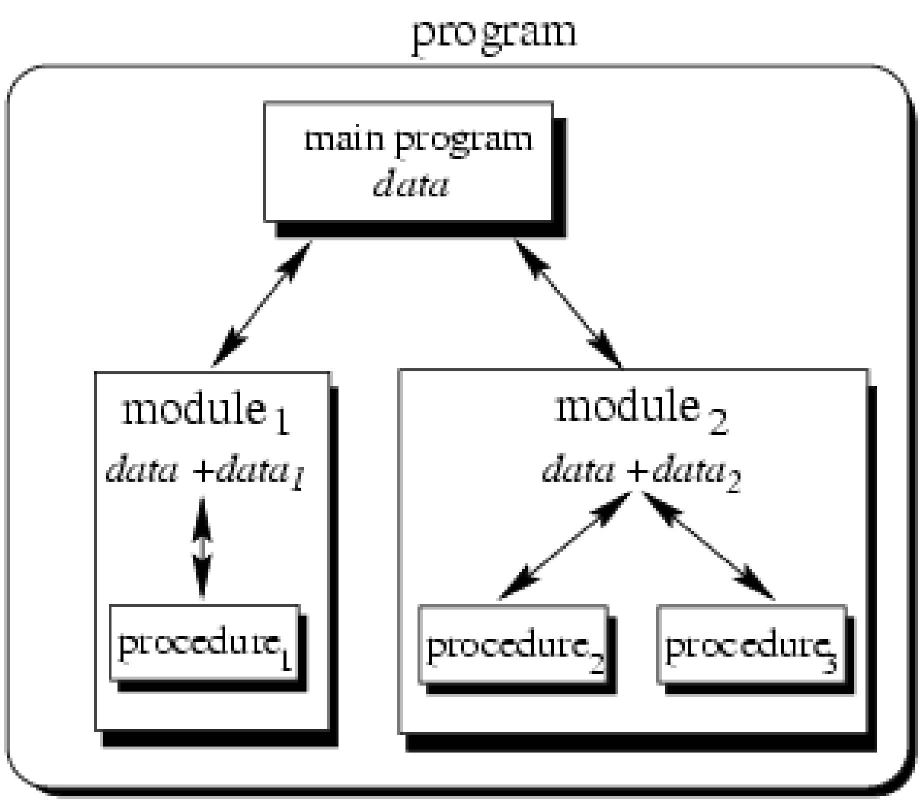
Motivation

- Types such as `int, double`, and `char` are "dumb" objects.
- They can only answer one question: "What value do you contain?"



Programming Paradigm: Procedural Concept

- The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters

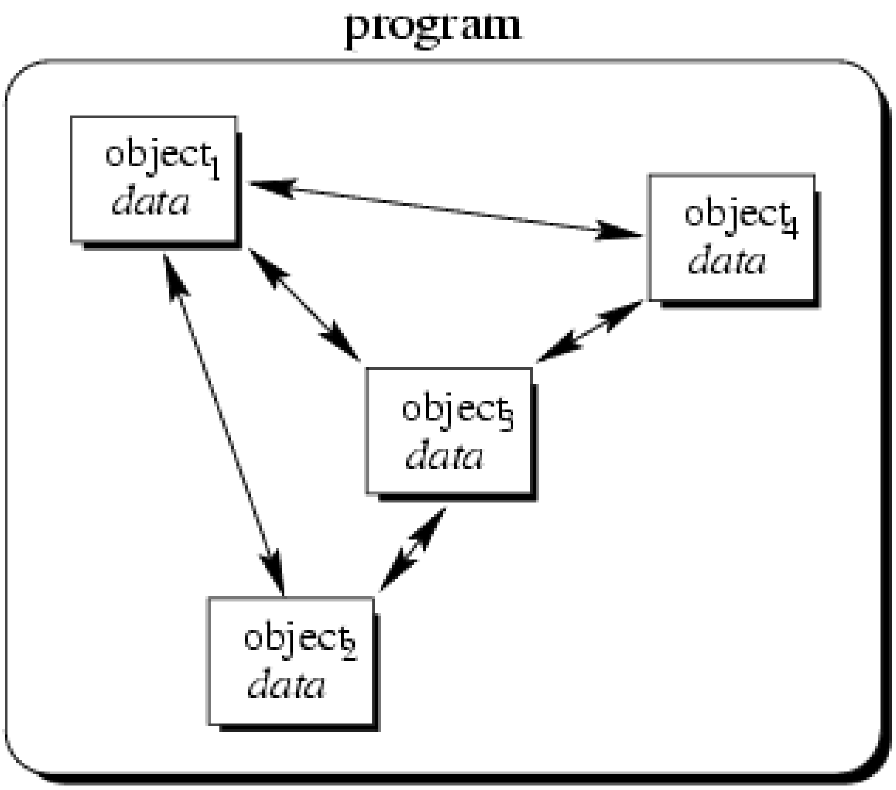


Procedural Concept - Problems

- Designing operations on data
  - The resulting module structure is oriented on the operations of input data
  - The defined operations specify the data to be used
- The design is: Given data we have, what operations we need on manipulating it?
  - E.g., `add( int a, int b );`

Object-Oriented Concept (C++)

- Objects of the program interact by sending messages to the objects
  - `obj.RunCommand();`
  - `obj1.add(obj2); // obj1 + obj2;`
- The objects are then "wired" by their output and flow control
  - `If( obj1.speed() > obj2.speed() )…`



Procedural vs. Object Oriented

Procedural

- **Action**-oriented – concentrates on the verbs

Programmers:

- Identify basic tasks to solve problem given existing data
- Implement actions to do tasks as subprograms (procedures/ functions/subroutines)
- Group subprograms into programs/modules/libraries, together make up a complete system for solving the problem

Object-oriented

- Focuses on the **nouns** of problem specification

Programmers:

- Determine objects needed for the problem
- Determine the operations of each object
- Determine how objects should work together to solve the problem
- Create types called *classes* with
  - *data members*
  - *function members* to operate on the data
- Instances of a type (class) are called *objects*

Classes

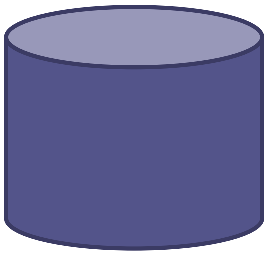
- *Classes* allow you to build "smart" objects that can answer many questions (and perform various actions).
  - "What is your temperature?"
  - "What is your temperature in Fahrenheit?"
  - "What is your temperature in Kelvin?"
- Objects may send messages to each other, which in turn affects the operations of the objects. This leads to different outcomes of the program.
  - `obj1.transfer( weapon, obj2);`

OOD: Object-Oriented Design

- Identify the objects in the problem's specification
- Identify the operations or tasks to manipulate the objects

	FinancialAidAward
data	amount source
operations	getAmount() getsource() display() setAmount() setSource()

Think of them as Containers



First Look at ADTs & Implementations

- For a programming task we must identify
  - The collection of data items
  - Basic operations or algorithms to be performed on them
- Taken together (data items & operations) are called an Abstract Data Type (ADT)
- As an application developer, you do not need to worry how ADT is implemented --- you only need to worry about how they are used
  - ADT hence hides implementation details from its users

