Intermediate Programming

Day 29

Outline

- Exercise 28
- Copy constructor
- Function overloading
- Operator overloading
- Review questions

Exercise 28

Define the constructor.

grade_list.cpp

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

Exercise 28

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

Exercise 28

Define the **clear** member function.

grade_list.cpp

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

Exercise 28

grade_list.cpp

```
...
void GradeList::clear( void )
{
```

```
>> valgrind --leak-check=full ./main1
...
==1538562==
==1538562== HEAP SUMMARY:
==1538562==     in use at exit: 64 bytes in 1 blocks
==1538562==   total heap usage: 9 allocs, 8 frees, 74,016 bytes allocated
==1538562==
==1538562== 64 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1538562==    at 0x484322F: operator new[](unsigned long) (vg_replace_malloc.c:640)
==1538562==    by 0x401757: GradeList::add(double) (grade_list.cpp:44)
==1538562==    by 0x40183F: GradeList::add(int, double*) (grade_list.cpp:59)
==1538562==    by 0x401431: main (main1.cpp:24)
==1538562==
==1538562== LEAK SUMMARY:
==1538562==    definitely lost: 64 bytes in 1 blocks
==1538562==    indirectly lost: 0 bytes in 0 blocks
==1538562==      possibly lost: 0 bytes in 0 blocks
==1538562==    still reachable: 0 bytes in 0 blocks
==1538562==         suppressed: 0 bytes in 0 blocks
==1538562==
==1538562== For lists of detected and suppressed errors, rerun with: -s
==1538562== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
>>
```

Exercise 28

Declare and define the destructor.

grade_list.cpp

```
...
GradeList::~GradeList( void ){ delete[] grades; }
...
```

grade_list.h

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

Exercise 28

Declare and define the destructor.

grade_list.cpp

```
...
GradeList::~GradeList( void ){ delete[] grades; }
...
```

grade_list.h

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

```
>> valgrind --leak-check=full ./main1
...
==1537987==
==1537987== HEAP SUMMARY:
==1537987==     in use at exit: 0 bytes in 0 blocks
==1537987==   total heap usage: 9 allocs, 9 frees, 74,016 bytes allocated
==1537987==
==1537987== All heap blocks were freed -- no leaks are possible
==1537987==
==1537987== For lists of detected and suppressed errors, rerun with: -s
==1537987== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
>>
```

Exercise 28

Declare and define the default constructor.

grade_list.h

```
...
class GradeList
{
public:
        ...
        GradeList( int capcity );
        ...
};
...
```

grade_list.h

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

Exercise 28

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

grade_list.h

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

Outline

- Exercise 28
- Copy constructor
- Function overloading
- Operator overloading
- Review questions

Copy Constructor

In addition to the default and non-default constructors C++ supports a *copy constructor* to construct one object from another.

Copy Constructor

In addition to the default and non-defa
copy constructor to construct one objec

- If you don't define one, C++ will create an implicitly defined copy constructor for you, which (recursively) copy-constructs the member data.
  - As opposed to the default constructor, a copy constructor will be created even if other (e.g. non-default) constructors are defined.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
      Rectangle r1(10,20);   // non-default ctor
      Rectangle r2(r1);      // copy ctor:
                             //   _w and _h copied
                             //   into r2 from r1
      return 1;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
      double _w , _h;
public:
      Rectangle( double w=0 , double h=0 )
            : _w(w) , _h(h) { }
};
#endif // RECTANGLE_INCLUDED
```

Copy Constructor

In addition to the default and non-defau
copy constructor to construct one objec

- If you don't define one, C++ will create an implicitly defined copy constructor for you, which (recursively) copy-constructs the member data.
- But you may want to create your own.

main.cpp

```
#include <iostream>
class Array
{
public:
      int sz , *values;
      Array( int s )
            : sz( s ) , values( new int [sz] ) {}
      ~Array( void ){ delete[] values; }
};
int main( void )
{
      Array a( 10 );
      Array b( a );
      return 0;
}
```

```
>> ./a.out
free(): double free detected in tcache 2
Abort (core dumped)
>>
```

Copy Constructor

In addition to the default and non-defau
copy constructor to construct one objec

main.cpp

```
#include <iostream>
class Array
{
public:
    int sz , *values;
    Array( int s )
        : sz( s ) , values( new int [sz] ) {}
    ~Array( void ){ delete[] values; }
```

```
>> valgrind --leak-check=full ./a.out
...
==1568619== Invalid free() / delete / delete[] / realloc()
==1568619==    at 0x484565B: operator delete[](void*) (vg_replace_malloc.c:1103)
==1568619==    by 0x401290: Array::~Array() (foo.cpp:7)
==1568619==    by 0x4011BC: main (foo.cpp:14)
==1568619==  Address 0x4db6c80 is 0 bytes inside a block of size 40 free'd
==1568619==    at 0x484565B: operator delete[](void*) (vg_replace_malloc.c:1103)
==1568619==    by 0x401290: Array::~Array() (foo.cpp:7)
==1568619==    by 0x4011B0: main (foo.cpp:14)
==1568619==  Block was alloc'd at
==1568619==    at 0x484322F: operator new[](unsigned long) (vg_replace_malloc.c:640)
==1568619==    by 0x401259: Array::Array(int) (foo.cpp:6)
==1568619==    by 0x40118F: main (foo.cpp:11)
...
>>
```

