Object-Oriented Programming in C++

Pre-Lecture 9: Advanced topics

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

Room 7.07, Schuster Building

Version: March 15, 2020

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 the lifetime of the 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>
class my_class
{
private:
  int x{};
  static int n_objects;
public:
  my_class() : x{} {n_objects++;}
  my_class(int x_in) : x{x_in} {n_objects++;}
  ~my_class() {n_objects--;}
  void show() {std::cout<<"x="<<x<<", n_objects="<<n_objects<<std::endl;}
};
int my_class::n_objects{}; // define static data member outside class!
void test()
{
  my_class a3{3};
  a3.show();
}
int main()
{
  my_class a1{1};
  a1.show();
  my_class a2{2};
  a2.show();
  test();
  a1.show();
  return 0;
}
```

Listing 1 : selection of PL9/staticdata.cpp

Static data

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

```
static int n_objects;
```

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

```
int my_class::n_objects{}; // define static data member ↩
    outside class!
```

- 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, n_objects=1
x=2, n_objects=2
x=3, n_objects=3
x=1, n_objects=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 an lvalue

Templates: functions

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

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

Listing 2 : selection of PL9/functiontemplate.cpp

- Output

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

Templates: functions

- The function template started with

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

  before defining the function itself
- The statement `<class c_type>` tells the compiler the template is for a generic type `c_type` - known as a template parameter
- The remainder is like any function except a specific datatype is replaced with `c_type`
- 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.

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

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

Templates: Classes

Templates: classes

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

```
#include<iostream>
class pair_of_numbers
{
private:
  int x;
  int y;
public:
  pair_of_numbers() : x{},y{} {}
  pair_of_numbers(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};
  pair_of_numbers ip{x,y};
  std::cout<<"x+y="<<ip.add()<<std::endl;
  std::cout<<"x-y="<<ip.sub()<<std::endl;
  return 0;
}
```

Listing 3 : selection of PL9/twonum.cpp

- Might want another version for doubles...

Templates: classes

So change code as follows:

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

Listing 4 : selection of PL9/twonum2.cpp

Templates: classes

- Modified declaration of class as class template with template parameter

```
template <class c_type> class pair_of_numbers {
```

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

```
  pair_of_numbers(c_type xx, c_type yy) : x{xx},y{yy} {}
```

- Instances of the class are created as

```
  pair_of_numbers<int> ip{x,y};
```

- Then for an object of double type, we write

```
  pair_of_numbers<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<c_type>`
- Compiler will then replace `c_type` with `int`, `double`, etc. as appropriate when creating template class
- Example: write a simple copy constructor
  `twonum(const twonum<c_type> &tn) x=tn.x; y=tn.y;`
- For member functions defined outside class, we prototype inside class as before, e.g.
  `twonum(const twonum<c_type> &tn); // prototype`
- Then we define the function itself as follows

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

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

Namespaces

Namespaces

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

```cpp
#include<iostream>
class my_class
{
private:
  int x;
public:
  my_class() : x{} {}
  my_class(int xx) : x{xx} {}
  ~my_class(){}
  void show(){std::cout<<"x="<<x<<std::endl;}
};
class my_class
{
private:
  int x,y;
public:
  my_class() : x{},y{} {}
  my_class(int xx, int yy) : x{xx},y{yy} {}
  ~my_class(){}
  void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
};
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

Namespaces

C++ has a solution: namespaces

```cpp
#include<iostream>
namespace my_ns1 {
  const double ab{1.5};
  class my_class
  {
  private:
    int x;
  public:
    my_class() : x{} {}
    my_class(int xx) : x{xx} {}
    ~my_class(){}
    void show(){std::cout<<"x="<<x<<std::endl;}
  };
}
namespace my_ns2
{
  const double ab{2.5};
  class my_class
  {
  private:
    int x,y;
  public:
    my_class() : x{},y{} {} // shorter method!
    my_class(int xx, int yy) : x{xx},y{yy} {}
    ~my_class(){}
    void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
  };
```

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, `::`

