Object-Oriented Programming in C++

Pre-Lecture 9: Advanced topics

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

March 24, 2015

Prelecture 9

Outline

This prelecture largely covers some advanced C++ topics on program structure

- Static data
- Function and class templates
- Namespaces
- Header and multiple source files

Static class members

Static data

- Recall: an object is an instance of a class
  - Each object has unique set of values for data members
  - Object may not exist for lifetime of program (e.g. object destroyed when exiting function - goes out of scope)
- Sometimes we may want all objects from a given class to share access to (and be able to modify) some data (“global data”)
- Need to create static data members - memory is reserved for lifetime of program and can be accessed by all objects
- Here is how to implement one...

Static data

```cpp
#include<iostream>
using namespace std;
// Example using static data members
class myclass
{
private:
  int x;
  static int nobjects;
public:
  myclass(int xin) : x(xin) {nobjects++;}
  ~myclass() {nobjects--;}
  void show() {cout<<"x="<<x<<", nobjects="<<nobjects<<endl;}
};
int myclass::nobjects(0); // define static data member
void test()
{
  myclass a3(3);
  a3.show();
}
int main()
{
  myclass a1(1);
  a1.show();
  myclass a2(2);
  a2.show();
  test();
  a1.show();
  return 0;
}
```

Listing 1 : PL9/staticdata.cpp

Static data

- We first declare[1] a static data member within our class

```
  static int nobjects;
```

- We then define and initialize it after our class (this is where memory is set aside)

```
int myclass::nobjects(0); // define static data member
```

- Every object instantiated from our class can see the same `nobjects` and modify it
- In our example, we used it to contain the current number of objects (changed in constructor and destructor)
- Program outputs

```
x=1, nobjects=1
x=2, nobjects=2
x=3, nobjects=3
x=1, nobjects=2
```

[1] You declare what something is, you define what something does

Templates: Functions

Templates: functions

- Templates allow functions and classes to be created for generic datatypes
- Consider functions first - example (remember lecture 2)

```
double maxval(double a, double b)  {return (a>b) ? a : b;}
int maxval(int a, int b)  {return (a>b) ? a : b;}
```

- Used overloading to re-write function for integer parameters
- Second function performs identical task to first (maximum of two numbers) but with different data type
- Used ternary operator (`test ? iftrue : iffalse` - good for true-or-false tests returning a value

Templates: functions

- Overloading is good but laborious (a function for every type)
- Solution: write single function template

```
#include<iostream>
using namespace std;
template <class T> T maxval(T a, T b)
{
  return (a > b) ? a : b;
}
int main()
{
  double x1(1),x2(1.5);
  cout<<"Maximum value (doubles) = "<<maxval<double>(x1,x2)<<endl;
  int i1(1),i2(-1);
  cout<<"Maximum value (ints) = "<<maxval<int>(i1,i2)<<endl;
  return 0;
}
```

Listing 2 : PL9/functiontemplate.cpp

- Output

```
Maximum value (doubles) = 1.5
Maximum value (ints) = 1
```

Templates: functions

- The function template started with

```
template <class T> T maxval(T a, T b)
```

  before defining the function itself
- The statement `<class T>` tells the compiler the template is for a generic type `T` - known as a template parameter
- The remainder is like any function except a specific datatype is replaced with `T`
- NB: the compiler will not use the function template until an instance is created (known as a template function)
- We did this twice in the program itself, e.g.

```
cout<<"Maximum value (doubles) = "<<maxval<double>(x1,x2)<<endl;
```

  which requires a template function to be created that replaces `T` with `double`

Templates: Classes

Templates: classes

- Can also write a class template
- Example class for a pair of integers

```
#include<iostream>
using namespace std;
class twonum
{
private:
  int x,y;
public:
  twonum() : x(0),y(0) {}
  twonum(int xx, int yy) : x(xx),y(yy) {}
  int add() {return x+y;}
  int sub() {return x-y;}
};

int main()
{
  int x(1),y(2);

  twonum ip(x,y);
  cout<<"x+y="<<ip.add()<<endl;
  cout<<"x-y="<<ip.sub()<<endl;

  return 0;
}
```

Listing 3 : PL9/twonum.cpp

- Might want another version for doubles...

