TP _ C++ OOP

TABUEU FOTSO Laurent Cabrel
PhD Candidate

Use case 1 : Student.h file

```
#include <iostream>
#include <cstring>
using namespace std;
class Student
{
// private members are accessible only within the scope of
// the class (e.g. within member functions or friends)
private:
    char *firstName; // data members
    char *lastName;
    char middleInitial;
    float gpa;
    char *currentCourse;
    char *studentId;
public:
// public members are accessible from any scope
// member function prototypes
    void Initialize();
    void Initialize(const char *, const char *, char, float,
                    const char *, const char *);
    void CleanUp();
    void Print();
};
```

Use case 1 : Student.cpp file (1/3)

```
#include "Student.h"
void Student::Initialize()
{
    firstName = lastName = 0; // NULL pointer
    MiddleInitial = '\0'; // null character
    gpa = 0.0;
    currentCourse = studentId = 0;
}
// Overloaded member function definition
void Student::Initialize(const char *fn, const char *ln, char mi, float avg, const char
*course, const char *id)
{
    firstName = new char [strlen(fn) + 1];
    strcpy(firstName, fn);
    lastName = new char [strlen(ln) + 1];
    strcpy(lastName, ln);
    middleInitial = mi;
    gpa = avg;
    currentCourse = new char [strlen(course) + 1];
    strcpy(currentCourse, course);
    studentId = new char [strlen(id) + 1];
    strcpy (studentId, id);
```

Use case 1 : Student.cpp file (2/3)

```
// Member function definition
void Student::CleanUp()
{
delete firstName;
delete lastName;
delete currentCourse;
delete studentId;
}
// Member function definition
void Student::Print()
{
cout << firstName << " " << middleInitial << ". ";
cout << lastName << " with id: " << studentId;
cout << " has gpa: " << gpa << " and enrolled in: ";
cout << currentCourse << endl;
}
```

Use case 1 : main.cpp (3/3)

```
include <iostream>
#include "Student.h"

using namespace std;

int main()
{
    Student s1;
    // Initialize() is public; accessible from any scope
    s1.Initialize("Laurent", "Fotso", 'I', 3.9, "C++", "178GW");
    s1.Print(); // Print() is public, accessible from main()
// Error! firstName is private; not accessible in main()
//cout << s1.firstName << endl;

// CleanUp() is public, accessible from any scope
    s1.CleanUp();
    return 0;
}
```

Member Functions

In order to write any member function's body, it is usually done outside the definition (the only difference between regular functions and members is the presence of ClassName:: as part of the name) in the following format:

```
ClassName::MemberFunction()
{
}
```

When a member function definition inside the class definition is followed by the **const** keyword, it means that the function will not change any of the (nonstatic) data members of that object**.**

```
class MDate {
public:
    bool IsValid() const;
private:
    int M, D, Y;
};
```

This means that the code inside this function will not change any of the nonstatic data members, such as M, D, and Y.

Inline Functions

Inline functions can be created within the class definition, without the **inline** keyword. The format requires the function's body to follow the member definition:

```
class Example
{
public:
Example(); // Constructor
~Example (); // Destructor
int IsExampleEmpty( void ) { return( IsEmpty ); }; // Automatic inline function
private:
int IsEmpty;
};
```

Constructor and Destructor

Inside the public section just shown, the functions **ClassName** and **~ClassName** are defined (like in a prototype). These are the constructor and destructor for this class. The constructor is called whenever an instance of a class is created, and the destructor is called when the instance is destroyed.

NOTE: The constructor and destructor are functions with the same name as the class, and have no return type. The destructor name is preceded with the tilde (~), and never accepts an argument. The constructor may be overloaded. Constructors and destructors are not required, but are most commonly implemented.

The copy constructor (instead of the default constructor) is called when an object is passed to a function, for autoinitialization, or when an object is returned from a function.

```
class Mobject
{
public:
MObject(); // Regular constructor
Mobject( const Mobject &Original );// Note Original is not a required name.
}
```

Use Case on Constructor

Let's introduce a simple example to understand constructor basics:

University.h file

```
#include <iostream>
#include <cstring>
using namespace std;
class University
{
private:
    char name[30];
    int numStudents;
public:
// constructor prototypes
    University(); // default constructor
    University(const char *, int);
    void Print();
};
```

University.cpp file

```
#include "University.h"

University::University()
{
    name[0] = '\0';
    numStudents = 0;
}
University::University(const char * n, int num)
{
    strcpy(name, n);
    numStudents = num;
}
void University::Print()
{
    cout << "University: " << name;
    cout << " Enrollment: " << numStudents << endl;
}
```

Use Case on Copy Constructor (Student.h file)

```
#include <iostream>
#include <cstring> // though we'll prefer std::string, we will need to illustrate one dynamically
                   //allocated
// data member to show several important concepts  (so we'll use a single char *)
using namespace std;
class Student
{
private:
    // data members
    string firstName;
    string lastName;
    char middleInitial;
    float gpa;
    char *currentCourse;  // though we'll prefer std::string, this data member will help us
                          // illustrate a few important ideas
public:
    // member function prototypes
    Student();  // default constructor
    Student(const string &, const string &, char, float, const char *);
    Student(const Student &);  // copy constructor prototype
    void CleanUp();
    void Print();
    void SetFirstName(const string &);
```

