Intermediate Programming

Day 29

Outline

- Exercise 10-2
- Function overloading
- Operator overloading
- Review questions

Exercise 10-2 (part 2)

Define the constructor and the **add** and **clear** member functions.

grade_list.cpp

```
...
GradeList::GradeList( int capacity ) : capacity(capacity) , count(0)
{
      assert( capacity>0 );
      grades = new double[ capacity ];
      assert( grades );
}
```

Exercise 10-2 (part 2)

Define the constructor and the **add** and **clear** member functions.

grade_list.cpp

```
...
void GradeList::add( double grade )
{
    if( count==capacity )
    {
        capacity *=2;
        double *temp = new double[ capacity ];
        for( int i=0 ; i<count ; i++ ) temp[i] = grades[i];
        delete[] grades;
        grades = temp;
    }
    grades[ count++ ] = grade;
}
```

Exercise 10-2 (part 2)

Define the constructor and the **add** and **clear** member functions.

grade_list.cpp

```
...
void GradeList::add( int howmany , double *grades )
{
     for( int i=0 ; i<howmany ; i++ ) add( grades[i] );
}
```

Exercise 10-2 (part 2)

Define the constructor and the **add** and **clear** member functions.

grade_list.cpp

```
...
void GradeList::clear( void )
{
      delete[] grades;
      capacity = 1;
      grades = new double[capacity];
      assert( grades );
      count = 0;
}
```

Exercise 10-2 (part 3)

Declare and define the destructor

grade_list.h

```
class GradeList
{
public:
        ...
        ~GradeList( void );
        ...
};
```

grade_list.cpp

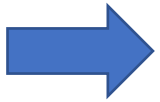
```
...
GradeList::~GradeList( void )
{
        delete[] grades;
        capacity = 0;
        count = 0;
}
```

Exercise 10-2 (part 4)

Declare and define the default constructor and the **begin** and **end** member functions.

grade_list.h

```
class GradeList
{
public:
        ...
        GradeList( int capacity );
```



grade_list.h

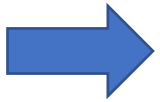
```
class GradeList
{
public:
        ...
        GradeList( int capacity=1 );
```

Exercise 10-2 (part 4)

Declare and define the default constructor and the **begin** and **end** member functions.

grade_list.h

```
class GradeList
{
public:
        ...
        GradeList( int capacity );
```



grade_list.h

```
class GradeList
{
public:
        ...
        GradeList( int capacity=1 );
        double *begin( void ){ return grades; }
        double *  end( void ){ return grades+count; };
```

Outline

- Exercise 10-2
- Function overloading
- Operator overloading
- Review questions

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
  - But if the argument type does not match one of the types with which the function is defined, the compiler can't figure out which way to cast!

main.cpp

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

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

int main(void)
{
      PrintType( 1.0 );
      return 0;
```

```
>> ++ main.cpp -std=c++11 -pedantic -Wall -Wextra
main.cpp:9:18: error: call of overloaded âPrintType(double)â is ambiguous
   PrintType( 1.0 );
                  ^
...
```

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 cannot 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; }
       ^~~
>>
```

Outline

- Exercise 10-2
- Function overloading
- Operator overloading
- Review questions

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=0 , float y=0 );
	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=0 , float y=0 );
	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=0 , float y=0 );
	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=0 , float y=0 );
	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.cpp

```
…
Point2D::Point2D( 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=0 , float y=0 );
	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.cpp

```
…
Point2D::Point2D( 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=0 , float y=0 );
	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.cpp

```
…
Point2D::Point2D( 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=0 , float y=0 );
	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.cpp

```
...
Point2D::Point2D( 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=0 , float y=0 );
	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.cpp

```
...
Point2D::Point2D( 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=0 , float y=0 );
	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.cpp

```
...
Point2D::Point2D( 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=0 , float y=0 );
	float operator[] ( int i ) const{ return _v[i]; }
	Point2D operator + ( Point2D p ) const;
	Point2D operator - ( Point2D p ) const;
	Point2D operator * ( float s ) const;
```

Point2D.cpp

```
...
Point2D::Point2D( 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; }
```

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

Overloading

- In all these implementations, the overloaded operators return a locally declared Point2D object.
- To do this, C++ invokes the copy constructor to replicate the local object:

  Point2D(const Point2D &);
- Like the default constructor, if your code does not provide one, C++ generates one for you that copies the member data.

Point2D.cpp

```
…
Point2D::Point2D( 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

- We could also overload the operators **+=**, **-=**, ***=**, **/=** etc.

Point2D.h

```
class Point2D
{
      float _v[2];
public:
      …
      Point2D& operator += ( Point2D p );
      Point2D& operator -= ( Point2D p );
      Point2D& operator *= ( float s );
      Point2D& operator /= ( float s );
};
```

Point2D.cpp

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

Note:
These operators return a reference to the object itself, allowing us to chain operators like (p+=q) *= 3;

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>
class Point2D
{
	float _v[2];
public:

	…
};
```

Point2D.cpp

```
…
std::ostream& operator << ( std::ostream& os , Point2D p )
{
	return os << "( " << p[0] << " , " << p[1] << " )";
}
```

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 an external function, operator, or class access to the private class members

Point2D.h

```
#include <iostream>
class Point2D
{
      float _v[2];
public:
      friend std::ostream& operator << (std::ostream & , Point2D );
      …
};
```

Point2D.cpp

```
…
std::ostream& operator << ( std::ostream& os , Point2D p )
{
      return os << "( " << p._v[0] << " , " << p._v[1] << " )";
}
```

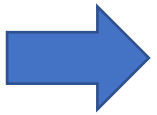
Overloading

- 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

- Exercise 10-2
- Function overloading
- Operator overloading
- Review questions

Review questions

1. What is overloading in C++?

When we create two functions with the same name but different arguments

Review questions

2. Can you overload a function with the same name, same parameters, but different return type?

No

Review questions

3. Is it true that we can overload all the operators of a class?

Almost (operators like `::` and `.` cannot be overloaded)

Review questions

4. What is a copy constructor? When will it be called?

A copy constructor initializes a new object by copying information from the argument. It is called when making an explicit call to the copy constructor, sending an object to a function by argument using pass-by-value, and returning a class object from a function by value.

Review questions

5. What happens if you don't define a copy constructor?

C++ generates a default (shallow) copy constructor that copies over the individual fields.

Review questions

6. What is the `friend` keyword? When do we use it?

This keyword signifies that some other class/function has access to an object's private members. It's used when we would like to define functions (like stream insertion/extraction) that need access to the private data but are not (can't be) members of the class.

Exercise 10-3

- Website -> Course Materials -> ex10-3