Week 9 Lecture 2

NWEN 241
Systems Programming

Alvin Valera

alvin.valera@ecs.vuw.ac.nz

Content

- More on Classes in C++

Recap: Classes

Classes generalizes user defined data types in an object-oriented sense:

- Classes are types representing groups of similar instances
- Each instance has certain fields that define it (instance variables)
- Instances also have functions that can be applied to them– also known as *methods* in OOP
- Access to parts of the class can be limited

Classes allow the combination of data and operations in a single unit

Recap: Defining a Class

- A class is a collection of fixed number of components called **members** of the class

- General syntax for defining a class:

```
class class_identifier {
      class_member_list
};
```

  - `class_member_list` consists of variable declarations and/or methods

Recap: Example

```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Member access specifiers

Possible specifiers:

- `private`
- `protected`
- `public`

Recap: Member Access Specifier

- **Private members** – can only be accessed by member functions (and friends) and not accessible by descendant classes

- **Public members** – can be accessed outside the class and inherited by descendant classes

- **Protected members** – can only be accessed by member functions (and friends) and inherited by descendant classes

- When member access specifier is not indicated, default access is `private`

Recap: Example

```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Constructors

- Named after class name
- Similar to Java.

When class performs dynamic memory allocation, **destructor** is also needed

Types of Constructors

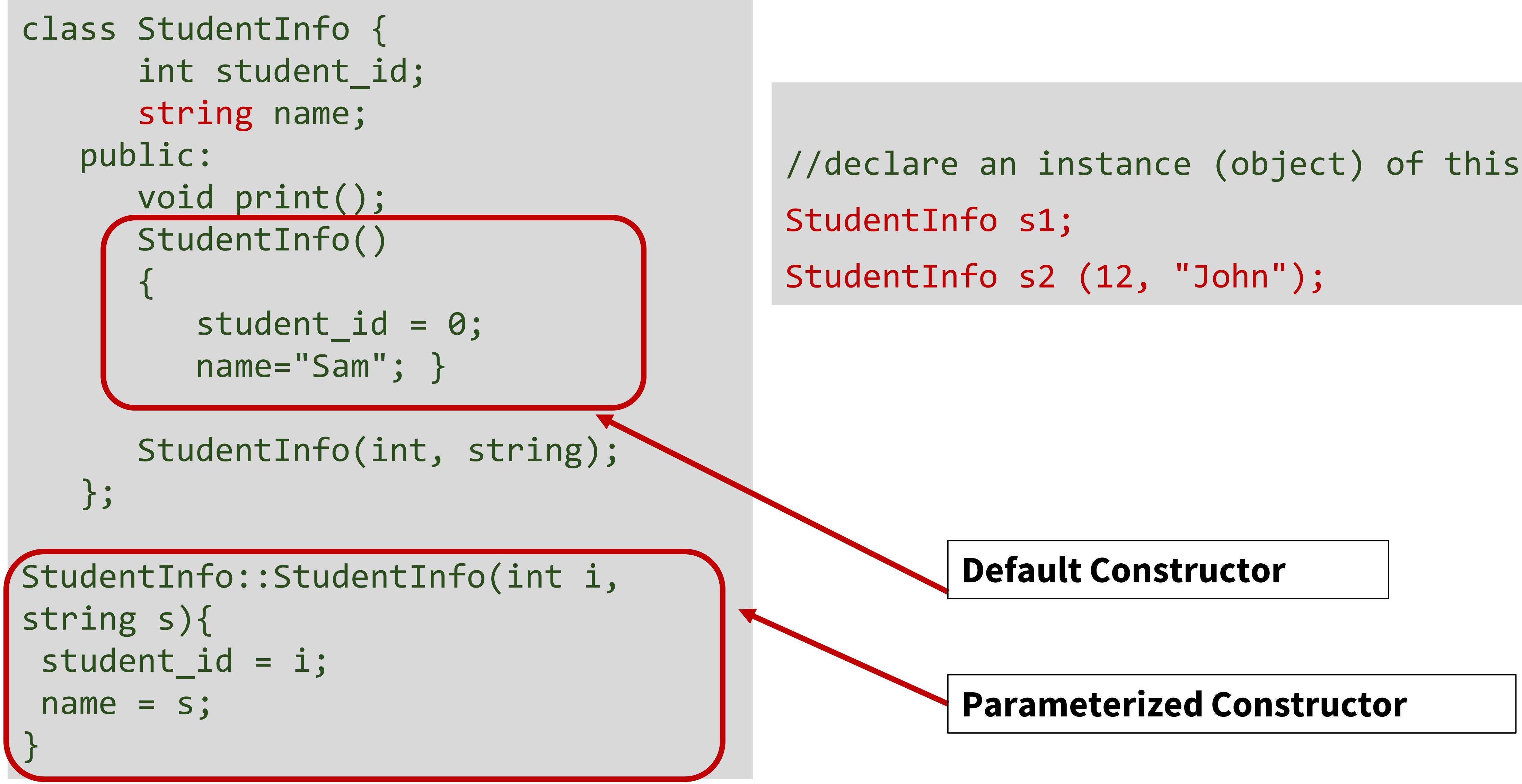
- **Default Constructors (Non – parameterized Constructor)**
  - Accepts no arguments
  - `class_name()`

