Annotated slides

CS319: Scientific Computing (with C++)

Niall Madden (Niall.Madden@NUIGalway.ie)

Week 11: `static`, `const` and inheritance

9am, 27 April, and 4pm, 28 April, 2021

1. Part 1: The STL again
   - Algorithm
   - The password frequency problem
2. Part 2: `static`
   - (i) `static` variables
   - (ii) `static` variable in a class
   - (iii) `static` functions
3. Part 3: `const`
   - `const` functions
   - `const` function parameters
4. Part 4: Inheritance
   - `protected`
   - Replacing members

Tuesday

Wednesday.

Reminder: Assessment for CS319

The assessment for CS319 is based on

1. 50% based on lab assignments:
   - 10% for each of Lab 2 and Lab 3; ← Solutions posted online.
   - 15% for Lab 4+5 and
   - 15% for Lab 7 (due today at 5pm).
2. 50% based on your project work:
   - (i) Initial Project Plan/Discussion [**5 Marks**]
   - (ii) 250 word Project Proposal [**5 Marks**], detailing
     - (a) An external data source, so that you can show your expertise in read from and/or writing to files.
     - (b) A `class` (or set of classes) that you design yourself
     - (c) An algorithm that preforms some type of useful calculation
   - (iii) 3 page project report [**15 Marks**]
   - (iv) Project code [**25 Marks**]

Proposed deadline for report and code is 5pm, Friday ~~13~~ 14. May. THOUGHTS???

CS319 – Week 11: `static`, `const` and inheritance

Start of ...

PART 1: The STL again

The **Standard Template Library** is a collection of storage types, and algorithms to work with them.

Part 1: The STL again

Last week, we introduced the **Standard Template Library** (**STL**). It provides

(1) **Containers:** ways of collecting/storing items of some type (template....). The most important types of containers are:

- `vector`: an indexed sequence. All the items are of the same type. It can be resized, and have new items added to the end.
- `set`: a collection of unique items (of the same type), stored in order. When defined relative to a user-defined class, an overloaded `operator::operator<` (less than) must be provided for correct operation.
- `multiset`: an ordered collection, like a set, but can have repeated values.
- `list`: a doubly linked list.
- `stack`: a stack.

We looked at `set`s, `multiset`s and `vector`s.

(2) **Iterators:** an object used to select (or move between) elements in a container. They are various types:

- forward, reverse, and bidirectional iterators;
- random-access/indexed-access iterators;
- input and output iterators;

Part 1: The STL again

(3) **Algorithms:** for operating on the contents of containers, such as finding a particular item, or sorting (a subset) of them. (**Today!**)

(4) **functors:** (also called "function objects"). Essentially, a class which defines the operator(). (Not covered in CS319).

Today

Algorithm

To use `algorithm`, we must

```
#include <algorithm>
```

Useful functions that this provides include

- `for_each` → iterate over all elements
- `sort` and `partial_sort` — used for vector.
- `search`
- `copy` and `fill` — used to set entries of a container to a particular value.
- `merge`
- `set_union`, `set_difference` — note - can be used to define set intersection.
- etc.

Remember the Password Frequency Problem from Week 7: Given a file with 30,000,000 passwords, construct a list of unique words (i.e., no repetition), listed in order of frequency.

With the STL we can make a simple(r) solution to this.

1. Write a class, `pwd`, that represents a password, and its frequency. Write a function that compares such objects according to their frequency;
2. Construct a `multiset` of `string`s to store the passwords.
3. Construct a `vector` of `pwd`s; copy each password (once), and its frequency, into it.
4. Sort the vector by `pwd` frequency.

Full code is in `01SortPasswordsSTL.cpp`. Here is a summary...

01SortPasswordsSTL.cpp

```
16 #include <set>       // multiset
   #include <vector>    // vector
   #include <algorithm> // sort
```

Class for password.

01SortPasswordsSTL.cpp

```
20 class pwd {
   private:
     std::string word;
22   int freq;
   public:
24   pwd(std::string s, int f) {word=s; freq=f;};
     std::string getword(void) const {return(word);};
26   int getfreq(void) const {return(freq);};
   };

   bool compare (pwd p, pwd q)
30 {
     return (p.getfreq() > q.getfreq());
32 }

34 int FileLength(std::ifstream &InFile, int &LongestWord);
```

constructor.

true of false

note: not a member of class.

