Intermediate Programming

Day 28

Outline

- Exercise 27
- Constructors
- Destructors
- The `this` keyword
- Review questions

Exercise 27

Implement the **mean** and the **median** functions.

grade_list.cpp

```
...
double GradeList::mean(void)
{
	double mean = 0;
	for( unsigned int i=0 ; i<grades.size() ; i++ ) mean += grades[i];
	return mean / grades.size();
}

double GradeList::median(void)
{
	return percentile(50);
}
```

```
>> ./main1
80th percentile was: 75
mean was: 40.2222
median was: 40
>>
```

Exercise 27

Fix the code (to remove the error with the **main** function trying to access a **private** member)

grade_list.h

```
...
class GradeList
{
public:
        ...



private:
        std::vector< double > grades;
};
```

main2.cpp

```
int main( void )
{
        ...
        for (size_t i = 0; i < gl.grades.size(); i++)
        {
                if( gl.grades[i]<min_so_far )
                {
                        min_so_far = gl.grades[i];
                }
        }
        ...
}
```

Exercise 27

Fix the code (to remove the error with the **main** function trying to access a **private** member)

grade_list.h

```
...
class GradeList
{
public:
        ...
        const std::vector< double > &get_grades( void ) const
                { return grades; }
        ...
private:
        std::vector< double > grades;
};
```

main2.cpp

```
int main( void )
{
        ...
        for (size_t i = 0; i < gl.grades.size(); i++)
        {
                if( gl.grades[i]<min_so_far )
                {
                        min_so_far = gl.grades[i];
                }
        }
        ...
}
```

main2.cpp

```
int main( void )
{
        ...
        for( size_t i=0 ; i<gl.get_grades().size() ; i++)
        {
                if( gl.get_grades()[i]<min_so_far )
                {
                        min_so_far = gl.get_grades()[i];
                }
        }
        ...
}
```

Exercise 27

Fix the code (to remove the error with the **main** function trying to access a **private** member)

grade_list.h

```
...
class GradeList
{
public:
	...
	const std::vector< double > &get_grades( void ) const
		{ return grades; }
	...
private:
	std::vector< double > grades;
};
```

main2.cpp

```
int main( void )
{
	...
	for (size_t i = 0; i < gl.grades.size(); i++)
	{
		if( gl.grades[i]<min_so_far )
		{
			min_so_far = gl.grades[i];
		}
	}
	...
}
```

main2.cpp

```
int main( void )
{
	...
	const std::vector< grades > &grades = gl.get_grades();
	for( size_t i=0 ; i<grades.size() ; i++)
	{
		if( grades [i]<min_so_far )
		{
			min_so_far = grades[i];
		}
	}
	...
}
```

Exercise 27

Write code to set a GradeList object with all even number in the range [0,100] and compute and print the summary statistics.

main3.cpp

```
#include <iostream>
#include "grade_list.h"

int main( void )
{
	// Initialize
	GradeList gl;
	for( unsigned int i=0 ; i<=100 ; i+=2 ) gl.add( i );

	// Get stats
	const std::vector< double > &grades = gl.get_grades();
	double min = grades[0] , max = grades[0];
	for( unsigned int i=0 ; i<grades.size() ; i++ )
	{
		min = std::min( grades[i] , min );
		max = std::max( grades[i] , max );
	}

	// Print stats
	std::cout << "Min / Max: " << min << " / " << max << std::endl;
	std::cout << "Median / Mean: " << gl.median() << " / " << gl.mean() << std::endl;
	std::cout << "75-th Percentile: " << gl.percentile(75) << std::endl;

	return 0;
}
```

Exercise 27

Write code to set a *GradeList* object with all even number in the range [0,100] and compute and print the summary statistics.

main3.cpp

```
#include <iostream>
#include "grade_list.h"

int main( void )
{
	// Initialize
	GradeList gl;
	for( unsigned int i=0 ; i<=100 ; i+=2 ) gl.add( i );

	// Get stats
	const std::vector< double > &grades = gl.get_grades();
	double min = grades[0] , max = grades[0];
	for( unsigned int i=0 ; i<grades.size() ; i++ )
	{
		min = std::min( grades[i] , min );
		max = std::max( grades[i] , max );
	}

	// Print stats
	std::cout << "Min / Max: " << min << " / " << max << std::endl;
	std::cout << "Median / Mean: " << gl.median() << " / " << gl.mean() << std::endl;
	std::cout << "

	return 0;
}
```

```
>> ./main3
Min / Max: 0 / 100
Median / Mean: 50 / 50
75-th Percentile: 76
>>
```

Outline

- Exercise 27
- Constructors
- Destructors
- Review questions

C++ Constructors

- The *default constructor* is called when no initialization parameters are passed.
  - If you do not provide a constructor, C++ will implicitly define one for you
    - It calls the constructor for each (non-POD) member datum

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }


	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

- The *default constructor* is called when no initialization parameters are passed.
  - If you do not provide a constructor, C++ will implicitly define one for you
    - It calls the constructor for each (non-POD) member datum
- We can also overload and define a *non-default constructor* which takes arguments.