```
tected in tcache 2
```

Copy Constructor

In addition to the default and non-defau
copy constructor to construct one objec

- If you don't define one, C++ will create an implicitly defined copy constructor for you, which (recursively) copy-constructs the member data.
- But you may want to create your own.

main.cpp

```
#include <iostream>
class Array
{
public:
      int sz , *values;
      Array( int s )
            : sz( s ) , values( new int [sz] ) {}
      ~Array( void ){ delete[] values; }
};
int main( void )
{
      Array a( 10 );
      Array b( a );
      return 0;
}
```

```
>> ./a.out
free(): double free detected in tcache 2
Abort (core dumped)
>>
```

The default constructor sets b.values equal to a.values so both point to the same memory.
⇒ When destructor is called for a, it tries to delete memory that was already deleted when the destructor for b was called.

Copy Constructor

In addition to the default and non-defau
copy constructor to construct one objec

- If you don't define one, C++ will create an implicitly defined copy constructor for you, which (recursively) copy-constructs the member data.
- But you may want to create your own.

main.cpp

```
#include <iostream>
class Array
{
public:
	int sz , *values;
	Array( int s )
		: sz( s ) , values( new int [sz] ) {}
	Array( const Array &a )
		: sz( a.sz ) , values( new int[sz] )
		{
			for( unsigned int i=0 ; i<sz ; i++ )
				values[i] = a.values[i];
		}
	~Array( void ){ delete[] values; }
};
int main( void )
{
	Array a( 10 );
	Array b( a );
	return 0;
}
```

Copy Constructor

In addition to the default and non-defau
copy constructor to construct one objec

- If you don't define one, C++ will create an implicitly defined copy constructor for you, which (recursively) copy-constructs the member data.

main.cpp

```
#include <iostream>
class Array
{
public:
    int sz , *values;
    Array( int s )
        : sz( s ) , values( new int [sz] ) {}
    Array( const Array &a )
        : sz( a.sz ) , values( new int[sz] )
        {
            for( unsigned int i=0 ; i<sz ; i++ )
                values[i] = a.values[i];
        }
                                                alues; }
```

```
>> valgrind --leak-check=full ./a.out
...
==1570511== HEAP SUMMARY:
==1570511==     in use at exit: 0 bytes in 0 blocks
==1570511==   total heap usage: 3 allocs, 3 frees, 72,784 bytes allocated
==1570511==
==1570511== All heap blocks were freed -- no leaks are possible
==1570511==
==1570511== For lists of detected and suppressed errors, rerun with: -s
==1570511== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
>>
```

Copy Constructor

In addition to the default and non-default constructors C++ supports a *copy constructor* to construct one object from another.

It is called when:

- Constructing an object using another (including using the assignment operator, “**=**”, when declaring a variable)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

**Return value optimization* may keep it from being invoked in this case.

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (including using the assignment operator, "=", when declaring a variable)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (including using the assignment operator, "=", when declaring a variable)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- **Constructing an object using another** (including using the assignment operator, "=", when declaring a variable)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (**including using the assignment operator, "=", when declaring a variable**)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (including using the assignment operator, "=", when declaring a variable)
- Passing an argument to a function by value
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (including using the assignment operator, "=", when declaring a variable)
- Passing an argument to a function by value
- ~~[Possibly] returning an object from a function (defined on the function stack)~~*

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

***Return value optimization** may keep it from being invoked in thi

Copy Constructor

In addition to the default and non-de
copy constructor to construct one obj

It is called when:

- Constructing an object using another (including using the assignment operator, "=", when declaring a variable)
- **Passing an argument to a function by value**
- [Possibly] returning an object from a function (defined on the function stack)*

Return value optimization may keep it from being invoked in this

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct S
{
      S( void ){ cout << "default ctor called" << endl; }
      S( const S &s ){ cout << "copy ctor called" << endl; }
};

S foo1( void )
{
      S s;
      return s;
}

void foo2( S s ){}

int main( void )
{
      S s1;
      S s2(s1) , s3=s1;
      s1 = foo1();
      foo2( s1 );
      return 1;
}
```

```
>> ./a.out
default ctor called
copy ctor called
copy ctor called
default ctor called
copy ctor called
>>
```