Same as Week 7.

01SortPasswordsSTL.cpp

```
36 int main(void )
   {
38   std::ifstream InFile;
     std::string InFileName="UserAccount-1e5.txt";
40   std::multiset <std::string> multi_pwd;
     std::multiset <std::string>::iterator multi_pwd_i;
42   std::vector <pwd> vector_pwd;

44   InFile.open(InFileName.c_str());
     if (InFile.fail() )
46   {
       std::cerr << "Error: Cannot open " << InFileName <<
48       " for reading." << std::endl;
       exit(1);
50   }

52   // Need to know the number of lines, and the length of the longest one
     int LineCount=0, LongestLine;
54   LineCount =  FileLength(InFile, LongestLine);
     std::cout  << InFileName << " has " << LineCount << " lines.\n";
56   std::cout  << "\tThe longest has " << LongestLine << " characters.\n";
```

→ multiset for passwords
→ iterator for looping,
— vector of pwd objects.

Same as Week 7

01SortPasswordsSTL.cpp

```
58  // Read the lines of the password file.
    char *c_string_word;
60  c_string_word = new char [LongestLine+1];
    for (int i=0; i<LineCount; i++)
62  {
      InFile.getline(c_string_word, LongestLine+1);
64    multi_pwd.insert(c_string_word);
    }
```

Each string is inserted into the multiset, and sorted as we go.

01SortPasswordsSTL.cpp

```
   // Copy the passwords to the pwd vector
76 multi_pwd_i = multi_pwd.begin();
   vector_pwd.push_back(
78   pwd(*multi_pwd_i, multi_pwd.count(*multi_pwd_i)));
   multi_pwd_i++;
80 while (multi_pwd_i != multi_pwd.end())
   {
82   if ((vector_pwd.back()).getword() != *multi_pwd_i)
       vector_pwd.push_back(
84       pwd(*multi_pwd_i, multi_pwd.count(*multi_pwd_i)));
     multi_pwd_i++;
86 }
```

point to 1st item in the multiset.

returns the frequency.

Calling the pwd constructor.

01SortPasswordsSTL.cpp

```
88    std::sort (vector_pwd.begin(), vector_pwd.end(), compare);

90    std::cout << "Top 10 passwords are: " << std::endl;
      for (unsigned int i=0; i<10; i++)
92      std::cout <<  std::setw(12) << (vector_pwd[i]).getword() <<
          std::setw(6) << (vector_pwd[i]).getfreq() << std::endl;

      InFile.close();
96    return(0);
```

CS319 – Week 11: `static`, `const` and inheritance

END OF PART 1

CS319 – Week 11: static, const and inheritance

Start of ...

PART 2: static

Basically, a variable that is shared by every instance of the class.

or function

Part 2: `static`

Next we introduce a new C++ keyword: `static`.

Usually, variables exist only in their scope. That means that **it exists only within the program block in which it is defined**.

For example, if a variable is *local* to a function every time that function is called, a new instance of the variable is created, and every time the function ends, the variable (and its value) are lost.

However, if a variable is declared to the `static` then it 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.

A `static` **variable** belongs to a function. It can be accessed only within that function, but it retains its value between calls.

02staticVarInFunction.cpp

```
10  void Example1(void)
    {
      static int count=0;
12    count++;

14    std::cout << "Function Example 1 has been called " <<
        count << " times." << std::endl;
16  }

18  int main(void)
    {
20    Example1();
      Example1();
22    Example1();
      return(0);
24  }
```

This is a trivial example. A much better use would be to count the number of function evaluations in out optimization problem from Lab 3.

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

```
10  class Student
    {
12  private:
      std::string name;
14    std::string university;
    public:
16    void set_name(std::string n) {name=n;};
      void set_university(std::string u) {university=u;};
18    std::string get_name(void) {return(name);};
      std::string get_university(void) {return(university);};
20  };
```

But if all `Students` are enroled in the same University, a name change of the institution for one student should change them for all.

