CS319: Scientific Computing (with C++)

Static variables; Inheritance; Projects

Week 11: **9am** and **4pm**, 22 March 2017

Annotated slides

1. `static`
   - (i) `static` variables
   - (ii) `static` variable in a class
   - (iii) `static` functions

2. Inheritance
   - `protected`

3. Much of the 4pm class is dedicated to working on the project.

static

If we exit the block, k is deleted.

Today we start with a new C++ keyword: `static`.

Usually, variable exist only in their scope. That means:, for example

```
for (i=0; i<10; i++)
{
  int K=0;
  K++;
}
```

Here,
- k is local to the bloch (so it is unknown outside the block).
- Every iteration k is declared, initialised to 0, then incremented to 1.

we enter & exit the block 10 times — so k is only ever 0 or 1.

But if they are declared to the `static` they can persist.

There are three uses of `static`

1. A `static` variable within a block/function.
2. A `static` member **variable** in a class.
3. A `static` member **function** in a class.

Idea:

```
for (i=0; i<10; i++)
{
  static int K=0;
  K++;
}
```

Here, K is still local to the block.

But it comes "back into existence" each time we enter the block.

Furthermore, it is not reinitialised each time. So now K takes the values 1, 2, 3, 4, ..., 10.

01StaticVarInFunction.cpp

```
10  void Example1(void)
    {
12    static int count=0;
      count++;
14
      std::cout << "Function Example 1 has been called " <<
16      count << " times." << std::endl;
    }
18
    int main(void)
20  {
      Example1();
22    Example1();
      Example1();
24    return(0);
    }
```

1. Try accessing count from main.
2. Try deleting "static" from Line 12.

Every data member of an object is unique to that object (recall: an object is an *instance* of a class).

Suppose we have a class defined as

```
class Student
{
private:
  std::string name;
  std::string university;
public:
  void set_name(std::string n) {name=n;};
  void set_university(std::string u) {university=u;};
  std::string get_name(void) {return(name);};
  std::string get_university(void) {return(university);};
};
```

So, if we declare
Student A, B;
Then A.name ≠ B.name & A.university ≠ B.university
(In particular, &A.university ≠ &B.university)

But there are cases where we would like objects to share a data member. This is achieved using a `static` variable.

Static data members of a class are also known as **class variables**, because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.

This means that

- The static variable is created when the class is defined.
- Therefore, it exists even before a object of that class type is declared.
- It must be initialised outside the class, in the global scope, using the **scope resolution operator**.

An object is an instance of a class. Eg, in the previous slide, A & B are objects, & "student" is a class.

02StaticVarInClass.cpp

```
12  class Student
    {
    private:
14    std::string name;
      static std::string university;
16  public:
      void set_name(std::string n) {name=n;};
18    void set_university(std::string u) {university=u;};
      std::string get_name(void) {return(name);};
20    std::string get_university(void) {return(university);};
    };
22
    // the static member must be initialised in global space
24  std::string Student::university="QCG";
```

copare with slide 5.

we use the class name, along with the scope resolution operator, to access "university".

Note; if it were public, we could access A.university, but that is bad practice.

02StaticVarInClass.cpp

```
Student A, B, C;

A.set_name("Alice Perry"); // Set the values of A.name
std::cout << "Object A: \t\t" << A.get_name() <<
  " is a student at " << A.get_university() << std::endl;

// Set the values of B.name and B.university
B.set_name("Breandan O hEithir"); B.set_university("UCG");
std::cout << "Object B: \t\t" << B.get_name() <<
  " is a student at " << B.get_university() << std::endl;

// And check the values of A again
std::cout << "Now the data for A is : " << A.get_name() <<
  " is a student at " << A.get_university() << std::endl;
```

Output

```
Object A:                Alice Perry is a student at QCG
Object B:                Breandan O hEithir is a student at UCG
Now the data for A is :  Alice Perry is a student at UCG
```

Another, less trivial example of the use of `static` variables, is to associate with the class an integer variable that counts the number of objects that have been created.

This is done (in this example) by having a `static int count` variable in the class. It is initialised to zero and incremented when the constructor is called.

We also have a `static function` called *`getCount()`*. Because this is static, it can be called before an instance of the object is declared.

The following example is adapted from Deitel, *C++ How to Program*, 4th Ed, Example 7.17. (*Refer to lecture notes for explanation*).

03Employee.cpp

```
12  class Employee {
    public:
14    Employee(const std::string, const std::string); // constructor
      ~Employee(void); // destructor
16    const std::string getGivenName() const {return (GivenName);};
      const std::string getFamilyName() const {return (FamilyName);};
18    static int getCount() {return(count);};

20  private:
      std::string GivenName;
22    std::string FamilyName;
      static int count; // number of objects instantiated
24  };

26  // Define and initialize static data member
    int Employee::count = 0;
```

arguments : 1st & 2nd name.