Templates: classes

So change code as follows:

```
#include<iostream>
using namespace std;
// Class template
template <class T> class twonum {
private:
  T x,y;
public:
  twonum() : x(0),y(0) {}
  twonum(T xx, T yy) : x(xx),y(yy) {}
  T add() {return x+y;}
  T sub() {return x-y;}
};
int main()
{
  int x(1),y(2);
  double a(-1.5),b(-2.5);
  // Use class template for object representing pair of integers
  twonum<int> ip(x,y);
  cout<<"x+y="<<ip.add()<<endl;
  cout<<"x-y="<<ip.sub()<<endl;
  // Now for  a pair of doubles
  twonum<double> dp(a,b);
  cout<<"a+b="<<dp.add()<<endl;
  cout<<"a-b="<<dp.sub()<<endl;
  return 0;
}
```

Listing 4 : PL9/twonum2.cpp

Templates: classes

- Modified declaration of class as class template with template parameter

```
template <class T> class twonum {
```

- Then replace appropriate data type in class with `T`, e.g. for parameterised constructor

```
  T add() {return x+y;}
```

- Instances of the class are created as

```
  twonum<int> ip(x,y);
```

- Then for an object of double type, we write

```
  twonum<double> dp(a,b);
```

- Again, compiler uses class template to create two instances (or template classes), one for each type, as required
- Seen this already: `vector<double>` (`vector` is a class template and `vector<double>` creates a template class for vector of doubles)

Templates: classes

- If a member function contains parameter that is an instance of a template class (i.e. object), must refer to its type as `twonum<T>`
- Compiler will then replace `T` with `int`, `double`, etc. as appropriate when creating template class
- Example: write a simple copy constructor
  `twonum(const twonum<T> &tn) x=tn.x; y=tn.y;`
- For member functions defined outside class, we prototype inside class as before, e.g.
  `twonum(const twonum<T> &tn); // prototype`
- Then we define the function itself as follows

```
template <class T> twonum<T>::twonum(const twonum<T> &tn)
         {x=tn.x; y=tn.y;}
```

- Must also modify class name (before `::`) to `twonum<T>` as referring to template class

Namespaces

Namespaces

- Imagine if we tried to include two classes with same name:

```
1  #include<iostream>
2  class myclass {
3  private:
4    int x;
5  public:
6    myclass() : x(0) {}
7    myclass(int xx) : x(xx) {}
8    ~myclass(){}
9    void show(){std::cout<<"x="<<x<<std::endl;}
10 };
11 class myclass {
12 private:
13   int x,y;
14 public:
15   myclass() : x(0),y(0) {}
16   myclass(int xx, int yy) : x(xx),y(yy) {}
17   ~myclass(){}
18   void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
19 };
20 int main() {  return 0; }
```

Listing 5 : PL9/namespacewrong.cpp

- Will result in compilation error: have a (class) name collision
- Same applies to variables and functions with same name and parameter list (overloading not possible)
- But might be unavoidable in large programs – e.g. when including multiple external libraries

Namespaces

C++ has a solution: namespaces

```
3  namespace myns1
4  {
5    const double ab=1.5;
6    class myclass
7    {
8    private:
9      int x;
10   public:
11     myclass() : x(0) {} // shorter method!
12     myclass(int xx) : x(xx) {}
13     ~myclass(){}
14     void show(){std::cout<<"x="<<x<<std::endl;}
15   };
16 }
17 namespace myns2
18 {
19   const double ab=2.5;
20   class myclass
21   {
22   private:
23     int x,y;
24   public:
25     myclass() : x(0),y(0) {} // shorter method!
26     myclass(int xx, int yy) : x(xx),y(yy) {}
27     ~myclass(){}
28     void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
29   };
30 }
```

Listing 6 : selection of PL9/namespaceright.cpp

Namespaces

- Namespaces are like boxes: allow us to keep class definitions distinct and we choose which ones to use
- We can implement namespaces in two ways
- First is direct reference to namespace using scope resolution operator, `::`

```
31 int main()
32 {
33   myns1::myclass c1(1); // utilizes myclass from myns1
34   c1.show();
35   myns2::myclass c2(1,2); // now different myclass from myns2
36   c2.show();
37   return 0;
38 }
```

Listing 7 : selection of PL9/namespaceright.cpp

Namespaces

- Second method appropriate when choosing to use one namespace in particular

```
31 int main()
32 {
33   using namespace myns1
34   myclass c1(1);
35   c1.show();
36   return 0;
37 }
```

Listing 8 : selection of PL9/namespaceright2.cpp

- Can then refer to `myclass` (from `myns1`) directly as 2nd `myclass` within `myns2` is not used
- Note: already very familiar with one particular namespace

```
2 using namespace std;
```

- This namespace contains all standard library definitions (e.g. for `cout`)
- Although we used first method above when using

```
28     void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
```

Listing 9 : selection of PL9/namespaceright2.cpp

Headers and multiple files

Headers and multiple source files

- When our code grows large, might want to spread across files
- First thing to consider is where to put constants, class definitions and function declarations
- Normal place is in a header file
- We include the contents of header files as follows

```
#include<iostream>  // system include file (C++ standard library)
#include<cmath> // another one (from C library)
#include "myheader.h" //  our include file
```

- Note differences between system header files and our own
- We can then include this header file in every `.cpp` file that makes up our program
- Header files are for class definitions and function declarations: where should we put function definitions?