Class Declaration Syntax

- Class members are private by default, but can also be declared private

```
class ClassName
{
      public:
      // Declarations of public members

      private:
      // Declarations of private members
};
```

Designing a Class

- Data members are normally placed in `private:` section of a class
  - Can only be manipulated directly *inside* the member functions of the same class
  - Cannot be accessed/called outside the class or by other objects
- Function members are usually in `public:` section
  - Can be called by other objects
- Conventionally `public:` section followed by `private:`
  - although not required by compiler
- There is also a `protected:` keyword
  - Treated as private members against access outside the class
  - Allow direct access to the members for the *derived* classes in inheritance and polymorphism (later)

Private and Public Access

- Attributes (data members)
  - Exist throughout the life of the object
  - Each object of class maintains its own independent copy of attributes
- The access-specifier **`private`** makes a data member or member function accessible only to member functions of the *same* class
  - **`private`** is the default access for class members
  - Users cannot access and manipulate the data directly → Data hiding
- As a rule, data members should be declared `private` and member functions should be declared `public`
- It is appropriate to declare certain member functions private, if
  - they are *helper* functions to be accessed only by other member functions of the same class

Example: Gradebook Class

- A simple object (book) with course name
- Class definition
  - Tells compiler what member functions and data members belong to the class
- Keyword `class` followed by the class's name
- Class body is enclosed in braces (`{ };`)
  - Specifies data members and member functions

Gradebook1b.cpp

- We can separate the *declaration* of member functions from their *definitions*
- Use the :: keyword
- Gradebook1b.cpp

Gradebook2.cpp Sample Output

```
Initial course name is:          (there is nothing there)

Please enter the course name:
COMP2012 OOP and Date Structures

Welcome to the grade book for
COMP2012 OOP and Date Structures!
```

Gradebook Examples (Summary)

- Gradebook1.cpp
  - Your simplest OOP program with class and object creation
  - Class member function is implemented with its declaration
  - Accessing public member functions and variables using '.'
  - No private variables
- Gradebook1b.cpp
  - Same as Gradebook1.cpp but with the member function implemented outside the class
- Gradebook2.cpp
  - private member variable `courseName`
  - Right of direct access to private variable within and outside the class
  - Public member functions that allow clients of a class to set the values of private data members are called mutators
  - Public member functions that allow clients of a class to get the values of private data members are called accessors
  - Calling member function within a member function (`getCourseName`)
  - Get input from users using `getline` and `string` STL (standard template library)

Data Integrity

- Data integrity are not automatic by putting data members as private
  - The programmer must provide appropriate validity checking and report the errors
- Member functions that set the values of private data should verify that the intended new values are proper
  - They should place the private data members into an appropriate state
- *set* functions can be used to validate data besides simply setting the value
  - Known as validity checking
  - Keeps object in a consistent state
    - The data member contains a valid value
  - Can return message indicating that attempts were made to assign invalid data

Information hiding with set and get functions

- Using set and get functions control how clients access private data
  - Can be called by functions of other classes
- *Should* be used by other member functions of the *same* class
  - even though the private data members can be accessed directly
- Localize the effects of changes to a class's data members by accessing and manipulating the data members through these get and set functions

Caution with Set and Get Function

- Be careful when returning a reference to a variable
  - Return a reference returns an acceptable *lvalue* that can be set a value, i.e., may be used on the *left* side of an assignment statement
  - The returned space can be alias to another variable
- One (dangerous) way to use this “return of reference”: A public member function of a class returns a reference to a private data member of that class
  - Client code **could** alter private data members
  - Same problem would occur if a pointer to private data were returned

Caution with Set and Get Function

- A bad setHour function

```
// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
class Time{
  public:
    int & badSetHour( int );
  private:
    int hour;
};

int & Time::badSetHour( int hh )
 {
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

Boundary check: Good

private data member

Problems with the Above Example

- Modifying a private data member through a returned reference and set it to invalid number without going through boundary check!

```
Time t;
// initialize hourRef with the reference returned
int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
// use hourRef as alias to set invalid value in Time t
hourRef = 30;
```

- For below, we have just modified private data by using the resturned lvalue without going through boundary check!

```
// assign another invalid value to hour
t.badSetHour( 12 ) = 74;
```

- To protect against the above two cases, we should return `const int &`, or return value instead
  - If a function returns a const reference, that reference cannot be used as a modifiable lvalue
- Note that sometimes we do return a reference (e.g., overloading >> and <<)

Constructors

- A *constructor* is a special method/function that describes how an instance of the class (called *object*) is constructed
  - May be called implicitly when object is created
  - Must be defined in your program with the same name as the class
  - Cannot return values, not even `void`
- Whenever an instance of the class is created, its constructor is called.
- C++ provides a *default constructor* for each class, which is a constructor with no input parameters (e.g., `foo f;`)
  - The compiler will provide one when a class does not explicitly include a constructor
  - Compiler's default constructor only calls constructors of data members of the class
  - The data members will have undefined values
- One can define multiple constructors for the same class, and may even redefine the default constructor

Gradebook3.cpp

- Constructor syntax
- Default constructor and parameterized constructor
- Ways to construct objects
- Constructor codes may call member functions

Class Definitions: Another Example

- A C++ class consists of *data members* and *methods (member functions)*.

```
class IntCell
{
  public:
        explicit IntCell( int initialValue = 0 )
          : storedValue( initialValue ) {}

        int read( ) const
          { return storedValue;}
        void write( int x )
          { storedValue = x; }
  private:
        int storedValue;
}
```

Avoid implicit type conversion

Initializer list: used to initialize the data members directly. They are NOT functions

Member functions

Indicates that the member's invocation does not change any of the data members.

Data member(s)

explicit and implicit constructor statements

```
main(){
  int x = 4;      // same as int x(4) or int x = int(4);
  IntCell z(x), k(5.2); // storedValue is set to 4 and 5, resp.
  IntCell t;      // storedValue is now 0 (default
                  // constructor)
  IntCell u = IntCell( x );  // u's storedValue is now 4
  // conversion constructor is called and then
  // copy construct as u
  IntCell y = x; // invalid implicit conversion: y = IntCell(x)
…
}
```

Invalid because `x` has to be first *implicitly* converted to type `IntCell` (by calling `IntCell(x)`) before the assignment `y=x` is done (i.e., doing `y=IntCell(x)` implicitly).

However, if the `explicit` keyword is missing, the above codes would work without compiler complaining.

Constant Object and Member Functions

- Principle of least privilege
  - One of the most fundamental principles of good software engineering
  - Applies to objects, too
- const objects
  - Keyword `const`
  - Specifies that an object is not modifiable
  - Attempts to modify the object will result in compilation errors

const member functions

- Member functions declared `const` are not allowed to modify the object
- A function is specified as `const` ***both*** in its class prototype and in its definition
- For a `const` object, only its `const` member function can be called
  - Because all the other functions may modify its value
- `const` declarations are not allowed for constructors and destructors
  - Because by definition they modify the object

Const Member Functions Only Apply to the Member Variables, NOT on the Heap