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w , double h )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

- The *default constructor* is called when
are pa
  - If y
C++
    - (non-POD) member datum
- We can also overload and define a *non-default constructor* which takes arguments.
  - If you **only** define a non-default constructor, C++ will **not** define a default constructor for you.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.print();
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra -pedantic main.cpp
main.cpp: In function ‘int main()’:
main.cpp:5:13: error: no matching function for call to ‘Rectangle::Rectangle()’
    5 |   Rectangle r;
      |             ^
...
```

```
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w , double h )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

Warning:

When you create an array of objects (statically or with the keyword **new**) C++ invokes the default constructor for each element of the array.

⇒ If you only have a non-default constructor you will not be able to allocate the array.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r[3];
	r[0].print();
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w , double h )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

Warning:

When you create an array of objects (statically or with the keyword **new**) C++ invokes the default constructor for each element of the array.

⇒ If you only have a non-default constructor you will not be able to allocate the array.

You can get around this using initializer lists (but it's not pretty).

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r[] = { {0,0} , {1,1} , {2,2} };
	r[0].print();
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w , double h )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	…
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

Default arguments:

Often, the default constructor is a special case of the non-default constructor:

- For the Rectangle class the default constructor acts like the non-default constructor with arguments w=0 and h=0.

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w , double h )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

C++ Constructors

<u>Default arguments</u>:

Often, the default constructor is a special case of the non-default constructor.

C++ allows us to assign default values for the **last** argument(s) of constructors (and functions).

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r1(10,20);	// Standard ctor
	Rectangle r2;		// Same as "r2(0,0)"
	Rectangle r3(5);	// Same as "r3(5,0)"
	r.print();
	return 0;
}
```

```
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ) : _w(0) , _h(0) { }
	Rectangle( double w=0 , double h=0 )
		: _w(w) , _h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

Outline

- Exercise 27
- Constructors
- Destructors
- The `this` keyword
- Review questions

C++ Destructors

- A class *constructor*'s job is to initialize the fields of the object
  - A constructor can obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed

C++ Destructors

- A class *constructor*'s job is to initialize th
  - A constructor can obtain a resource (alloca should be released when the object is dest
- A class *destructor* is a method called by C++ when the object goes out of scope or is deallocated (e.g. using delete)

main.cpp

```
#include <iostream>
#include <cassert>
class MyArray
{
public:
	size_t sz;
	int* values;
	MyArray( int s ) : sz( s )
	{
		values = new int[sz];
		assert( values );
	}

};
int main( void )
{
	MyArray a( 10 );
	return 0;
}
```

C++ Destructors

- A class *constructor*'s job is to initialize th
  - A constructor can obtain a resource (alloca should be released when the object is dest
- A class *destructor* is a method called by C++ when the object goes out of scope or is deallocated (e.g. using delete)
    - Looks like a function:
        - Whose name is the class name
            - prepended with a "~"
        - With no (void) arguments
        - With no return type
    - This should be public

main.cpp
```
#include <iostream>
#include <cassert>
class MyArray
{
public:
	size_t sz;
	int* values;
	MyArray( int s ) : sz( s )
	{
		values = new int[sz];
		assert( values );
	}
	~MyArray( void ){ delete[] values; }
};
int main( void )
{
	MyArray a( 10 );
	return 0;
}
```

Outline

- Exercise 27
- Constructors
- Destructors
- The `this` keyword
- Review questions

The **this** keyword

- Within a class member function, we can get access to a pointer to the object that "owns" the function 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; }
};
```

The **this** keyword

- But we could also access the member data without the **this** pointer, so whey do we need it?

foo.h

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

foo.h

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

The **this** keyword

- But we could also access the member data without the **this** pointer, so whey do we need it?
  - Scope:
    If a member function argument has the same name as the member data, the member data goes out of scope.

main.cpp