- **Parameterized constructor**
  - Accepts arguments
  - `class_name(parameters)`

- **Copy constructor**
  - Copies another existing object
  - `class_name (const class_name & )`

& - Reference operator, used to provide an alternative name for an existing variable

Example



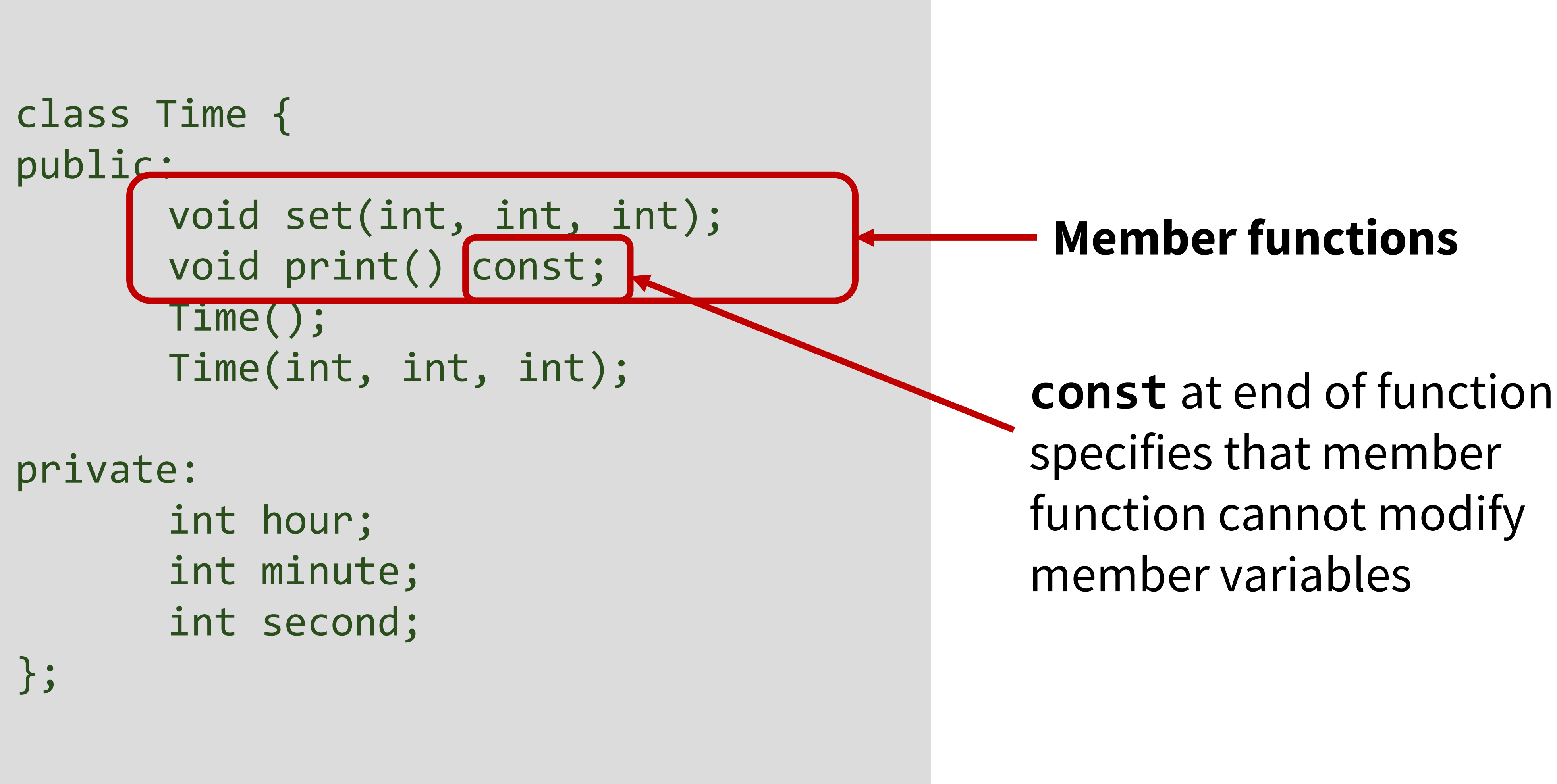
```
class StudentInfo {
      int student_id;
      string name;
   public:
      void print();
      StudentInfo()
      {
         student_id = 0;
         name="Sam"; }

      StudentInfo(int, string);
   };

StudentInfo::StudentInfo(int i,
string s){
 student_id = i;
 name = s;
}
```

```
//declare an instance (object) of this class
StudentInfo s1;
StudentInfo s2 (12, "John");
```

