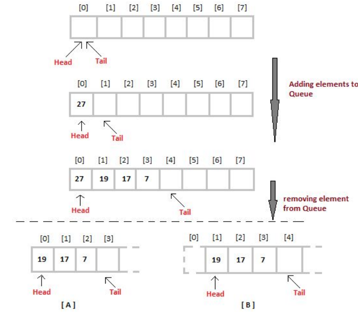
STL - Principles and Practice

Victor Ciura - Technical Lead, Advanced Installer
Gabriel Diaconița - Senior Software Developer, Advanced Installer
http://www.advancedinstaller.com
CAPHYON

Agenda

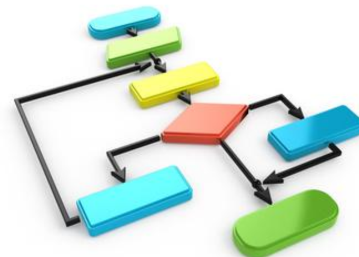
Part 0: STL Intro.

Part 1: Containers and Iterators



Part 2: STL Function Objects and Utilities

Part 3-4: STL Algorithms Principles and Practice



Part 2:

STL Function Objects and Utilities

Function Objects

- A function object, or functor, is any **`class/struct`** that implements **`operator()`**
- This operator is referred to as ***the call operator*** or sometimes the application operator.
- STL uses function objects primarily as **sorting** criteria for **containers** and in **algorithms**.
- Functions objects (functors) pervade the STL. **`#include <functional>`**
- Associative containers use them to keep their elements in order.
- Algorithms such as **`find_if()`** use them to control their behavior.
- Adapters like **`not1()`** and **`bind()`** actively produce them.
- STL provides many built-in function objects.

Function Objects

Arithmetic operations

- **plus** function object implementing `x + y`
- **minus** function object implementing `x - y`
- **multiplies** function object implementing `x * y`
- **divides** function object implementing `x / y`
- **modulus** function object implementing `x % y`
- **negate** function object implementing `-x`

Comparisons

- **equal_to** function object implementing `x == y`
- **not_equal_to** function object implementing `x != y`
- **greater** function object implementing `x > y`
- **less** function object implementing `x < y`
- **greater_equal** function object implementing `x >= y`
- **less_equal** function object implementing `x <= y`

Function Objects

Logical operations

- `logical_and` function object implementing `x && y`
- `logical_or` function object implementing `x || y`
- `logical_not` function object implementing `!x`

Bitwise operations

- `bit_and` function object implementing `x & y`
- `bit_or` function object implementing `x | y`
- `bit_xor` function object implementing `x ^ y`
- `bit_not` function object implementing `~x`

Function Objects

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

Function Objects

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

Function Objects

```
// already defined in STL
template<typename T>
struct greater
{
  bool operator()(const T & lhs, const T & rhs) const
  {
    return lhs > rhs;
  }
};
```

Eg. Changing the behavior (ordering) of std:: **set**

```
std::set<int> = { 14, 2, 5, 2, 9, 11, 5, 7 };
// => 2, 5, 7, 9, 11, 14

std::set<int, std::greater<int>> = { 14, 2, 5, 2, 9, 11, 5, 7 };
// => 14, 11, 9, 7, 5, 2
```

Function Objects

```
template<class InputIt, class UnaryFunction>
void std::for_each( InputIt first, InputIt last,  UnaryFunction func )
{
  for(; first != last; ++first)
    func( *first );
}

struct Printer // our custom functor for console output
{
  void operator()(const std::string & str)
  {
    std::cout << str << std::endl;
  }
};

std::vector<std::string> vec = { “STL”, “function”, “objects”, “rule” };

std::for_each(vec.begin(), vec.end(), Printer());
```

Function Objects

`std::function()`

- Class template **`std::function`** is a general-purpose polymorphic function wrapper.
- Instances of **`std::function`** can store, copy, and invoke any ***callable*** *target*:
    - functions
    - lambda expressions
    - bind expressions
    - function objects
    - pointers to member functions
- The stored callable object is called the ***target*** of std::function (can be empty).
- Usually, the purpose is ***storing*** a callable target code for ***later invocation***.