So 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**.

ie, not in a constructor

03staticVarInClass.cpp

```
10  class Student
    {
12  private:
      std::string name;
14    static std::string university;
    public:
16    void set_name(std::string n) {name=n;};
      void set_university(std::string u) {university=u;};
18    std::string get_name(void) {return(name);};
      std::string get_university(void) {return(university);};
20  };

22  // the static member must be initialised in global space
    std::string Student::university="QCG";
```

This is shared by all objects of this class.

not in main()

"Queens College Galway".

So A.university has changed from Line 31 to line 40. (because of line 34).

03staticVarInClass.cpp

```
  Student A, B, C;

  A.set_name("Alice Perry"); // Set the values of A.name
30  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
34  B.set_name("Breandan O hEithir"); B.set_university("UCG");
    std::cout << "Object B: \t\t" << B.get_name() <<
36    " is a student at "  << B.get_university() <<  std::endl;

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

Output

```
1 Object A:                 Alice Perry is a student at QCG
  Object B:                 Breandan O hEithir is a student at UCG
3 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.

04Employee.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;
```

04Employee.cpp

```
30 // The constructor sets values of (string) GivenName
   // and (string) FamilyName, and it increments the counter
   Employee::Employee(const std::string given,
32                    const std::string family){
     GivenName = given;
34   FamilyName = family;
     count++; // increment static count of employees
36 }

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

So, someone leaves.

04Employee.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;
- The `*this` pointer is not defined within the function;
- It can only act on other static members;
- It should be considered to be 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.

CS319 – Week 11: static, const and inheritance

END OF PART 2

Finished here Tuesday.

[Back at 10 for Lab]

CS319 – Week 11: static, const and inheritance

Start of ...

PART 3: const

Modifier that ensures a "variable" does not vary

A expression is constant it is value never changes. We are familiar with **literal consants**. Examples:

- `'x'`
- `"Hello"`
- `(int) 5`
- `(float) 3.14159`

(assuming, of course, the base is unchanged)

C++ also has three **keyword literals**: `true`, `false` and `nullptr`.

(1 *NULL.

Part 3: const

We can also define our own constants, by putting the qualifier `const` before a "variable" declaration:

05const.cpp

```
   const double Pi=3.14159;
16 std::cout << "Pi=" << Pi << std::endl;

18 // Now change Pi
19 Pi = 3.14; // This will not (and should not) work

   std::cout << "Now Pi=" << Pi << std::endl;
```

Now, if we compile:

```
01const.cpp: In function 'int main()':
01const.cpp:19:8: error: assignment of read-only variable 'Pi'
   Pi = 3.14;
      ^
```

cannot be changed. ("write once").

We can add the `const` "type qualifier" (modifier) to any data type, including a `class` that we define ourselves.

However, that creates issues.

Specifically, if we write our own class, and then create a `const` object of this class, it can only call `const` methods.

Consider the following code, adpated from `Week11/04StaticVarInClass.cpp`

Bad example: Part 1

```
12 class Student
   {
14 private:
     std::string name;
16   static std::string university;
   public:
18   Student(std::string n) {name=n;};
     void set_name(std::string n) {name=n;};
20   void set_university(std::string u) {university=u;};
     std::string get_name(void) {return(name);};
22   std::string get_university(void) {return(university);};
   };
```

Bad example: Part 2

```
34   const Student B("Agnes Perry");
     std::cout << "Object B: \t" << B.get_name() <<
36     " is a student at " << B.get_university() <<  std::endl;
```

```
tmp.cpp:35:45: error: passing ‘const Student’ as ‘this’ argument discards qualifiers
   std::cout << "Object B: \t" << B.get_name() <<
```

06constFunction.cpp

```
12 class Student
   {
14 private:
     std::string name;
16   static std::string university;
   public:
18   Student(std::string n) {name=n;};
     void set_name(std::string n) {name=n;};
20   void set_university(std::string u) {university=u;};
     std::string get_name(void) const {return(name);};
22   std::string get_university(void) const {return(university);};
   };
```

const function parameters

When writing a function header, we can designate some of the parameters as `const` meaning that the **copy** of the parameter that is passed cannot be changed in the function.

There are some times when this is necessary. Consider the following example...

