600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Member / method modifiers
  - const
  - static
- enums

const modifier

- If we declare a method as **const**, we are declaring that the object should not be modified within the method
  - Member values cannot be modified

main.cpp

```
#include <iostream>
class Foo
{
	int _bar;
public:
	void bar( void ) const { _bar++; }
};
int main( void )
{
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In member function âvoid Foo::bar() constâ:
main.cpp:7:32: error: increment of member Foo::_bar in read-only object
   void bar( void ) const { _bar++; }
                                ^~
>>
```

const modifier

- If we declare a method as const, we are declaring that the object should not be modified within the method
  - Member values cannot be modified
  - Non-const methods cannot be called (even if they don't change state)

main.cpp

```
#include <iostream>
class Foo
{
public:
	void bar1( void ) const { bar2(); }
	void bar2( void ) { ; }
};
int main( void )
{
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In member function void Foo::bar1() const:
main.cpp:7:34: error: passing const Foo as this argument discards qualifiers [-fpermissive]
   void bar1( void ) const { bar2(); }
                                  ^
foo.cpp:8:8: note:   in call to âvoid Foo::bar2()â
   void bar2( void ) { ; }
        ^~~~
>>
```

`const` modifier

- If we declare a method as `const`, we are declaring that the object should not be modified within the method
  - Member values cannot be modified
  - Non-`const` methods cannot be called
  - A reference / pointer to the object (or its members) will not be passed as a non-`const` argument to some other function / method

main.cpp

```
#include <iostream>
using namespace std;
void Print( int& i ){ cout << i << endl; }
class Foo
{
	int _bar;
public:
	void print( void ) const { Print( _bar ); }
};
int main( void )
{
	return 0;
}
```

const modifier

- If we declare a method as const, we are declaring that the object should not be modified within the method
  - Member values cannot be modified
  - Non-const methods cannot be called
  - A reference / pointer to the object

main.cpp

```
#include <iostream>
using namespace std;
void Print( int& i ){ cout << i << endl; }
class Foo
{
	int _bar;
public:
	void print( void ) const { Print( _bar ); }
};
int main( void )
{
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In member function âvoid Foo::bar() constâ:
main.cpp:8:33: error: binding âconst intâ to reference of type âint&â discards qualifiers
   void bar( void ) const { Bar( _bar ); }
                                 ^~~~
foo.cpp:2:6: note:   initializing argument 1 of âvoid Bar(int&)â
 void Bar( int& i ){ }
      ^~~
>>
```

const modifier

- Same rules hold for arguments of a function that are declared const
  - Member values cannot be modified
  - Non-const methods cannot be called
  - A reference / pointer to th
    (or its members) will not b
    as a non-const argument
    other function / method

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	int _bar;
public:
	void bar( void ){ _bar++; }
};
void Bar( const Foo& f ){ foo.bar(); }
int main( void )
{
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function void Bar(const Foo&) :
main.cpp:9:27: error: foo was not declared in this scope
 void Bar( const Foo& f ){ foo.bar(); }
                           ^~~
>>
```

const modifier

- The **const** keyword is part of the signature of the method and can be used to overload it

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
public:
	void bar1( void ) const { bar2(); }
	void bar2( void ) { cout << "hi" << endl; }
	void bar2( void ) const { cout << "bye" << endl; }
};
int main( void )
{
	Foo foo;
	foo.bar1();
	return 0;
}
```

```
>> ./a.out
bye
>>
```

const modifier

IntVector.h

```
class IntVector
{
        int* _values;
public:
        …
        int& operator[] ( int idx ){ return _values[idx]; }
        int operator[] ( int idx ) const { return _values[idx]; }
        …
};
```

- The const keyword is part of the signature of the method and can be used to overload it
  - For contexts where we are allowed to modify the object, we have methods returning references to the members
  - For contexts where we are not, we can have members returning copies (or const references) of the member values

const modifier

- The **const** keyword is part of the signature of the method and can be used to overload it
  - If a **const** method is defined outside the class, the **const** qualifier is required for the definition as well as the declaration

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	int _bar;
public:
	void print( void ) const;
};
void Foo::print( void ) const { cout << _bar << endl; }
int main( void )
{
	return 0;
}
```

`static` modifier

- In C, we use the keyword `static` to indicate that a variable shouldn't be destroyed at the end of a block of code
  - Instead of living on the heap, static variables live in the data segment

- In C++ classes, when we want all instances of a class of objects to share a variable, we declare it `static`
  - Contrast this with an instance variable, a new copy of which is created for every instance of the class

static modifier

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1 , foo2;
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

static modifier

- Use static to declare a class member or method

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1(-24) , foo2(17);
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

static modifier

- Use static to declare a class member or method
- Members defined outside the class, in the .cpp file:

  (unless the member is an integral type that is declared const)

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1(-24) , foo2(17);
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

static modifier

- Use static to declare a class member or method
- Members defined outside the class, in the .cpp file
- External access is via the class

  Class::MemberName

  rather than the object:

  object.MemberName

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1(-24) , foo2(17);
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

static modifier

- Use static to declare a class member or method
- Members defined outside the class, in the .cpp file
- External access is via the class
- Internal access is through the member / method name

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1(-24) , foo2(17);
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

static modifier

- Use static to declare a class member or method
- Members defined outside the class, in the .cpp file
- External access is via the class
- Internal access is through the member / method name

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
	static int _Count;
	int _myCount;
public:
	Foo( int count ) : _myCount( count ){ _Count++; }
	static int Count( void ){ return _Count; }
};
int Foo::_Count = 0;
int main( void )
{
	Foo foo1(-24) , foo2(17);
	cout << "Constructed " << Foo::Count() << " objects" << endl;
	return 0;
}
```

```
>> ./a.out
constructed 2 objects
>>
```