```
class foo{
public:
  foo(){
    pointer = new int[10];
  }

  void set_el() const{  // compiler ok to have const
    pointer[1] = 10;    // modify the heap
  }

  void set_ptr() {  // cannot have const here
    delete [] pointer;
    pointer = new int[ 100 ];  // modify member variable
  }

private:
  int * pointer;
};
```

- The rule is that if the member function modifies the data member it stores, cannot use `const` function.
- If the member function only modifies some internal hidden book-keeping variables, using `const` is fine.

Constructors Syntax

```
ClassName::ClassName (parameter_list)
: member_initializer_list
{
      // body of constructor definition
}
```

- Member initializer list
  - Invoke the *constructors* for the data members of the object whose memory has been allocated
  - Particularly important if you have reference or constant variable which has to be initialized with a variable
  - After the member initailizers are finished , the body of the constructor is executed
    - You can further change the values of the data members through some function calls here.

Member Initializer

- Required for initializing
  - Data members that are references
  - const data members
- Member initializer list
  - Appears between a constructor's parameter list with a colon (:) and the left brace ({) that begins the constructor's body
  - Each member initializer consists of the data member name followed by parentheses containing the member's construction and its initial value
  - Multiple member initializers are separated by commas
  - Executes before the body of the constructor executes

Initializer to Initialize Variables on Its Construction

OK

```
class foo{
public:
  foo(): i(j), m(3), k(m), j(4) // any order
  {
    cout << i << j << k << m << endl;
  }
private:
  const int & i;
  const int j; // ANSI C++ cannot have const int j = 4;
  int & k;
  int m; // ANSI C++ cannot have int m = 3;
};
```

4433

OK

```
class foo{
public:
  foo(): i(j), k(m), j(4){
    m=3;
    cout << i << j << k << m << endl;
  }
private:
  const int & i;
  const int j;
  int & k;
  int m;
};
```

NOT OK

```
class foo{
public:
  foo(): i(j), k(m){
    m=3;
    j = 4;  // compiler complains: assignment of read-only member `foo::j'
    cout << i << j << k << m << endl;
  }
private:
  const int & i;
  const int j;
  int & k;
  int m;
};
```

Increment Example for Initializer

- Increment.h
  - Class definition with a constant integer
  - No initialization at class definition
- Increment.cpp
  - Initializer list to initialize constant integer
  - const data member *increment* must be initialized using a member initializer
  - Not providing a member initializer for a const data member is a compilation error
- const2.cpp
  - Driver program

Some Final Words on Constructor

- The compiler will always find the closest match among all of your constructor statements
- Once a parameter in a constructor has a default value, all its *following* parameters *must* have one.

```
class foobar{
  public:
    foobar( int a = 1, double d ){  // compiler complains:
    // default argument missing for parameter 2
      i = a; j = d;
    }
  private:
    int i;
    double j;
};
```

```
#include <iostream>
using namespace std;
class foo{
public:
  foo( double d = 4.0 ){
    i = -1;
    j = d;
  } // compiler will match foo f(1.2) to this
  foo( int a = 10 ){
    i = a;
    j = -2.0;
    }    // compiler will match foo f(1) to this
  void print( void ) const{
    cout << i << " " << j << endl;
  }
private:
  int i;
  double j;
};

int main(){
  //  foo a;  // compiler complains: call of overloaded `foo()' is ambiguous
  foo b(1);   // ok - match to foo( int )
  b.print();
  foo c(1.0); // ok - match to foo( double )
  c.print();

  return 1;
}
```

```
1 -2
-1 1
```

```
class bar{
public:
  bar( int a = 1, double d = 2.2 ){
    i = a;
    j = d;
  }
  //  bar();  //if put this here, compiler complains (ambiguous constructor)
  void print( void ) const{
    cout << i << " " << j << endl;
  }
private:
  int i;
  double j;
};


int main(){
  bar d;        // ok
  d.print();
  bar e(2);     // ok
  e.print();
  bar f(4.5);   // ok; a gets 4
  f.print();

  bar *bptr = new bar [10]; // ok: all objects with default of a = 1 and d = 2.2
  bar *bptr2 = new bar(); // same as bar *bptr = new bar ;

  bar g();     // NOT a constructor; it is a function prototype stating
               // that g is a FUNCTION returning bar
  bar h(void);  // NOT a constructor: a function prototype; does nothing

  return 1;
}
```

1 2.2
2 2.2
4 2.2

GradeBook4.h and GradeBook4.cpp

- Separation of definitions of class and functions from their usage
- Same as Gradebook3.cpp, but broken into 2 files with `main()` in Gradebook4.cpp
- GradeBook4.h
  - Implementation details of class
- GradeBook4.cpp
  - Usage of class
  - #include "GradeBook4.h"to read in GradeBook4.h

Interface and Implementation

- In C++ it is more common to separate the *class interface* from its *implementation*.
  - Abstract data type
- The *interface* lists the class and its members (data and functions).
- The *implementation* provides implementations of the functions.

Separate File for Reusability

- Header files
  - Separate files in which class definitions are placed
  - Allow compiler to recognize the classes when used elsewhere
  - Generally have .h filename extensions
- .cpp file is known as a source-code file to implement the functions
- Driver files
  - Program used to test software (such as classes)
  - Contains a main function so it can be executed

#include preprocessor directive

```
#include "GradeBook.h"
```

- Used to include header files
  - Instructs C++ preprocessor to replace directive with a copy of the contents of the specified file
- Quotes indicate user-defined header files
  - Preprocessor first looks in current directory
  - If the file is not found, looks in C++ Standard Library directory
- Angle brackets indicate C++ Standard Library
  - Preprocessor looks only in C++ Standard Library directory
  - #include <iostream>

Interface

- Describes what services a class's clients can use and how to request those services
- But does not reveal how the class carries out the services
- A class definition that lists only member function names, return types and parameter types
  - Function prototypes
- A class's interface consists of the class's public member functions (services)

```cpp
class IntCell
{
    public:
        explicit IntCell( int initialValue = 0 );
        int read( ) const;
        void write( int x );
    private:
        int storedValue;
}
```

IntCell.h

