

Exception Handling

About

When we are writing structures that can be re-used across multiple programs, we will also want to add in error handling. But how do we handle errors that we detect? – That's where Exception Handling comes in!

Topics

1. Error codes

5. Throwing exceptions

2. What kinds of errors are there?

6. Listening for exceptions with try/catch

3. Listening & repsonding

7. Creating your own exception class

4. Exception safety

1. *Error codes*

I. *Error codes*

Have you ever been using a program, and when it crashed all it gave you was some number code that wasn't helpful at all in diagnosing the problem?



I. *Error codes*

Have you ever been using a program, and when it crashed all it gave you was some number code that wasn't helpful at all in diagnosing the problem?

Passing around error codes used to be how programmers would find the errors in a program – though not much use to the user, or some other programmer using a library that throws codes!



I. *Error codes*

This is why C++ programs “return 0” at the end – traditionally, if something went wrong, you would return some number *other than 0* to represent what error occurred.

If your program returns 0, then it means “everything is OK.”



I. *Error codes*

This isn't a great way to handle errors, though, and languages have since evolved to have more built-in error handling.

This is where **exception handling** comes in!

2. What kind of errors are there?

2. What kind of errors are there?

What kinds of errors do we need to look out for, especially when it comes to structures built to be used in multiple programs?

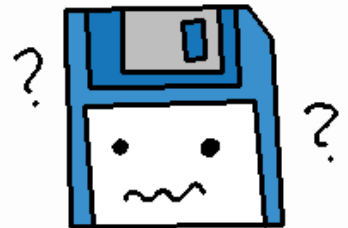
Think of times your programs have crashed during runtime – out-of-bounds access to an array, de-referencing a bad pointer, doing some bad math, and so on...

When we're writing a structure, we will want to make sure anyone using our structure won't have the ability to crash the program with it.

2. *What kind of errors are there?*

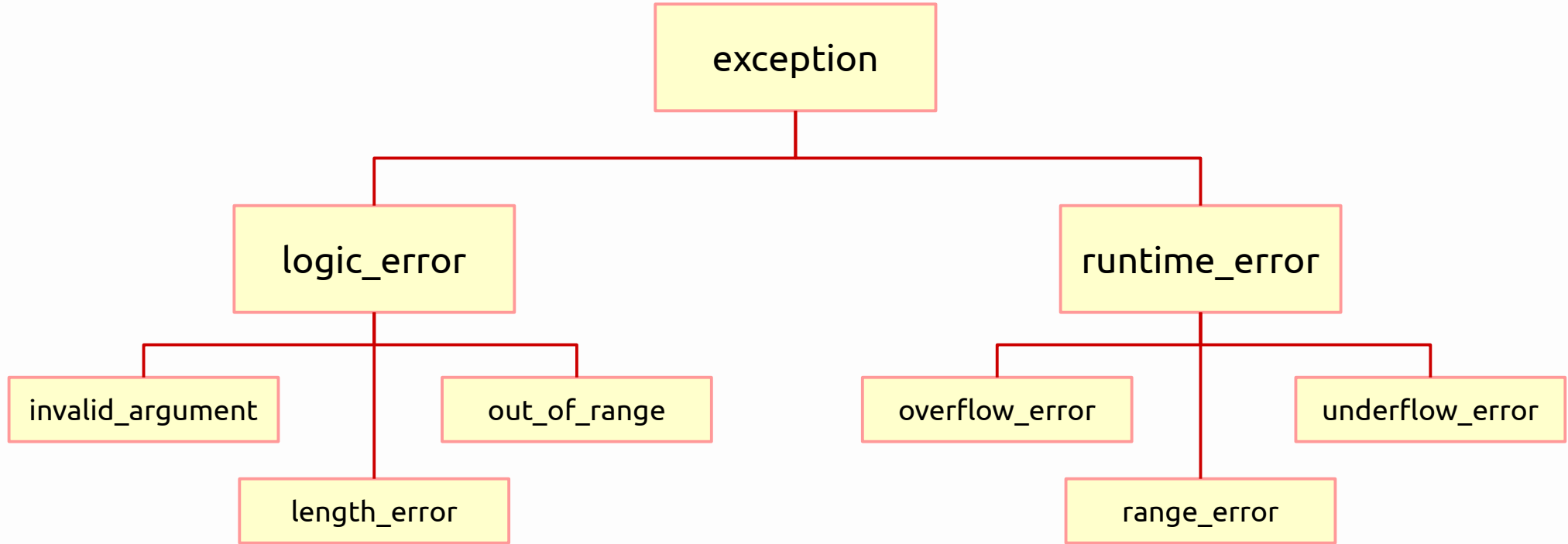
The C++ Standard Library has a family of exception types, so while you can have an error message passed as a **string** (human-readable), you can also pass a specific **type** of exception (program-readable).

