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

- The `this` keyword
- Overloading
- Implicitly defined constructors

The **this** keyword

- When defining a class method, we can get access to a pointer to the object that "owns" the method using the keyword **this**.

foo.h

```
using namespace std;
class Foo
{
	int _i;
public:
	void set( int i ){ _i = i; }
	int get( void ) const { return this->_i; }
};
```

Outline

- The `this` keyword
- Overloading
- Implicitly defined constructors

Overloading

- In C++, the compiler can distinguish between functions which have the same name but different numbers/types of parameters
  - The compiler will use the argument types to determine which function to call

main.cpp

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

void PrintType( int ){ cout << "int" << endl; }
void PrintType( float ){ cout << "float" << endl; }

int main(void)
{
	PrintType( 1 );
	PrintType( 1.f );
	return 0;
}
```

```
>> ./a.out
int
float
>>
```

Overloading

- In C++, the compiler can distinguish between functions which have the same name but different numbers/types of parameters
  - The compiler will use the argument types to determine which function to call
  - It <u>cannot</u> distinguish between functions based on their output type

main.cpp

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

int GetType( void ){ return 1; }
float GetType( void ){ return 1.f; }

int main(void)
{
    int i = GetType();
```

```
>>  g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function float GetType() :
main.cpp:5:7: error: ambiguating new declaration of float GetType()
 float GetType ( void ){ return 1.f; }
       ^~~
>>
```

Overloading

- In C++, the compiler can distinguish between functions which have the same name but different numbers/types of parameters
- Similarly, it can distinguish between class methods which have the same name but different numbers/types of parameters

main.cpp

```
#include <iostream>
using namespace std;
class Foo
{
public:
	void printType( int ){ cout << "int" << endl; }
	void printType( float ){ cout << "float" << endl; }
};
int main(void)
{
	Foo f;
	f.printType( 1 ) ; f.printType( 1.f );
	return 0;
}
```

```
>> ./a.out
int
float
>>
```

Overloading

- Some classes "naturally" define operators
  - Using full-fledged names can get cumbersome and hard to read

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float x( void ) const { return _v[0]; }
	float y( void ) const { return _v[1]; }
};
Point2D Add( Point2D p1 , Point2D p2 );
Point2D Scale( Point2D p , float s );
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(0,0) , q(1,1);
	Point2D avg = Scale( Add(p,q) , 0.5f );
	cout << "( " << avg.x() << " , " << avg.y() << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 0.5 , 0.5 )
>>
```

Overloading

- In C++, we can also overload operators:
  - +, -, *, / , < , | , & , [] , == , != , << , etc.

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
};
Point2D operator + ( Point2D p1 , Point2D p2 );
Point2D operator - ( Point2D p1 , Point2D p2 );
Point2D operator * ( Point2D p , float s );
Point2D operator / ( Point2D p , float s );
Point2D operator * ( float s , Point2D p );
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(0,0) , q(1,1);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 0.5 , 0.5 )
>>
```

Overloading

- We can also have class methods be operators
  - The first argument is the object itself

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(0,0) , q(1,1);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 0.5 , 0.5 )
>>
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
Point2D Point2D::operator * ( float s ) const
{
	return Point2D( _v[0] * s , _v[1] * s );
}
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
Point2D Point2D::operator * ( float s ) const
{
	return Point2D( _v[0] * s , _v[1] * s );
}
Point2D Point2D::operator - ( Point2D p ) const
{
	return (*this) + ( p * -1.f );
}
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
Point2D Point2D::operator * ( float s ) const
{
	return Point2D( _v[0] * s , _v[1] * s );
}
Point2D Point2D::operator - ( Point2D p ) const
{
	return (*this) + ( p * -1.f );
}
Point2D Point2D::operator / ( float s ) const
{
	return (*this) * (1.f/s);
}
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
	Point2D operator / ( float s ) const;
};
Point2D operator * ( float s , Point2D p );
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
Point2D Point2D::operator * ( float s ) const
{
	return Point2D( _v[0] * s , _v[1] * s );
}
Point2D Point2D::operator - ( Point2D p ) const
{
	return (*this) + ( p * -1.f );
}
Point2D Point2D::operator / ( float s ) const
{
	return (*this) * (1.f/s);
}
Point2D operator * ( float s , Point2D p ){ return p*s; }
```

Overloading

- In terms of implementation:

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x , float y );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
```

Point2D.h

```
…
Point2D::Point( float x , float y ){ _v[0] = x , _v[1] = y };
Point2D Point2D::operator + ( Point2D p ) const
{
	return Point2D( _v[0] + p._v[0] , _v[1] + p._v[1] );
}
Point2D Point2D::operator * ( float s ) const
{
	return Point2D( _v[0] * s , _v[1] * s );
}
Point2D Point2D::operator - ( Point2D p ) const
{
	return (*this) + ( p * -1.f );
}
Point2D Point2D::operator / ( float s ) const
{
```