```cpp
IntCell::IntCell( int initialValue )
    : storedValue ( initialValue ) 
{ }

int IntCell::read( ) const
    { return storedValue; }

void IntCell::write( int x )
     { storedValue = x; }
```

IntCell.cpp

- The interface is typically placed in a file that ends with *.h*. The member functions are defined as:

  ReturnType FunctionName(*parameterList*);

- The implementation file typically ends with *.cpp*, *.cc*, or *.C*. The member functions are defined as follows:

  ReturnType ClassName::FunctionName(*parameterList)*
  { ...... }

Scoping operator

Separating Interface from Implementation

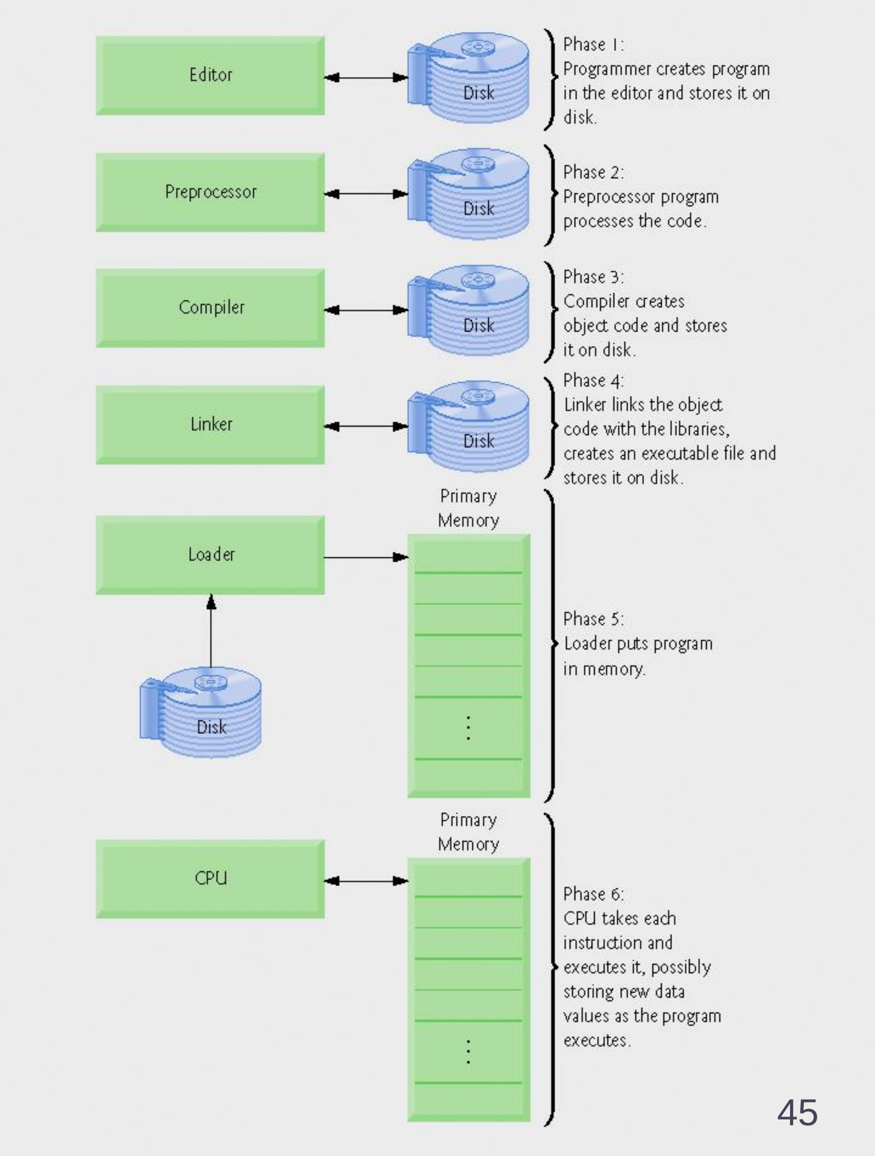
- Client code should not break if the implementation changes, as long as the interface stays the same
- Define member functions outside the class definition, in a separate source-code file
- In source-code file for a class
  - Use binary scope resolution operator (::) to “tie” each member function to the class definition
- Implementation details are hidden
  - Client code does not need to know the implementation
- In the header file for a class
  - Function prototypes describe the class’s public interface

Separating Interface from Implementation (Cont.)

- Makes it easier to modify programs
  - Changes in the class's implementation do not affect the client as long as the class's interface remains unchanged
- Things are not quite this rosy
  - Header files do contain some portions of the implementation and hint about others
  - private members are listed in the class definition in the header file

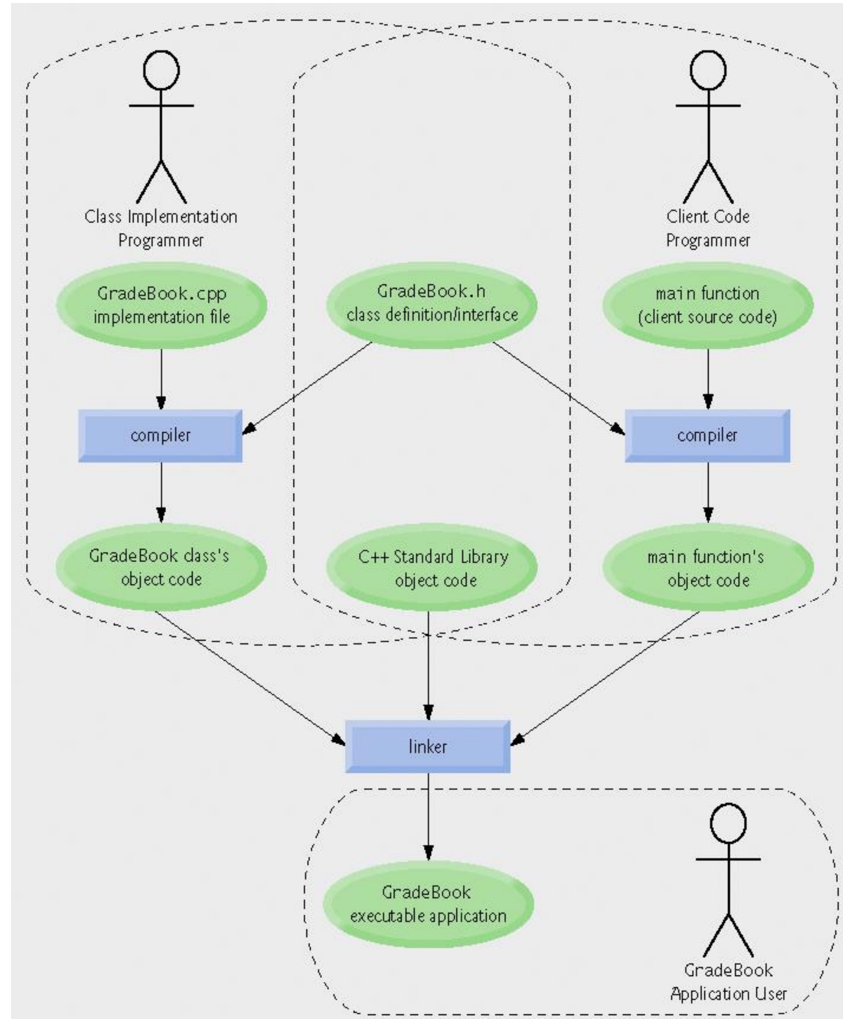
Typical C++ Development Environment

- Edit
  - Programmer writes program (and stores source code on disk)
- Preprocess
  - Perform certain manipulations and file I/O to prepare for compilation
- Compile
  - Compiler translates C++ programs into machine languages in object codes
- Link
  - Link object codes with missing functions and data
- Load
  - Transfer executable image to memory
- Execute
  - Execute the program one instruction at a time



The Compilation and Linking Process

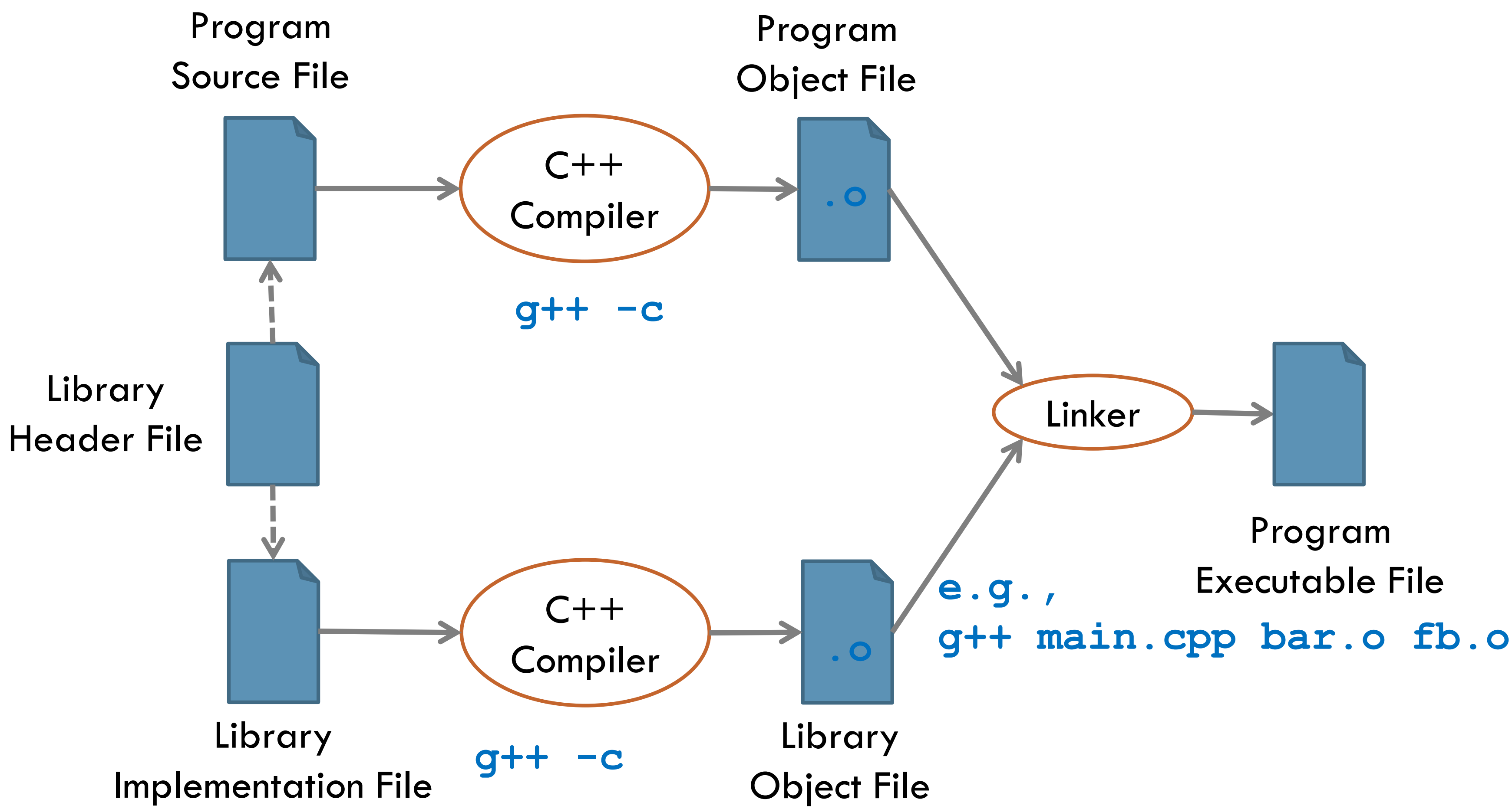
- Source-code file is compiled to create the class's object code (source-code file must #include header file)
  - Class implementation programmer only needs to provide header file and object code to client
- Client must #include header file in their own code
  - So compiler can ensure that the main function creates and manipulates objects of the class correctly



Class Libraries

- Class declarations placed in header file
  - Given `.h` extension
  - Contains data items and prototypes
- Implementation file
  - Same prefix name as header file
  - Given `.cpp` extension
- Programs which use this class library called client programs

Compilation Process



library.h