07constParam.cpp

```
12 class Number
   {
   private:
14   double x;
   public:
16   Number() {x=0.0;};
     Number(const Number &copy) {x=copy.x;}; // notice const
18   void set_x(double X) {x=X;};
     double get_x(void) {return(x);};
20   Number operator+(Number &b)
       {
22       Number sum;
         sum.x = x + b.x;
24       return(sum);
       }
26 };
```

copy constructor

07constParam.cpp

```
30 int main(void)
   {
     Number a;
32   a.set_x(121.234);
     Number b(a+a);
34   std::cout << "a.x= \t" << a.get_x() <<  std::endl;
     std::cout << "b.x= \t" << b.get_x() <<  std::endl;
36   return(0);
   }
```

calling copy constructor.

Why this works...

Why would this not work with out const in the copy constructor ?

Ans: with const "a+a" is designed as being constant.

CS319 – Week 11: `static`, `const` and inheritance

END OF PART 3

CS319 – Week 11: `static`, `const` and inheritance

Start of ...

PART 4: Inheritance

Making new classes from old ones

We'll finish with one final **big** idea that we'll study CS319: **Inheritance**.

Basic concept in Inheritance

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.

methods.

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

Part 4: Inheritance

In our first example, `08InheritExample.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.

08InheritExample.cpp

```
12 // The base class
   class Point1D {
14 private:
     int x;
16 public:
     Point1D(int X=0) {x=X;};
18   int GetX(void) {return(x);};
     void SetX(int X) {x=X;};
20 };

22 // The derived class will inherit x, GetX, SetX
   class Point2D : public Point1D {
24 private:
     int y;
26 public:
     Point2D(int X=0, int Y=0) {SetX(X); y=Y;};
28   int GetY(void) {return(y);};
     void SetY(int Y) {y=Y;};
30   void PrintXY(void);
   };

   void Point2D :: PrintXY(void){
34   std::cout << "(" << GetX() << "," << y << ")" << std::endl;
   }
```

(for a one-dimensional point).

constructor.

will be inherited by Point2D

(for a two-dim point).

member y is accessed directly.

x not accessed 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 `09Protected.cpp` with our earlier example.

09Protected.cpp

```
14 class Point1D {
   protected:  // this was private in 08InheritExample.cpp
16   int x;
   public:
18   Point1D(int X=0) {x=X;};
     int GetX(void) {return(x);};
20   void SetX(int X) {x=X;};
   };

   // The derivated class will inherit x, GetX, SetX
24 class Point2D : public Point1D {
   private:
26   int y;
   public:
28   // Don't have to use SetX in the constructor
     Point2D(int X=0, int Y=0) {x=X; y=Y;};
30   int GetY(void) {return(y);};
     void SetY(int Y) {y=Y;};
32   void PrintXY(void);
   };

   void Point2D:: PrintXY(void)
36 {
     // Don't have to use GetX
38   std::cout << "(" << x << "," << y << ")" << std::endl;
   }
```

Same as detor e . .

Now x is accessed directly.

Suppose, for example, that a base class has a method `void PrintData(void)`. It is possible to make a derived class that also contains a (new) method call `void PrintData(void)`.

The version of `PrintData(void)` from the derived class will superceed the one from the base class.

However, the original one does not disappear, it can still be called, but you need to use the base class name and scope resolution operator.

Note: If the base and derived classes have functions with the same names, but different **"signatures"**, then the function is overloaded, not replaced.

10Replacement.cpp

```
36 class Point2D : public Point1D {
   private:
38   int y;
   public:
40   Point2D(int X=0, int Y=0) {x=X; y=Y;};
     int GetY(void) {return(y);};
42   void SetY(int Y) {y=Y;};
     void PrintData(void);
44 };

46 // The Point2D class also has a method
   // called PrintData()
48 void Point2D::PrintData(void)
   {
50   std::cout << "(" << x << "," << y << ")\n";
   }
```

instead of PrintXY

In this version, Point1D also has a method called PrintData

10Replacement.cpp

```
int main(void)
{
  Point2D d;

  d.SetX(1); d.SetY(2);
  std::cout << "Calling the Point2D version of d.PrintData(): d=";
  d.PrintData();

  std::cout << "Calling the Point1D version of d.PrintData(): d=";
  d.Point1D::PrintData();
  return(0);
}
```

There are other times when we need to take care which of two functions that have the same *signature* and belonging the base and derived class is called, and need to declare one of them as `virtual`, but that is beyond the scope of this class...

function name + org list.