Overview

- Exceptions
    - Syntax
    - Exceptions in a function/method
    - Base Class exception
    - User Defined Exception Classes

- File Access
    - Write Data
    - Read Data

Exceptions

- Errors may occur within a program at any time in any situation
- Handling an error within a function/method is often not appropriate
- Caller of a function/method may handle the error
- Exceptions automate error propagation
    - As long as an error is not handled it is propagated to the previous caller on the stack
    - Unhandled errors abort the program

Exceptions

Typical Exceptions

- Division by 0
- Array overflow
- File I/O errors
- Network connection errors
- NULL pointer exceptions

New languages have built-in exception handling

- Java, C#
- C++ has no built-in exception handling, doesn't throw exception automatically

Exceptions

- Exceptions may have any data type
- Exceptions are not checked by the compiler

```
try{
 // Encapsulate critical code that may cause an exception in a try
block,
}catch(type name){
 // Catch an exception of a given type.
}
```

You can throw an exception manually

```
throw exceptionObject;
```

Exceptions Example

```
int divide (int a, int b){
 if (b == 0) {
  throw 0;
 }
 return a/b;
}

int main (){
 try {
  cout << divide(10, 0);
 } catch (int exception) {
  cout << "Division by Zero.";
 } catch (...) { // all other exceptions
  cout << "An error occurred.";
 }
}
```

Exceptions

- Functions can be restricted
- Defines what types of exceptions are thrown
- Keyword: `throw`

```
int divide (int a, int b) throw (int){
 if (b == 0) {
  throw 0; // only throwing int allowed
 }
 return a/b;
}

int divide (int a, int b) throw (type){} // throws an exception of
type
int divide (int a, int b) throw (){} // doesn't throw any exception
```

Base class `exception`

C++ standard library provides a base class `exceptions`

- defined in the `exception` header file
- under the namespace `std`

It has a

- default and copy constructors, operators and destructors,
- virtual member function `what()` which can be overridden (returns a null-terminated character sequence)

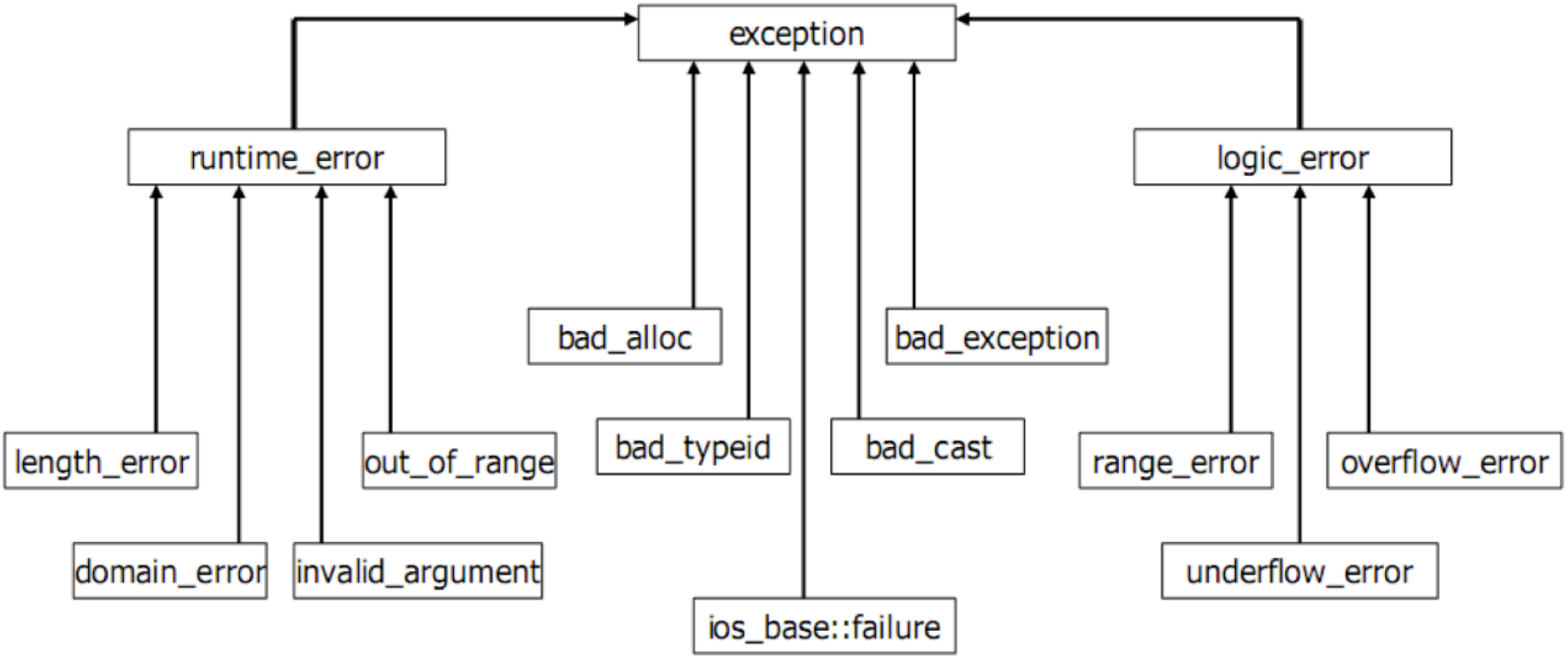
Base class `exception`

```
class exception {
public:
 exception() throw();
 exception(const exception&) throw();
 exception& operator=(const exception&) throw();
 virtual ~exception() throw();
 virtual const char* what() const throw();
private:
 ...
};
```