```
#ifndef ABC
#define ABC

~~int TestInt = 99;~~ extern int TestInt;

int functionA( int );
#endif
```

main.cpp

```
#include <iostream>
#include "library.h"
using namespace std;
int TestInt=99;

int main(){
  cout << "Hello"<<endl;
  TestInt = 10;
  cout << functionA(100) << endl;
  return 0;
}
```

source.cpp

```
#include "library.h"

int functionA( int i ){
  return TestInt * i;
}
```

```
> g++ main.cpp source.cpp
ld: fatal: symbol `TestInt' is multiply-defined:
        (file /var/tmp/ccvkmxE2.o type=OBJT; file /var/tmp/ccgj1SDu.o
type=OBJT);
ld: fatal: File processing errors. No output written to a.out
collect2: ld returned 1 exit status
```

Output:
Hello
1000

Why #ifndefine #define #endif Statement?

- It is ok to have multiple declarations of a function *prototype*, but not for its definition
  - In the .h file, put the prototypes there
  - .h files are likely to be multiply-included
- In creating the .o file, there may be nested #include statements
- The nested #include statement may be recursive
  - In main.cpp, #include “foo.h”
  - In foo.h, #include “bar.h”
  - In bar.h, #include “foo.h”
- To break the infinite “recursive” inclusion, use #ifndefine #define to define a “variable” in the compilation process of .o file
- If a variable has been defined, the compiler will skip the code segment between #ifndefine and #endif.

GradeBook6

- GradeBook6.h
  - Header file
  - Only specifies how the class functions can be used, not how they are implemented
  - #ifndefine… #define… #endif
- GradeBook6.cpp
  - Implementation file
  - Only specifies how the functions are implemented
  - No main()
  - #include "GradeBook6.h"
- driver6.cpp
  - Driver program with main()
  - Uses the class functions
  - #include "GradeBook6.h"
- In Linux, compile them all together using

```
g++ Gradebook6.cpp driver6.cpp
```

- Or using object files:

```
g++ -c Gradebook6.cpp; g++ -c driver6.cpp; g++ Gradebook6.o
  driver6.o
```

driver6.cpp Sample Output

```
Name "COMP2011 Introduction to Programming in C++" exceeds maximum
length (25).
Limiting courseName to first 25 characters.

