

Exception Handling

(A light introduction)

Today's Plan



Announcements

Motivation

Exceptions (light)

Something should really bother you about the List class...

What?

```
template<typename ItemType>
ItemType List<ItemType>::getItem(size_t position) const
{
    Node<T>* pos_ptr = getPointerTo(position);
    if(pos_ptr != nullptr)
        return pos_ptr->getItem();
    else
        //MUST RETURN SOMETHING!!!!
}
}
```

```
template<typename ItemType>
T List<T>::getItem(size_t position) const
{
    T dummy;
    Node<T>* pos_ptr = getPointerTo(position);
    if(pos_ptr != nullptr)
        return pos_ptr->getItem();
    else
        return dummy;
}
```

If there is no item at position, can we just return a dummy object?

```

template<typename ItemType>
T List<T>::getItem(size_t position) const
{
    T dummy;
    Node<T>* pos_ptr = getPointerTo(position);
    if(pos_ptr != nullptr)
        return pos_ptr->getItem();
    else
        return dummy; //problem/warning may return
                       // uninitialized object
}

```

The calling function has no way of knowing the returned object is uninitialized -> UNDEFINED BEHAVIOR



Fail-safe Programming

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const T& newEntry);`
- Use **sentinel value**: return error codes

Fail-safe Programming

What happens when preconditions are not met or input data is malformed?

???

- Do nothing
- Return false - `bool add(const T& newEntry);`
- Use **sentinel value**: return error codes

Fail-safe Programming

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const T& newEntry);`
- Use **sentinel value**: return error codes

Rely on user to handle problem

Rely on user to handle problem

Sometimes it is not possible to return an error code

Fail-safe Programming

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const T& newEntry);`
- Use **sentinel value**: return error codes

What happens if there is no item at position when calling `getItem(size_t position)?`

assert

```
#include <cassert>
```

```
// ...
```

```
assert(getPointerTo(position) != nullptr);
```



Make sure this is true

If assertion is false, program execution terminates

assert

```
#include <cassert>
```

Make sure this is true

```
// ...
```

```
assert(getPointerTo(position) != nullptr);
```

If assertion is false, program execution terminates

Good for testing and debugging

So drastic! Give me
another chance!



Exceptions: A Light Introduction

Exceptions



Software: calling function

Client might be able to recover from a violation or unexpected condition

Communicate **Exception** (error) to client:

- Bypass normal execution
- Return control to client
- Communicate error

Exceptions

Client might be able to recover from a violation or unexpected condition

Communicate **Exception** (error) to client:

