Intermediate Programming

Day 27

Outline

- Exercise 26
- C++ classes
- Constructors
- Review questions

Exercise 26

Compute the cumulative distribution from the probability distribution.

distribution.cpp

```
...
typedef std::vector< double >::const_iterator citer;
void make_cumulative( std::vector< double > &pdf );
...
void make_cumulative( std::vector< double > &pdf )
{
	for( unsigned int i=1 ; i<pdf.size() ; i++ ) pdf[i] += pdf[i-1];
}
...
```

Exercise 26

Implement the function finding the last iterator less than or equal to the prescribed value (naively).

distribution.cpp

```
...
typedef std::vector< double >::const_iterator citer;
void make_cumulative( std::vector< double > &pdf );
...
void make_cumulative( std::vector< double > &pdf )
{
	for( unsigned int i=1 ; i<pdf.size() ; i++ ) pdf[i] += pdf[i-1];
}

citer naive_find_last_iterator( citer begin , citer end , double v )
{
	for( citer it=begin ; it!=end ; ++it ) if( *it>v ) return it-1;
	return end;
}
...
```

Exercise 26

Implement the function finding the last iterator less than or equal to the prescribed value (efficiently).

distribution.cpp

```
...
typedef std::vector< double >::const_iterator citer;
void make_cumulative( std::vector< double > &pdf );
...
void make_cumulative( std::vector< double > &pdf )
{
	for( unsigned int i=1 ; i<pdf.size() ; i++ ) pdf[i] += pdf[i-1];
}

citer naive_find_last_iterator( citer begin , citer end , double v )
{
	for( citer it=begin ; it!=end ; ++it ) if( *it>v ) return it-1;
	return end;
}

citer fast_find_last_iterator( citer begin , citer end , double v )
{
	if( end==begin+1 )
	{
		if( *begin<=v ) return begin;
		else            return end;
	}
	citer mid = begin + (end-begin)/2;
	if( *mid<v ) return fast_find_last_iterator( mid , end , v );
	else         return fast_find_last_iterator( begin , mid , v );
}
```

Exercise 26

Confirm that the efficient implementation is, in fact, more efficient.

distribution.cpp

```
...
typedef std::vector< double >::const_iterator citer;
void make_cumulative( std::vector< double > &pdf );
...
void make_cumulative( std::vector< double > &pdf )
{
	for( unsigned int i=1 ; i<pdf.size() ; i++ ) pdf[i] += pdf[i-1];
}
citer naive_find_last_iterator( citer begin , citer end , double v )
{
	for( citer it=begin ; it!=end ; ++it ) if( *it>v ) return it-1;
	return end;
}
citer fast_find_last_iterator( citer begin , citer end , double v )
{
	if( end==begin+1 )
	{
		if( *begin<=v ) return begin;
		else            return end;
```

```
>> echo 1000000 10000 1000 | ./distribution
Number of bins: Number of random samples: Number of find tests: Confirmed that the CDF seems reasonable
Confirmed that the naive find seems reasonable
Confirmed that the fast find seems reasonable
Naive find time = 10444(ms)
 Fast find time = 1(ms)
>>
```

Outline

- Exercise 26
- C++ class**es**
- Constructors
- Review questions

C `struct`s

✓In C, we can use `struct`s to encapsulate heterogenous data.

main.c

```
#include <stdlib.h>
#include "rectangle.h"
int main( void )
{
	struct Rectangle r;
	r.w = r.h = 10;
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
struct Rectangle
{
	double w , h;
};
double area( struct Rectangle r );
#endif // RECTANGLE_INCLUDED
```

C structs

- ✓ In C, we can use structs to encapsulate heterogenous data.
- ✗ But if we want to support struct-specific functionality we have to declare/define it **outside** the struct.

main.c

```
#include <stdlib.h>
#include "rectangle.h"
int main( void )
{
	struct Rectangle r;
	r.w = r.h = 10;
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
struct Rectangle
{
	double w , h;
};
double area( struct Rectangle r );
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include "rectangle.h"
double area( struct Rectangle r )
{
	return r.w * r.h;
};
```