Name "COMP2012 OOP and Data Structures" exceeds maximum length
(25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: COMP1004 Introduction to
gradeBook2's initial course name is: COMP2012 OOP and Data Str

gradeBook1's course name is: COMP104 C++ Programming
gradeBook2's course name is: COMP2012 OOP and Data Str
```

Time.h and Time.cpp

- Display and change time
  - Starting from 12:00am (midnight) to 11:59pm
- Keep a military time
  - Converting a normal time to a 4-digit integer
  - 2:05am ←→ 205
  - 4:15pm ←→ 1615
  - 12:00am (midnight) ←→ 0000 (or simply 0)
- Constructor
  - Initializer list
  - Default constructor
  - Explicit-value constructor

Overloading Functions

- Note existence of multiple functions with the same name

```
Time();
Time(unsigned initHours,
     unsigned initMinutes,
     char initAMPM);
```

- Known as overloading
- Compiler compares numbers and types of arguments of overloaded functions
  - Checks the "signature" of the functions

Default Arguments

- Can be combined to specify default values for constructor arguments

```
Time(unsigned initHours = 12,
     unsigned initMinutes = 0,
     char initAMPM = 'A');
```

```
Time t1, t2(5), t3(6,30), t4(8,15,'P');
```

t1	
myHours	12
myMinutes	0
myAMorPM	A
myMilTime	0

t2	
myHours	5
myMinutes	0
myAMorPM	A
myMilTime	500

t3	
myHours	6
myMinutes	30
myAMorPM	A
myMilTime	630

t4	
myHours	8
myMinutes	15
myAMorPM	P
myMilTime	2015

Copy Constructor and Assignment

- Copy constructor (default):

  ```
  Time t = bedTime;
  //calls Time t(bedTime);
  ```

t	
myHours	11
myMinutes	30
myAMorPM	P
myMilTime	2330

bedTime	
myHours	11
myMinutes	30
myAMorPM	P
myMilTime	2330

- During assignment

  ```
  t = midnight;
  ```

t	
myHours	12
myMinutes	0
myAMorPM	A
myMilTime	0

midnight	
myHours	12
myMinutes	0
myAMorPM	A
myMilTime	0

Display Functions

- Two functions used for output
  - `void display(ostream &)` inside the class as member function
  - `ostream & operator<<(ostream &, const Time &)` outside the class as an external function
- The display function:

```
void Time::display(ostream & out) const
{
  out << myHours << ':'
      << (myMinutes < 10 ? "0" : "") << myMinutes
      << ' ' << myAMorPM << ".M.  ("
      << myMilTime << " mil. time)";
}
```

- We'd like to have `cout << t1 << t2;`

Implementing Output by Overloading <<

- Use the public display() function to display the object
- Declaration in .h file

```
class Time {
        …
};

ostream & operator<<(ostream & out, const Time & t);
```

- Definition in .cpp file

```
ostream & operator<<(ostream & out, const Time & t)
{
  t.display(out);
  return out;
}
```

Read Functions

- Two functions used for intput
  - **`read()`** inside the class as member function
  - **`Operator>>()`** outside the class as an external function
- The read function:

```
void Time::read(istream & in){
  unsigned hours,    // Local variables to hold input values from in so
           minutes;  //    they can be checked against the class invariant
  char     am_pm,    //    before putting them in the data members
           ch;       // To gobble up ':' and the 'M' in input

  in >> hours >> ch >> minutes >> am_pm >> ch; // e.g., 3:18 PM

  set(hours, minutes, am_pm);  // use mutator to check validity
}
```

- We'd like to have `cin >> t1 >> t2;`

Implementing Input by Overloading >>

- Use the public display() function to display the object
- Declaration in .h file

```
class Time {
        …
};

istream & operator>>(istream & in, Time & t);
```

- Definition in .cpp file

```
istream & operator>>(istream & in, Time & t)
{
  t.read(in);
  return in;  // return in for input cascading
}
```

Relational Operators

- In Time.cpp

```
bool operator<(const Time & t1, const Time & t2) {
  return t1.getMilTime() < t2.getMilTime();
}

bool operator>(const Time & t1, const Time & t2) {
   return t1.getMilTime() > t2.getMilTime();
}

bool operator==(const Time & t1, const Time & t2) {
   return t1.getMilTime() == t2.getMilTime();

// may also return ( !(t1 < t2) && !(t1 > t2) );
}

bool operator<=(const Time & t1, const Time & t2) {
   return t1.getMilTime() <= t2.getMilTime();

// or return !(t1 > t2);
}
```

`friend` Functions

- It is possible to specify an operator, e.g., `operator<<()`, as a "friend" function
  - Thus give "permission" to an external function to access private data elements directly
- Declaration in .h file

```
class Time {
        …

friend ostream & operator<<(ostream & out, const
Time & t);

};
```

`friend` Functions (Cont.)

- Definition in .cpp file

```
ostream & operator<<(ostream &out, const Time &t)
{
  out << t.myHours<<":"
      << (t.myMinutes< 10? "0": "")  //print,e.g., 05
      << t.myMinutes
      << ' '<<t.myAMorPM<<".M.";
  return out;
}
```

- `cout << t` is converted to `operator<<(cout, t)`
- Note that the function can directly access private data members without going through accessor functions
- Remember to return ostream as a reference as we require it to be used in cascade
- A friend function is NOT a member function
  - not qualified with class name and ::
  - receives class object on which it operates as a parameter

3 Ways of Operator Overloading

- As an external function (external view)
  - Has to use accessors and mutators to get or set variables
  - Discussed in Time.h
  - Best used when the original class cannot be modified
- As a `friend` of an external function
  - Can directly access data members
  - Discussed just now in the slides
  - Best used when efficiency is needed without affecting the original class codes
- As a member function (internal view of the object)
  - Can directly access data members
  - Best used when the operator overloading are developed with the class

Internal Function: Operator Overloading for a Complex Class

```
class Complex {
  public:
    ...// constructor with two parameters: Complex( double, double);
                    self
    Complex operator +(const Complex &op) {
      double real = _real + op._real,
             imag = _imag + op._imag;
      return(Complex(real, imag));  //construct a Complex object
    }
    ...
  };
```

- An expression of the form

```
c = a + b;
```

is translated into a method call

```
c = a.operator +(b);
```

- We need to return the result in a complex object so that we can compute `a+b+c`
- We have made the operator + a *member* of class Complex. This is an *internal* view of the object (the object is added to `op`), which differs from the *external* declaration of adding two objects to be discussed next: `Complex operator+(const Complex &a, const Complex &b);`

External Function: Operator Overloading for Complex Objects

- The overloaded operator may not be a member of a class: It can rather be defined outside the class as a normal overloaded function. For example, we could define operator +, which takes two arguments, in this way:

```
class Complex {
  public:
    ...
    double real() const { return _real; }
    double imag() const { return _imag; }

    // No need to define any operator here!
  private:
    double _real, _imag;
  };

//add two objects together
  Complex operator +(const Complex &op1, const Complex &op2) {
    double real = op1.real() + op2.real(), // cannot access private data member
           imag = op1.imag() + op2.imag();
    return(Complex(real, imag)); // call constructor
  }
```

- A call of `a+b` is then converted to `operator+(a,b)`

Friend for Complex Objects

- We can define functions or classes to be friends of a class to allow them direct access to its private data members

```
class Complex {
    public:
    ...
    friend Complex operator +(
      const Complex &,
      const Complex &
    );  // NOT member function
  };