Class hierarchy

They are defined in the `stdexcept` library

```
#include <stdexcept>
using namespace std;
```


exception
runtime_error
logic_error
bad_alloc
bad_exception
length_error
out_of_range
bad_typeid
bad_cast
range_error
overflow_error
domain_error
invalid_argument
underflow_error
ios_base::failure

Exceptions Example

Check if memory can be allocated

- Exception of type `bad_alloc` is thrown

```
try{
 int* array = new int[100];
}catch(bad_alloc&){
 cout << "Error allocating memory" << endl;
}
```

User Defined Exception Class

```
class MyException : public exception{
 int _errorCode;
public:
 MyException() : exception(){
  _errorCode = 0;
 }

 MyException(int errorCode) : exception(){
  _errorCode = errorCode;
 }

 const char* what() throw(){
  stringstream sstream;
  sstream << "exception: " << m_errorCode;
  string message = sstream.str();

  return message.c_str();
 }
}
```

Exceptions

- Exceptions need not to be catched
- Exceptions can be explicitly forwarded to the caller
- Exception objects may contain data (see previous slide)
- Exceptions may be organized in class hierarchies
- Missing compared to Java
    - finally block
    - Checked exceptions (throws statement)
    - Built-in exceptions (e.g. for arrays or pointers)

Exceptions Example - `SaveArray`

```
class SaveArray{
 int _maxSize;
 int* _array;
public:
 SaveArray(int size);
 int& operator[](int index);
};

class RangeException : public exception {
 int _index;
public:
 RangeException(int index);
 const char* what() throw();
};

class SizeException : public exception {
 int _size;
public:
 SizeException(int size);
 const char* what() throw();
};
```

Exceptions Example - `SaveArray`

```
RangeException::RangeException(int index) throw(){
 _index = index;
}

const char* RangeException::what() const throw(){
 stringstream sstream;
 sstream << "range exception for index " << _index;
 string message = sstream.str();

 return message.c_str();
}

SizeException::SizeException(int size) throw(){
 _size = size;
}

const char* SizeException::what() const throw(){
 stringstream sstream;
 sstream << "size exception for size " << _size;
 string message = sstream.str();

 return message.c_str();
```

Exceptions Example - `SaveArray`

Implementation of `SaveArray`

```
SaveArray::SaveArray(int size){
 if(size < 0){
  throw SizeException(size);
 }
}

int& SaveArray::operator[](int index){
 if(index < 0 || index >= _maxSize){
  throw RangeException(index);
 }
 return _array[index];
}
```