Then, when the program using your structure detects the exception, it can look at what *type* of error happened, and respond appropriately.



2. What kind of errors are there?

The C++ Exception classes are:



You can view additional information about exceptions at the [cplusplus.com](http://www.cplusplus.com/doc/tutorial/exceptions/) page:
<http://www.cplusplus.com/doc/tutorial/exceptions/>

2. *What kind of errors are there?*

logic_error

“reports errors that are a consequence of faulty logic within the program such as violating logical preconditions or class invariants and may be preventable.”

From http://en.cppreference.com/w/cpp/error/logic_error

invalid_argument

“reports errors that arise because an argument value has not been accepted.”

From http://en.cppreference.com/w/cpp/error/invalid_argument

length_error

“reports errors that result from attempts to exceed implementation defined length limits for some object.”

From http://en.cppreference.com/w/cpp/error/length_error

out_of_range

“reports errors that are consequence of attempt to access elements out of defined range.”

From http://en.cppreference.com/w/cpp/error/out_of_range

2. *What kind of errors are there?*

runtime_error

“reports errors that are due to events beyond the scope of the program and can not be easily predicted.”

From http://en.cppreference.com/w/cpp/error/logic_error

overflow_error

“report arithmetic overflow errors (that is, situations where a result of a computation is too large for the destination type)”

From http://en.cppreference.com/w/cpp/error/logic_error

range_error

“report range errors (that is, situations where a result of a computation cannot be represented by the destination type)”

From http://en.cppreference.com/w/cpp/error/logic_error

underflow_error

“report arithmetic underflow errors (that is, situations where the result of a computation is a subnormal floating-point value)”

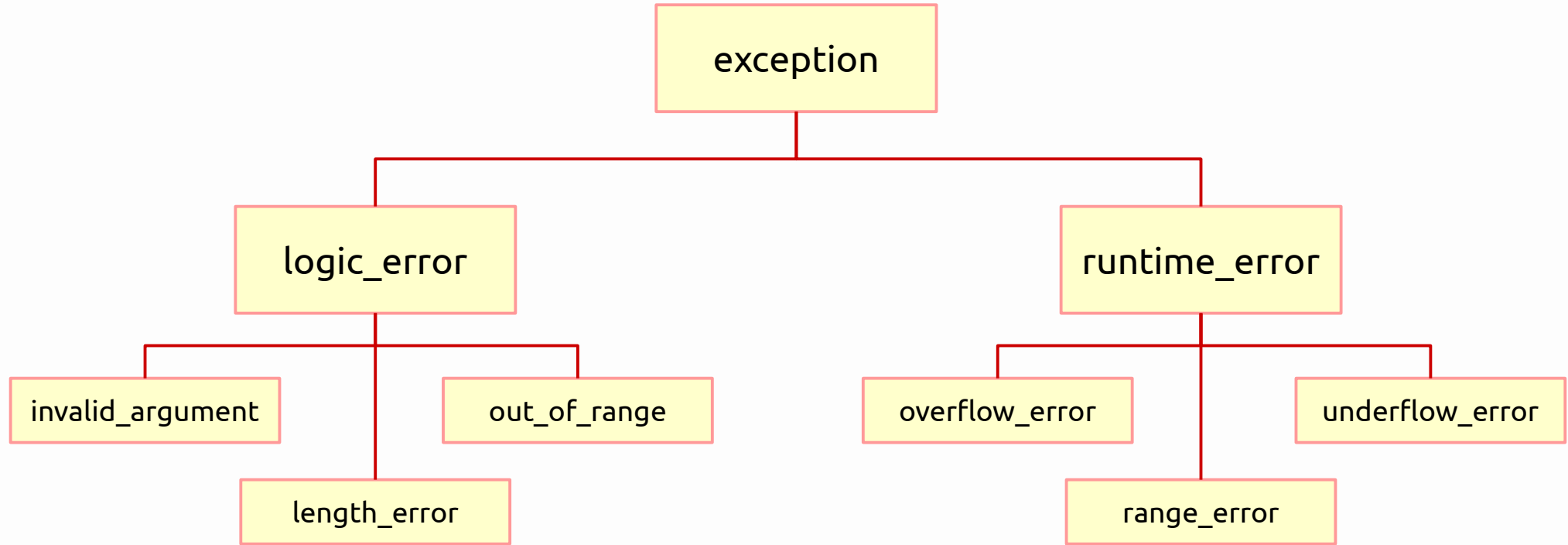
From http://en.cppreference.com/w/cpp/error/logic_error