  Complex operator +(const Complex &op1, const Complex &op2) {
    double real = op1._real + op2._real,  //access private data members due to friend
           imag = op1._imag + op2._imag;
    return(Complex(real, imag));
  }
```

Destructor

- C++ destroys an object when it goes out of scope; called implicitly when an object is destroyed
  - When functions returns; program execution leaves the scope in which that object was instantiated
  - When `delete` is called on the object
- A special member function
- Name is the tilde character (~) followed by the class name
  - e.g., `~Time();`
- The default destructor is to free up all the private members
  - Pointers are not traversed, and hence may have leak problem!
- To declare a destructor, use a member function which has no return and no parameters: `~foo();`

Destructor (Cont.)

- C++ provides a *default destructor* for each class
  - If the programmer does not explicitly provide a destructor, the compiler creates an "empty" destructor
  - The default simply applies the destructor on each data member.
  - We can redefine the destructor of a class.
- A C++ class can have only one destructor
  - Destructor overloading is not allowed
- Receives no parameters and returns no value
  - May not specify a return type—not even void
- It is a syntax error to attempt to
  - pass arguments to a destructor
  - specify a return type for a destructor (even void cannot be specified)
  - return values from a destructor
  - overload a destructor

Some Words on Destructor

- *Outside* a class, you should almost never call a destructor :

```
foo f;
f.~foo(); // not ok, as it does not destroy the object.
       // Please let the system takes care of the local variables
foo *fptr = new foo;
fptr -> ~foo(); // not ok, use delete fptr; instead
```

- *Within* a class, you may call the destructor as a member function to execute the destructor body (which is NOT to destroy the whole object):

```
void foo::bar(){
  ~foo(); // execute the destructor body
  // some other codes here
}
```

Other Issues

Constant Object and Constant Member Functions

- Member functions declared const are not allowed to modify the object
- A function is specified as const BOTH in its prototype and in its definition
- Const declarations are not allowed for constructors and destructors
- Const objects can *only* call const member functions
  - Therefore declare `const` in a function if it does not modify the object, so that a const object can use it
- Const object can access both constant and non-constant member *variables*
- Declaring const has another advantage: if the member function is inadvertently written to modify the object, the compiler will issue an error message
- const data members
  - It is an error to modify a const data member
  - Prevents accidental changes to a data member in any member functions
  - Must be initialized with a member initializer

```cpp
#include <iostream>
using namespace std;

class foo{
public:
  int i;
  const int j;
  foo(): j(2), i(3){}
  void print( void ) const {
    cout << i << endl;  cout << j << endl;
  }
  void print2( void ) {
    cout << i << endl;  cout << j << endl;
  }
};

int main(){
  const foo f;
  //  f.j = 10;  Compilation error
  //  f.i = 2;    Compilation error
  cout << f.i << endl; // access non-const data member
  cout << f.j << endl;
  f.print();
  //  f.print2(); Compilation error
}
```

```
3
2
3
2
```

A Member Function Returning a Reference

- Note that we can have a member function which returns a reference. For example, if a member function returns an integer reference, there are 4 possibilities.
- **`1) int & bar();`**
  - Constant object cannot call it; this is for non-constant objects. It returns an integer reference and hence can be subsequently changed.
  - E.g., for a non-constant object `ncfoo`, we can call `ncfoo.bar() = 10;` or `i = ncfoo.bar();`
- **`2) const int & bar();`**
  - Constant object cannot call it; this is for non-constant objects. It has to be a rvalue.
  - `i = ncfoo.bar(); // good`
  - `ncfoo.bar() = 10; // wrong: compilation error`

A Member Function Returning a Reference (Cont.)

- **`3) const int & bar() const;`**
  - This is for both constant and non-constant objects (constant object can call it only). It returns a constant reference and hence can only be rvalue.
  - `i = cfoo.bar(); // good; or i= ncfoo.bar();`
  - `cfoo.bar() = 10; //wrong; and nor ncfoo.bar() = 10;`
- **`4) int & bar() const;`**
  - A constant function not modifying the object
  - If it is for a constant object, it cannot be a lvalue → Use the third case above
  - If it is for a non-constant object, there is no need to have the keyword `const`
  - To conclude, there is no point in using this.
- In a program, therefore, you can have
  - Either first (1) <u>or</u> second (2) for non-constant objects depending on what you want on the return value (cannot have both in your program); and/or
  - The third one (3) for constant objects
  - The compiler will make the call depending on whether the object is constant or not.
- So there can be 5 possibilities: 1, 2, 3, (1,3), or (2,3)

Summary

- (const) int & foo::bar() (const);
  - Can always be rvalue

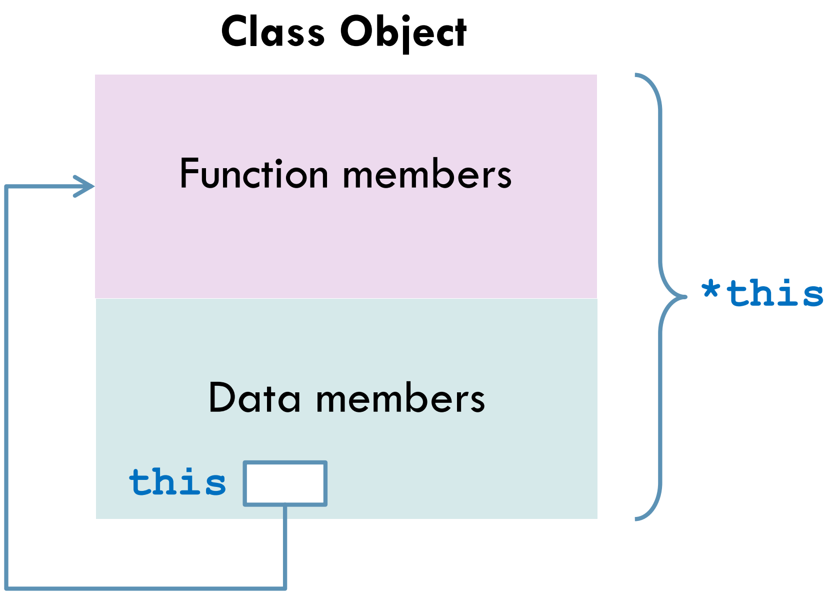
	int &	const int &
::bar();	•For non-constant object only •Can be lvalue	•For non-constant object only •Cannot be lvalue (can only be rvalue)
::bar() const;	•Constant object can call it, but it returns a reference which may be lvalue •Should put `const int &` •→ No use	•For constant or non-constant objects •Cannot be lvalue

Composition: Objects as Members of Classes

- Sometimes referred to as a has-a relationship
- A class can have objects of other classes as members
- Example: AlarmClock object with a Time object as a member
- Initializing member objects
  - Member initializers pass arguments from the object's constructor to member-object constructors
  - Before the enclosing class object (host object) is constructed
  - If a member initializer is not provided, the member object's default constructor will be called implicitly
- Example: Date.h, Date.cpp, Employee.h, Employee.cpp and composition.cpp

The **`this`** Pointer

- Every class has a keyword, **`this`**
  - a pointer whose value is the address of the object
  - Value of **`*this`** would be the object itself



Using `this` Pointer

- Every object has access to its own address through a pointer called this (a C++ keyword)
- Objects use the `this` pointer implicitly or explicitly
  - Implicitly when accessing members directly
  - Explicitly when using keyword this
  - Type of the `this` pointer (i.e., whether it can be modified or not) depends on the type of the object and whether the executing member function is declared const
- Usually used when you want to return the modified object for concatenation:

