Introduction to the C++ Programming Language

Day 5

Aleksandra Rylund Glesaaen
aleksandra@glesaaen.com

October 2nd 2015

What will we learn?

- ~~Basic C++ syntax~~
- ~~Control structures~~
- ~~Functions~~
- ~~Structs and classes~~
- Templates and STL (Thursday and today)
- Exceptions (today)

Today's topics

1. Standard Template Library
2. Exceptions
3. Where to go from here
4. Programming Practices
5. Recap

Standard Template Library

What is the STL?

A compilation of template classes and functions with a consistent interface design, it contains

- Container classes
- Iterators
- Generic algorithms
- Smart pointers {C++11}
- Random number generation {C++11}
- ...and more

Motivation

Using the STL is fun, but using it effectively is outrageous fun, the kind of fun where they have to drag you away from the keyboard, because you just can't believe the good time you're having.

Scott Meyers

Containers

We have seen two ways of storing larger chunks of data in C++ so far

- arrays
- linked lists

The STL contain these two as well as many more containers you can use for all your data storage needs

Standard Template Library

All the storage containers are templates

- `std::vector<Penguin>`
- `std::list<Song>`
- `std::stack<Card>`
- `std::array<Hedgehog,10>`
- `std::map<Coordinate,Treasure>`

Sequential containers

Containers where the data is stored one after another

- `std::vector`
- `std::array` {C++11}

Can (and should) replace C arrays

- `std::list`
- `std::stack`

First in, first out

- `std::queue`

vector and array

Elements are also stored sequential in memory

Can access the elements in the arrays through:

- The access operator `[ ]`
- The `at()` function ← Includes bounds check

The vector class has almost no overhead and is always preferred to dynamic C arrays

vector

Container that does sequential dynamic arrays

Notable functions

(constructor)(size_t)	sets the initial size
operator[](size_t)	access nth element
at(size_t)	access nth element w/ bounds check
resize(size_t)	resize the vector
front(), back()	access first/last element
size()	get current size of vector

#include<vector>

Linked lists

Elements are not sequential in memory

No direct access of individual elements, we need to navigate through the list structure

Deleting and inserting are both really cheap

Initialiser lists

A convenient way of initialising containers is by listing their initial content

```
std::vector<int> lucky_numbers {12, 5, 42};
std::list<char> the_word {'b', 'i', 'r', 'd'};
```

Can create similar constructors for our own classes by using the std::initializer_list container

#include<initializer_list> {C++11}

C++11 constructors

Note

Using C++11 constructor notation {} will pick out initialiser list constructors first

```
std::vector zero_vector {0};
std::vector zero_vector (0);
```

Vector of length 1, with element 0

{C++11}

C++11 constructors

Note

Using C++11 constructor notation {} will pick out initialiser list constructors first

```
std::vector zero_vector {0};
std::vector zero_vector (0);
```

Vector of length 0

{C++11}

Associative containers

The elements are sorted, and searches are very quick

- `std::set`
  collection of unique elements, sorted
- `std::map`
  collection of key-value pairs, keys unique and sorted

Complexity

	Access	Search	Insert
Array	O(1)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)
List	O(n)	O(n)	O(1)
Map	-	O(1)	O(1)
Binary tree	O(log(n))	O(log(n))	O(log(n))

*best case

What container to choose

99% of the time you will use vector or array

What container to choose

Otherwise make a careful decision based on what you need the container for

- Are you going to insert elements in the middle of the container?
- What are your iterator needs?
- Do you need a fixed ordering?
- Is memory consistency important?

Iterators

For many of the containers in STL, direct access to an element is not possible, we somehow have to traverse the container structure, iterate if you will

Iterators

An iterator points to an element of a container

```
*iterator
```

and it can move to the next element of the container

```
++iterator
```

Iterator example

```
#include<set>
#include<cctype>

int main()
{
  std::set<char> lower_set {'a', 'q', 'k', 'p'};
  std::set<char> upper_set;

  for (auto it  = lower_set.begin();
            it != lower_set.end();   ++it)
    upper_set.insert(std::toupper(*it));
}
```

Iterator interface

Most of the containers in STL have iterators, and their interface is uniform

```
container.begin();
container.end();

++iterator;
 *iterator;
```

First element of the container

One past the final element

Iterator interface

Most of the containers in STL have iterators, and their interface is uniform

```
container.begin();
container.end();
```

```
++iterator;    ← Move to the next element
 *iterator;    ← Value of current element
```

Range based for loops

A cleaner way of iterating through containers

```
std::vector<Employee> employees;

//...

for (auto & worker : employees) {
  worker.work();
}
```

