Week 11 Lecture 1

NWEN 241
Systems Programming

Alvin Valera

Alvin.valera@ecs.vuw.ac.nz

Content

- Files
- Revisit Constructors
- Dynamic Memory Allocation

Recap: Steps for File I/O

1. Create file stream objects

```
ifstream fsIn;    //input
ofstream fsOut;   // output
fstream  fsBoth; //input & output
```

2. Open the file

File open mode

```
fsIn.open("data.txt", fileopenmode);
```

OR Combine the two steps

```
ifstream fsIn("data.txt", fileopenmode);
```

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

Steps for File I/O

3. Check if the file is opened properly. Every stream object has an `is_open( )` method that returns `true` if a file is open and associated with the stream object, otherwise it returns `false`.

```
if(!fsOut.is_open())
    cerr<<"Failed to open file"
```

```
fsOut.bad() //true if reading writing operation fails
fsOut.fail() // true if bad + format errors
fsOut.eof() // true if file has reached the end
```

True if none of the stream's error state flags (eofbit, failbit and badbit) is set

```
if(fsOut.good())
    fsOut<<"Hello World";
```

4. Read and write to files using the functions defined in the stream's public interface

Steps for File I/O

5. Close the file after use.

```
fsIn.close(); //Close files
fsOut.close();
fsBoth.close();
```

Note: All `istreams` (and `ifstreams`) have a method **`eof( )`** that returns a boolean value of true if an attempt is made to read past the end of the file, and false otherwise.

Formatted Output

- Formatted output is carried out on streams using the stream insertion << operator for all basic and overloaded data types.

```
int a = 1;
char c = 'A';
string s = " Sam P";

ofstream fout("example.txt");

fout << a << " " << c << " " << s;
fout.close();
```

example.txt

```
1 A  Sam P
```

Formatted Input

- Formatted input is carried out on streams using the stream extraction operator >> for all basic and overloaded data types.

```
int a1 ;
char c1 ;
string s1 ;

ifstream fin("example.txt");

fin >> a1 >> c1 >> s1;
cout << a1 << c1 << s1;
fin.close();
```

example.txt

```
1 A  Sam P
```

Output

```
1 A Sam
```

Note: >> (extraction operation) Operator does not consider white space characters as part of strings (white space characters causes extraction to terminate)

Unformatted I/O

Single Character - Input	
Name	**Description**
`int get ();`	Extracts a character from a stream and returns as int.
`istream & get (char & c);`	Extracts a character from a stream, stores it in c and return the invoking istream reference

Single Character - Output	
Name	**Description**
`ostream & put (char c);`	Inserts character c into the stream and return the invoking ostream reference

Allows cascaded use – as they return the invoking stream reference.

Example

```
char c;
ofstream fout("Data.txt");

while( (c=cin.get()) != '\n' )
    fout.put(c);

fout.close();
```

Input

```
Welcome NWEN241
```

Data.txt

```
Welcome NWEN241
```

`int get()` returns ASCII value of the character read

Example

```
int main ()
{
  char c1, c2;
  cout<<"Enter 2 characters without space: ";

  cin.get(c1).get(c2);

  cout<<"\nYou entered: ";
  cout.put(c1).put(' ').put(c2);

  return 0;

}
```

`get(ch) and put(ch)` can be cascaded, as they returns the invoking stream reference

Unformatted I/O

- C - String

1.
```
istream& get (char* s, streamsize n);
```

2.
```
istream& get (char* s, streamsize n, char delim);
```

c-string

Extracts n-1 characters from the stream

User specified delimiter

- `Default` `Delim` in (1) is '\n'
- Extracts characters until either the extracted character is `delim` or `n-1` characters have been read.
- **Appends** null character ('\0') to s.
- **Keeps** the `delim` char in the input stream

Unformatted I/O

- C - String

c-string

Extracts n-1 characters from the stream

User specified delimiter

3.
```
istream& getline (char* s, streamsize n);
```

4.
```
istream& getline (char* s, streamsize n, char delim);
```

- Same as `get()`, but **extracts and discards** `delim` char from the input stream

Example

```
int main ()
{
    char c[20];
    char c1;

    cin.get(c,20);
    cin.get(c1);

    cout << c << " " << c1;
}
```

```
int main ()
{
    char c[20];
    char c1;

    cin.getline(c,20);
    cin.getline(c1);

    cout << c << " " << c1;
}
```

- `get(c, 20)` keeps '\n' in the input stream.
- Reads `'\n'` into c1

- `getline(c,20)`Discards '\n' from the input stream.