Exceptions Example - `SaveArray`

Exception when accessing array element

```
try{
 SaveArray array(10);
 array[11] = 0;
}catch(RangeException& e){
 cout << e.what() << endl; // range exception for index 11
}
```

Exception when creating array

```
try{
 SaveArray array(-1);
}catch(SizeException& e){
 throw e; // Explicitly forward to caller
}
```

File Access

Classes to perform output and input to/from files

- `ofstream` : stream class to write to files
- `ifstream` : stream class to read from files
- `fstream` : stream class to both read and write from/to files

Classes are derived from `istream` and `ostream`

- Use insertion operator `<<` to write data to a text file
- Use extraction operator `>>` to read data from a text file

File Access

Steps for reading from a file

1. Open file

```
fstream file;
file.open(filename, mode)
```

2. Read/Write to file
   If you can read or write to a file depends on the opening mode (see next slide)
3. Close file
   Close a file to make resource available

```
file.close();
```

File Access - File Open

Open the file `my_text_file.txt`

```
fstream file;
file.open("my_text_file.txt", ios::read); // open the file for read
access
fail(file);
```

Methods to check state of the file stream

- `is_open()` : returns true when the file is open
- `bad()` : true if reading or writing operation failed
- `fail()` : true for the same cases as `bad()` but also for format errors (get integer when trying to read a character)
- `eof()` : returns true when we reached the end of a file
- `good()` returns true when `bad(), fail(), eof()` returns false

File Access - File Open Modes

Mode	Description
`ios::in`	Open for input operations (read)
`ios::out`	Open for output operation (write)
`ios::binary`	Open in binary mode
`ios::ate`	Initial position is at the file end, If this flag is not provided, initial position is the beginning at default.
`ios::app`	Appending data to the end of the file, can only be used for output-only operations.
`ios::trunc`	If file exists and opened for output operations, it gets replaced by a new one.

File Access - File Open Modes

Mode flags can be combined using the bitwise operator OR

```
fstream file;
file.open("my_text_file.txt", ios::binary | ios::in);
```

Use method `fail()` to check if operation was successful

```
if(file.fail()){
 cout << "Couldn't open file" << endl;
}
```

File Access - File Open Modes

ofstream, ifstream and fstream have a default mode when calling `open()` without second argument

Class	Default mode parameter
`ofstream`	`ios::out`
`ifstream`	`ios::in`
`fstream`	`ios::in | ios::out`

File Access - Write Data

Write binary data in general

```
ostream& ostream::write(const char* s, streamsize n);
```

- `s` : pointer to the data which should be written
- `n` : size (in characters = bytes) of the data to write

```
int arr[4] = { 1, 2, 3, 4 };
double amount = 12.34;

ofstream file;
file.open("medialibrary1.dat", ios::binary);
file.write((char*)arr, sizeof(int) * 4);
file.write((char*) &amount, sizeof(double));
file.close();
```

File Access - Read Data

Read binary data in general

```
istream& istream::read(const char* s, streamsize n);
```

- `s` : pointer to the data where the content read will be stored
- `n` : size (in characters = bytes) of the data to be read

```
int arr[4];
double amount;

ifstream file;
file.open("medialibrary1.dat", ios::binary);
file.read((char*)arr, sizeof(int) * 4);
file.read((char*) &amount, sizeof(double));
file.close();
```

The `const` Mysterium

The keyword `const` can be used mostly everywhere

```
static const Rectangle* const createNew(const Rectangle* const rect){
 rect->x = 0; // Error: const Rectangle
 rect = &Rectangle(); // Error: Rectangle* const
 ...
}

int main(){
 Rectangle rect1;
 const Rectangle* const rect2 = Rectangle::createNew(&rect1);
 rect2->x = 0; // Error: const Rectangle

 Rectangle rect3;
 rect2 = &rect3; // Error: Rectangle* const
}
```

Read from left-to-right