Function Objects

std::function()

```
void DoWork() {...}
void Print(int flags) {...}
bool IsPrime() {...}
int GetLength(const string & str) {...}
int GetCoef(const char * name, int flags, bool raw) {...}

std::function<void ()> f1 = DoWork;                                        f1();
std::function<void (int)> f2 = Print;                                      f2(5);
std::function<bool ()> f3 = IsPrime;                                       if ( f3() ) {...}
std::function<int (const string &)> f4 = GetLength;                        len = f4("example");
std::function<int (const char *, int, bool)> f5 = GetCoef;                 k = f5("key",7,false);
```

Lambda Functions

```
struct Printer // our custom functor for console output
{
  void operator()(const string & str)
  {
    cout << str << endl;
  }
};

std::vector<string> vec = { "STL", "function", "objects", "rule" };

std::for_each(vec.begin(), vec.end(), Printer());



// using a lambda

std::for_each(vec.begin(), vec.end(),
              [](const string & str) { cout << str << endl; });
```

Lambda Functions

```
[ capture-list ] ( params ) mutable(optional) -> ret { body }

[ capture-list ] ( params ) -> ret { body }

[ capture-list ] ( params ) { body }

[ capture-list ] { body }
```

Capture list can be passed as follows :

- **[a, &b]** where **a** is captured by ***value*** and **b** is captured by ***reference***.
- **[this]** captures the **this** pointer by ***value***
- **[&]** captures all automatic variables **used** in the body of the lambda by ***reference***
- **[=]** captures all automatic variables **used** in the body of the lambda by ***value***
- **[]** captures ***nothing***

Anatomy of A Lambda

Lambdas == Functors

[captures] (params) -> ret { statements; }

```
class __functor {
private:
  CaptureTypes __captures;
public:
  __functor( CaptureTypes captures )
    : __captures( captures ) { }

  auto operator() ( params ) -> ret
    { statements; }
};
```

credit: Herb Sutter - "*Lambdas, Lambdas Everywhere*"
https://www.youtube.com/watch?v=rcgRY7sOA58

Anatomy of A Lambda

Capture Example

```
[ c1, &c2 ]  { f( c1, c2 ); }
```

```
class __functor {
  private:
    C1 __c1;   C2& __c2;
  public:
    __functor( C1 c1, C2& c2 )
      : __c1(c1), __c2(c2) { }

    void operator()() { f( __c1, __c2 ); }
};
```

credit: Herb Sutter - “*Lambdas, Lambdas Everywhere*”
https://www.youtube.com/watch?v=rcgRY7sOA58

Anatomy of A Lambda

Parameter Example

```
[]  ( P1 p1, const P2& p2 )   { f( p1, p2 ); }
```

```
class __functor {

public:
  void operator()( P1 p1, const P2& p2 ) {
    f( p1, p2 );
  }

};
```

credit: Herb Sutter - “*Lambdas, Lambdas Everywhere*”
https://www.youtube.com/watch?v=rcgRY7sOA58

Lambda Functions

```
std::list<Person> members = {...};
unsigned int minAge = GetMinimumAge();
members.remove_if( [minAge](const Person & p) { return p.age < minAge; } );

// compiler generated code:
namespace {
struct lambda3
{
  lambda3(unsigned int age) : minAge(age) {}
  bool operator()(const Person & p) { return p.age < minAge; }
  unsigned int minAge;
}; }

members.remove_if( lambda3(minAge) );
```

Prefer Function Objects or Lambdas to Free Functions

```
vector<int> v = { … };

bool GreaterInt(int i1, int i2) { return i1 > i2; }

sort(v.begin(), v.end(), GreaterInt); // pass function pointer

sort(v.begin(), v.end(), greater<>());

sort(v.begin(), v.end(), [](int i1, int i2) { return i1 > i2; });
```

Function Objects and Lambdas leverage operator() **inlining**
vs.
indirect **function call** through a *function pointer*

This is the main reason **std::sort()** outperforms **qsort()** from **C**-runtime by at least 500% in typical scenarios, on large collections.

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

```
string binStr;
binStr.reserve(32);

// recursive implementation
void bin(unsigned int n)
{
  if (n > 1)
    bin( n/2 );

  binStr += (n % 2) ? "1" : "0";
}
```

Is this implementation correct / complete ?

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.
* without leading zeros

```
// iterative implementation
string bin(unsigned int n)
{
  string binStr;
  binStr.reserve(32);

  for (unsigned int i = 1u << 31; i > 0; i = i / 2)
    binStr += (n & i) ? "1" : "0";

  return binStr;
}
```

Is this implementation correct / complete ?