2. What kind of errors are there?

There are additional types of exceptions, including exception types added in the 2011 and 2017 updates to C++. You can view a full list at:

<http://en.cppreference.com/w/cpp/error/exception>

2. What kind of errors are there?



You can also write a class that **inherits** from any of these, to make your own exception type.

3. Listening and responding

3. Listening and responding

```
class StudentList
{
    public:
        string GetStudent( int index );
        void SetStudent( int index, string name );
    private:
        string students[10];
};
```

For example, let's say we are writing a structure to store Students which contains an **array** to store data. The array has space for 10 elements, so indexes 0 through 9 are valid.

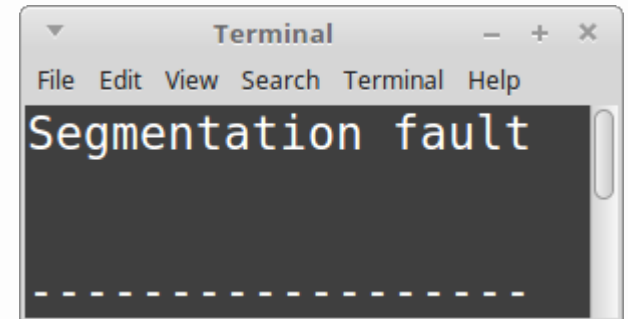
We also have functions to **Get** and **Set** student names, allowing the user to pass in an index.

3. Listening and responding

```
string StudentList::GetStudent( int index )  
{  
    return students[ index ];  
}  
  
void StudentList::SetStudent( int index, string name )  
{  
    students[ index ] = name;  
}
```

Without any error checking, something can call the **GetStudent** or **SetStudent** functions, passing in an invalid index (less than 0, or greater than 9).

When the program goes to try to access a student at an invalid index, the program would crash.



3. Listening and responding

```
string StudentList::GetStudent( int index )  
{  
    if ( index < 0 || index >= 10 )  
    {  
        // error!  
        return "";  
    }  
    return students[ index ];  
}
```

At the same time, a basic error check isn't always sufficient, such as this function that returns a value. If the user passes in a bad index, what do we return as the required **string**? Nothing? "Error"?

How do we communicate with the **caller** to let it know something went wrong?

3. Listening and responding

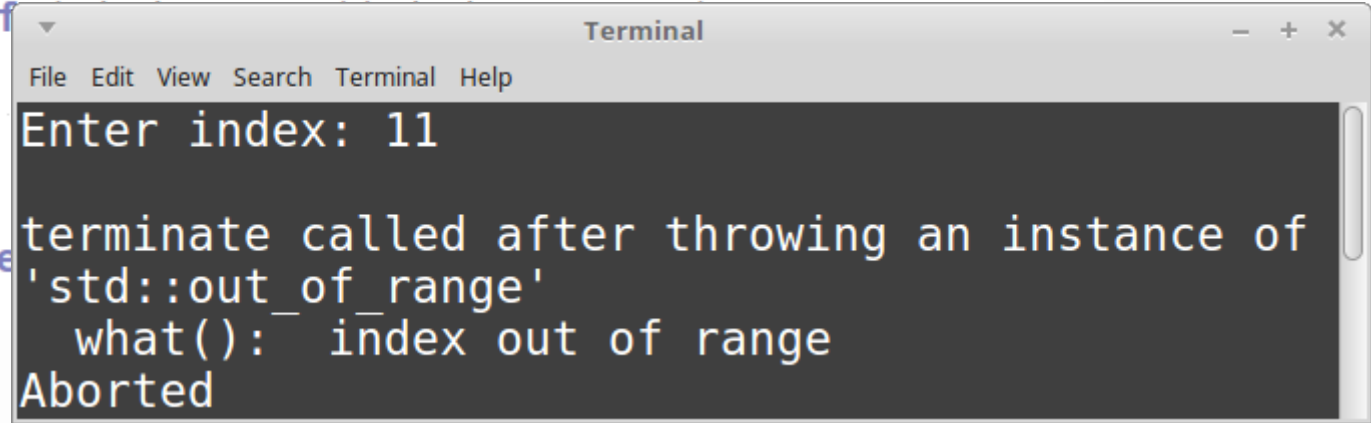
```
string StudentList::GetStudent( int index )
{
    if ( index < 0 || index >= 10 )
    {
        throw out_of_range( "index out of range" );
    }

    return students[ index ];
}
```

