Week 11 Lecture 1

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Content

- Files - Random access (carry forward from last week)
- Dynamic memory allocation (carry forward from last week)
- Revisit Constructors and Destructors
- Friends

Files - Random access

Recap: File Streams

- `ifstream` – stream class to **read** from files
- `ofstream` – stream class to **write** to files.
- `fstream` - stream class to both **read (from) and write (to)** files.

Recap: File Open Modes

Name	Description
`ios::in`	Open file to read (default for ifstream)
`ios::out`	Open file to write (default for ofstream)
`ios::app`	Output operations happen at the end of the file, appending to its existing contents.
`ios::ate`	The stream's position indicator is set to the end of the file.
`ios::trunc`	Deletes all previous content in the file. (empties the file)
`ios::nocreate`	If the file does not exists, new file is not created
`ios::noreplace`	If the file exists, trying to open it with the open() function, returns an error.
`ios::binary`	Opens the file in binary mode.

`ios::in` is default for ifstream
`ios::out` is default for ofstream
`ios::in |ios::out` is the default for fstream

Random access

- Each file stream contains a **file stream pointer** to track current read / write position within a file.

- To know current position :
  - `tellg()` function (for input) and
  - `tellp()` function(for output) files

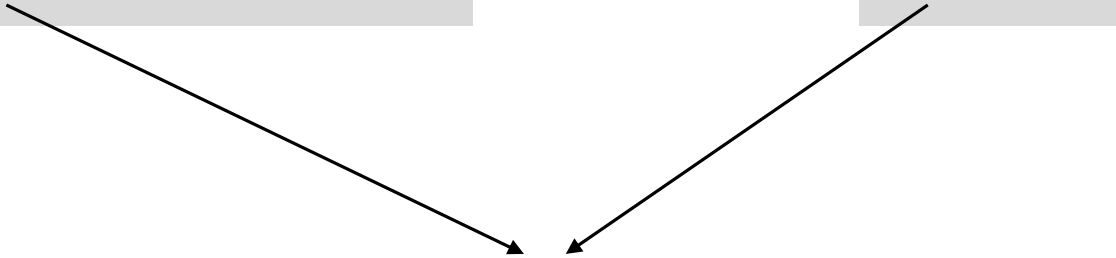
- For manipulating the file pointer:
  - `seekg()` for input and
  - `seekp()` for output.

Random access

- To know current position :
  - `tellg()` function returns the position of the current character in the input stream.
  - `tellp()` function returns the position of the current character in the output stream.

```
streampos tellg();
```

```
streampos tellp();
```



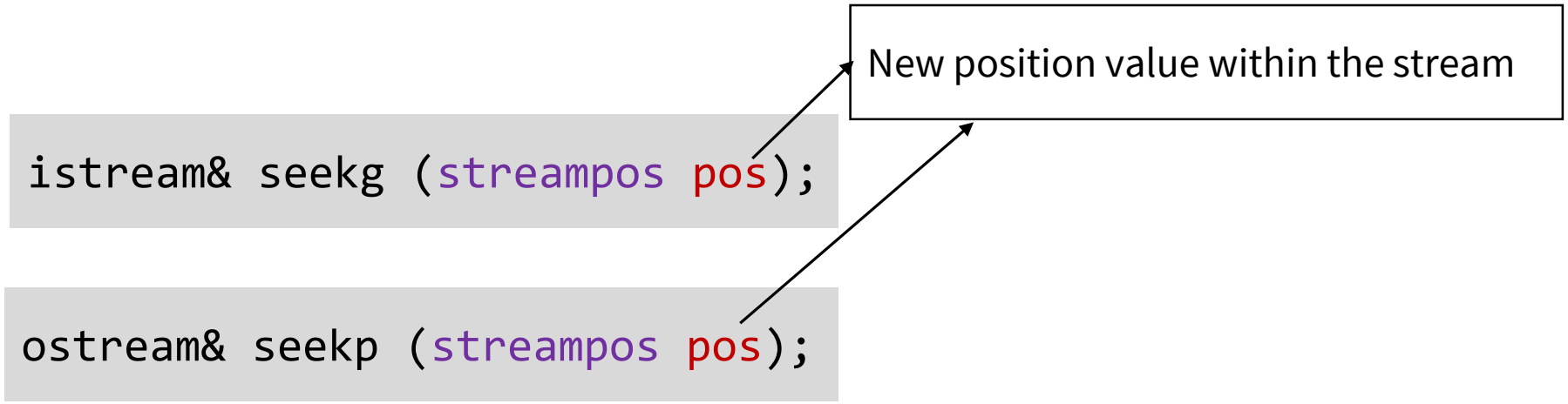
File position data type. Can be converted to integers