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

```
// STL implementation
string bin(unsigned int n)
{
  string binStr = std::bitset<32>(n).to_string();

  // erase leading zeros, if any
  binStr.erase(0, binStr.find_first_not_of('0'));
  return binStr;
}
```

Learn what's available in your standard toolbox !

SAMPLE: Function pointers vs Functors

Scenario: implement a generic and scalable solution for sending event confirmation emails

- An event has limited seat availability
- Applicants may be accepted based on (customizable) criteria
- Attendees will receive confirmation emails

SAMPLE: Function pointers vs Functors

Person
Class

Fields
- mEducationLevel
- mEmail
- mName

Methods
- GetEducationLevel
- GetEmail
- GetName
- Person (+ 1 overload)

Nested Types

EducationLevel
Enum Class

- Elev
- Student
- Angajat

SOLUTION: Function pointers vs Functors

```
typedef bool(*AcceptanceCriterion)(const Person &);

void ProcessApplication(const Person & registeredPerson,
                        AcceptanceCriterion qualifiesAsAtendee,
                        const int totalSeats, int & seatsTaken)
{
  if (seatsTaken >= totalSeats)
    return;
  if (!qualifiesAsAtendee(registeredPerson))
    return;
  SendConfirmationMail(registeredPerson.GetEmail());
  seatsTaken += 1;
}
```

SOLUTION: Function pointers vs Functors

```
class EventRegistrationFunctor
{
public:
  typedef function<bool(const Person &)> AcceptanceCriterion;

  EventRegistrationFunctor(string eventName, int availableEventSeats,
                           AcceptanceCriterion acceptanceCriterium)
  : totalSeats(availableEventSeats), qualifiesAsAtendee(acceptanceCriterium), seatsTaken(0)
  {
  }

  void operator()(const Person & registeredPerson)
  {
    if (seatsTaken >= totalSeats)
      return;
    if (! qualifiesAsAtendee(registeredPerson))
      return;
    SendConfirmationMail(registeredPerson.GetEmail());
    seatsTaken += 1;
  }

private:
  string               name;
  int                  totalSeats, seatsTaken;
  AcceptanceCriterion  qualifiesAsAtendee;
};
```

The same code as before

USAGE: Function pointers vs Functors

```
vector<Person> firstThreeRegistered =
{
  { "Andrei Cristescu",           "foo 1 @bar.net", Person::EducationLevel::Elev    },
  { "Cioromela Valentin Stefan", "foo 2 @bar.net", Person::EducationLevel::Student },
  { "Istinie Cristian Danut",     "foo 3 @bar.net", Person::EducationLevel::Student }
};

bool CaphyonSummerSchoolCriterion(const Person &)
{
  return true; // everyone is welcome
}

{ // C++ STL Principles and Practice
  const int totalSeats = 40;
  int seatsTaken = 0;
  for (size_t i = 0; i < firstThreeRegistered .size(); ++i)
    ProcessApplication(firstFiveRegistered[i], CaphyonSummerSchoolCriterion, totalSeats, seatsTaken);
}

{ // Modern web programming
  const int totalSeats = 40;
  int seatsTaken = 0;
  for (size_t i = 0; i < firstThreeRegistered .size(); ++i)
    ProcessApplication(firstFiveRegistered[i], CaphyonSummerSchoolCriterion, totalSeats, seatsTaken);
}
```

USAGE: Function pointers vs Functors

```
vector<Person> firstThreeRegistered =
{
  { "Andrei Cristescu",          "foo 1 @bar.net", Person::EducationLevel::Elev    },
  { "Cioromela Valentin Stefan", "foo 2 @bar.net", Person::EducationLevel::Student },
  { "Istinie Cristian Danut",    "foo 3 @bar.net", Person::EducationLevel::Student }
};
```

```
auto caphyonSummerSchoolCriterion = [](const Person &) { return true; /* everyone is welcome */ };

EventRegistrationFunctor stlSummerSchoolApply("C++ STL Principles and Practice", 40, caphyonCriterion);

for_each( begin(firstThreeRegistered), end(firstThreeRegistered), stlSummerSchoolApply );

EventRegistrationFunctor webSummerSchoolApply("Modern web programming", 40, caphyonCriterion);
for_each( begin(firstThreeRegistered), end(firstThreeRegistered), webSummerSchoolApply );
```