Instead of trying to return some default value and otherwise not doing very good error reporting, instead we can use the **throw** command and throw some *type* of exception – in this case, an **out_of_range** exception that's part of C++'s standard library – and we can include a string message with that exception to give more information on what went wrong, in a human-readable way.

3. Listening and responding

```
string StudentList::GetStudent( int index )  
{  
    if  
{  
}  
}  
re  
}
```

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the prompt "Enter index: 11" followed by a C++ exception message: "terminate called after throwing an instance of 'std::out_of_range'", "what(): index out of range", and "Aborted".

```
Terminal  
File Edit View Search Terminal Help  
Enter index: 11  
terminate called after throwing an instance of  
'std::out_of_range'  
what(): index out of range  
Aborted
```

If the exception is thrown and nothing in the program **catches** that exception, it will at least be reported and then the program will exit.

However, we can also write code to **catch** an exception, resolve it, and move on so the program continues working properly.

4. Exception safety

4. Exception safety

When errors are encountered in a piece of software, crashing and exiting isn't always *an option*.

If your video game crashes and exits in the middle of a competitive multiplayer, it is inconvenient but not life-threatening. A crash here is acceptable. However, if software is driving your car, or controlling the amount of radiation a cancer patient is receiving, errors need to be taken care of. It is best to deal with the exceptions, make sure the program is still in a **valid state**, and continue running after cleaning up.

When we're writing functions, we can specify some **exception guarantee** to let users of our work know how safe any given function is.

4. Exception safety

The levels of exception safety are:

- 1) **No-throw guarantee**, also known as **failure transparency**: Operations are guaranteed to succeed and satisfy all requirements even in exceptional situations. If an exception occurs, it will be handled internally and not observed by clients.
- 2) **Strong exception safety**, also known as **commit or rollback semantics**: Operations can fail, but failed operations are guaranteed to have no side effects, so all data retain their original values.
- 3) **Basic exception safety**, also known as a **no-leak guarantee**: Partial execution of failed operations can cause side effects, but all invariants are preserved and there are no resource leaks (including memory leaks). Any stored data will contain valid values, even if they differ from what they were before the exception.
- 4) **No exception safety**: No guarantees are made.

From https://en.wikipedia.org/wiki/Exception_safety

5. Throwing exceptions

5. Throwing exceptions

There are two parts of working with exceptions:

Inside (Inside a function that may have an error)

Detect an error, then **throw** an exception.

Outside (Calling the function)

Try to call the function,
and **catch** any exceptions that occur.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

When should you add an exception throw to your function?

When you're doing error checking!

You might not be in the practice of checking for errors (such as bad user input, bad pointer address, or other things), but we will have error checking in our data structures.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

Whether your code is being re-used across multiple programs, or you're working with multiple people in one program, or even working alone, you should get into the habit of checking for errors and handling them appropriately!

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

Some examples of errors to look out for are...

Validate function inputs

Are the parameters within a valid range?
What do you do if something's wrong?

Validate objects are initialized

With dynamic arrays, is it pointing to nullptr or is it initialized?
Has a file been opened already for reading, or is it closed?

Are we outside of bounds?

Is the index to be accessed within a valid range?

Is it OK to free the memory?

Don't use delete on nullptr

Are we about to divide by zero?

