Intermediate Programming

Day 28

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The `this` keyword
- Destructors
- Review questions

Exercise 10-1 (part 2)

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

grade_list.cpp

```
...
double GradeList::mean( void )
{
	double mean = 0;
	for( size_t i=0 ; i<grades.size() ; i++ ) mean += grades[i];
	return mean / grades.size();
}
double GradeList::median( void ){ return percentile(50.); }
```

Exercise 10-1 (part 3)

Fix the code, which does not work as written.

grade_list.h

```
class GradeList
{
public:
        ...



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

main2.cpp

```
...
for( size_t i=0 ; i<gl.grades.size() ; i++ )
        if( gl.grades[i] < min_so_far )
                min_so_far = gl.grades[i];
...
```

Exercise 10-1 (part 3)

Fix the code, which does not work as written.

grade_list.h

```
class GradeList
{
public:
        ...
        size_t size( void ) const { return grades.size(); }
        double &operator [] ( size_t idx ){ return grades[idx]; }
private:
        std::vector< double > grades;
        bool is_sorted;
};
```

main2.cpp

```
...
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

```
...
for( size_t i=0 ; i<gl.size() ; i++ )
        if( gl[i] < min_so_far )
                min_so_far = gl[i];
...
```

Exercise 10-1 (part 4)

Follow the instructions in the comments in `main3.cpp`.

main3.cpp

```
#include "grade_list.h"
#include <iostream>
int main( void )
{

    GradeList gl;
    for( int i=0 ; i<=100 ; i+=2 ) gl.add( i );

    std::cout << "minimum: “          << gl.percentile(0)   << std::endl;
    std::cout << "maximum: “          << gl.percentile(100) << std::endl;
    std::cout << "median: “           << gl.median()        << std::endl;
    std::cout << "mean: “             << gl.mean()          << std::endl;
    std::cout << "75th percentile: “  << gl.percentile(75)  << std::endl;
   return 0;
}
```

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The `this` keyword
- Destructors
- Review questions

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;  // Default ctor called here
    …
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:

    double area( void ) const ;
};

#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - If no constructor is given, C++ implicitly defines one which calls the default constructors of the member data
    - ~~For plain old data (POD) like ints, floats, etc., values are initialized to zero~~

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:

	double area( void ) const ;
};

#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

en no initialization

- ~~For plain old data (POD) like ints, floats, etc., values are initialized to zero~~

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;    // Default ctor called here
    …
}
```

The members **r._w** and **r._h** are undefined

rectangle.h

```
ndef RECTANGLE_INCLUDED
fine RECTANGLE_INCLUDED
s Rectangle

    double _w , _h;
public:

    double area( void ) const ;
};

#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - Or the class can provide its own
    - Looks like a function:
      - Whose name is the class name
      - With no (void) arguments
      - With no return type
    - (Usually) this should be public

rectangle.h

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

#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - Or the class can provide its own
    - It can be defined in the class definition (if it's short)
    - Or it can be declared in the class and defined outside of it

rectangle.h

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

C++ Non-Default Construc

- Constructors can also take arguments, allowing the caller to “customize” the object

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
        Rectangle r1 , r2( 5 , 5 );
        std::cout << r1.area() << std::endl;
        std::cout << r2.area() << std::endl;
        return 0;
}
```

```
>> ./a.out
0
25
>>
```

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

C++ Non-Default Constructors

- Constructors can also take arguments, allowing the caller to “customize” the object
  - As with default constructors, we can use initializer lists for non-default constructors.

rectangle.h

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

C++ Non-Default Constructors

- Constructors can also take arguments, allowing the caller to “customize” the object
  - As with default constructors, we can use initializer lists for non-default constructors.
  - Initializer lists are the only way to initialize reference member data.

main.cpp

```
class C
{
public:
        int &r;
        C( int &i ) : r(i){ }
};
int main( void )
{
        int a;
        C c( a );
        return 0;
};
```

C++ Non-Default Construc

Note:
If we supply a constructor, C++ will not supply a default constructor!

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r1 , r2( 5 , 5 );
	std::cout << r1.area() << std::endl;
	std::cout << r2.area() << std::endl;
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:

	Rectangle( int w , int h ) : _w(w) , _h(h) {;}
	double area( void ) const { return _w*_h; }
};
#endif // RECTANGLE_INCLUDED
```

```
>> g++ main.cpp ...
main.cpp: In function ‘int main()’:
main.cpp:5:12: error: no matching function for call to
‘Rectangle::Rectangle()’
   5 |  Rectangle r1 , r2( 5 , 5 );
     |             ^
>>
```

C++ Non-Default Construc

<u>Note</u>:
Declaring an array of objects initializes each of the objects with the default constructor

⇒ The default constructor must exist

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r[2];
	r[0] = Rectangle( 2 , 3 );
	r[1] = Rectangle( 4 , 5 );
	return 0;
}
```

```
#i
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:

	Rectangle( int w , int h ) : _w(w) , _h(h) {;}
	double area( void ) const { return _w*_h; }
};
#endif // RECTANGLE_INCLUDED
```

