Intermediate Programming Day 35

Outline

- Enumerated types
- Exceptions
- Review questions

Enumerated types

- 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

3

char.h

Card(int r , int s) : rank(r) , suit(s) {}

class Card

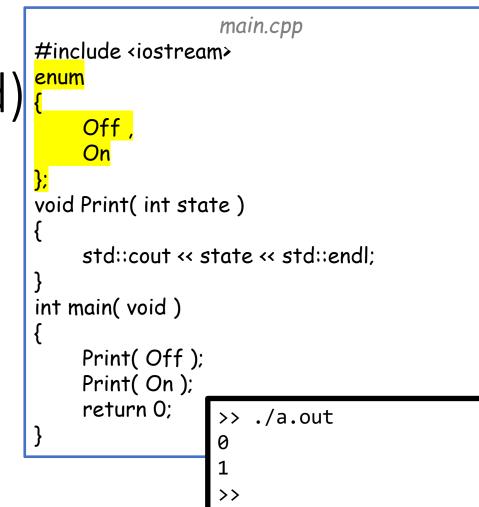
int rank , suit;

public:

};

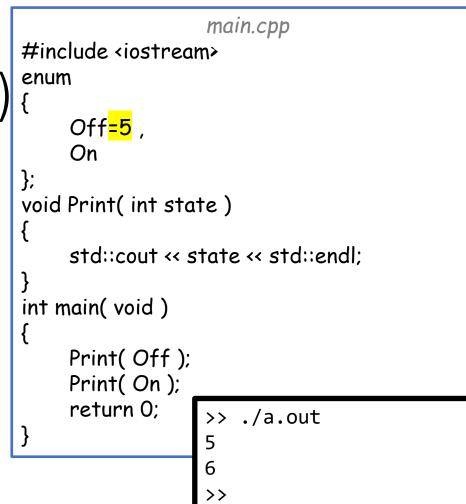
Enumerated types (unscoped)

- 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



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



Enumerated types (unscope

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

```
main.cpp
#include <iostream>
enum State { Off , On };
void Print (States)
     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;</pre>
int main(void)
     Print(On);
     int on = On;
     Print( on );
     Print( (int)On );
                         >> ./a.out
     return 0;
                         on
                         1
                         1
```

>>

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- enums 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;</pre> int main(void) Print(On); int on = On;Print(on); Print((int)On); >> ./a.out return 0: on 1 >>

Enumerated types (unscope

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- enums 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 (States)
     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;</pre>
int main(void)
     Print(On);
     int on = On;
     Print( on );
     Print( (int)On );
                         >> ./a.out
     return 0:
                         on
                         1
                         >>
```

#include <iostream>

Enumerated types (unscope enum State1 { Off, On }; enum State2 { Left Right };

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

```
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;</pre>
     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 enums
 - 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;</pre>

return 0;

main.cpp

#include <iostream>

Enumerated types (scoped struct State1 { enum State { On, Off }; }; struct State2 { enum State { On, Below, In }; };

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

```
void Print( State1::State 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 struct State1 { enum State { On , Off }; }; struct State2 { enum State { On , Below , In }; };

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

× Requires extra indirection

void Print(State1::State s)

#include <iostream>

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

main.cpp

int main(void)

Print(State1::Off); std::cout << (int)State1::Off << std::endl; return 0;

>> ./a.out
Off
1
>>

main.cpp

enum class State1 { On , Off }; enum class State2 { On , Below , In }; 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 conflicts by encapsulating each enum within its own struct, class, or namespace
 - C++ lets you do this directly by using enum class <name> { ... };

```
void Print( State1 s )
```

#include <iostream>

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

```
int main(void)
```

Print(State1::Off); std::cout << (int)State1::Off << std::endl;</pre> return 0;

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

Enumerated types (scoped enum class State1 { enum class State2 {

- An enumeration is a type consisting of a fixed set of integer-like enumerators
- We can define multiple enums
 - But we need to beware of naming conflicts {
 - We can bypass conflicts 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==<mark>State1::On</mark>) std::cout << "On" << std::endl; std::cout << "Off" << std::endl;</pre> else int main(void) Print(State1::Off); std::cout << (int)State1::Off << std::endl;</pre> 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 };
```