But it is only syntactic sugar

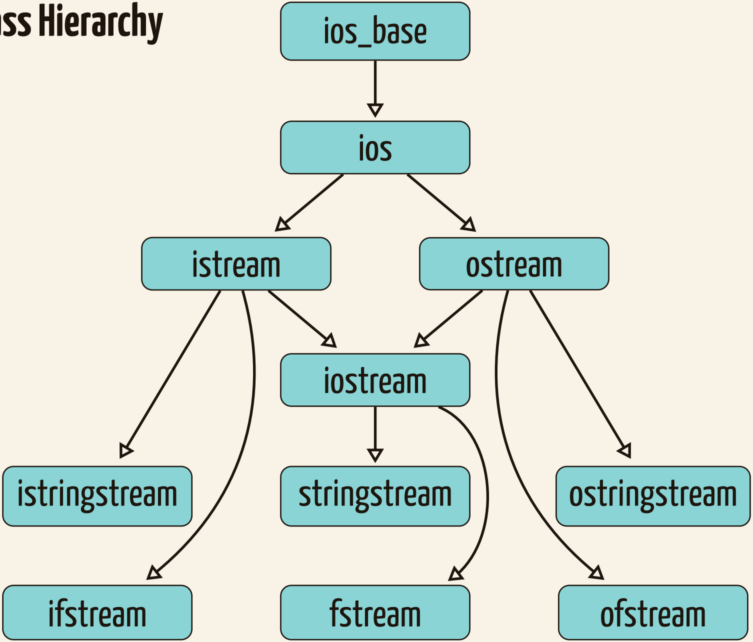
{C++11}

Streams

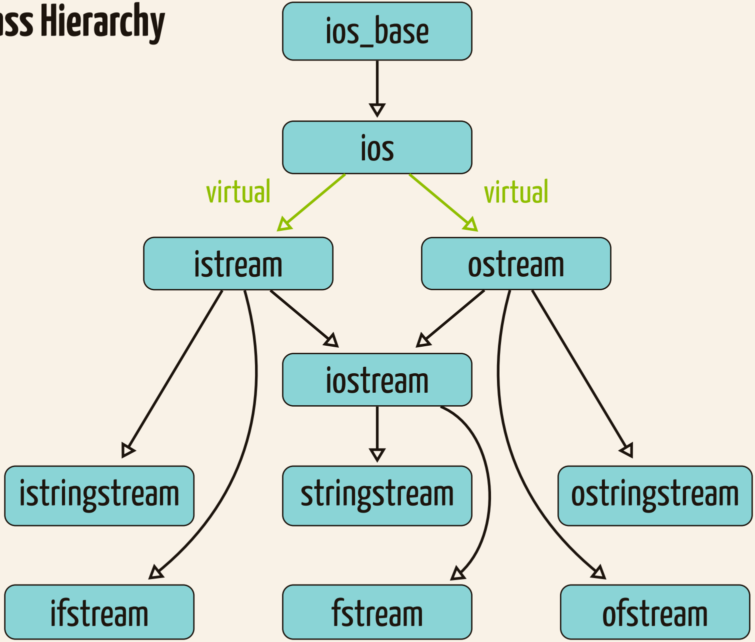
Streams are standardised input/output objects in C++

It is possible to create your own input- or output stream objects that have other sinks and sources than the two we have used so far

I/O Class Hierarchy



I/O Class Hierarchy



ifstream, ofstream, fstream

Stream object for files

Notable functions

(constructor)(std::string)	opens the file with the given filename
operator<<(...)	writes to the file
operator>>(...)	reads from the file
open(std::string)	opens the file with the given filename
close( )	closes the file buffer
is_open( )	checks if the file was opened correctly

#include<fstream>

File streams - example

```
std::ofstream to_file_stream {"file.txt"};

if (!to_file_stream) {
  std::cerr << "Could not open file";
  return 1;
}

to_file_stream << "Hello world" << std::endl;
to_file_stream.close();
```

istringstream, ostringstream, stringstream

Stream object for the std::string class

Notable functions

(constructor)(std::string)	set initial value of the string object
operator<<(...)	writes to the string
operator>>(...)	reads from the string
str( )	access the underlying string object
str(std::string)	change value of the string object

#include<sstream>

Stream example

```cpp
void sayHello(std::ostream & os)
{
  os << "Hello!" << std::endl;
}

int main()
{
  std::ofstream ofs {"file.txt"};
  sayHello(ofs);
  ofs.close();

  std::ostringstream oss;
  sayHello(oss);

  auto hello_string = oss.str();
}
```

Function objects

Objects with an overloaded call operator ()

```
struct Greater
{
  bool operator() (const double a, const double b)
  {
    return a > b;
  }
};

auto greater = Greater {};
```

Function objects

Objects with an overloaded call operator ()

```
auto greater = [](const double a, const double b)
{
  return a > b;
};
```

The functional library

The STL has a uniform interface for these

```
template <class R, class... Args>
class function<R(Args...)> {...};
```

