Intermediate Programming

Day 35

Outline

- Enumerated types
- Exceptions
- Review questions

Enumerated types

char.h

```
class Card
{
public:
        int rank , suit;
        Card( int r , int s ) : rank(r) , suit(s) {}
};
```

- Consider writing a card game…
  - Could use ints for representing the rank and the suit
    - ✓ Makes sense for 2-10
    - ✓ Sort of makes sense that ace=1, jack=11, queen=12, king=13
    - ✗ Doesn't make sense why hearts=1, clubs=2, spades=3, diamonds=4
    - ✗ Have to remember when calling the constructor that the first argument is the rank and the second is the suit

Enumerated types (unscoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
  - By default:
    - The first value is 0
    - The next is the previous plus one

main.cpp

```
#include <iostream>
enum
{
        Off ,
        On
};
void Print( int state )
{
        std::cout << state << std::endl;
}
int main( void )
{
        Print( Off );
        Print( On );
        return 0;
}
```

```
>> ./a.out
0
1
>>
```

Enumerated types (unscoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
  - By default:
    - The first value is 0
    - The next is the previous plus one
  - We can force prescribed values if we like

main.cpp

```
#include <iostream>
enum
{
        Off=5 ,
        On
};
void Print( int state )
{
        std::cout << state << std::endl;
}
int main( void )
{
        Print( Off );
        Print( On );
        return 0;
}
```

```
>> ./a.out
5
6
>>
```

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- **enum**s can be named

main.cpp

```
#include <iostream>
enum State { Off , On };

void Print ( State s )
{
        if( s==On ) std::cout << "on" << std:: endl;
        if( s==Off ) std:: cout << "off" << std:: endl;
}
void Print( int s )
{
        std::cout << s << std::endl;
}

int main( void )
{
        Print( On );
        int on = On;
        Print( on );
        Print( (int)On );
        return 0;
}
```

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

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- **enum**s can be named
  - The name of the **enum** is the type of the enumerator

main.cpp

```
#include <iostream>
enum State { Off , On };

void Print ( State s )
{
        if( s==On ) std::cout << "on" << std:: endl;
        if( s==Off ) std:: cout << "off" << std:: endl;
}
void Print( int s )
{
        std::cout << s << std::endl;
}

int main( void )
{
        Print( On );
        int on = On;
        Print( on );
        Print( (int)On );
        return 0;
}
```

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

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- **enum**s can be named
  - The name of the **enum** is the type of the enumerator
  - The enumerator can be cast to an **int**

main.cpp

```
#include <iostream>
enum State { Off , On };

void Print ( State s )
{
      if( s==On ) std::cout << "on" << std:: endl;
      if( s==Off ) std:: cout << "off" << std:: endl;
}
void Print( int s )
{
      std::cout << s << std::endl;
}

int main( void )
{
      Print( On );
      int on = On;
      Print( on );
      Print( (int)On );
      return 0;
}
```

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

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- **enum**s can be named
  - The name of the **enum** is the type of the enumerator
  - The enumerator can be cast to an **int**
  - The names can be used for overloading

main.cpp

```
#include <iostream>

enum State1 { Off , On };
enum State2 { Left , Right };

void Print( State1 s )
{
	if( s==On ) std::cout << "on" << std::endl;
	if( s==Off ) std::cout << "off" << std::endl;
}
void Print( State2 s )
{
	if( s==Left ) std::cout << "left" << std::endl;
	if( s==Right ) std::cout << "right" << std::endl;
}

int main( void )
{
	Print( On );
	Print( Right );
	return 0;
}
```

```
>> ./a.out
on
right
>>
```

Enumerated types (unscoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts

main.cpp

```
#include <iostream>
enum State1 { On , Off };
enum State2 { On , Below , In };
int main( void )
{
      std::cout << (int)On << std::endl;
      std::cout << (int)Off << std:: endl;
      return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp:3:15: error: ‘On’ conflicts with a previous declaration
    3 | enum State2 { On , Below , In };
      |               ^~
main.cpp:2:15: note: previous declaration ‘State1 On’
    2 | enum State1 { On , Off };
      |               ^~
>>
```

Enumerated types (scoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**

main.cpp

```
#include <iostream>

struct State1 { enum State { On , Off }; };
struct State2 { enum State { On , Below , In }; };

void Print( State1::State s )
{
	if( s==State1::State::On )
		std::cout << "On" << std::endl;
	else
		std::cout << "Off" << std::endl;
}

int main( void )
{
	Print( State1::State::Off );
	std::cout << (int)State1::State::Off << std::endl;
	return 0;
}
```