Outline

- Exercise 28
- Copy constructor
- 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 functions have different *signatures.*
  - The compiler will use the argument types to infer 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
>>
```

*Note: a decimal number appended with an "f" is interpreted as a `float`. Otherwise it's interpreted as a `double`.

Overloading

- In C++, the compiler can distinguish between functions which have the same name but different numbers/types of parameters. The functions have different *signatures.*
  - The compiler will use the argument types to infer which function to call
    - Note:
      If the argument type does not match one of the types with which the function is defined, the compiler won't know which to cast to

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 functions have different *signatures.*
  - The compiler will use the argument types to infer which function to call
  - It **cannot** distinguish between functions based on their output type – the return type is not part of the signature.

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 betv
  the same name but different numbers/t
  The functions have different *signatures.*
  - The compiler will use the argument types to infer which function to call
  - It **cannot** distinguish between functions based on their output type – the return type is not part of the signature.
  - You can overload member functions (and constructors).

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct MyStruct
{
      void print( int ) { cout << "int" << endl; }
      void print( float ) { cout << "float" << endl; }
};

int main(void)
{
      MyStruct ms;
      ms.print( 1 );
      ms.print( 1.f );
      return 0;
}
```

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

Overloading

- In C++, the compiler can distinguish betv
  the same name but different numbers/t
  The functions have different *signatures.*
  - The compiler will use the argument types to infer which function to call
  - It **cannot** distinguish between functions based on their output type – the return type is not part of the signature.
  - You can overload member functions (and constructors).
  - You can overload based on whether the argument, or even the member function itself, is `const`. – the `const` designator is part of the signature.

main.cpp

```
#include <iostream>
using std::cout ; using std::endl;

struct MyStruct
{
      void print() const { cout << "const" << endl; }
      void print() { cout << "non-const" << endl; }
};
void PrintConst( const MyStruct &ms )
{
      ms.print();
}
void PrintNonConst( MyStruct &ms )
{
      ms.print();
}

int main(void)
{
      MyStruct ms;
      PrintConst( ms );
      PrintNonConst( m
      return 0;
}
```

```
>>   ./a.out
const
non-const
>>
```

Outline

- Exercise 28
- Copy constructor
- 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(1,2) , q(2,3);
	Point2D avg = Scale( Add(p,q) , 0.5f );
	cout << "( " << avg.x() << " , " << avg.y() << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

Overloading

- In C++, using the keyword **operator**, we can overload operators like +, -, *, / , += , -=, *= , /= , < , | , & , [] , () , == , != , << , etc.

```
<return type> operator <operator name> ( <operator arg(s)> )
{
        <operator body>
}
```

Overloading

- In C++, using the keyword **operator**, we can overload operators like +, -, *, / , += , -=, *= , /= , < , | , & , [] , () , == , != , << , etc.

<return type> operator <operator name> (<operator arg(s)>)

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x=0 , float y=0 );
	float operator[] ( int i ) const;
};
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(1,2) , q(2,3);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

Overloading

- In C++, using the keyword **operator**, we can overload operators like +, -, *, / , += , -=, *= , /= , < , | , & , [] , () , == , != , << , etc.

<return type> operator <operator name> (<operator arg(s)>)

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x=0 , float y=0 );
	float operator[] ( int i ) const;
};
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(1,2) , q(2,3);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

Overloading

- In C++, using the keyword **operator**, we can overload operators like +, -, *, / , += , -=, *= , /= , < , | , & , [] , () , == , != , << , etc.

<return type> operator <operator name> (<operator arg(s)>)

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x=0 , float y=0 );
	float operator[] ( int i ) const;
};
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(1,2) , q(2,3);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

Overloading

- In C++, using the keyword **operator**, we can overload operators like +, -, *, / , += , -=, *= , /= , < , | , & , [] , () , == , != , << , etc.

<return type> operator <operator name> (<operator arg(s)>)

Point2D.h

```
class Point2D
{
	float _v[2];
public:
	Point2D( float x=0 , float y=0 );
	float operator[] ( int i ) const;
};
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(1,2) , q(2,3);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

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;
	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(1,2) , q(2,3);
	Point2D avg = ( p + q ) / 2;
	cout << "( " << avg[0] << " , " << avg[1] << " )" << endl;
	return 0;
}
```

```
>> ./a.out
( 1.5 , 2.5 )
>>
```

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;
	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 };
float operator []( int i ) const
{
	assert( i==0 || i==1 );
	return _v[i];
}
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 operator + ( p * -1.f );
}
Point2D Point2D::operator / ( float s ) const
{
	return operator * (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 operator += ( p * -1.f );
}
Point2D &Point2D::operator /= ( float s )
{
      return operator *= (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

  Input:
  - The output stream
  - The object to be written

  Output:
  - 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

  Input:
  - The output stream
  - The object to be written

  Output:
  - 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] << " )";
}
```

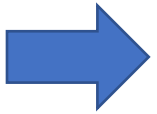
Overloading

- Operator overloading allows us to write succinct and clear code

main.cpp

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

int main( void )
{
	Point2D p(1,2) , q(2,3);
	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(1,2) , q(2,3);
	cout << ( p + q ) / 2 << endl;
	return 0;
}
```

Outline

- Exercise 28
- Copy constructor
- 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 29

- Website -> Course Materials -> Exercise 29