Random access

- To manipulate current position :
  - `seekg()` function sets the position of the next character to be extracted from the input stream
  - `seekp()` function sets the position where the next character is to be inserted into the output stream.
  - Two methods to set a position :
    - Set an **absolute** position
    - Set a **relative** position

Random access

- Absolute Position



Streampos:
File position data type. Can be converted to integers

Random access

- Relative Position

```
istream& seekg (streamoff off, ios_base::seekdir way);
```

Offset value relative to the way parameter

```
ostream& seekp (streamoff off, ios_base::seekdir way);
```

Object of type ios_base::seekdir. Can take any one of the following values

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

Dynamic Memory Allocation

Using malloc( ) and calloc( ) in C++

```
void *malloc(size_t size);
```

```
int main(){

    int *p;
    p= malloc(sizeof(int));
    *p=5;
    printf("%d", *p);
}
```

Valid in C. C allows void pointers to be implicitly converted to any other pointer type.

Invalid in C++: invalid conversion from ‘void*’ to ‘int*

Strong type checking in C++

```
void *calloc(size_t nmemb, size_t size);
```

```
int main(){

    int *p;
    p= (int*)malloc(sizeof(int));
    *p=5;
    printf("%d", *p);
}
```

Valid in C.

Valid in C++

Dynamic Memory Allocation

- C++ uses `new` and `delete` **operators** to create and destroy dynamic variables

- The `new` operator allocates memory for the variable and returns a pointer to it.

```
new datatype;
```

Allocates memory for datatype and returns a pointer to this memory location **(single object form)**

```
new datatype[an expression that evaluates to an integer];
```

Allocates memory for an **array** of datatype and returns a pointer to this memory location

Dynamic Memory Allocation

- The `delete` operator is used to return / release the memory that was allocated using the `new` operator.

```
delete ptr;
```

→ Deallocates memory pointed to by ptr**(single object form)**

```
delete [] ptr;
```

→ Deallocates memory block pointed to by ptr **(array form)**

Note:
new and delete call the constructors and destructors of objects respectively, whereas malloc, calloc, realloc and free do not.
new returns pointer to exact data type, while malloc() returns void *
On failure, malloc() returns NULL where as new throws bad_alloc exception.

Revisit Constructors and Destructors

Recap: Constructors

- A constructor is a member function which **initializes** an object of a class or structure.

- A constructor has:

  the **same name** as the class or structure itself and has **no return type** .

- You may declare more than one constructor for a class or structure, each one must have a different function prototype(different arguments)

- Concept similar to Java. Difference is in the way they are invoked.

Recap: 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 {

      string name;
      int student_id;

 public:
      void print();
      student_info()
     {
        student_id = 1;
        name = "John";}
      };
```

Constructor

Implicit call

```
//declare an instance (object) of
this class
student_info  s;

student_info s1 = student_info();
```

Explicit call

Example

```
class student_info {
   int student_id;
   string name;

 public:
   void print();
   student_info(int,string);
   student_info(const student_info &s);
};
student_info:: student_info(int id, string s){
   student_id =id;
    name =s;  }

student_info:: student_info(const student_info &s){
   student_id = s.student_id;
   name = s.name; }
```

```
student_info s(1, "John");
student_info s1(s);
```

Parameterized Constructor

Copy Constructor

Constructors

- A constructor is called **automatically** whenever a new instance of a class or structure is created

- If no user-declared constructors of any kind are provided for a class, the compiler will implicitly define a **default constructor** and a **copy constructor** as public members of its class.

- An implicitly defined default constructor expects **no parameters and has an empty body**.

- Default constructor is **not** automatically provided, if class definition includes any constructor definition.

Example

```
class student_info {

      int student_id;
      string name;

 public:
      void print();

};
```

Valid, Default constructor implicitly defined by compiler, if no other constructor defined.