Default Constructor

Parameterized Constructor

Example



```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Member functions

`const` at end of function specifies that member function cannot modify member variables

Example

```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Member variables

Example

```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour = 0 ;
        int minute = 0;
        int second = 0;
};
```

Member variables

Default values for member variables can be initialized during declaration

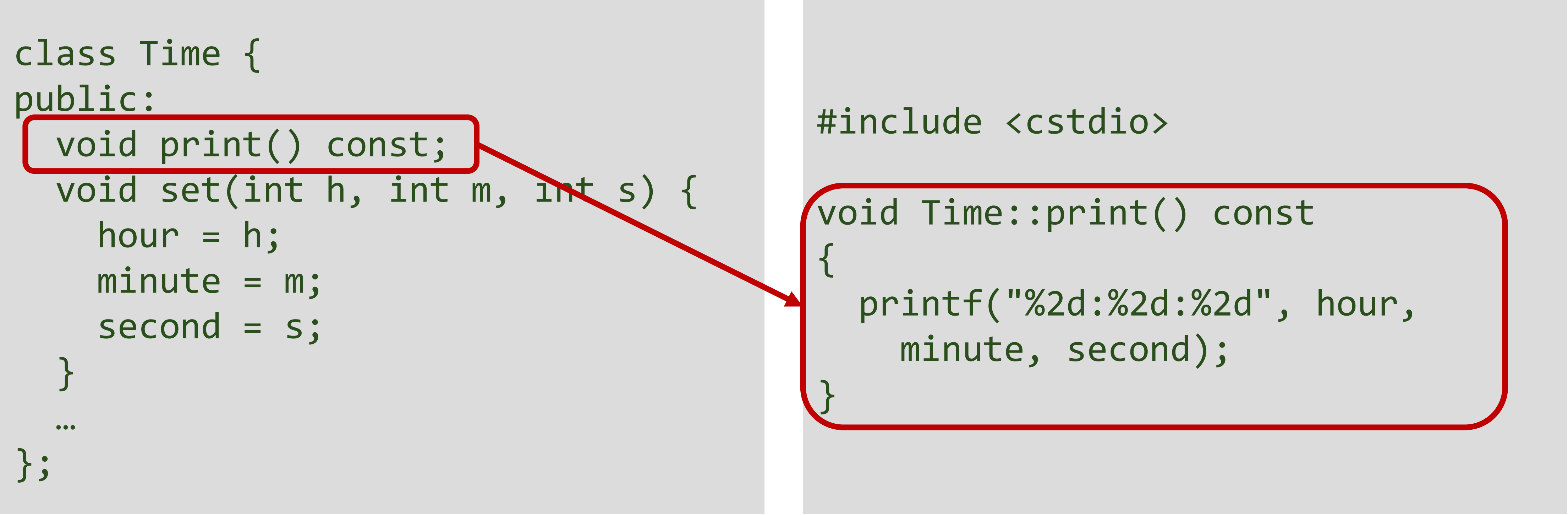
Member Functions

- Member functions can be declared in 2 ways:
  - By specifying the function prototype
  - By specifying the function implementation
- Java allows only the second method

```
class Time {
public:
    void print() const;
    void set(int h, int m, int s) {
        hour = h;
        minute = m;
        second = s;
    }
    Time();
    Time(int, int, int);

private:
    int hour;
    int minute;
    int second;
};
```

Implementing Functions Separately

- For member functions that are not implemented in the class declaration, they must be implemented separately



```
class Time {
public:
  void print() const;
  void set(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
#include <cstdio>

void Time::print() const
{
  printf("%2d:%2d:%2d", hour,
    minute, second);
}
```

Inline Functions

- Including the implementation of a function within the class definition is an implicit *request* (to the compiler) to make a function **`inline`**

- When a function is inline, the compiler does not make a function call
  - The code of the function is used in place of the function call (function call is replaced by function code and appropriate argument substitutions made)

  - Compiled code may be slightly larger, but will execute faster because function call overhead is avoided

- To explicitly request to make member functions inline
  - Add `inline` keyword before return type in function declaration and definition

Explicit Inline Request

- Add `inline` keyword before return type in function declaration and definition

```
class Time {
public:
  inline void print() const;
  void set(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
#include <cstdio>

inline void Time::print() const
{
  printf("%2d:%2d:%2d", hour,
    minute, second);
}
```

Inline Functions

- **Not** all inline requests are granted by the compiler