C++ classes

As with C structs, we can define new types (classes) for storing member data.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	class Rectangle r;
	r.w = r.h = 10;
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
};
#endif // RECTANGLE_INCLUDED
```

C++ classes

As with C structs, we can define new types (classes) for storing member data.

Unlike C structs, C++ classes support object-oriented-programming, with member functions defined **within** the class.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	class Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

C++ classes

As with C structs, we can define new types (classes) for storing member data.

Unlike C structs, C++ classes support object-oriented-programming, with member functions defined **within** the class.

- As with C structs, the definition is preceded by the keyword class and terminated with a semi-colon.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	class Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

C++ classes

As with C structs, we can define new types (classes) for storing member data.

Unlike C structs, C++ classes support object-oriented-programming, with member functions defined **within** the class.

- As with C structs, the definition is preceded by the keyword class and terminated with a semi-colon.
- Unlike C structs, we don’t need to use the keyword class to use the type (so we don’t need to use the keyword typedef in the declaration).

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

C++ classes

In C++ We can define new classes for storing member data and member functions.

- As with C structs, these need to be <u>declared</u> in a header file (with header guards).

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

C++ classes

In C++ We can define new classes for storing member data and member functions.

- As with C structs, these need to be declared in a header file (with heade
- Member functions can be defined in the header file if they are short.

main.cpp

```
#include <iostream>
#include “rectangle.h”
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
#include <iostream>
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << w << “ , " << h << std::endl; }
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << w << " , " << h << endl; }
double Rectangle::area( void ) const { return w * h; }
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

In C++ We can define new `class`es for storing member data and member functions.

- As with C `struct`s, these need to be <u>declared</u> in a header file (with header guards).
- Member functions can be <u>defined</u> in the header file if they are short.
- Otherwise they should be defined in a `.cpp` file.

rectangle.cpp

```cpp
#include <iostream>
#include "rectangle.h"
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << w << " , " << h << endl; }
double Rectangle::area( void ) const { return w * h; }
```

In C++ We can define new classes for storing member data and member functions.

- As with C structs, these need to be declared in a header file (with header guards).
- Member functions can be defined in the header file if they are short.
- Otherwise they should be defined in a .cpp file.
  - The member function name is preceded by the name of the class and "::" to indicate which class the function belongs to.

main.cpp

```cpp
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

rectangle.cpp

```
#include <iostream>
#include "rectangle.h“
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << w << “ , " << h << endl; }
double Rectangle::area( void ) const { return w * h; }
```

In C++ We can define new `class`es for storing member data and member functions.

- As with C `struct`s, these need to be <u>declared</u> in a header file (with header guards).
- Member functions can be <u>defined</u> in the header file if they are short.
- Otherwise they should be defined in a `.cpp` file.
- Either way, in the function body we do not need to specify who the members belong to. They belong to the object calling the member function.

main.cpp

```
#include <iostream>
#include “rectangle.h”
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << w << " , " << h << endl; }
double Rectangle::area( void ) const { return w * h; }
```

In C++ We can define new classes for storing member data and member functions.

- When member functions are declared const we are "promising" that calling the member function will not change the state of the class's member data.
- Note: If a member function is const, both the declaration and the definition need to use the const keyword.

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	r.w = r.h = 10;
	std::cout << r.area() << std::endl;
	return 0;
}
```

rectangle.h

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

C++ classes

- Members can be public or private (or protected, discussed later)
- We use public: and private: to divide the class definition into sections according to whether members are public or private
  - <u>All</u> members declared following a public / private keyword are public / private (until the next public / private keyword)
- Everything is private by default
- A public member can be accessed by any code with access to the class definition (code that includes the .h file)
- A private member can be accessed from other member functions in the class, but *not* by a user of the class

rectangle.h

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

C++ classes

⇒ We can protect members by making them **private**

rectangle.h

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

rectangle.cpp

```
#include <iostream>
#include "rectangle.h“
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << _w << “ , " << _h << endl; }
double Rectangle::area( void ) const { return _w * _h; }
```

C++ classes

⇒ We can protect members by making them **private**

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	std::cout << r._w << std::endl;
	return 0;
}
```

rectangle.h

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

```
#include <iostream>
#include "rectangle.h"
using std::cout ; using
void Rectangle::print(
double Rectangle::area( void ) const { return _w * _h; }
```