```
>> g++ main.cpp ...
main.cpp: In function ‘int main()’:
main.cpp:5:16: error: no matching function for call to
‘Rectangle::Rectangle()’
   5 |  Rectangle r[2];
     |                ^
>>
```

C++ Non-Default Construc

Note:

Declaring an array of objects initializes each of the objects with the default constructor

⇒ The default constructor must exist

Work-arounds:

1. Use initializer lists

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r[] = { {2,3} , {4,5} };
    return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:

    Rectangle( int w , int h ) : _w(w) , _h(h) {;}
    double area( void ) const { return _w*_h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Non-Default Construc

Note:
Declaring an array of objects initializes each of the objects with the default constructor

⇒ The default constructor must exist

Work-arounds:

1. Use initializer lists
2. Grow an `std::vector` with `emplace_back` or `push_back`

main.cpp

```
#include <iostream>
#include <vector>
#include "rectangle.h"
int main( void )
{
	std::vector< Rectangle > r;
	r.reserve( 2 );
	r.emplace_back( 2 , 3 );
	r.push_back( Rectangle( 4 , 5 ) );
	return 0;
}
```

```
#i
#d
cla
{
	double _w , _h;
public:

	Rectangle( int w , int h ) : _w(w) , _h(h) {;}
	double area( void ) const { return _w*_h; }
};
#endif // RECTANGLE_INCLUDED
```

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The `this` keyword
- Destructors
- Review questions

Default arguments

- In C++ we can specify default values for function (or constructor) arguments in the declaration.
  - Effectively creates multiple (overloaded) versions of the function
  - Missing arguments will have their values substituted in sequentially, <u>from right to left</u>.
  - There cannot be non-default arguments after the default arguments.

Default arguments

- In C++ we can specify default values for function (or constructor) arguments in the declaration

) ve

s su

afte

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
#include <iostream>
class Rectangle
{
     double _w , _h;
public:
     Rectangle( int w=0 , int h=0 );
     double area( void ) const { return _w*_h; }
     …
};
Rectangle::Rectangle( int w , int h ) : _w(w) , _h(h) {;}
#endif // RECTANGLE_INCLUDED
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
     Rectangle r1 , r2( 1 ) , r3( 2 , 3 );
     std::cout << r1 << std::endl;
     std::cout << r2 << std::endl;
     std::cout << r3 << std::endl;
     retu
}
```

```
>> ./a.out
Rectangle[ 0 , 0 ]
Rectangle[ 1 , 0 ]
Rectangle[ 2 , 3 ]
>>
```

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The this keyword
- Destructors
- Review questions

The **this** keyword

When defining a member function (or constructor), we may want to use a local variable with the same name as one of the object’s members (instance variables).

- The local variable hides the instance variable.

circle.h

```
#ifndef CIRCLE_INCLUDED
#define CIRCLE _INCLUDED
class Circle
{
public:
	double r;
	Circle( double r=0 ){ r=r; }
	double area( void ) const { return r*r*3.14; }
};
#endif // CIRCLE _INCLUDED
```

main.cpp

```
#include <iostream>
#include “circle.h"
int main( void )
{
	Circle c(10);
	std::cout << c.area() << std::endl;
	return 0;
}
```

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

The **this** keyword

When defining a member function (or constructor), we may want to use a local variable with the same name as one of the object's members (instance variables).

- ✗ The local variable hides the instance variable.
- ✓ We can disambiguate by using the **this** keyword to get a pointer to the object whose member function we invoke.

circle.h

```
#ifndef CIRCLE_INCLUDED
#define CIRCLE _INCLUDED
class Circle
{
public:
        double r;
        Circle( double r=0 ){ this->r=r; }
        double area( void ) const { return r*r*3.14; }
};
#endif // CIRCLE _INCLUDED
```

main.cpp

```
#include <iostream>
#include "circle.h"
int main( void )
{
        Circle c(10);
        std::cout << c.area() << std::endl;
        return 0;
}
```

```
>> ./a.out
314
>>
```

The **this** keyword

When defining a member function (or constructor), we may want to use a local variable with the same name as one of the object’s members (instance variables).

- ✗ The local variable hides the instance variable.
- ✓ We can disambiguate by using the **this** keyword to get a pointer to the object whose member function we invoke.
- ✓ For constructors with initializer lists, the context disambiguates.

circle.h

```
#ifndef CIRCLE_INCLUDED
#define CIRCLE _INCLUDED
class Circle
{
public:
    double r;
    Circle( double r=0 ) : r(r) {;}
    double area( void ) const { return r*r*3.14; }
};
#endif // CIRCLE _INCLUDED
```

main.cpp

```
#include <iostream>
#include “circle.h"
int main( void )
{
    Circle c(10);
    std::cout << c.area() << std::endl;
    return 0;
}
```

```
>> ./a.out
314
>>
```

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The `this` keyword
- Destructors
- Review questions

C++ Destructors

- A class *constructor*’s job is to initialize the fields of the object
    - It’s common for a constructor to 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
  - It's common for a constructor to obtain file, etc.) that should be released when
- 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( size_t 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
  - It's common for a constructor to obtain file, etc.) that should be released when
- 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( size_t s ) : sz( s )
	{
		values = new int[sz];
		assert( values );
	}
	~MyArray( void ){ delete[] values; }
};

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

C++ Destructors

- A class *constructor*'s job is to initialize
  - It's common for a constructor to obtain file, etc.) that should be released when
- A class *destructor* is a method called by C++ when the object goes out of scope or is deallocated (e.g. using `delete`)
  - As with other methods, it can be declared in the class and defined outside of it

main.cpp

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

Outline

- Exercise 10-1
- Constructors
- Default arguments
- The this keyword
- Destructors
- 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 an instance variable

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 10-2

- Website -> Course Materials -> ex10-2