Use Case on Copy Constructor (Student.cpp file)

```
// default constructor
Student::Student()
{
    // Remember, firstName and lastName are member objects of type string; they are default
//constructed and hence
    // 'empty' by default. They HAVE been initialized.
    middleInitial = '\0';
    gpa = 0.0;
    currentCourse = nullptr;
}

// Alternate constructor member function definition
Student::Student(const string &fn, const string &ln, char mi, float avg, const char *course)
{
    firstName = fn;
    lastName = ln;
    middleInitial = mi;
    gpa = avg;
   // remember to dynamically allocate the memory for data members that are pointers
    currentCourse = new char [strlen(course) + 1];

    strcpy(currentCourse, course);
}
```

Use Case on Copy Constructor (Student.cpp file)

```
// Copy constructor definition - implements a deep copy
Student::Student(const Student &s)
{
    firstName = s.firstName;   // string will do a deep assignment for us
    lastName = s.lastName;
    middleInitial = s.middleInitial;
    gpa = s.gpa;
    // for ptr data member -- allocate necessary memory for destination string
    currentCourse = new char [strlen(s.currentCourse) + 1];
    // then copy source to destination string
    strcpy(currentCourse, s.currentCourse);
}
```

Use Case on Copy Constructor (Student.cpp file)

```
// Member function definition
void Student::CleanUp()
{
    delete currentCourse;   // deallocate the memory for data members that are pointers
}

// Member function definition
void Student::Print()
{
    cout << firstName << " " << middleInitial << ". ";
    cout << lastName << " has a gpa of: " << gpa;
    cout << " and is enrolled in: " << currentCourse << endl;
}

void Student::SetFirstName(const string &fn)
{
    firstName = fn;
}
```

Use Case on Copy Constructor (Student.cpp file)

```
// Member function definition
void Student::CleanUp()
{
    delete currentCourse;   // deallocate the memory for data members that are pointers
}

// Member function definition
void Student::Print()
{
    cout << firstName << " " << middleInitial << ". ";
    cout << lastName << " has a gpa of: " << gpa;
    cout << " and is enrolled in: " << currentCourse << endl;
}

void Student::SetFirstName(const string &fn)
{
    firstName = fn;
}
```

Use Case on Copy Constructor (Main.cpp file)

```
#include <iostream>
#include "Student.h"
using namespace std;
int main()
{
        // instantiate two Students
    Student s1("Zachary", "Moon", 'R', 3.7, "C++");
    Student s2("Gabrielle", "Doone", 'A', 3.7, "C++");
    // These initializations implicitly invoke copy constructor
    Student s3(s1);
    Student s4 = s2;
    s3.SetFirstName("Zack");  // alter each object slightly
    s4.SetFirstName("Gabby");
    // This sequence does not invoke copy constructor
    // This is instead an assignment.
    // Student s5("Giselle", "LeBrun", 'A', 3.1, "C++");
    // Student s6;
    // s6 = s5;  // this is an assignment, not initialization
    s1.Print();   // print each instance
    s3.Print();
    s2.Print();
    s4.Print();
    s1.CleanUp();  // Since some data members are pointers,
    s2.CleanUp();  // lets call CleanUp() to delete() them
    s3.CleanUp();
    s4.CleanUp();
    return 0;
```

Creating **conversion constructors**

Type conversions can be performed from one user defined type to another, or from a standard type to a user defined type. A conversion constructor is a language mechanism that allows such conversions to occur.

NOTE: A **conversion constructor** is a constructor that accepts one explicit argument of a
standard or user defined type, and applies a reasonable conversion or transformation on that object to initialize the object being instantiated.

Let's take a look at an example illustrating this idea. Though the example will be
broken into several segments and also abbreviated

```
#include <iostream>
#include <cstring>
using namespace std;
class Student;
// forward declaration of Student class
class Employee
{
private:
char firstName[20];
char lastName[20];
float salary;
public:
Employee();
Employee(const char *, const char *,
float);
Employee(Student &); // conversion
constructor
void Print();
};
```

Creating **conversion constructors**

Any constructor that takes a single argument is considered a conversion constructor, which can potentially be used to convert the parameter type to the object type of the class to which it belongs. For example, if you have a constructor in the Student class that takes only a float, this constructor could be employed not only in the manner shown in the example above, but also in places where an argument of type Student is expected (such as a function call) when an argument of type float is instead supplied. This may not be what you intend, which is why this interesting feature is being called out. If you don't want implicit conversions to take place, you can disable this behavior by declaring the constructor with the explicit keyword at the beginning of its prototype.

```
int main()
{
Student s1("Giselle", "LeBrun", 'A', 3.5, "C++");
Employee e1(s1); // conversion constructor
e1.Print();
s1.CleanUp(); // CleanUp() will delete() s1's dynamically
return 0;
// allocated data members
}
```

Applying qualifiers to data members and member functions :
Adding const data members and the member initialization list

Data members that should never be modified should be qualified as const . A const data member is one that may only be initialized, and never assigned a new value.