```
//declare an instance (object)
of this class
student_info s1;

student_info s2(12 ,"John");
```

Not valid, constructor with arguments is not defined

Example

```
class student_info {
      int student_id;
      string name;
   public:
      void print();
      student_info(int, string);
   };

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

Not valid, Default constructor not defined implicitly.

```
//declare an instance
(object) of this class
student_info  s1;

student_info s2(12 ,"John");
```

Valid, parameterized constructor is defined

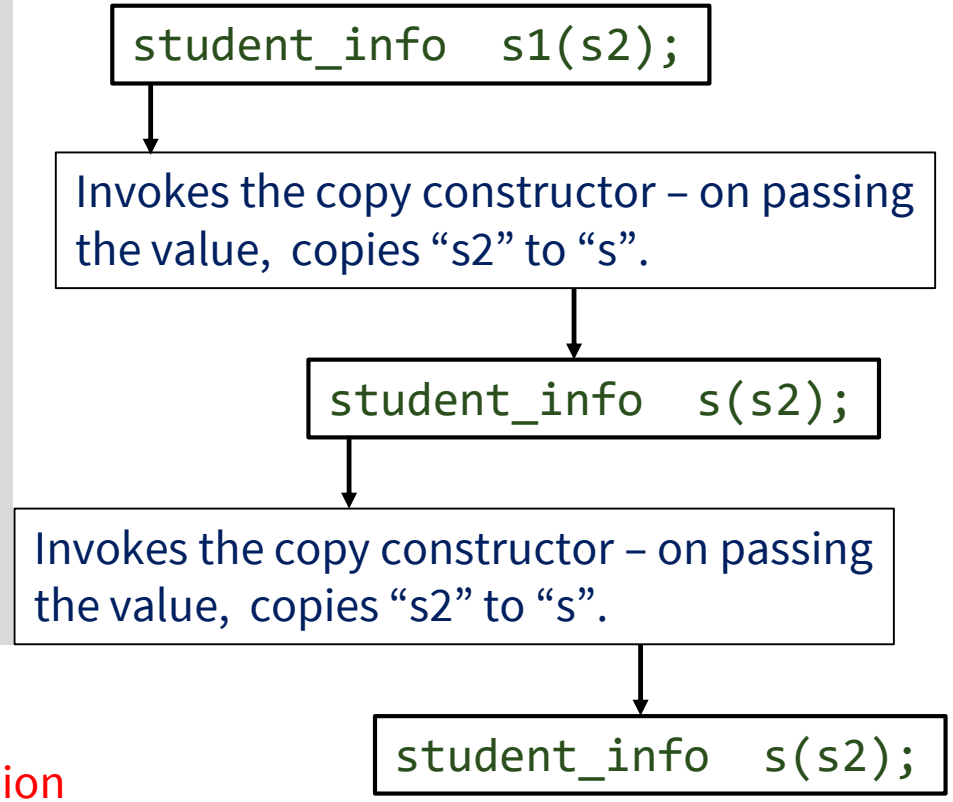
A clarification on Copy constructor

- It is necessary to declare a parameter to a copy constructor as a reference.

```
class student_info {
   int student_id;
   string name;

 public:
  student_info(const student_info s);
};

student_info:: student_info(const student_info s){
   student_id = s.student_id;
   name = s.name; }
```



Infinite recursion

Copy Constructor

A copy constructor is invoked automatically whenever a new object is created from the same class's existing object.

It happens in the following case:

a) When defining an object, it is initialized with an existing object of the same class.

b) When an object is passed by value to a function.

c) When an object is returned by value from a function.

Using member initializer list

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

OR
//Since C++11
student_info :: student_info(int id, string s):
student_id{id},name{s} {}
```

Initializing Objects

- **Default Initialization**
  - Initialization using a default constructor
  - Undefined values if default constructor is not included

```
student_info s;
```

- **Value Initialization**
  - Initialization using a default constructor.
  - If no constructor given, values of data members are set to zero (0 for `int`, `'\0'` for string etc.)

