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.

```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 `void Foo::bar() const':
main.cpp:7:32: error: increment of member Foo::_bar in read-only object
    void bar( void ) const { _bar++; }
    ^~
```

```bash
$ g++ -std=c++11 -Wall -Wextra main.cpp
```
**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)

```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

```cpp
#include <iostream>
using namespace std;

void Print( int& i ){ cout << i << endl; }

class Foo
{

    int _bar;

cpublic:

    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

```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

- 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 the object (or its members) will not be passed as a non-**const** argument to some other function / method

main.cpp

```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;
}
```
**const modifier**

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

- 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.

```cpp
#include <iostream>
using namespace std;

class Foo {
private:
    int _bar;

public:
    void 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

```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

```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`)

```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`

```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

```cpp
#include <iostream>
using namespace std;
class Foo
{
    static int _Count;
    int _myCount;
public:
    Foo( int count ) : _myCount( count ){ _Count++; }  // Constructor
    static int Count( void ){ return _Count; }

    int main( void )
    {
        Foo foo1(-24), foo2(17);
        cout << "Constructed " << Foo::Count() << " objects" << endl;
        return 0;
    }
};
```

```bash
$ ./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

```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;
}
```

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

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

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

```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 enums
  • But we need to beware of naming conflicts

```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 };
   ^~
```

```cpp
```
enums

• An enumeration is a type consisting of a fixed set of integer-like values

• We can define multiple enums

• They can be named
  • Values accessed like static class members

```
#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 enums
• They can be named
  • Values accessed like static class members
  • Or directly...

```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 **enums**
- They can be named
  - Values accessed like **static** class members
  - Or directly...
  - The names can be used for overloading

```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;
}
```

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

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

```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;
}
```

```
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 enums
• They can be named
• We can treat the values as integers
  • Treating an integer as a value requires an explicit cast

```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;
}
```

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

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

```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 enums
• They can be named
• We can treat the values as integers
• We can define enums within a class
  • Access is like a static member
  • The class can be used to resolve naming conflicts

```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
>>