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

Enumerated types (scoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**
    - ✗ Requires extra indirection

main.cpp

```
#include <iostream>

struct State1 { enum State { On , Off }; };
struct State2 { enum State { On , Below , In }; };

void Print( State1::State s )
{
	if( s==State1::State::On )
		std::cout << "On" << std::endl;
	else
		std::cout << "Off" << std::endl;
}

int main( void )
{
	Print( State1::State::Off );
	std::cout << (int)State1::State::Off << std::endl;
	return 0;
}
```

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

Enumerated types (scoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**
    - C++ lets you do this directly by using
      enum class <name> { … };

main.cpp

```
#include <iostream>

enum class State1 { On , Off };
enum class State2 { On , Below , In };

void Print( State1 s )
{
	if( s==State1::On )
		std::cout << "On" << std::endl;
	else
		std::cout << "Off" << std::endl;
}

int main( void )
{
	Print( State1::Off );
	std::cout << (int)State1::Off << std::endl;
	return 0;
}
```

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

Enumerated types (scoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**
    - C++ lets you do this directly by using
      enum class <name> { … }
    - Need to specify which class the **name** belongs to

main.cpp

```
#include <iostream>

enum class State1 { On , Off };
enum class State2 { On , Below , In };

void Print( State1 s )
{
	if( s==State1::On )
		std::cout << "On" << std::endl;
	else
		std::cout << "Off" << std::endl;
}

int main( void )
{
	Print( State1::Off );
	std::cout << (int)State1::Off << std::endl;
	return 0;
}
```

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

Enumerated types (scoped)

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**
    - C++ lets you do this directly by using
      enum class <name> { … }
    - Need to specify which class the **enum** belongs to
    - Still can cast to an **int**

main.cpp

```
#include <iostream>

enum class State1 { On , Off };
enum class State2 { On , Below , In };

void Print( State1 s )
{
	if( s==State1::On )
		std::cout << "On" << std::endl;
	else
		std::cout << "Off" << std::endl;
}

int main( void )
{
	Print( State1::Off );
	std::cout << (int)State1::Off << std::endl;
	return 0;
}
```

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

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp:15:20: error: cannot convert ‘State1’ to ‘int’ in initialization
   15 |  int off = State1::Off
      |            ~~~~~~~~^~~
      |                    |
      |                    State1
...
>>
```

An enumeration is a type consisting of a fixed set of integer-like enumerators

- We can define multiple **enum**s
  - But we need to beware of naming conflicts
  - We can bypass this by encapsulating each **enum** within its own **struct**, **class**, or **namespace**
    - C++ lets you do this directly by using
      enum class <name> { … }
    - Need to specify which class the **enum** belongs t
    - Still can cast to an **int**, but now it has to be an explicit cast

```
;

    if( s==State1::On )
        std::cout << "On" << std::endl;
    else
        std::cout << "Off" << std::endl;
}

int main( void )
{
    Print( State1::Off );
    int off = State1::Off;
    std::cout << off << std::endl;
    std::cout << (int)State1::Off << std::endl;
    return 0;
}
```

Enumerated types (scoped)

- Consider writing a card game…
  - Could create a scoped **enum** for the rank and a scoped **enum** for the suit
    - Need to initialize aces to be "1", but everything else follows
  - Can define two different constructors supporting either order of rank/suit

char.h

```
enum class Rank
{
        ACE=1, TWO, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK,
        QUEEN, KING
};

enum class Suit
{
        HEARTS, DIAMONDS, CLUBS, SPADES
};

class Card
{
public:
        Rank rank;
        Suit suit;
        Card( Rank r , Suit s ) : rank(r) , suit(s) {}
        Card( Suit s , Rank r ) : rank(r) , suit(s) {}
};
```

Outline

- Enumerated types
- Exceptions
- Review questions

Exceptions

- Things can go wrong at run-time:
  - Invalid data, file I/O problems, arithmetic operation problems, ...
- How should we deal with run-time error conditions?
  - Have a function return an error code
  - Display error messages (often using `cerr`)
  - Bail out using `exit( int )` - need `cstdlib`
    - "clean exit": destructors get called, files get closed, etc.
  - Bail out using `abort( void )`
    - "hard exit": nothings gets called or cleaned up

We may not know the right response without a larger context. And propagating to a more global context is annoying.

No recovery mechanism

Exceptions

- Exceptions are objects that help us manage run-time error situations
  - The class `std::exception` is a type in the standard library
  - But we can define our own exception classes too

- We may employ `throw` statements when we identify an error situation that we don't want to handle immediately (or at all)...