Which can bind to anything with the correct call operator

The functional library - example

```cpp
void printInt(int i)
{
  std::cout << i;
}

struct IntPrint
{
  void operator() (int i)
  {
    std::cout << i;
  }
};

int main()
{
  std::function<void(int)> f1 = printInt;
  std::function<void(int)> f2 = IntPrint {};
  std::function<void(int)> f3 = [](int i){printInt(i);};
}
```

Argument capture

Capture a by value

```
[a,&b] (/* args */) {/* body */};
```

Capture b by reference

{C++11}

Argument capture

```
[&] (/* args */) {/* body */};
```

Capture everything by reference

{C++11}

Argument capture

Capture everything by value

```
[=] (/* args */) {/* body */};
```

{C++11}

Closures in C++

A closure is the concept of storing values inside of functions

We can use the lambda function capture for that

Closures in C++

```
std::function<std::string(std::string)>
Surround(std::string surr)
{
  return [surr](std::string expr)
  {
    return surr[0] + expr + surr[1];
  };
}

int main()
{
  auto square_brackets = Surround("[]");
  auto quotation_marks = Surround("\"\"");

  std::cout << square_brackets("Hello") << std::endl;
  std::cout << quotation_marks("Hello") << std::endl;
}
```

{C++11}

Closures in C++

```
std::function<std::string(std::string)>
Surround(std::string surr)
{
  return [surr](std::string expr)
  {
    return surr[0] + expr + surr[1];
  };
}

int main()
{
  auto square_brackets = Surround("[]");
  auto quotation_marks = Surround("\"\"");

  std::cout << square_brackets("Hello") << std::endl;
  std::cout << quotation_marks("Hello") << std::endl;
}
```

Prints [Hello]

Prints "Hello"

{C++11}

Algorithms

There is a large number of commonly used algorithms in the STL

Uniform interface that go well with the iterators

#include<algorithm>

Algorithms

```
template <class Itt, class T>
Itt find(Itt begin, Itt end, const T& value);
```

Find the first element equal to value in a container, returns an iterator pointing to the element, or end if not found

#include<algorithm>

Algorithms

```
template <class Itt, class Unary>
Unary for_each(Itt begin, Itt end, Unary f);
```

Apply the function f to every element in the container returns the final state of the function object f

Algorithms

```
template <class Itt>
void sort(Itt begin, Itt end);
```

Sorts the range specified by begin and end

#include<algorithm>

Algorithms

```
template <class Itt, class Compare>
void sort(Itt begin, Itt end, Compare compare);
```

Sorts the range specified by begin and end using the supplied comparison operator

Less than operator is king

STL uses the less than operator for all comparisons

Equality:

```
!(a < b) and !(b < a)
```

Inequality:

```
(a < b) or (b < a)
```

It is important to implement a proper less than

Capture in algorithms

```
std::vector<double> earnings {};

// ...

double total {0.};

std::for_each(earnings.begin(), earnings.end(),
  [&total](auto val){ total += val; });
```

{C++11}

Smart pointers

We discussed the dangers of dynamic memory management using raw pointers on day two

The smart pointer library in STL is here to rescue us

Smart pointers act as if they were pointers, but provide additional functionality

#include<memory> {C++11}

The big question

```
SmartPointer<Type> p {};
auto q = p;
```

What happens here?

std::unique_ptr

The object completely own the resource

Copying is disallowed, can only move

```
std::unique_ptr<Type> p {};
auto q = p;   ← Compile error
```

#include<memory> {C++11}

std::unique_ptr

The object completely own the resource

Copying is disallowed, can only move

```
std::unique_ptr<Type> p {};
auto q = std::move(p);  ←— OK
```

#include<memory> {C++11}

std::shared_ptr

Keeps a count of all the references to the resource

Only deletes the resource when all hooks are gone

```
std::shared_ptr<Type> p {};
auto q = p;
```

.use_count() = 1
.use_count() = 2

#include<memory> {C++11}

Guideline

Don't use explicit new, delete and owning * pointers, except in rare cases encapsulated inside the implementation of low-level data structures.

Herb Sutter

Other libraries

- Random number generation
- Duration
- Regular expressions
- Thread support
- Atomic operations

Exceptions

Motivating example

```
template <class Type>
class Vector
{
public:
  Type& operator[](const std::size_t index)
  {
    if (index >= size) {



    }

    // ...
  }
};
```

What do you do here?

The old style

C programmers mostly use error flags for this

```
template <class Type>
int accessVector(
  Type& res, const Vector<Type> &v, std::size_t i)
{
  if (i >= v.size()) {
    return 1;
  }

  res = v[i];
  return 0;
}
```

Disadvantage: They can be ignored

Exception detection - throw

When an exception is detected, we **throw**

Exception detection - throw

What do we throw?