- Reasons for not granting inline requests:
  - Function is recursive
  - Function contains `switch` or `goto` statement
  - Function return type is other than void, and the return statement doesn’t exist in function body
  - Function contains a loop (`for`, `while`, `do-while`)
  - Function contains static variables

Example: Accessing Members

```cpp
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

```cpp
// Creates instance using
// default constructor
Time myTime;

// Invokes member function
myTime.set(10, 30, 0);

// This is not allowed.
myTime.hour = 12;
```

Member access operator, assuming Time() and set() are defined

Static Members

- **C++ classes can contain static members**
- A static member variable is a variable that is **shared** by all instances of a class
    - Non-static members are not shared: every object maintains a copy of non-static data members
- Static member variables are often used to declare class constants
- A static member function is a special member function, which is used to access only static data members
- Member functions and variables can be made static by using the **`static`** qualifier
- Static members can be accessed using class name

Example

```
class Time {
public:
  void set(int, int, int);
  void print() const;
  static int getCounter();
  Time();
  Time(int, int, int);

private:
  int hour;
  int minute;
  int second;
  static int counter;
};
```

```
Time::Time() {
  hour = 0; minute = 0; second = 0;
  counter++;
}

Time::Time(int h, int m, int s){
  hour = h; minute = m; second = s;
  counter++;
}
…
// Initialize static member variable
int Time::counter = 0;

// Define static member function
int Time::getCounter()
{
  return counter;
}
```

Static members are **only declared in a class declaration**. They must be **explicitly defined** outside the class using the scope resolution operator. The **static keyword is only used with the declaration** of a static member, inside the class definition, but not with the definition of that static member

Example (continued)

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

…

int main(void)
{
    cout << Time::getCounter() << "\n";
    Time t1;
    cout << Time::getCounter() << "\n";
    Time t2(10,0,0);
    cout << Time::getCounter() << "\n";

    return 0;
}
```

Output:

```
0
1
2
```

Overloading

- Create two or more members having the **same name** declared in the same scope.