Note:
A method cannot be both const and static at the same time

Outline

- Member / method modifiers
  - const
  - static
- enums

enums

- An enumeration is a type consisting of a fixed set of integer-like values
  - By default:
    - The first value is 0
    - The next is the previous plus one

main.cpp

```
#include <iostream>
using namespace std;
enum
{
        On ,
        Off
};
int main( void )
{
        cout << On << " " << Off << endl;
        return 0;
}
```

```
>> ./a.out
0 1
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
  - By default:
    - The first value is 0
    - The next is the previous plus one
  - We can force prescribed values if we like

main.cpp

```
#include <iostream>
using namespace std;
enum
{
        On = 5 ,
        Off
};
int main( void )
{
        cout << On << " " << Off << endl;
        return 0;
}
```

```
>> ./a.out
5 6
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
  - But we need to beware of naming conflicts

main.cpp

```
#include <iostream>
using namespace std;
enum{ On , Off };
enum{ On , Below , In };
int main( void )
{
      cout << On << " " << Off << endl;
      return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra foo.cpp
main.cpp:4:7: error: redeclaration of On
 enum{ On , Below , In };
       ^~
main.cpp:3:7: note: previous declaration <anonymous enum> On
 enum{ On , Off };
       ^~
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
  - Values accessed like **static** class members

main.cpp

```
#include <iostream>
using namespace std;
enum State { Off , On };

void PrintState( State s )
{
	if( s==State::On ) cout << "on" << endl;
	if( s==State::Off ) cout << "off" << endl;
}

int main( void )
{
	PrintState( State::On );
	return 0;
}
```

```
>> ./a.out
on
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
  - Values accessed like **static** class members
  - Or directly…

main.cpp

```
#include <iostream>
using namespace std;
enum State { Off , On };

void PrintState( State s )
{
	if( s==On ) cout << "on" << endl;
	if( s==Off ) cout << "off" << endl;
}

int main( void )
{
	PrintState( On );
	return 0;
}
```

```
>> ./a.out
on
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
  - Values accessed like **static** class members
  - Or directly…
  - The names can be used for overloading

main.cpp

```
#include <iostream>
using namespace std;
enum State1 { Off , On };
enum State2 { Left , Right };

void PrintState( State1 s )
{
	if( s==On ) cout << "on" << endl;
	if( s==Off ) cout << "off" << endl;
}
void PrintState( State2 s )
{
	if( s==Left ) cout << "left" << endl;
	if( s==Right ) cout << "right" << endl;
}

int main( void )
{
	PrintState( On );
	PrintState( Right );
	return 0;
}
```

```
>> ./a.out
on
right
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
- We can treat the value as an integer

main.cpp

```
#include <iostream>
using namespace std;
enum State { Off , On , Count };

int ToggleState( State s )
{
	int _s = (s+1) % Count;
	return _s;
}

int main( void )
{
	State s = On;
	cout << s << endl;
	cout << ToggleState( s ) << endl;
	return 0;
}
```

```
>> ./a.out
1
0
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
- We can treat the values as integers
  - Treating an integer as a value requires an explicit cast

main.cpp

```
#include <iostream>
using namespace std;
enum State { Off , On , Count };

State ToggleState( State s )
{
	int _s = (s+1) % Count;
	return (State)( _s );
}

int main( void )
{
	State s = Off;
	cout << s << endl;
	cout << ToggleState( s ) << endl;
	return 0;
}
```

```
>> ./a.out
1
0
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
- We can treat the values as integers
  - Treating an integer as a value requires an explicit cast

main.cpp

```
#include <iostream>
using namespace std;
enum State { Off , On , Count };

State ToggleState( State s )
{
	int _s = (s+1) % Count;
	return static_cast< State >( _s );
}

int main( void )
{
	State s = Off;
	cout << s << endl;
	cout << ToggleState( s ) << endl;
	return 0;
}
```

```
>> ./a.out
1
0
>>
```

enums

- An enumeration is a type consisting of a fixed set of integer-like values
- We can define multiple **enum**s
- They can be named
- We can treat the values as integers
- We can define **enum**s within a class
  - Access is like a **static** member
  - The class can be used to resolve naming conflicts

main.cpp

```
#include <iostream>
using namespace std;
class Foo1
{
public:
	enum{ Off , On };
};
class Foo2
{
public:
	enum{ On=5 , Below , In };
};
int main( void )
{
	cout << Foo1::On << endl;
	cout << Foo2::On << endl;
	return 0;
}
```

```
>> ./a.out
1
5
>>
```