Unformatted I/O

- String class

Input stream

String object

1.
```
istream& getline (istream& is, string& s);
```

User specified delimiter

2.
```
istream& getline (istream& is, string& s, char delim);
```

- `Default` `Delim` in (1) is '\n'
- Extracts characters until the extracted character is `delim` character is found.
- **Discards** the `delim` char

Example

```
int main(void)
{
    string s;
    ofstream fOut;
   fOut.open("Data.txt");

    getline(cin, s);

    fOut<<s;
    fOut.close();
    return 0;
}
```

Input	Welcome !!
Data.txt	Welcome !!

Revisit Constructors

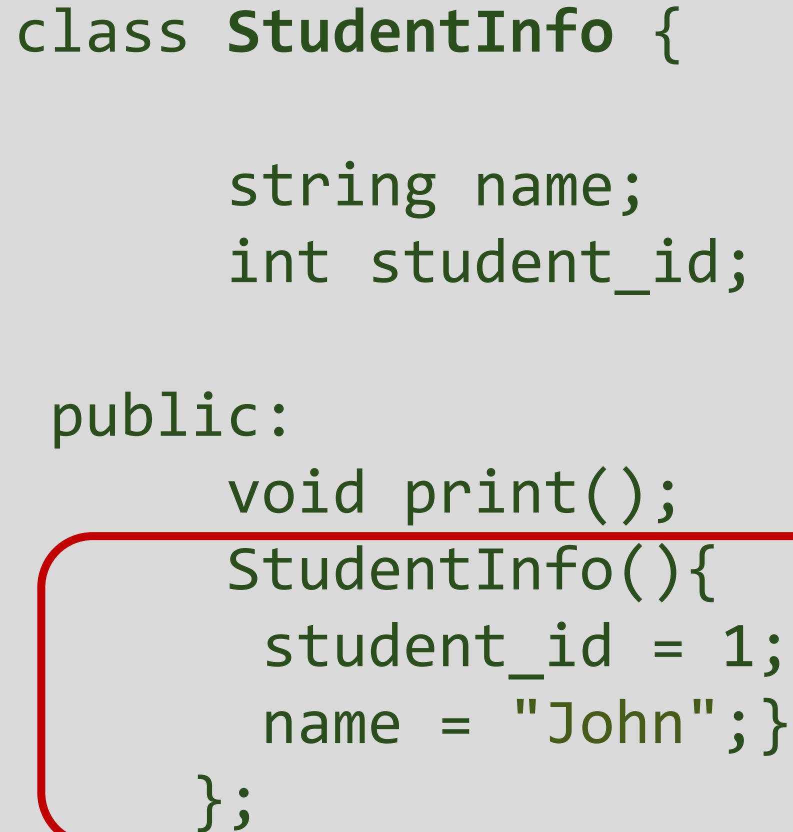
Recap: Types of Constructors

- **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 & )`

Implicit vs Explicit Call



```
//declare an instance (object) of this class
```

Parameterized vs Copy Constructor

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

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

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

```
StudentInfo s(1, "John");
StudentInfo 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 StudentInfo {
      int student_id;
      string name;
public:
      void print();

};
```

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

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

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

Not valid, constructor with arguments is not defined

Example

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

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

Not valid, Default constructor not defined implicitly.

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

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

Valid, parameterized constructor is defined

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.

```
StudentInfo s(1,"Sam");
StudentInfo s2  = s1;
```

```
print(StudentInfo s) {
    …
}
```

```
StudentInfo returnStudent() {
    return s3; // s3 is StudentInfo
}
…
StudentInfo s = returnStudent();
```

Constructor Definition: Using member initializer list

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

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

More About Initializing Objects

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

```
StudentInfo 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.)

```
StudentInfo s{};
```

More About Initializing Objects

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

```
StudentInfo s{1, "John"};
StudentInfo s(1, "John");
StudentInfo 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)

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

Constructors in Implicit Conversion

```
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

Next Lecture

- Dynamic Memory Allocation
- Friends
- Templates