- C++ supports

  - **Function (Method) overloading**

  - **Operator overloading**

Function Overloading

- Two or more function with the same name, but different in parameters.
- Function overloading increases the readability of the program because you don't need to use different names for the same action.

```
class Cal {
public:
    int add(int a, int b) {
        return a + b; }
    int add(int a, int b, int c) {
        return a + b + c; }
};
```

```
int main(void) {
    Cal C;
    cout << C.add(10, 20) << " ";
    cout << C.add(12, 20, 23);
    return 0;
}
```

Output:

```
30  55
```

Operator Overloading

- Operators have different implementations (meanings) with different arguments
- The extraction operator >> and the insertion operator << are overloaded
  - They perform the I / O operation based on the type of argument
- Operators can be overloaded to have different meaning for user defined classes (will be covered later)

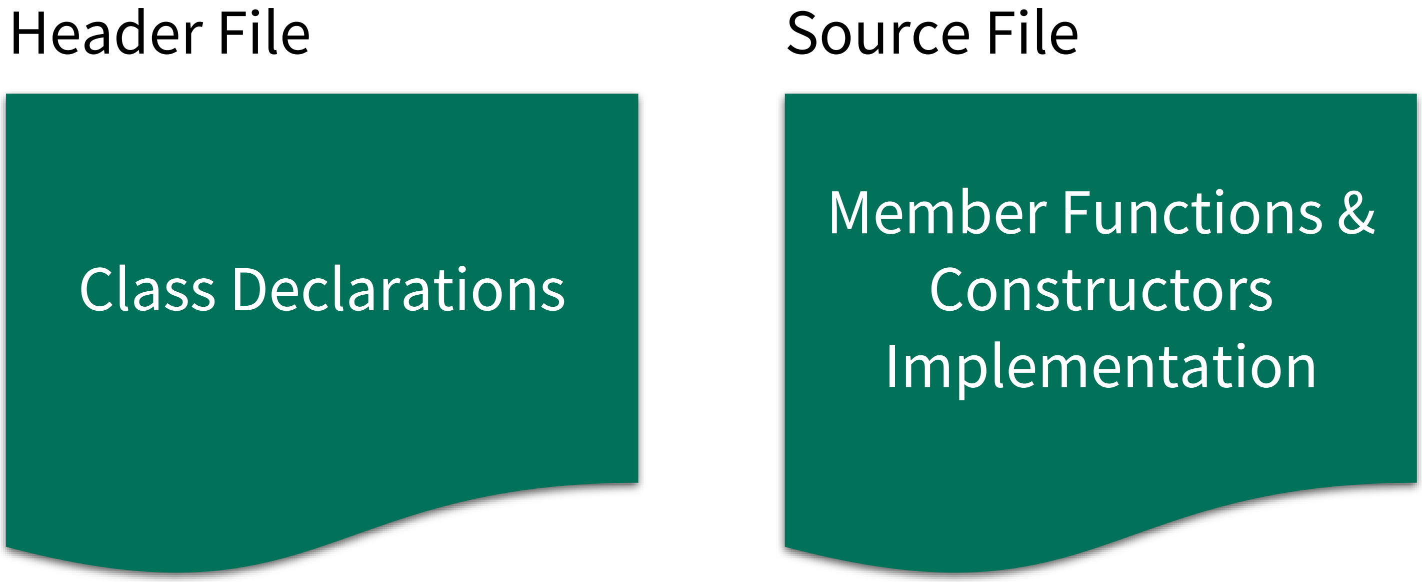
```
int a = 10, b = 20;
string s = "Hello", s1 = "World";
s = s + " " + s1;
a = a + b;
cout << "a = " << a << endl << "s = " << s;
```

Output:

```
a = 30
s = Hello World
```

Where to Declare and Implement Classes and Member Functions

- Good programming practice is to declare the class in a header file
- Separate the implementation of the member functions (and possibly constructors) in another source file



Header File

Class Declarations

Source File

Member Functions & Constructors Implementation

Example

time.h

```
class Time {
public:
  void set(int, int, int);
  void print() const;
  Time();
  Time(int, int, int);

private:
  int hour;
  int minute;
  int second;
};
```

time.cpp

```
…
#include "time.h"
Time::Time() {
  hour = 0; minute = 0; second = 0;
}

Time::Time(int h, int m, int s){
  hour = h; minute = m; second = s;
}

void Time::set(int h, int m, int s) {
  hour = h; minute = m; second = s;
}

void Time::print() const {
  printf("%2d:%2d:%2d", hour, minute, second);
}
```

Note other extensions can also be used. Common examples are .cc, .cp for source files; and .hh, .hpp for header files.

Next Lecture

- Inheritance
- Containers