```
>> g++ main.cpp rectangle.cpp -std=c++11 -pedantic -Wall -Wextra
main.cpp: In function int main() :
main.cpp:6:18: error: double Rectangle::_w is private within this context
   std::cout << r._w << std::endl;
                  ^~
...
>>
```

C++ classes

- We can give read access to **private** member data by defining member functions that return a copy (or a **const** reference)

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	std::cout << r.width() << std::endl;
	return 0;
}
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h“
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << _w << “ , " << _h << endl; }
double Rectangle::area( void ) const { return _w * _h; }
```

rectangle.h

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

C++ classes

- Why make members **private**?
  - To ensure that the member data is within a specific range
  - To ensure that member data in the class be consistent

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h , _a;
public:
	void print( void ) const;
	double area( void ) const { return _a; }
	double width( void ) const { return _w; }
	double height( void ) const { return _h; }
};
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include <iostream>
#include <cassert>
#include "rectangle.h"
using std::cout ; using std::endl;
void Rectangle::print( void ) const { cout << _w << " , " << _h << endl;
void Rectangle::set( double w , double h )
{
	_w = w , _h = h , _a = w*h;
}
```

C++ classes

- C++ also allows us to define a struct
  - This is identical to a class only by default all members are public

rectangle.cpp

```
#include <iostream>
#include <cassert>
#include "rectangle.h"
void Rectangle::print( void ) const { std::cout << "width
void Rectangle::set( double w , double h )
{
    _w = w , _h = h , _a = w*h;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
struct Rectangle
{
    void print( void ) const;
    double area( void ) const { return _a; }
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }
private:
    double _w , _h , _a;
};
#endif // RECTANGLE_INCLUDED
```

Outline

- Exercise 26
- C++ classes
- Constructors
- Review questions

C++ Default Constructors

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

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
        double w , h;

        void print( void ) const;
        double area( void ) const ;

       dif // RECTANGLE_INCLUDED
```

main.cpp

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

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 constructor for each of the member data.
    - This is only true for classes.
      Plain Old Data (POD) like `int`s, `float`s, etc., values are still not initialized in C++

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;

	void print( void ) const;
	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
{
public:
	double w , h;
	Rectangle( void ){ w = 10 , h = 20; }
	void print( void ) const;
	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)

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	Rectangle( void ){ w = 10 , h = 20; }
	void print( void ) const;
	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 `.h` file and defined the `.cpp` file

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
        double w , h;
        Rectangle( void );
        void print( void ) const;
        double area( void ) const ;
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
Rectangle::Rectangle( void ){ w = 10 , h = 20; }
…
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - You cannot call the constructor directly.
  - A constructor is called when:
    - a new object is declared on the stack, or
    - when it is created on the heap using **new**

main.cpp

```
#include "rectangle.h"
int main( void )
{
	Rectangle r;
	Rectangle *rPtr = new Rectangle();
	…
	return 0;
}
```

C++ Default Constructors

<u>Note</u>:

- We’ve been using default constructors behind the scenes

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
	std::string name;
	std::cout << "Please enter your first name: ";
	std::cin >> name;
	std::cout << "Hello, " << name << "!" << std::endl;
	return 0;
}
```

C++ Non-Default Constructors

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

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
        std::string s1( "hello" );
        std::string s2 = "goodbye";           // std::string s2 = std::string( "goodbye" );
        std::cout << s1 << " " << s2 << std::endl;
        return 0;
}
```

```
>> ./a.out
hello goodbye
>>
```

C++ Non-Default Constructors

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

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
	double _w , _h;
public:
	Rectangle( void ){ _w = 10 , _h = 20; }
	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 ( 5 , 5 );
	std::cout << r1.area() << std::endl;
	std::cout << r2.area() << std::endl;
	return 0;
}
```

```
>> ./a.out
200
25
>>
```

C++ Non-Default Constructors

- Constructors can also take arguments, allowing the caller to "customize" the object
  - In this case we can have two functions with the same name but with different arguments*