```
#include <iostream>
class C
{
	int i;
public:
	C( void ) : i(0) { }
	void set( int i ) { i = i; }
	int get( void ) const { return i; }
};
int main( void )
{
	C c;
	c.set( 1 );
	std::cout << c.get() << std::endl;
	return 0;
}
```

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

The this keyword

- But we could also access the member data without the this pointer, so whey do we need it?
  - <u>Scope</u>:
    If a member function argument has the same name as the member data, the member data goes out of scope.
    We can bring it back into scope by using the this pointer.

main.cpp

```
#include <iostream>
class C
{
	int i;
public:
	C( void ) : i(0) { }
	void set( int i ) { this->i = i; }
	int get( void ) const { return i; }
};
int main( void )
{
	C c;
	c.set( 1 );
	std::cout << c.get() << std::endl;
	return 0;
}
```

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

The **this** keyword

- But we could also access the member data without it, so whey do we need it?
  - Scope:
    If a member function argument has the same name as the member data, the member data goes out of scope.
    We can bring it back into scope by using the **this** pointer.
  - Returning a reference:
    If we want a member function to return a reference to the object, we can return the dereferenced **this** pointer.

account.h

```
class Account
{
public:
	Account( double b=0. ) : _balance( b ) { }
	void credit( double amt ) { _balance += amt; }
	void debit( double amt ) { _balance -= amt; }
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

main.cpp

```
#include <iostream>
#include "account.h"
int main( void )
{
	Account a( 100 );
	a.credit( 5 );
	a.debit( 2 );
	a.debit( 3 );
	std::cout << a.balance() << std::endl;
	return 0;
};
```

The **this** keyword

```
account.h
class Account
{
public:
	Account( double b=0. ) : _balance( b ) { }
	Account& credit( double amt ) { _balance += amt; return *this; }
	Account& debit( double amt ) { _balance -= amt; return *this; }
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

- But we could also access t
  so whey do we need it?
  - <u>Scope</u>:
    If a member function argument has the same name as the member data, the member data goes out of scope.
    We can bring it back into scope by using the **this** pointer.
  - <u>Returning a reference</u>:
    If we want a member function to return a reference to the object, we can return the dereferenced **this** pointer
    This is useful if we would like to chain functions

```
main.cpp
#include <iostream>
#include “account.h”
int main( void )
{
	Account a( 100 );
	a.credit( 5 );
	a.debit( 2 );
	a.debit( 3 );
	std::cout << a.balance() << std::endl;
	return 0;
};
```

The this keyword

- But we could also access t
  so whey do we need it?
  - Scope:
    If a member function argument has the same name as the member data, the member data goes out of scope.
    We can bring it back into scope by using the **this** pointer.
  - Returning a reference:
    If we want a member function to return a reference to the object, we can return the dereferenced **this** pointer
    This is useful if we would like to chain functions

account.h

```
class Account
{
public:
	Account( double b=0. ) : _balance( b ) { }
	Account& credit( double amt ) { _balance += amt; return *this; }
	Account& debit( double amt ) { _balance -= amt; return *this; }
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

main.cpp

```
#include <iostream>
#include "account.h"
int main( void )
{
	Account a( 100 );
	a.credit( 5 ).debit( 2 ).debit( 3 );
	std::cout << a.balance() << std::endl;
	return 0;
};
```

The **this** keyword

<u>Note</u>:

In the case of an initializer list, we do not to need to use the **this** keyword to disambiguate since:

- The variable used for initialization is locally scoped (the argument to the constructor), and
- The variable being initialized can only be the **class**'s member data.

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	Rectangle( double w=0 , double h=0 )
		: w(w) , h(h) { }
	void print( void ) const;
	double area( void ) const;
	...
};
#endif // RECTANGLE_INCLUDED
```

Outline

- Exercise 27
- Constructors
- Destructors
- The `this` keyword
- Review questions

Review Questions

1. What is a non-default (or "alternative") constructor?

A constructor that takes arguments

Review Questions

2. If we define a non-default constructor, will C++ generate an implicitly defined default constructor?

No

Review Questions

3. When do we use the `this` keyword?

When a local variable hides a member variable.
When we want to return a reference to the object itself.

Review Questions

4. What is a destructor?

A method called by C++ when an object's lifetime ends or it is otherwise deallocated

Review Questions

5. A destructor will automatically release memories that are allocated in the constructor – true or false?

false

Exercise 28

- Website -> Course Materials -> Exercise 28