Headers and multiple source files

- Function definitions (what functions actually do) usually go in a `.cpp` file, especially when substantial.
- We can create a second `.cpp` file to hold these.
- Example: put the function definition for `show()` in a separate file (`myclass.cpp`)
- We now have 3 files: `myclass.h`, `myclass.cpp` and `myproject.cpp`
- In practic we name files as appropriate; usually same name for header and implementation (.h or .cpp extension)
- Keep all these files in projects folder

Headers and multiple source files

one definition rule

- Important: definitions can be made only once.
- Functions in `.cpp` file OK - included only once.
- Headers (containing class definitions) may be included more than once (e.g., include in multiple other headers)- need a header guard to prevent multiple definition.
- We can use pre-processor directives to ensure this.
- See the header file `myclass.h` for an example,

```
#ifndef MY_CLASS_H // Will only be true the once!
#define MY_CLASS_H

namespace myns1
{
  class myclass
  {
  private:
    int x;
  public:
    myclass() : x(0) {}
    myclass(int xx) : x(xx) {}
    ~myclass(){}
    // Prototypes
    void show();
  };
}
#endif
```

Listing 10 : PL9/myclass.h

Headers and multiple source files

```
#include<iostream>
#include "myclass.h"
using namespace myns1;
using namespace std;
void myclass::show()
{
  cout<<"x="<<x<<endl;
}
```

Listing 11 : PL9/myclass.cpp

```
// Example of including a header file
#include<iostream>
#include "myclass.h"
using namespace myns1;
using namespace std;
int main()
{
  myclass c1(1);
  c1.show();
  return 0;
}
```

Listing 12 : PL9/myproject.cpp

Headers/Source for templates

Headers and multiple source files

templates Health Warning

- Using the method for splitting code in multiple files discussed above can cause linker errors when using templates.
- Template classes and functions are generated on demand.
- There is a consequence: compiler needs to see **both** declarations and definitions in the same file as the code that uses the templates.
- The default rule above was that there are no function definitions inside a header file. You are allowed to break this for templates.
- Solution - below namespace (containing the class definition) in header file:
  - Add `using namespace myns` (or equivalent);
  - Then add all template function definitions;
  - Include this header file in any `.cpp` file where objects are instantiated from this class template.

move vs copy: assignment

```cpp
#include<iostream>
#include"twonum3.h"
using namespace std;
using namespace two_num;
int main()
{
  int x(1),y(2);
  double a(-1.5),b(-2.5);
  // Use class template for object ↩
    representing pair of integers
  twonum<int> ip(x,y);
  cout<<"x+y="<<ip.add()<<endl;
  cout<<"x-y="<<ip.sub()<<endl;
  // Now for  a pair of doubles
  twonum<double> dp(a,b);
  cout<<"a+b="<<dp.add()<<endl;
  cout<<"a-b="<<dp.sub()<<endl;
  return 0;
}
```

Listing 13 : PL9/twonum3.cpp

```cpp
#ifndef TWO_NUM_H // Will only be true↩
     the once!
#define TWO_NUM_H
// Class template
namespace two_num
{
  template <class T> class twonum {
  private:
    T x,y;
  public:
    twonum() : x(0),y(0) {};
    twonum(T xx, T yy) : x(xx),y(yy) ↩
    {};
    T add();
    T sub();
  };
}
using namespace two_num;
template<class T> T twonum<T>::add() {↩
    return x+y;};
template<class T> T twonum<T>::sub() {↩
```

Headers and multiple source files

- You need to be specific about relationship between a template class and friends (as template functions).
- This is particularly important for the inserion operator <<.
- Here's how to do it:

Headers and multiple source files

templates Advanced

- Before class declaration, add following lines:

```
// Forward declaration of class
template <class Ttype> class myclass;
// So that we can declare friend function as a template function
template <class Ttype>
 std::ostream & operator<<(std::ostream &os, const myclass<Ttype> &↩
  myobject);
```

- Then in body of class, declare friend as follows:

```
friend std::ostream & operator<< <Ttype> (std::ostream &os, const myclass↩
   <Ttype> &myobject);
```

- Finally, define operator<< (refers to class' namespace)

```
// Function to overload << operator
 template <class Ttype>
 std::ostream & myns::operator<<(std::ostream &os, const myclass<Ttype> &↩
    myobject)
{ ....
  return os;
}
```

Summary

Prelecture 9

Outline

We covered

- Static class members
- Function and class templates
- Namespaces
- Header and multiple source files
  - and use for templates