```
                    rectangle.h
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
        double _w , _h;
public:
        Rectangle( void ){ _w = 10 , _h = 20; }
        Rectangle( int w , int h ){ _w=w , _h=h; }
        ...
};
#endif // RECTANGLE_INCLUDED
```

*More on function overloading later.

C++ Constructors

- Before the body of the constructor is called, C++ calls the default constructor for each of the member data.

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
	std::string str;
	MyString( void ){ str = "hello"; }
};
int main( void )
{
	MyString s;
	std::cout << s.str << std::endl;
	return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Constructors

- Before the body of the constructor is called, C++ calls the default constructor for each of the member data.
  - This is inefficient because the default constructor of `MyString` undoes the default construction of `str` with the results of a different constructor
  - We would like to be able to invoke the non-default constructor directly

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
	std::string str;
	MyString( void ){ str = "hello"; }
};
int main( void )
{
	MyString s;
	std::cout << s.str << std::endl;
	return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  - Before defining the body of the constructor:
    - a ":" followed by a comma-separated list of member constructors

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
	std::string str;
	MyString( void )
		: str( "hello" )
	{}
};
int main( void )
{
	MyString s;
	std::cout << s.str << std::endl;
	return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  - Before defining the body of the constructor:
    - a ":" followed by a comma-separated list of member constructors
  - Can do this to initialize POD member data that do not have constructors

main.cpp

```
#include <iostream>
class Foo
{
public:
	int x , y;
	Foo( void ) : x(5) , y(10) {}
};
int main( void )
{
	Foo f;
	std::cout << f.x << " " << f.y << std::endl;
	return 0;
}
```

```
>> ./a.out
5 10
>>
```

C++ Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  - Before defining the body of the constructor:
    - a ":" followed by a comma-separated list of member constructors
  - Can do this to initialize POD member data that do not have constructors
  - And also for reference member data
    - These *have to* be initialized within an initializer list (otherwise they are in an un-initialized state).

main.cpp

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

C++ Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly.

[WARNING]

The order of initialization is the order in which the member data is declared, not the order in which it appears in the list.

This, becomes an issue when you use the value of one member data to initialize the other.

main.cpp

```
class MyIntArray
{
public:
	int *values;
	int size;
	MyIntArray( int s )
		: size(s) , values(new int[size]){ }
};
int main( void )
{
	MyIntArray mia(5);
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra -pedantic main.cpp
main.cpp: In constructor ‘MyIntArray::MyIntArray(int)’:
main.cpp:5:13: warning: ‘MyIntArray::size’ will be initialized after [-Wreorder]
    5 |         int size;
      |             ^~~~
main.cpp:4:14: warning:   ‘int* MyIntArray::values’ [-Wreorder]
    4 |         int *values;
      |              ^~~~~~
main.cpp:6:9: warning:   when initialized here [-Wreorder]
    6 |         MyIntArray( int s )
      |         ^~~~~~~~~~
main.cpp: In constructor ‘MyIntArray::MyIntArray(int)’:
main.cpp:7:44: warning: ‘*this.MyIntArray::size’ is used uninitialized [-Wuninitialized]
    7 |                 : size(s) , values(new int[size]){ }
      |                                            ^~~~
>>
```

```
};
int main( void )
{
    MyIntArray mia(5);
    return 0;
}
```

The order of initialization is the order in which the member data is declared, not the order in which it appears in the list.

This, becomes an issue when you use the value of one member data to initialize the other.

Outline

- Exercise 26
- C++ classes
- Default constructors
- Review questions

Review Questions

1. What is object-oriented programming?

When the relative functionality is part of the object

Review Questions

2. What is the difference between a `public` and a `private` members?

A `public` member can be accessed freely by any code with access to the class definition. A `private` member can only be accessed from other member functions in the class.

Review Questions

3. Do class fields and member functions default to `public` or `private`?

`private`

Review Questions

4. Can we define member functions in a `struct` in C?
   How does C++ handle `struct`s?
   Can we do that in C++?

We cannot define member functions in a C `struct`.
In C++ a `struct` is like a C++ `class` but all members are default `public`.
A C++ `struct` can have member functions.

Review Questions

5. What is a default constructor?

A member function that C++ calls when you declare a new variable (on the stack or on the heap)

Review Questions

6. Why is using an initializer list in a `class` constructor a better choice than not using one?

Objects can be initialized with a non-default constructor, instead of having the default constructor called first and then resetting the value.

Exercise 27

- Website -> Course Materials -> Exercise 27