```
int main()
{
  my_ns1::my_class c1{1}; // utilizes my_class from myns1
  c1.show();
  my_ns2::my_class c2{1,2}; // now different my_class from myns2
  c2.show();
  return 0;
}
```

Listing 7 : selection of PL9/namespaceright.cpp

Namespaces

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

```
int main()
{
  using namespace my_ns1;
  my_class c1{1};
  c1.show();
  return 0;
}
```

Listing 8 : selection of PL9/namespaceright2.cpp

- Can then refer to `myclass` (from `myns1`) directly as 2nd `myclass` within `myns2` is not used
- Note: we are already very familiar with one particular namespace `std`
- This namespace contains all standard library definitions (e.g. for `cout`)
- Although we used first method above when using

```
  using namespace my_ns1;
```

Listing 9 : selection of PL9/namespaceright2.cpp

Headers and multiple files

Headers and multiple source files

- When our code grows large, we must divide code across files for readability
- 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`
- We name files as appropriate; the house style requires the 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)- we 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,

```
// PL9/myclass.h
// header file for class definition; also defined namespace
// Niels Walet, Last modified 06/12/2019
#ifndef MY_CLASS_H // Will only be true the once!
#define MY_CLASS_H
namespace my_ns1 {
  class my_class
  {
  private:
    int x;
  public:
    my_class() : x{} {}
    my_class(int xx) : x{xx} {}
    ~my_class(){}
    void show();
  };
}
#endif
```

Listing 10 : PL9/myclass.h

Headers and multiple source files

```
// PL9/myclass.cpp
// implementation file for class definition
// Niels Walet, Last modified 03/12/2019
#include<iostream>
#include "myclass.h"
using namespace my_ns1;
void my_class::show()
{
  std::cout<<"x="<<x<<std::endl;
}
```

Listing 11 : PL9/myclass.cpp

```
// PL9/myproject.cpp
// Using class with namespace defined in header
// Niels Walet, Last modified 03/12/2019
#include<iostream>
#include "myclass.h"
int main()
{
  my_ns1::my_class 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 expected 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
// PL9/twonum3.cpp
// Define a class template to hold a pair of numbers (header file)
// Niels Walet, Last modified 03/12/2019
#include<iostream>
#include"twonum3.h"
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
  pair_of_numbers<int> ip(x,y);
  std::cout<<"x+y="<<ip.add()<<std::endl;
  std::cout<<"x-y="<<ip.sub()<<std::endl;
  // Now for  a pair of doubles
  pair_of_numbers<double> dp(a,b);
  std::cout<<"a+b="<<dp.add()<<std::endl;
  std::cout<<"a-b="<<dp.sub()<<std::endl;
  return 0;
}
```

Listing 13 : PL9/twonum3.cpp

```cpp
// PL9/twonum3.h
// Header file to define a class template to hold a pair of numbers
// Niels Walet, Last modified 03/12/2019
#ifndef TWO_NUM_H // Will only be true the once!
#define TWO_NUM_H
namespace two_num
{
  template <class c_type> class pair_of_numbers {
  private:
    c_type x;
    c_type y;
  public:
  pair_of_numbers() : x{},y{} {};
  pair_of_numbers(const c_type xx, const c_type yy) : x{xx},y{yy} {};
    c_type add();
    c_type sub();
  };
}
using namespace two_num;
template<class c_type> c_type pair_of_numbers<c_type>::add() {return x+y;};
template<class c_type> c_type pair_of_numbers<c_type>::sub() {return x-y;};
#endif
```

Listing 14 : PL9/twonum3.h

Headers and multiple source files

templates Advanced

- 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 the class declaration, add the following lines:

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

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

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

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

```
// Function to overload << operator
 template <class c_type>
 std::ostream & myns::operator<<(std::ostream &os, const myclass<c_type↩
    > &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