Exception detection - throw

What do we throw?

An object that describes the error we encountered

Back to the vector

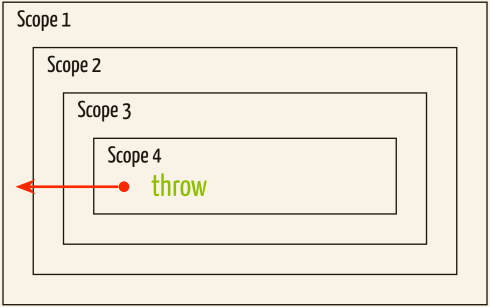
```
struct OutOfRangeError {};

template <class Type>
class Vector
{
public:
  Type& operator[](const std::size_t index)
  {
    if (index >= size) {
      throw OutOfRangeError {};
    }

    // ...
  }
};
```

Exception handling - catch

When an exception is thrown, the code will move outwards until it is caught



If the exception isn't caught, the program terminates

Exception handling - catch

Exception handling is done by try-catch blocks

```
try {
    executing code
} catch (ExceptionType & err) {
    exception handling
}
```

The catch block is only executed if an exception of the corresponding type is thrown in the try block

Exception handling - catch

Can do multiple catch statements

```
try {

} catch (std::runtime_error & e) {

} catch (std::exception & e) {

} catch (...) {

}
```

As with if-else, first match is executed

Exception classes

One should implement exception classes through inheritance

That way the handler doesn’t have to know about everything that can go wrong inside the try-block

Should always catch by reference to avoid slicing

exception

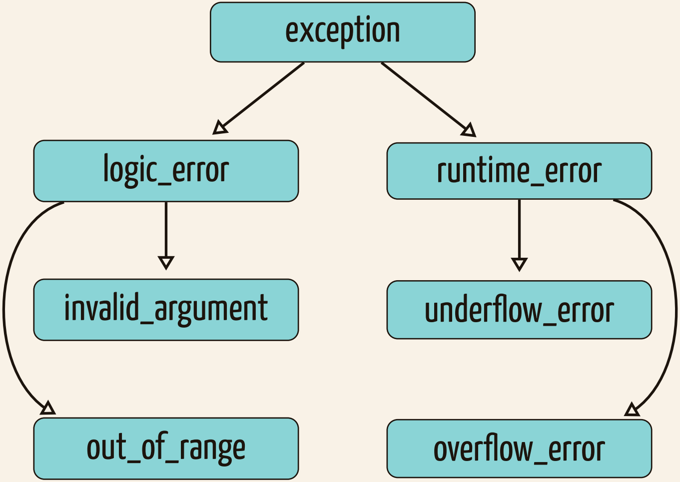
Base exception type in the standard library

Notable functions

virtual what()	returns an explanatory cstring

#include<exception>

Exception Class Hierarchy



#include<stdexcept>

Inheriting from std::exception

```
class Error : public std::exception
{
public:
  Error(std::string err)
    : error_message {std::move(err)} {}

  virtual const char* what() const noexcept override
  {
    return error_message.c_str();
  }

private:
  std::string error_message;
};
```

Performance impact

Biggest complaint from C users: exceptions are slow

Performance impact

Biggest complaint from C users: exceptions are slow

Well, then you are using them wrong...

Never use exceptions to steer program flow

If the code doesn't throw, there is no overhead

Where to go from here

Only the tip of the iceberg

boost

The best multi-functional library for C++ out there

- Filesystem
- Iostreams
- Iterator
- Multi-Array
- Multiprecision
- Phoenix
- Program Options
- Property Tree
- System
- ...and many more

Fun with templates

There is so much one can do with templates

- Metaprogramming
- Variadic templates
- Expression templates

Design patterns

Design patterns are simple and elegant solutions to specific problems one often encounters when coding

Design patterns are often language independent, and is an indispensable tool to any good programmer

All the details

cppreferen.com is my most visited webpage

Scott Meyers' "Effective ..." series is **really** good

Herb Sutter's "Guru of the Week" is very enlightening

Language development

There is a lot happening these days

`isocpp/CppCoreGuidelines`

Programming Practices

Good Programming Practices

- prefer std::array to static arrays
- prefer std::vector to dynamic arrays
- prefer algorithms calls to hand-written loops
- avoid the use of ”dumb” pointers

Good Programming Practices

When it comes to exceptions

- throw by value
- catch by reference
- re-throw if necessary
- inherit from std::exception

Recap

Recap Day 5

- Lots of useful tools in the STL
  - Containers
  - Streams
  - Smart pointers
  - ++
- Iterators give a generic way of working with all the different containers
- Algorithms for all your programming needs

Recap Day 5

- Use throw to signal an exception
- try-catch blocks to handle them
- One can inherit from std::exception to make a uniform exception interface

May the
FORCE
be with
YOU