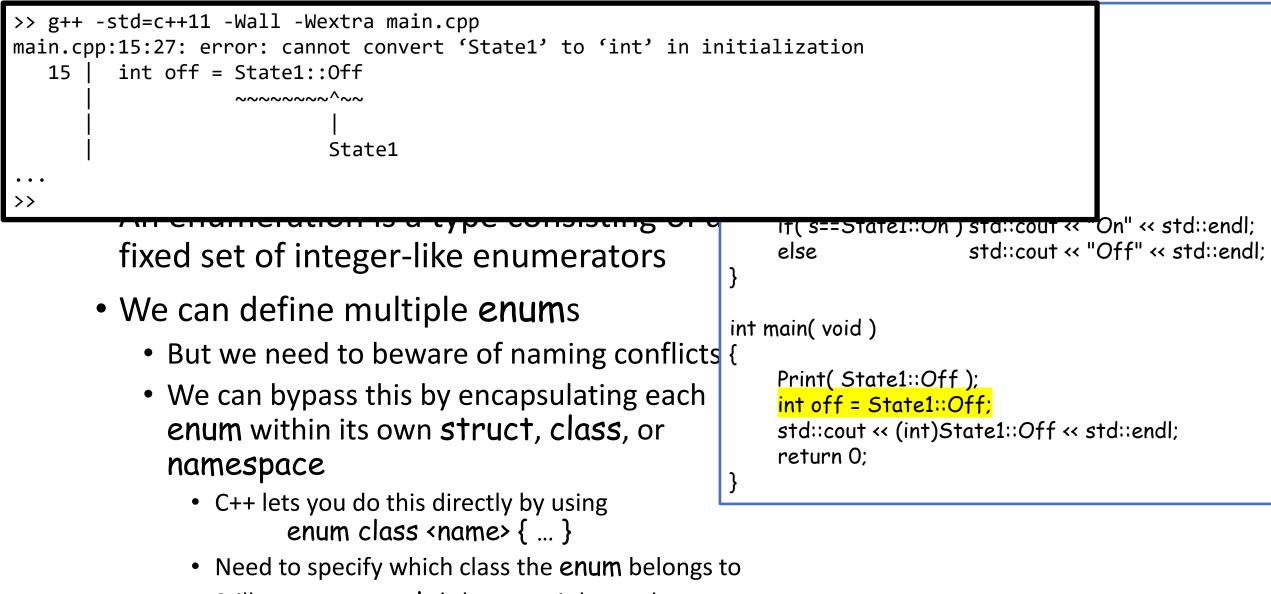
void Print(State1 s)

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

```
int main(void)
```

Print(State1::Off); std::cout << (int)State1::Off << std::endl;</pre> return 0;

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



Still can cast to an int, but now it has to be an explicit cast

Enumerated types (scoped)

- Consider writing a card game...
 - Could create a scoped enum for the rank and a scoped enum for the suit

```
char.h
enum class Rank
    ACE=1, TWO, THREE, FOUR, FIVE, SIX,
    SEVEN, EIGHT, NINE, TEN, JACK,
    QUEEN, KING
};
enum class Suit
ł
    HEARTS, DIAMONDS, CLUBS, SPADES
};
class Card
public:
    Rank rank;
    Suit suit:
    Card(Rank r, Suit s): rank(r), suit(s) {}
    Card(Suits, Rankr): rank(r), suit(s) {}
};
```

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

```
char.h
enum class Rank
    ACE=1, TWO, THREE, FOUR, FIVE, SIX,
    SEVEN, EIGHT, NINE, TEN, JACK,
    QUEEN, KING
};
enum class Suit
٦
    HEARTS, DIAMONDS, CLUBS, SPADES
};
class Card
public:
    Rank rank;
    Suit suit:
    Card(Rank r, Suit s): rank(r), suit(s) {}
    Card(Suits, Rankr): rank(r), suit(s) {}
};
```