A const member function is a member function that specifies (and enforces) that the method can only perform read-only activities on the object invoking the function.

The member initialization list must be used in a constructor to initialize any data members that are constant, or that are references. A member initialization list offers a mechanism to initialize data members that may never be l-values in an assignment. A member initialization list may also be used to initialize non-const data members. For performance reasons, the member initialization list is most often the preferred way to initialize any data member (const or non-const)

A member initialization list may appear in any constructor, and to indicate this list, simply place a : after the formal parameter list, followed by a comma-separated list of data members, paired with the initial value for each data member in parentheses. For example, here we use the member initialization list to set two data members, studentId and gpa :

Applying qualifiers to data members and member functions :
Adding const data members and the member initialization list

A member initialization list may appear in any constructor, and to indicate this list, simply place a : after the formal parameter list, followed by a comma-separated list of data members, paired with the initial value for each data member in parentheses. For example, here we use the member initialization list to set two data members, studentId and gpa :

```
Student::Student(): studentId(0), gpa(0.0)
{
firstName = lastName = 0; // NULL pointer
middleInitial = '\0';
currentCourse = 0;
}
```

Applying qualifiers to data members and member functions :
Utilizing static data members and static member functions

A static member function is one that encapsulates access to static data members within a class or structure. A static member function does not receive a this pointer; hence, it may only manipulate static data members and other external (global) variables.

```
#include <iostream>
#include <cstring>
using namespace std;
class Student
{
private:
// data members
char *firstName, *lastName, *currentCourse;
char middleInitial;
float gpa;
const char *studentId; // pointer to constant string
static int numStudents; // static data member
public:
// member function prototypes
Student(); // default constructor
Student(const char *, const char *,
char, float, const char *, const char *);
Student(const Student &); // copy constructor
~Student(); // destructor
void Print() const;
const char *GetFirstName() const { return
firstName; }
const char *GetLastName() const { return
lastName; }
char GetMiddleInitial() const { return
middleInitial; }
float GetGpa() const { return gpa; }
const char *GetCurrentCourse() const
{ return currentCourse; }
const char *GetStudentId() const { return
studentId; }
void SetCurrentCourse(const char *);
static int GetNumberStudents();
// static member function
};
```

Applying qualifiers to data members and member functions :
Utilizing static data members and static member functions

```
// definition for static data member
// (which is implemented as an external variable)
int Student::numStudents = 0; // notice initial value of 0
```

```
// Definition for static member function
inline int Student::GetNumberStudents()
{
return numStudents;
}
```

```
Student::Student(): studentId (0) // default constructor
{
firstName = lastName = 0; // NULL pointer
middleInitial = '\0';
gpa = 0.0;
currentCourse = 0;
numStudents++;
// increment static counter
}
```

```
Student::~Student()
// destructor definition
{
delete firstName;
delete lastName;
delete currentCourse;
delete (char *) studentId;
 // cast is necessary for delete
NumStudents--;  // decrement static counter
}
```

Applying qualifiers to data members and member functions :
Utilizing static data members and static member functions

```
int main()
{
   Student s1("Nick", "Cole", 'S', 3.65, "C++", "112HAV");
   Student s2("Alex", "Tost", 'A', 3.78, "C++", "674HOP");

   cout << s1.GetFirstName() << " " << s1.GetLastName();
   cout << " Enrolled in " << s1.GetCurrentCourse() << endl;
   cout << s2.GetFirstName() << " " << s2.GetLastName();
   cout << " Enrolled in " << s2.GetCurrentCourse() << endl;

   // call a static member function in the preferred manner
   cout << "There are " << Student::GetNumberStudents();
   cout << " students" << endl;

   // Though not preferable, we could also use:
   // cout << "There are " << s1.GetNumberStudents();
   // cout << " students" << endl;

   return 0;
}
```

Questions

1. Create a C++ program to encapsulate a Student . Try to do this yourself, rather than relying on any online code. You will need this class as a basis to move forward with future examples; now is a good time to try each feature on your own. Specifically:

a. Create or modify your previous Student class to fully encapsulate a student. Be sure to include several data members that are dynamically allocated. Provide several overloaded constructors to provide a means to initialize your class. Be sure to include a copy constructor. Also, include a destructor to release any dynamically allocated data members.

b. Add an assortment of access functions to your class to provide safe access to data members within your class. Decide for which data members you will offer a GetDataMember() interface, and if any of these data members should have the ability to be reset after construction with a SetDataMember() interface. Apply the const and inline qualifiers to these methods as appropriate.

Questions

c. Be sure to utilize appropriate access regions – private for data members, and possibly for some helper member functions to break up a larger task. Add public member functions as necessary above and beyond your access functions above.

d. Include at least one const data member in your class and utilize the member initialization list to set this member. Add at least one static data member and one static member function.

e. Instantiate a Student using each constructor signature, including the copy constructor. Make several instances dynamically allocated using new() . Be sure to delete() each of these instances when you are done with them (so that their destructor will be called)