```
foo & foo::bar(){
  // manipulate and transform data members
  // …

  return *this;
}
```

Pointers to Class Objects

- Possible to declare pointers to class objects

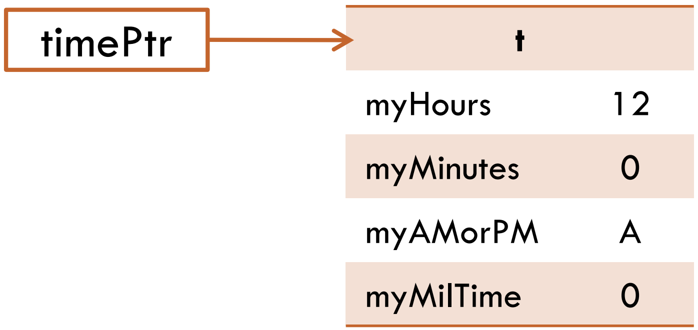
```
Time * timePtr = &t;

Time * timePtr = new Time( 12, 0, 'A', 0 );
```

- Access with

```
timePtr->getMilTime()
```

or

```
(*timePtr).getMilTime()
```



- Call delete to free the memory

```
delete timePtr; // call destructor
```

Static Variables

- Static variables are put somewhere in memory
- ct has only local scope and can only be accessed within the function. It is not deleted when the function exits.

```
int bar( void ){
  static int ct = 0;

  ct++;
  return ct;
}

int main(){
  // cout << ct; Compilation error
  cout << bar() << endl;
  cout << bar() << endl;

  return 0;
}
```

```
Output:
1
2
```

Static Class Members

- Only *one* copy of a variable or function shared by *all* objects of a class
  - "Class-wide" information
  - A property of the *class* shared by all instances, not a property of a specific object of the class
- Declaration begins with keyword **`static`**
- May seem like global variables but they have class scope
  - Outside the class, they cannot be accessed

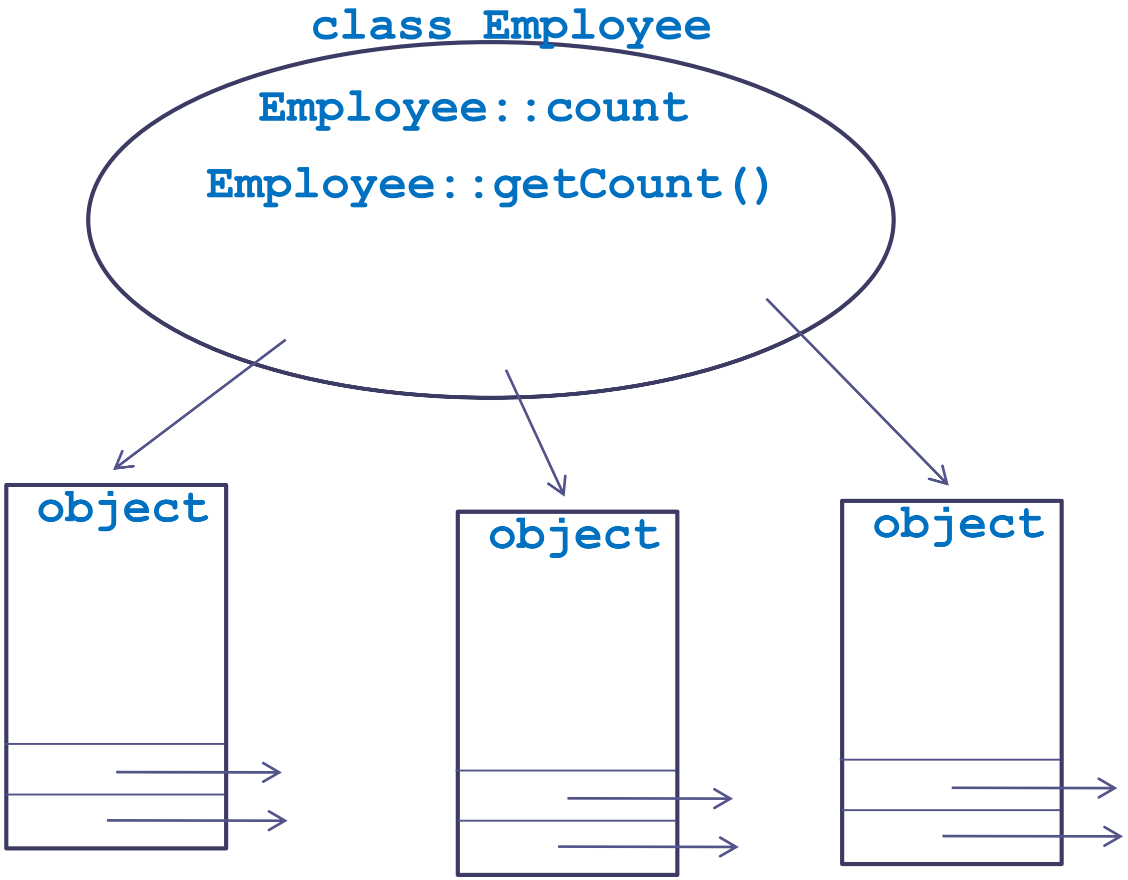
Static Class Members

- Can be declared public, private or protected
- Primitive (Fundamental-type) static data members
  - Initialized by default to 0
  - If you want a different initial value, a static data member can be initialized once (and only once)
- A *const static* data member of primitive or enum type can be initialized in its definition in the class definition
  - Alternatively, you can also initialize it in file scope
- All non-constant static data members must be defined at file scope, i.e., outside the body of the class definition
- static data members of class types (i.e., static member objects) that have default constructors need not be initialized because their default constructors will be called

Static Data and Function Members of a Class

- static member function
  - Is a service of the *class*, not a service of the *object* of the class
- Exist even when *no* objects of the class exists
- To access a public *static* class member when no objects of the class exist:
  - Prefix the class name and the binary scope resolution operator (::) to the name of the data member
  - Example: `Employee::count` or `Employee::getcount()`
- Also accessible through any object of that class
  - Use the object's name, the dot operator and the name of the member
  - Example: `Employee_object.count` or `Employee_object.getcount()`
- Example: SEmployee.h, SEmployee.cpp, static.cpp

Programmer's View



Constant Static Variable

```
#include <iostream>

using namespace std;

class foo{
public:
  static int getcount();
  // static member function cannot have `const' method qualifier
private:
  const static int count;  // may also be const static int count = 2;
};

// initialization of constant static variable: must be here (file scope); not in main()
const int foo::count = 2;

int foo::getcount(){
  cout << count;
}

int main(){

  foo::getcount();  // print out 2
  foo::getcount();  // print out 2
  cout << foo::count;  // wrong as 'const int foo::count' is private
  return 0;
}
```

`static` member function

- It cannot access non-static data members or non-static member functions of the class (because the object may not exist when the function is called)
- A static member function does not have a `this` pointer
- static data members and static member functions exist independently of any objects of a class, i.e., when a static member function is called, there might not be any objects of its class in memory