- Bypass normal execution
- Return control to client
- Communicate error



Throw and Exception

Throwing Exceptions

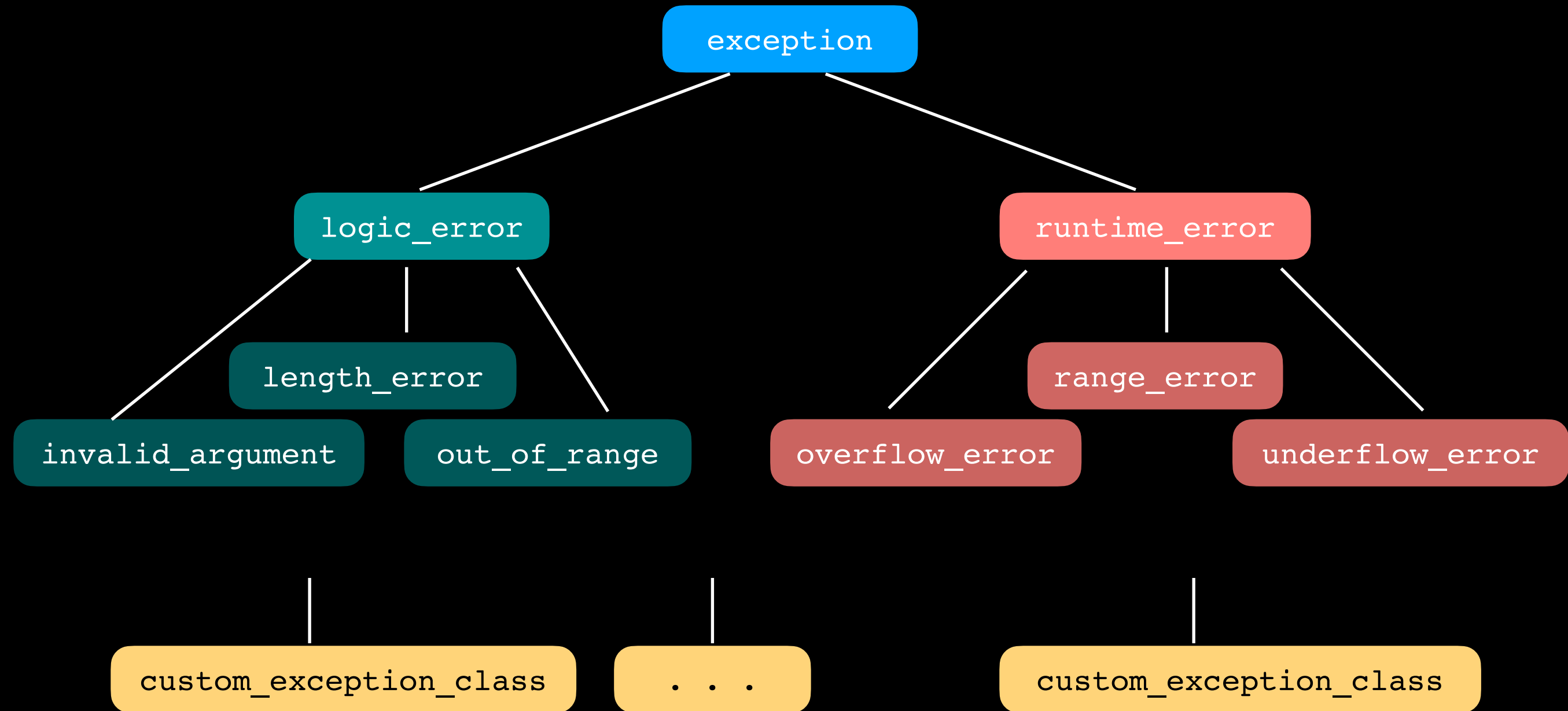
Type of Exception

throw(*ExceptionClass*(*stringArgument*))

Message describing
Exception

```
template<typename ItemType>
T List<T>::getItem(size_t position) const
{
    Node<T>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw (std::out_of_range("getItem called with empty list
                                or invalid position"));
    else
        return pos_ptr->getItem();
}
```


C++ Exception Classes



C++ Exception Classes

Control returned to
calling function

exception

Program Terminates

logic_error

runtime_error

length_error

range_error

invalid_argument

out_of_range

overflow_error

underflow_error

user_defined_class

...

user_defined_class

e.g. `PreconditionViolatedExcep`
thrown by `List`

| Exception Type | | Header File |
|----------------|-------------------|-------------|
| exception | | <exception> |
| | bad_alloc | <new> |
| | bad_cast | <typeinfo> |
| | bad_exception | <exception> |
| | bad_typeid | <typeinfo> |
| | ios_base::failure | <ios> |
| | logic_error | <stdexcept> |
| | length_error | <stdexcept> |
| | domain_error | <stdexcept> |
| | out_of_range | <stdexcept> |
| | invalid_argument | <stdexcept> |
| | runtime_error | <stdexcept> |
| | overflow_error | <stdexcept> |
| | range_error | <stdexcept> |
| | underflow_error | <stdexcept> |

Exception Handling



Can handle only exceptions of class `logic_error` and its derived classes

Exception Handling Syntax

```
try
{
    //statement(s) that might throw exception
}
catch(ExceptionClass1 identifier)
{
    //statement(s) that react to an exception
    // of type ExceptionClass1
}
```