Check denominator values prior to doing a computation

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

To implement throwing exceptions, declare your functions as normal. If you're 100% sure that a function won't throw an exception (including via ***other functions*** being called within your function), you can mark it as **noexcept**.

```
class StudentList
{
    public:
    void SetStudent( int index, string name );

    string GetStudent( int index );

    void SafeFunction( int abc ) noexcept;

    private:
    string students[MAX_STUDENTS];
};
```

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

In the function definition, do some error checking, and if you detect an error, use the **throw** command.

To use the **exception** family of classes, you will need to `#include <stdexcept>`

```
void StudentList::SetStudent( int index, string name )
{
    if ( index < 0 || index >= MAX_STUDENTS )
    {
        throw out_of_range( "Invalid index" );
    }

    students[ index ] = name;
}
```

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

Your functions can also throw multiple different types of exceptions as well, based on what you are detecting.

```
void SomeFunction( int error )
{
    if ( error == 1 )
    {
        throw logic_error( "Logic error" );
    }
    else if ( error == 2 )
    {
        throw runtime_error( "Runtime error" );
    }
    else if ( error == 3 )
    {
        throw out_of_range( "Out of range error" );
    }
}
```

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

When creating a function that can throw exceptions, we might want to add a **specifier** to that function.

In C++03 (the version from 2003), you could specify what *kind* of exception would be thrown:

```
string GetStudent( int index ) throw(out_of_range);
```

This function **declaration** specifies that it may throw an `out_of_range` error.

However, the **throw() function specifier** has been deprecated as of C++11; this means that you *can* still use this, but it may be removed or replaced in future versions of C++.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

5. Throwing exceptions

In C++11, instead of specifying

what kind of exception this function may throw,

we instead specify

whether this function may throw an exception

with the **noexcept** specifier.

```
void SafeFunction( int abc ) noexcept;
```

This function **declaration** specifies that this won't throw an exception.

```
string GetStudent( int index );
```

The function that *may* throw an exception has no extra specifier.

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

noexcept specifier
shows that a function
won't throw any
exceptions.

6. Listening for exceptions with try/catch

6. Listening for exceptions with try/catch

It isn't enough to just **throw** an exception when you detect an error. When a function that might cause an exception is **called**, the **caller** needs to listen... if the exception is thrown, the external code needs to **catch** it.

Just like we can **throw** different types of errors, we can also **catch** the different types of errors – and even catch multiple types of exceptions coming from one or more functions.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

6. Listening for exceptions with try/catch

In order to detect exceptions, we must wrap a portion of code within a **try/catch** block.

We can have as many **catch** blocks as we need, one catch per exception type, but there will be just one **try** per **try/catch** block. You can still have multiple **try/catch** statements in your program, though.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

6. Listening for exceptions with try/catch

When we are working with a function that may throw an exception, we can wrap it within the **try/catch** block.

```
try
{
    SomeFunction( num );
}
catch( exception& e )
{
    cout << e.what() << endl;
}
```

***This function
may throw an
exception.***



If any exceptions are thrown within the **try {}** block, then the code within the **catch {}** for the corresponding exception type is executed.

The base **exception** class contains a **what()** function, that returns a c-string of the error message.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

6. Listening for exceptions with try/catch

If we don't have a **catch** for a specific exception that will be thrown by the function, the exception can be caught by that exception's parent or ancestor.

```
void PizzaTopping( string topping )
{
    if ( topping == "anchovies" )
    {
        throw invalid_argument( "Invalid pizza topping: " + topping );
    }
}
```

This function throws **invalid_argument** exceptions

```
try
{
    PizzaTopping( topping );
}
catch( logic_error& e )
{
    cout << "Logic error: " << e.what() << endl << endl;
}
```

But the caller only detects **logic_error** exceptions.

Notes

#include <stdexcept>
to use exceptions!