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(Suits, Rankr): rank(r), suit(s) {}
};
```

Outline

- Enumerated types
- Exceptions
- Review questions

Exceptions

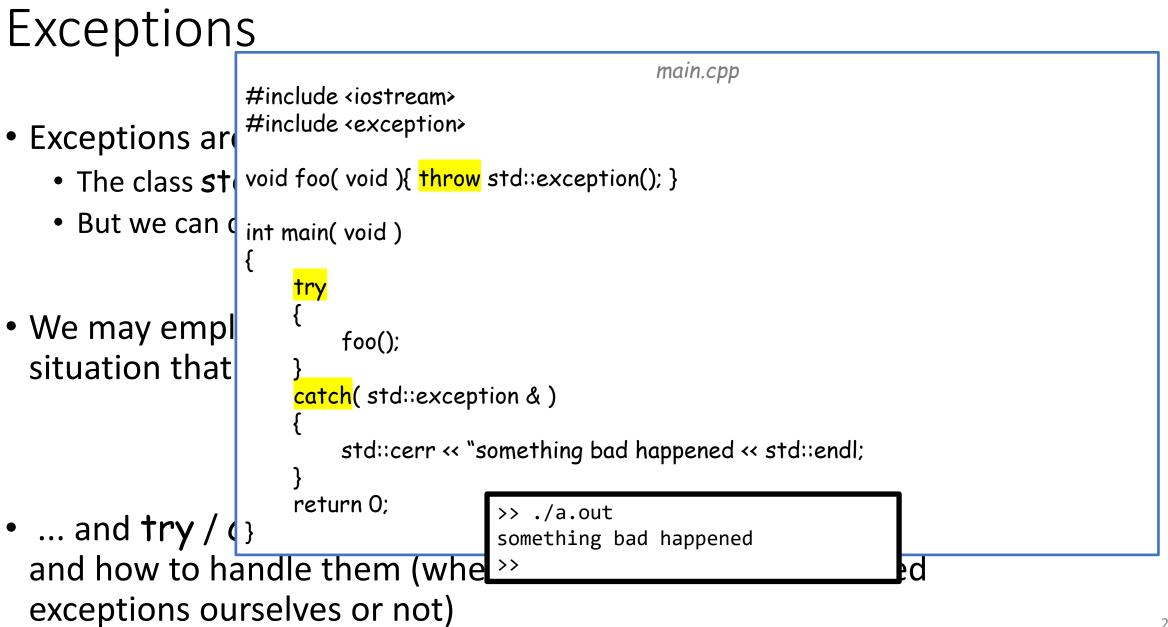
- Things can go wrong at run-time:
 - Invalid data, file I/O problems, arithmetic operation problems, ...
- How should we deal with run-time error conditions?
 - Have a function return an error code
 - Display error messages (often using cerr)
 - Bail out using exit(int) need cstdlib
 - "clean exit": 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)

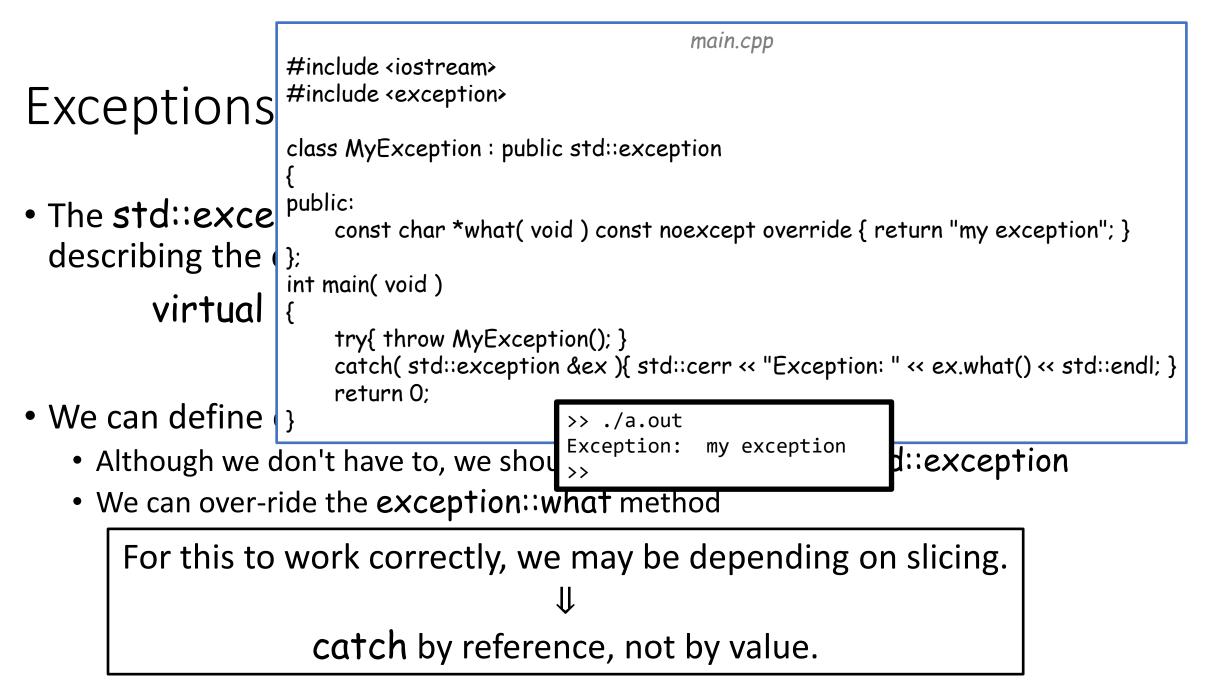


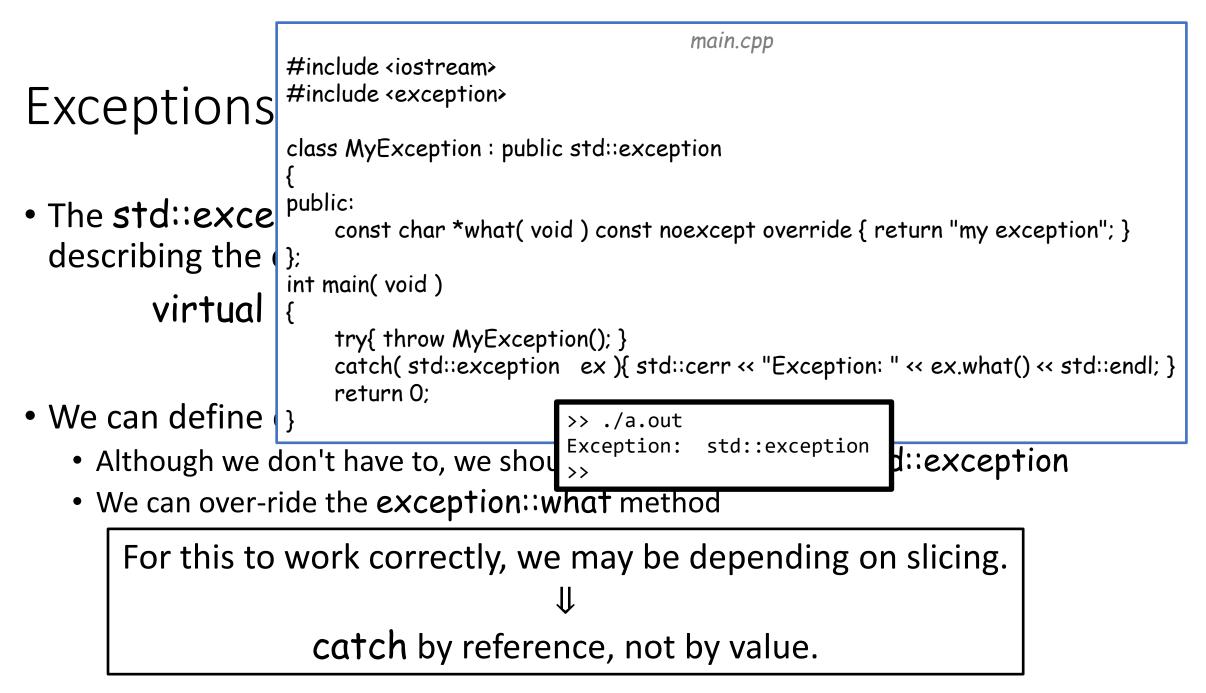
Exceptions

- The std::exception class has a virtual method that returns a C 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

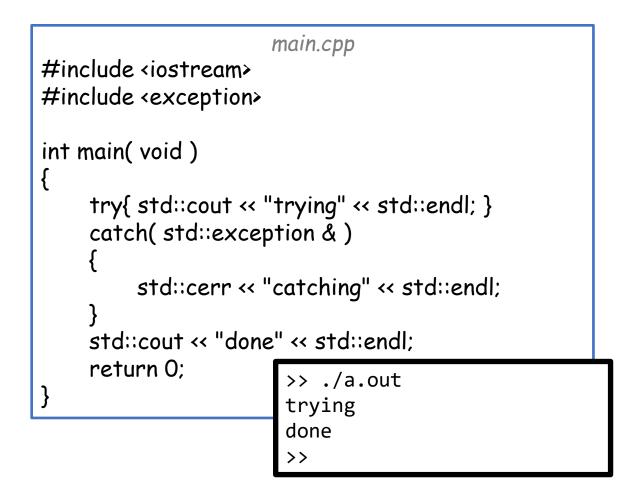
For this to work correctly, we may be depending on slicing. \downarrow **catch by reference, not by value!**

Exceptions	main.cpp #include <iostream> #include <exception></exception></iostream>
	<pre>int main(void) { try{ throw std::exception(); } catch(std::exception &ex){ std::cerr << "Exception: " << ex.what() << std::endl; } return 0; } don't have to, we shou ide the exception::what method</pre>
For this to	work correctly, we may be depending on slicing.
	\Downarrow
	catch by reference, not by value.





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



If an exception is thrown by code in a **try** block:

- Execution immediately jumps to the first matching catch block (if one exists)
- Code in that catch block is executed
- Then execution continues normally after the catch block(s)

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
3,
int main(void)
{
    try{ throw MyException(); }
    catch( MyException & )
         std::cerr << "caught mine" << std::endl;</pre>
    catch( std::exception & )
         std::cerr << "caught generic" << std::endl;</pre>
    std::cout << "done" << std::endl
    return 0;
                       >> ./a.out
                       caught mine
                       done
                       >>
```