```
throw std::exception();
```

- ... and `try` / `catch` blocks to indicate situations we'd like to handle, and how to handle them (whether we threw the associated exceptions ourselves or not)

Exceptions

- Exceptions ar
  - The class st
  - But we can c
- We may empl… situation that we don't want to …all)…

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
    try{ throw std::exception(); }
    catch( std::exception & ) { std::cerr << "something bad happened << std::endl; }
    return 0;
}
```

```
>> ./a.out
something bad happened
>>
```

throw std::exception();

- … and try / catch blocks to indicate situations we'd like to handle, and how to handle them (whether we threw the associated exceptions ourselves or not)

Exceptions

- The `std::exception` class has a virtual method that returns a string describing the exception (and will not throw an exception):

```
virtual const char *what( void ) const noexcept;
```

- We can define our own exception class that we can `throw` / `catch`
  - Although we don't have to, we should make it derive from `std::exception`
  - We can over-ride the `exception::what` method

For this to work correctly, we may be depending on slicing.

⇓

`catch` by reference, not by value.

Exceptions

- The std::exce
  describing the
  virtual
- We can define
  - Although we don't have to, we shou d::exception
  - We can over-ride the exception::what method

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
        try{ throw std::exception(); }
        catch( std::exception &ex ){ std::cerr << "Exception: " << ex.what() << std::endl; }
        return 0;
}
```

```
>> ./a.out
Exception:  std::exception
>>
```

For this to work correctly, we may be depending on slicing.

⇓

catch by reference, not by value.

Exceptions

- The std::exce
  describing the
  virtual
- We can define
  - Although we don't have to, we shou d::exception
  - We can over-ride the exception::what method

main.cpp

```
#include <iostream>
#include <exception>

class MyException : public std::exception
{
public:
	const char *what( void ) const noexcept override { return "my exception"; }
};
int main( void )
{
	try{ throw MyException(); }
	catch( std::exception &ex ){ std::cerr << "Exception: " << ex.what() << std::endl; }
	return 0;
}
```

```
>> ./a.out
Exception:  my exception
>>
```

For this to work correctly, we may be depending on slicing.

⇓

catch by reference, not by value.

Exceptions

- The std::exce
  describing the
  virtual
- We can define
  - Although we don't have to, we shou d::exception
  - We can over-ride the exception::what method

main.cpp

```
#include <iostream>
#include <exception>

class MyException : public std::exception
{
public:
	const char *what( void ) const noexcept override { return "my exception"; }
};
int main( void )
{
	try{ throw MyException(); }
	catch( std::exception   ex ){ std::cerr << "Exception: " << ex.what() << std::endl; }
	return 0;
}
```

```
>> ./a.out
Exception:  std::exception
>>
```

For this to work correctly, we may be depending on slicing.

⇓

catch by reference, not by value.

try/catch blocks

- If no exception is thrown by code in a try block, then catch block(s) are all skipped, and execution continues normally.

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
	try{ std::cout << "trying" << std::endl; }
	catch( std::exception & )
	{
		std::cerr << "catching" << std::endl;
	}
	std::cout << "done" << std::endl;
	return 0;
}
```

```
>> ./a.out
trying
done
>>
```

try/catch blocks

- If an exception is thrown by code in a try block, execution immediately jumps to the **first** matching catch block (if one exists), and code in that single catch block is executed, then execution continues normally after the catch block.
  - A catch block "matches" if the type of the exception is derived from the parameter type

main.cpp

```
#include <iostream>
#include <exception>
class MyException : public std::exception
{
};

int main( void )
{
	try{ throw MyException(); }
	catch( MyException & )
	{
		std::cerr << "caught mine" << std::endl;
	}
	catch( std::exception & )
	{
		std::cerr << "caught generic" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught mine
>>
```

try/catch blocks

- If an exception is thrown by code in a try block, execution immediately jumps to the **first** matching catch block (if one exists), and code in that single catch block is executed, then execution continues normally after the catch block.
  - A catch block "matches" if the type of the exception is derived from the parameter type
  - List the catch blocks in order from most derived to least!

main.cpp

```
#include <iostream>
#include <exception>
class MyException : public std::exception
{
};

int main( void )
{
	try{ throw MyException(); }
	catch( std::exception & )
	{
		std::cerr << "caught generic" << std::endl;
	}
	catch( MyException & )
	{
		std::cerr << "caught mine" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught generic
>>
```

try/catch blocks

- If an exception is thrown by code in try block, but no suitable catch block exists, the exception is passed up the call stack

main.cpp

```
#include <iostream>
#include <exception>

void foo( void ){ throw std::exception(); }

int main( void )
{
	try{ foo(); }
	catch( std::exception& )
	{
		std::cerr << "caught generic" << std::endl;
	}
	return 0;
}
```

```
>> ./a.out
caught generic
>>
```