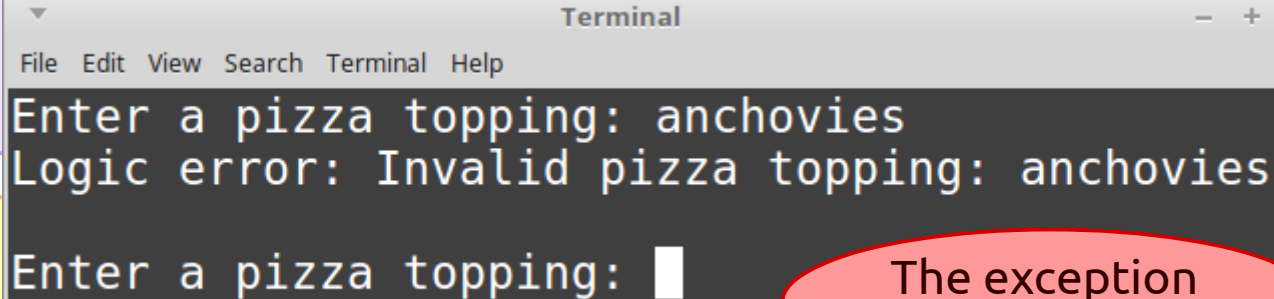
Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

6. Listening for exceptions with try/catch

If we don't have a **catch** for a specific exception that will be thrown by the function, the exception can be caught by that exception's parent or ancestor.

```
void PizzaTopping(string topping)
{
    if (topping == "anchovies")
    {
```



```
Terminal
File Edit View Search Terminal Help
Enter a pizza topping: anchovies
Logic error: Invalid pizza topping: anchovies
Enter a pizza topping: 
```

The exception was still caught!

```
    catch (logic_error& e)
    {
        cout << "Logic error: " << e.what() << endl << endl;
    }
```

But the caller only detects **logic_error** exceptions.

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

6. Listening for exceptions with try/catch

```
try
{
    SomeFunction( num );
}
catch( logic_error& e )
{
    cout << "LOGIC ERROR" << endl
          << e.what() << endl;
}
catch( runtime_error& e )
{
    cout << "RUNTIME ERROR" << endl
          << e.what() << endl;
}
catch( exception& e )
{
    cout << e.what() << endl;
}
```

Because exceptions are part of a family tree, you can use the parent **exception** as the default catch, if none of the other **catch** statements grab the exception.

However, make sure you put the more generic exception types after the more specific ones;

catch(exception& e)
should go on the bottom so it doesn't preempt the more specific exception handlers.

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

7. Creating your own exception class

7. Creating your own exception class

Using inheritance, you can also create your own exception class by inheriting from **exception**, or any of its children.

```
1 class exception {  
2 public:  
3     exception () noexcept;  
4     exception (const exception&) noexcept;  
5     exception& operator= (const exception&) noexcept;  
6     virtual ~exception();  
7     virtual const char* what() const noexcept;  
8 }
```

from <http://www.cplusplus.com/reference/exception/exception/>

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

7. Creating your own exception class

```
class SpecialException : public logic_error
{
public:
    SpecialException( const string& what_arg )
        : logic_error( what_arg )
    {
        // Do special things here
    }
};
```

Creating a custom exception, inheriting from **logic_error**

Notes

`#include <stdexcept>`
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

First, you create an exception class

7. Creating your own exception class

```
class SpecialException : public logic_error
```

```
{  
pub  
float RiskyFunction( int num, int denom )  
{  
    if ( denom == 0 )  
    {  
        throw SpecialException( "0 denominator not allowed" );  
    }  
    return float( num ) / float( denom );  
}
```

This function throws **invalid_argument** exceptions

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

Then you can throw it from a function

7. Creating your own exception class

```
class SpecialException : public logic_error
```

```
{
```

```
public: float RiskyFunction( int num, int denom )
```

```
{
```

```
    if (
```

```
    {
```

```
    }
```

```
};
```

Cre

```
return
```

```
}
```

```
try
```

```
{
```

```
    float result = RiskyFunction( num, denom );
```

```
    cout << "Result: " << result << endl;
```

```
}
```

```
catch (SpecialException ex)
```

```
{
```

```
    cout << "ERROR! " << ex.what() << endl;
```

```
}
```

Catching the special exception

Notes

#include <stdexcept>
to use exceptions!

Inside function:
throw exceptions

Outside function:
Try function,
Catch exceptions.

And catch the special exception in a try/catch.

Conclusion

Data Structures class is all about writing structures that store, maintain, and organize data. These aren't standalone classes that would just be used in a single project, but something to be used across multiple programs. Therefore, we need to make sure our structures are stable.

Additionally, you need to know how to listen for exceptions, as functions from other libraries (including the STL) will throw exceptions in some cases.