2D p){ return p*s; }

<u>Note</u>:
In this implementation, we have opted for consistency over efficiency.
(e.g. subtraction is implemented by first multiplying by -1 and then adding)

Overloading

- We would also like to support streaming output using the << operator
  - This is a function that takes two arguments
    - The output stream
    - the object to be written
  - And returns a reference to the output stream (so we can chain outputs)

Point2D.h

```
#include <iostream>
using namespace std;
class Point2D
{
	float _v[2];
public:
	…
};
ostream& operator << ( ostream& os , Point2D p )
{
	os << "( " << p[0] << " , " << p[1] << " )";
	return os;
}
```

Overloading

- We would also like to support streaming output using the << operator
  - This is a function that takes two arguments
    - The output stream
    - the object to be written
  - And returns a reference to the output stream (so we can chain outputs)
- Using the **friend** keyword, we can give a function / operator access to the private class members

Point2D.h

```
#include <iostream>
using namespace std;
class Point2D
{
	float _v[2];
public:
	friend ostream& operator << ( ostream& os , Point2D p );
	…
};
ostream& operator << ( ostream& os , Point2D p )
{
	os << "( " << p._v[0] << " , " << p._v[1] << " )";
	return os;
}
```

Overloading

- Operator overloading allows us to write succinct, but still readable, code

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(0,0) , q(1,1);
	Point2D avg = Scale( Add(p,q) , 0.5f );
	cout << "( " << avg.x() << " , " << avg.y() << " )" << endl;
	return 0;
}
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(0,0) , q(1,1);
	cout << ( p + q ) / 2 << endl;
	return 0;
}
```

Outline

- The `this` keyword
- Overloading
- Implicitly defined constructors

Implicitly defined constructors

- A *default constructor* is a constructor that takes no arguments
  - It is called when an object is created without an argument list

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( void ){ _v[0] = _v[1] = 0; }
};
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p;
	return 0;
}
```

Implicitly defined constructors

- A *default constructor* is a constructor that takes no arguments
  - It is called when an object is created without an argument list

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( void ){ _v[0] = _v[1] = 0; }
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p;
	return 0;
```

You want to provide your own default constructor if the members require special initialization:

- They are plain-old-data that won't be initialized
- They need to be constructed in a non-default way

Implicitly defined constructors

- If no constructor is provided, C++ will implicitly define a default constructor:
  - It will be public
  - It will call the default constructor for all members that have one (i.e. all members that are not plain-old-data types)

Implicitly defined constructors

- A *copy constructor* is a constructor that takes a single argument -- the object from which to copy the member data
  - It is called when:
    - An object is initialized
    - An object is passed as an argument to a function
    - An object is returned by a function call

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( const Point2D& p ){ _v[0] = p._v[0] , _v[1] = p._v[1];
	}
};
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(1,1);
	Point2D q(p);
	Point2D r = q;
	return 0;
}
```

Implicitly defined constructors

- A *copy constructor* is a constructor that takes a single argument -- the object from which to copy the member data
  - It is called when:
    - An object is initialized
    - An object is passed as an argument to a function
    - An object is returned by a function call

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( const Point2D& p ){ _v[0] = p._v[0] , _v[1] = p._v[1];
	}
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(1,1);
	Point2D q(p);
	Point2D r = q;
	return 0;
```

You want to provide your own copy constructor when you need to perform a deep copy.

Implicitly defined constructors

- If not provided, C++ will also implicitly declare a *copy constructor*:
  - It will be public
  - It will copy the values of the argument's members into the members of the object being constructed

Implicitly defined constructors

- An *assignment operator* is an operator that assigns the object's members using the members of another object
  - It is called when the = sign is used

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D& operator = ( const Point2D& p )
	{
		_v[0] = p._v[0] , _v[1] = p._v[1];
	}
};
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(1,1);
	Point2D q(p) , r;
	r = q;
	return 0;
}
```

Implicitly defined constructors

- An *assignment operator* is an operator that assigns the object's members using the members of another object
  - It is called when the = sign is used

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D& operator = ( const Point2D& p )
	{
		_v[0] = p._v[0] , _v[1] = p._v[1];
	}
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(1,1);
	Point2D q(p) , r;
	r = q;
	return 0;
```

You want to provide your own assignment operator when you need to perform a deep copy.

Implicitly defined constructors

- If not provided, C++ will also implicitly declare an *assignment operator*:
  - It will be public
  - It will copy the values of the argument's members into the members of the object being constructed

Implicitly defined constructors

<u>Note</u>:

- If a variable is declared and assigned at the same time, the constructor is used
- If it's declared first and assigned later, assignment is used

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( const Point2D& p );

	Point2D& operator = ( const Point2D& p );
};
```

main.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;

int main( void )
{
	Point2D p(1,1);
	Point2D q = p, r;
	r = p;
	return 0;
}
```

Final Project