Exception Handling Syntax

```
try
{
    //statement(s) that might throw exception
}
catch(ExceptionClass1 identifier)
{
    //statement(s) that react to an exception
    // of type ExceptionClass1
}
catch(ExceptionClass2 identifier)
{
    //statement(s) that react to an exception
    // of type ExceptionClass2
}
. . .
```

Exception Handling Syntax

Arrange catch blocks in order of specificity,
catching most specific first
(i.e. lower in the Exception Class Hierarchy first)

```
try
{
    //statement(s) that might throw exception
}
catch(const ExceptionClass1& identifier)
{
    //statement(s) that react to an exception
    // of type ExceptionClass1
}
catch(const ExceptionClass2& identifier)
{
    //statement(s) that react to an exception
    // of type ExceptionClass2
}
. . .
```

Good practice to catch exceptions by const reference whenever possible
(due to memory management, avoiding copying and slicing issues)

Exception Handling Usage

You know getItem() may throw an exception so call it in a try block

```
try
{
    some_object = my_list.getItem(n);
}
catch(const std::out_of_range& problem)
{
    //do something else instead
    bool object_not_found = true;
}
```



```

template<typename ItemType> ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw(std::out_of_range("getItem called with empty list or invalid position"));
    else
        return pos_ptr->getItem();
}

```

```

try
{
    some_object = my_list.getItem(n);
}
catch(const std::out_of_range& problem)
{
    std::cerr << problem.what() << std::endl;
    //do something else instead
    bool object_not_found = true;
}

```

Returns string
parameter to
thrown exception

Error Output Stream:

getItem called with empty list or invalid position

Uncaught Exceptions

```
template<typename ItemType> ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw(std::out_of_range("getItem called with empty list or invalid position"));
    else
        return pos_ptr->getItem();
}
```

out_of_range exception
thrown here

```
ItemType someFunction(const List<ItemType>& some_list) {
    ItemType an_item;
    //code here
    an_item = some_list.getItem(n);
}
```

out_of_range exception
not handled here

```
int main() {
    List<string> my_list;
    try {
        std::string some_string = someFunction(my_list);
    }
    catch(const std::out_of_range& problem) {
        //code to handle exception here
    }
    //more code here
    return 0;
}
```

out_of_range exception
handled here

Uncaught Exceptions

```
template<typename ItemType> ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw(std::out_of_range("getItem called with empty list or invalid position"));
    else
        return pos_ptr->getItem();
}
```

out_of_range exception
thrown here

```
ItemType someFunction(const List<ItemType>& some_list) {
    ItemType an_item;
    //code here
    an_item = some_list.getItem(n);
}
```

out_of_range exception
not handled here

```
int main() {
    List<string> my_list;
    std::string some_string = someFunction(my_list);
    //code here
    return 0;
}
```

out_of_range exception
not handled here

Abnormal program
termination

Implications

There could be several
... out of the scope of this course

We will discuss one:

What happens when program that dynamically allocated memory relinquishes control in the middle of execution because of an exception?

Implications and Complications

There could be many
... out of the scope of this course

We will discuss one:

What happens when program that dynamically
allocated memory relinquishes control in the mid-
of execution because of an exception?

Dynamically allocated memory never released!!!



Implications and Complications

Whenever using **dynamic memory allocation** and **exception handling** together must consider ways to **prevent memory leaks**

Uncaught Exceptions

Memory Leak

```
template<typename ItemType> ItemType List<ItemType>::getItem(size_t position) const
{
    Node<ItemType>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw(std::out_of_range("getItem called with empty list or invalid position"));
    else
        return pos_ptr->getItem();
}
```

out_of_range exception
thrown here

```
ItemType someFunction(const List<ItemType>& some_list) {
    //code here that dynamically allocates memory
    ItemType an_item;
    //code here
    an_item = some_list.getItem(n);
}
```

out_of_range exception
not handled here

```
int main() {
    List<string> my_list;
    try {
        std::string some_string = someFunction(my_list);
    }
    catch(const std::out_of_range& problem) {
        //code to handle exception here
    }
    //more code here
    return 0;
}
```

out_of_range exception
handled here

Possible solution coming soon