Week 9 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Content

- Classes in C++
  - Introduction
  - String in C++
  - Revisit Structures in the context of C++

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

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 functions

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`

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`

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

      student_info(int, string);
   };

 student_info:: student_info(int id,
string s)
{
     student_id = id;
     name=s;}
```

```
//declare an instance (object) of this class
Student_info s1;
student_info 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 contains a loop (for, while, do-while)
  - Function contains static variables
  - Function is recursive
  - Function return type is other than void, and the return statement doesn’t exist in function body
  - Function contains switch or goto statement

Example: Accessing Members

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

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

```
// 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

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;
}
```

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 stream 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

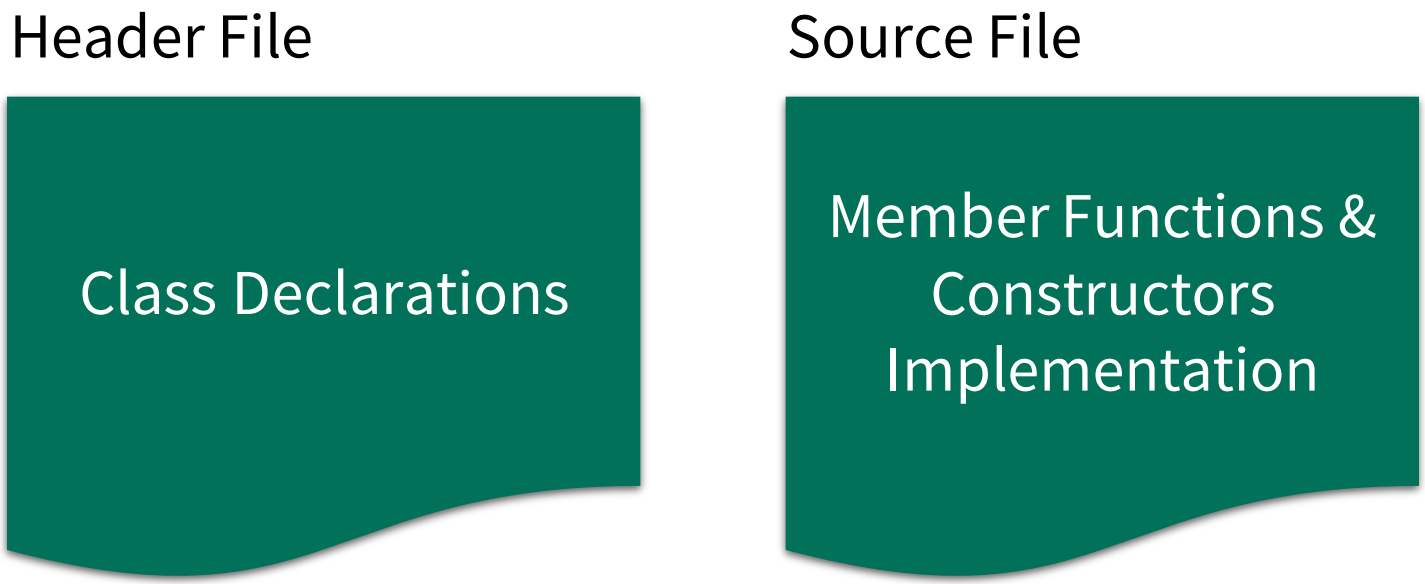
- You may declare classes and implement the member functions in the same C++ source file

- Disadvantage: other sources will not be able to use the class

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

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



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.

Strings in C++

Strings in C++

- **C Strings**
  - A one-dimensional array of characters.

- Standard C++ Library string class - **std::string**
  - A container for handling char arrays

Recap: C Strings

- A string is a sequence of characters that is terminated by a null character
- In C a string is actually a one-dimensional array of characters
- There are functions declared to manipulate strings in string.h
- Supported in C++. The cstring library can be used.
- Code safety / security is the responsibility of the programmer

strcpy(s1, s2); Copies string s2 into string s1.
strcat(s1, s2); Concatenates string s2 onto the end of string s1.
strlen(s1); Returns the length of string s1.
strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

std::string in C++

- C++ has a **`string`** class type that implements string datatype
- Not *exactly similar* to Java String class
  - Java strings are immutable reference types. C++ strings are mutable.
- Wide range of operators and member functions are available for variables declared as string type
- Include **`<string>`** header file
- Use **`standard`** namespace

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
 string s1,s2; // empty
 s1 = "Hello";
 s2 = "Hello World !";

 cout<<"String 1:"<<s1<<"Length:"<<s1.length()<<endl;

 cout<<"String 2:"<<s2<<"length:"<<s2.length()<<endl;

return 0;
}
```

Character array Vs String class in C++

String Operation	Character Array	String class
Copy string s2 into string s1.	`strcpy(s1, s2);`	`s1 = s2;`
Concatenates string s2 onto the end of string s1.	`strcat(s1, s2);`	`s1 + s2;`
Returns the length of string s1.	`strlen(s1);`	`s1.length();`
Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.	`strcmp(s1, s2);`	`==, <=, >, and >=` operators Or `s1.compare(s2)`
Returns a pointer to the first occurrence of character ch in string s1.	strchr(s1, ch);	`s1.find(ch)`
Returns a pointer to the first occurrence of string s2 in string s1.	strstr(s1, s2);	`s1.find(s2);`

Strings Examples – C++

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string s1,s2; // empty
s1 = "Hello";
s1 += " World !";
s2 = "Hello World !";

cout<<"String 1 "<<s1<<" Length:"<<s1.length()<<endl;
cout<<"String 2 "<<s2<<" length:"<<s2.length()<<endl;
if(s1==s2)
    cout<<"Strings are equal"<<endl;
else
    cout<<"Strings are not equal<<endl";
cout<<"First character of s1 is: "<<s1[0];
return 0;
}
```

Output:

String 1 Hello World ! Length:13
String 2 Hello World ! length:13
Strings are equal
First character of s1 is: H

Array of Strings Examples – Char array

```
#include <stdio.h>

int main(){
    char fish[][11] = {"terakihi","snapper","flounder","guppy" };
    printf("%s", fish[1]);
    return 0;
}
```

Output: snapper

Array of Strings Example – String class

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

int main(){
    string colour[4] = {"Violet", "Red", "Orange", "Yellow"};
    cout << colour[1] << "\n";

    return 0;

}
```

Output: Red

Revisit Structures in the context of C++

Structure in C Vs Structure in C++

- C++ structures adds **extra features** to C structures

- Same declaration syntax

- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

Structure in C Vs Structure in C++

- C++ structures adds **extra features** to C structures
- Same declaration syntax
- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct s
{
  int a;
  int b;
  void set()
    {
       a=10;
       b=20; } };
```

Structure in C Vs Structure in C++

- C++ structures adds **extra features** to C structures
- Same declaration syntax
- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct s
{
  int a;
  int b;
  void set()
    {
       a=10;
       b=20; } };

 struct s s1;
 s s1;
```

Structure in C Vs Structure in C++

- C++ structures adds **extra features** to C structures
- Same declaration syntax
- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

```
struct base
{
   :
   :
 };

struct derived: base {

}
```

Structure in C Vs Structure in C++

- C++ structures adds **extra features** to C structures

- Same declaration syntax

- C++ structures can –
  - have functions as members
  - treated like a *built-in* data type
  - be extended (supports inheritance)
  - define access specifiers (public, private, protected)

Members are public by default.

```
struct base
{
  public:
  int a;
  private:
  int b;
  protected:
  int c;
 };

struct derived: base {

}
```

Next Lecture

- More on Classes