↑ global scope.

03Employee.cpp

```
// The constructor sets values of (string) GivenName
// and (string) FamilyName, and it increments the counter
Employee::Employee(const std::string given, const std::string family){
  GivenName = given;
  FamilyName = family;
  count++;  // increment static count of employees
}

// denstructor decrements the count variable
Employee::~Employee(void){
  count--;  // decrement static count of employees
}
```

03Employee.cpp

```
44    std::cout << "Number of employees before instantiation is "
                << Employee::getCount() << std::endl;   // use class name
46
      Employee Driver("Bob", "Wood");
48    std::cout << "Now the number of employees is " <<
        Employee::getCount() << std::endl;
50
      { // New Block. Tom will be local to this block.
52      Employee Cleaner("Tom", "Broom");

54      std::cout << "Employee 1:" <<  Driver.getGivenName() << " "
                  << Driver.getFamilyName() << std::endl;
56
        std::cout << "Employee 2: " << Cleaner.getGivenName() << " "
58                << Cleaner.getFamilyName() << std::endl;

60      std::cout << "Cleaner.getCount() returns " <<
          Cleaner.getCount() << std::endl;
      }
```

Finally, we mention that it is possible to define a `static` function member of a class.

This means that the function belongs to the class, rather than an instance of the class. So

- It can be called even if no instance of the class is defined;
- It does not have `*this` pointer; ← "this" is the address of an object.
- It can only act on other static members;
- It should be classed as a member of the class, rather than an instance of the class.

Exercise

Read up on `static` functions. Write a function that shows how they are used.

Finished here 10 am.

Inheritance

Finally, we get 'round to one the final **big** idea that we'll study CS319: **Inheritance**.

Basic concept

Given a class, construct from it a new class that has the properties of the original class, plus some new ones.

- The original class is called the **base**, **parent** or **super** class.
- The new one, is called the **derived**, **child** or **sub** class.

The derived class will inherit

- All the data members of the base class, though it may not be able to access them directly,
- The ordinary function members, but not constructors, destructors, or assignment operators.

Inheritance

Some programming/syntax aspects:

- The syntax for defining a derived class

```
class Derived : public Base
```

  Here `public` means that elements that were public to the base class are publicly accessible to the derived class. (This is the most typical case.)

```
class Derived : private Base
```

  Here `private` means everything remains private to the base class, and so can't be access by the derived class. (Not very useful).

- To the `public` and `private` specifiers, we'll add *`protected`*
- `virtual` functions

Inheritance

In our first example, `04InheritExample.cpp`, we construct a simple base class that has a single data member. We'll then make a derived class.

Notice that, although the derived class inherits the base class's private datum, it must use the access method `SetX()` to change it, and `GetX()` to evaluate it.

Inheritance

integer.

```
class Point {
private:
  int x;
public:
  Point(int X=0) {x=X;};
  int GetX(void) {return(x);};
  void SetX(int X) {x=X;};
};

class TwoPoint : public Point {
private:
  int y;
public:
  TwoPoint(int X=0, int Y=0) {SetX(X); y=Y;};
  int GetY(void) {return(y);};
  void SetY(int Y) {y=Y;};
  void PrintXY(void);
};

void TwoPoint :: PrintXY(void){
  std::cout << "(" << GetX() << "," << y << ")";
}
```

Represents a single "point" (x)

all in-line.

a two-dimensional point (x, y).

(inherit int x)

notice set x using SetX, but set y directly.

To date, we designated the **accessibility** every data member and member function of a class with one of the following **access specifiers**:

- **private:** This specifies that the data/function member can only be accessed from within the class. For data members, this means that the data can be accessed or modified by a function that is a member or `friend` of the class. Similarly, private functions can be called by another member/friend function of the class. This is the default.
  OR
- **public:** specifies that a data or function member can be accessed from anywhere in your code.

Now we add: **protected:**. These members act the same as private ones, except that they are directly accessible within a **derived** class.

Compare `05Protected.cpp` with our earlier example.

```
class Point {
protected:  // this was private in 04InheritExample
  int x;
public:
  Point(int X=0) {x=X;};
  int GetX(void) {return(x);};
  void SetX(int X) {x=X;};
};

class TwoPoint : public Point {
private:
  int y;
public:
  // Don't have to use SetX in the constructor
  TwoPoint(int X=0, int Y=0) {x=X; y=Y;};
  int GetY(void) {return(y);};
  void SetY(int Y) {y=Y;};
  void PrintXY(void);
};

void TwoPoint:: PrintXY(void) {
  // Don't have to use GetX
  std::cout << "(" << x << "," << y << ")" << std::endl;
}
```

accessing x & y directly.