```
student_info s{};
```

Initializing Objects

- **Direct Initialization**
  - Searches for compatible constructor
  - Cannot be kept empty with parenthesis

```
student_info s{1, "John"};
student_info s(1, "John");
student_info s();//Not allowed
```

- **Copy Initialization**
  - Copies values from other objects
  - Uses the copy constructor
  - Copy constructor provided by default (if not defined by the programmer)

```
student_info s{1, "John"};
student_info s1(s);
OR

student_info s1 = s;
```

Since C++11, supports brace initialization of basic data types as well.
`int a = 5;` `int a(5);` and `int a{5};` are all valid.

Explicit Constructor

```
class A
{
public:
// single parameter constructor
  A (int x) : i (x) { }

  int  get() { return i; }

private:
  int i; };
```

```
void printobject (A a)
{
  int i = a.get();
  cout<<"Object : "<<i<<endl;
}
int main(){
    A a1(2);
    printobject(a1);

   printobject(10);
}
```

```
Output:
Object  : 2
Object : 10
```

The compiler is allowed to make implicit conversion to resolve the parameters to a function.

Explicit Constructor

- Prefixing the **explicit** keyword to the constructor prevents the compiler from using that constructor for implicit conversions.

```
class A
{
public:
// single parameter constructor
  explicit A (int x) : i (x) { }

  int  get() { return i; }

private:
  int i; };
```

```
void printobject (A a)
{
  int i = a.get();
  cout<<"Object : "<<i<<endl;
}
int main(){
    A a1(2);
   printobject(a1);

   printobject(10);
}
```

Compilation error

Destructors

- A destructor is a member function which is called when the instance (objects) memory is about to be destroyed
- It is the **clean up** function
- It is usually called when the object goes out of scope
- A destructor takes **no arguments and has no return type**.

```
class student_info {
      int student_id;
      string name;

 public:
      void print();
      student_info();
      student_info(int);
      ~student_info(); //destructor

};

student_info:: ~student_info() {
      cout<< "bye"<<endl;
}
```

Destructors

- If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor.

- When a class or structure contains a **pointer** that the programmer dynamically allocates, it is the programmers responsibility to **release the memory** before the instance (object) is destroyed.

- In general, if a constructor acquires resources, a destructor should release them.

Friends

Friend Declaration

- One of the important concepts of OOP is data hiding, i.e., a non-member function cannot access an object's private or protected data.
- But, sometimes this restriction may force programmer to write long and complex codes.
- So, there is mechanism built in C++ programming to access private or protected data from non-member functions.
- This is done using a **friend** declaration.
- A friend declaration can be applied to both a **function** and a **class**.

Friend Function

- If a function is defined as a friend function then, the private and protected data of a class can be accessed using this function.
- For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

```
class class_name {
       class_member_list;
     friend return type function_name(argument_list);
};

return type function_name(argument_list) {
   … .. …
   // Private and protected data of class_name can be accessed from this function
   … .. …
}
```

Example

```
class num
{
    private:
        int m;
    public:
        void set(int a)
        { m=a;
        }
        //friend function
        friend int add(num);
};
```

Function declared as friend of class num

Can accesses private member of num

```
// friend function definition
int add(num d)
{
    d.m += 5;
    return d.m;
}
int main()
{
    num N;
    N.set(5);
    cout<<"Num: "<< add(N);
    return 0;
}
```

Example

```
class num
{
    private:
        int m;
    public:
        void set(int a)
        { m=a;
        }
        int get(){
            return m;
        }
        //friend function
        friend num add(num,num);
};
```

```
num add(num a, num b)
{
    num c;
    c.m = a.m + b.m ;
    return c;
}
int main()
{
    num N1, N2, N3;
    N1.set(5);
    N2.set(6);
    N3= add(N1,N2);
    cout<<"Sum="<<N3.get();
    return 0;
}
```

Friend Class

- A class can also be made a friend of another class using keyword friend.
- When a class is made a friend class, all the member functions of that class becomes friend functions.

```
class A {
      class_member_list;

};

class B {
      class_member_list;

      friend class A;

};
```

all member functions of class A will be friend functions of class B

Next Lecture

- More on containers
- Templates