Note:
In particular, this means that the exception can be handled “further up the chain” where the wider context may give a better sense of how to handle the exception.

try/catch blocks

- If an exception is thrown by code in try block, but no suitable catch block exists, the exception is passed up the call stack
  - If the exception isn't caught in main, the code terminates

main.cpp

```
#include <iostream>
#include <exception>

void foo( void ){ throw std::exception(); }

int main( void )
{
      std:: cout << "pre foo" << std:: endl;
      foo();
      std::cout << "post foo" << std::endl;
      return 0;
}
```

```
>> ./a.out
pre foo
terminate called after throwing an instance of 'std::exception'
  what():  std::exception
Abort (core dumped)
>>
```

try/catch blocks

- Code after the **throw**ing of an exception is not executed

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
	try
	{
		std::cout << "a" << std::endl;
		throw std::exception();
		std::cout << "b" << std::endl;
	}
	catch( std::exception & ){ std::cout << "caught exception!" << std::endl; }
	return 0;
}
```

```
>> ./a.out
a
caught exception
>>
```

try/catch blocks

- `stdexcept` defines many useful (derived) exception classes.
  - Most have a constructor that takes a (descriptive) **string** as an argument

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

int main( void )
{
	try{ throw std::overflow_error( "ran out of space!" ); }
	catch( std::invalid_argument &e ){ std::cout << "got invalid argument: e.what() = "     << e.what() << std::endl; }
	catch( std::overflow_error   &e ){ std::cout << "got overflow exception: e.what() = " << e.what() << std::endl; }
	catch( std::exception        &e ){ std::cout << "got base exception: e.what() = "      << e.what() << std::endl; }
	return 0;
}
```

```
>> ./a.out
got overflow exception: e.what()=ran out of space!
>>
```

try/catch blocks

- `stdexcept` defines many useful (derived) exception classes.
  - Most have a constructor that takes a (descriptive) **string** as an argument

main.cpp

```
#include <iostream>
#include <exception>
#include <stdexcept>

int main( void )
{
	try
	{
		int *array = new int[100000000000];
		array[0] = 10;
	}
	catch( std::bad_alloc &bae ){ std::cout << "error while allocating: " << bae.what() << std::endl; }
	return 0;
}
```

```
>> ./a.out
error while allocating: std::bad_alloc
>>
```

Note: We are trying to allocate 4 terabytes of data. That's likely to exceed RAM.

try/catch blocks

- `stdexcept` defines many useful (derived) exception classes.
  - Most have a constructor that takes a (descriptive) `string` as an arugment

```
>> ./a.out
error with vector: vector::_M_range_check: __n (which is 10) >= this->size() (which is 10)
>>
```

```
#incl
#incl
#include <stdexcept>
#include <vector>
int main( void )
{
	try
	{
		std::vector<double> v(10);
		v.at(10) = 21;
	}
	catch( std::out_of_range &ex ){ std::cout << "error with vector: " << ex.what() << std::endl; }
	return 0;
}
```

Note: The `vector::at` method tests if the index is in bounds and throws an exception if it's not

try/catch blocks

- While not standard, you can throw/catch anything you want

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
      try{ throw 7; }
      catch( int &i ){ std::cout << "caught: " << i << std::endl; }
      return 0;
}
```

```
>> ./a.out
caught 7
>>
```

try/catch blocks

- If you want to catch all exceptions, use ellipsis

main.cpp

```
#include <iostream>
#include <exception>

int main( void )
{
      try{ throw 7; }
      catch( … ){ std::cout << "something bad happened: " << std::endl; }
      return 0;
}
```

```
>> ./a.out
Something bad happened
>>
```

Outline

- Enumerated types
- Exceptions
- Review questions

Review questions

1. What is the difference between an unscoped and a scoped `enum`?

Two scoped `enum`s can have enumerators with the same names.
But there are no implicit conversions from the values of a scoped `enum` to an `int`.

Review questions

2. Why do we use **exceptions**?

To indicate an error has occurred where there is no reasonable way to continue from the point of the error (but there might be a way to continue from somewhere else)

Review questions

3. What keyword is used to generate an exception?
   What keyword indicates that the block of code may generate an exception?
   What keyword indicates what should be done in the case of an exception?

throw, try, catch

Review questions

4. In the case of multiple matching `catch` blocks, which one `catch`es the exception?

The first one whose type equals to, or is a base of, the class of the thrown exception

Review questions

5. How do you get the message associated with an exception?

Call the exception's `what` member function

Exercise 35

- Website -> Course Materials -> Exercise 35