If an exception is thrown by code in a **try** block:

- Execution immediately jumps to the first matching catch block (if one exists)
- Code in that catch block is executed
- Then execution continues normally after the catch block(s)

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
3,
int main(void)
{
    try{ throw MyException(); }
    catch( std::exception & )
         std::cerr << "caught generic" << std::endl;</pre>
    catch( MyException & )
         std::cerr << "caught mine" << std::endl;</pre>
    std::cout << "done" << std::endl
    return 0:
                       >> ./a.out
                       caught generic
                       done
                       >>
```

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

Note:

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

```
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;</pre>
    return 0;
                         >> ./a.out
                         caught generic
                         >>
```

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

#include <iostream> #include <exception></exception></iostream>	main.cpp
void foo(void){ throw	w std::exception();
foo();	foo" << std:: endl; t foo" << std::endl;

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

• 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;</pre>
          throw std::exception();
          std::cout << "b" << std::endl;</pre>
     catch( std::exception & ){ std::cout << "caught exception!" << std::endl; }</pre>
     return 0;
                        >> ./a.out
                         а
                         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; }</pre>
     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; }</pre>
     return 0:
                              >> ./a.out
                              got overflow exception: e.what()=ran out of space!
                              >>
```

34

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>
                                                        >> ./a.out
                                                        error while allocating: std::bad_alloc
int main(void)
                                                        >>
    try
         int *array = new int[1000000000];
         array[0] = 10;
    catch( std::bad_alloc &bae ){ std::cout << "error while allocating: " << bae.what() << std::endl; }
    return 0:
```

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

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

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

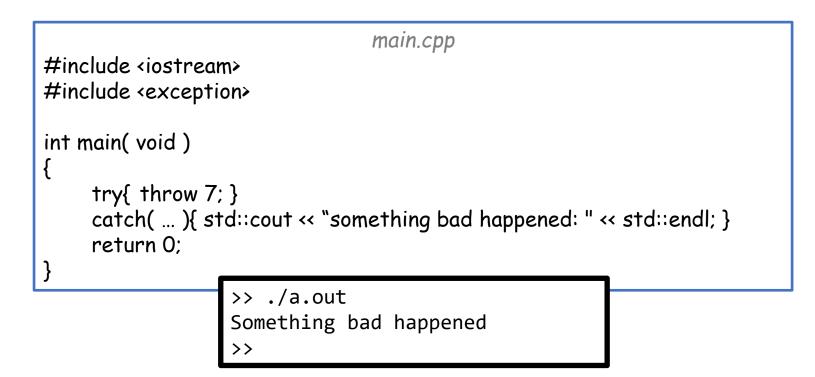
try/catch blocks

• 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



Outline

- Enumerated types
- Exceptions
- Review questions

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

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

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)

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

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

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