Modern C++

Luiz Alberto do Carmo Viana

June 8, 2020

Outline

History

- C with Classes: created by Bjarne Stroustrup at Bell Labs in 1979.
- C++98: first standard.
- C++03: requires that elements in `vector` are stored contiguously.
- C++11: legacy support; big changes; modern C++.
- C++14: better type deduction; generic lambdas; variable templates.
  - This is the default standard of GCC.
- C++17: `optional`; `any`; `variant`.
- C++20: to be finished (don't touch it, corona).

Hello World

```
#include <iostream>

int main(){
  std::cout << "Hello World" << std::endl;

  return 0;
}
```

Overview

- Extended C syntax.
- Large language: significant amount of legacy; focus on modern subset.
- Nearly no overhead abstractions: as C, good mapping to hardware.

Features

Uniform initialization {}:

```
// initialization: calls constructor
int a{}; // a == 0
int x{1};
std::string word{"apple"};
// assignment: calls operator=
int x = 1;
std::string word = "apple"; // calls constructor, but is misleading
word = "orange"; // here it does call operator=
// initialization does not perform invisible conversions
int y = 3.5; // y == 3
int y{3.5}; // error
// container constructors may bring ambiguities
std::vector<int> vec{10}; // vec contains one int element of value 10
std::vector<int> vec(10); // vec contains ten int elements of value 0
```

Features

- auto keyword: type deduction.
- Useful: less typing; type change propagation; generic programming.

```
auto a{true}; // bool
auto b{'a'}; // char
auto c{49}; // int
auto d{1.0}; // double
auto e{f(x)}; // e has the return type of f

std::vector<int> vec{1, 2, 3};
auto it = std::begin(vec); // std::vector<int>::iterator
```

Features

- `const`: variable that cannot be changed; function that promises not to change anything.
- `constexpr`: compile-time functions, variables and expressions.
- `constexpr const` is not redundant.

```
int global{3};

constexpr int f(int x){
  return x * 2;
}

int g(int x){
  return global + x;
}

constexpr int a{f(2)}; // evaluated at compile time
constexpr int b{f(a)}; // ok
// ...
// what if global has changed?
const int x{global};
constexpr int c{x}; // error
constexpr int d{f(x)}; // error
constexpr int e{global}; // error
```

Features

```
int arr[10]; //uninitialized array
int* ptr; // uninitialized pointer (dangerous)
int* ptr{arr + 2}; // initialized pointer
int x{*ptr}; // initialized with object that ptr points to
int& ref{x}; // reference
ref = 3; // x == 3
int y{4};
ref = y; // x == 4
int& ref2; // error: references must be initialized
```

- References offer indirect access to a variable.
- After initialization, a reference cannot refer to something else.
- Useful as function arguments.
- `const` references offer a view over a variable.

Features

- nullptr: more robust than NULL (NULL is still there, but forget it).
- NULL is just 0.
- nullptr is the only value of type nullptr_t.
- color is safe to call: no need to check node before calling it.

```
Color color(const Node* node){
  return node->color;
}

Color color(nullptr_t leaf){
  return Color::black;
}
```

Features

An `enum class` defines a scoped enumerator with a type.

```
enum class Color{red, black};
Color c{Color::red};
c = Color::black;
```

Features

- `namespace` declares or extends a collection of definitions.
- `std::string`: `string` is defined inside `namespace` `std`.
- `namespace` is useful for avoiding name conflicts between libraries.
- Standard C++ definitions belong to `namespace` `std`.

Features

Structured bindings make C++ feels modern.

```
std::pair<std::string, int> attach_length(const std::string& str){
  return std::make_pair(str, str.size());
}

std::string str1{"something"};
auto[str2, len] = attach_length(str1);

// use them together with uniform initialization
std::vector<std::pair<int, int>> vec{};

for(int i = 1; i <= 3; ++i){
  vec.push_back({i, i*10});
}

for(auto[a, b] : vec){
  std::cout << a << ' ' << b << std::endl;
}
```

Features

Move semantics: instead of copying, moves objects.

```
// copy semantics
std::string s1{"aaa"};
std::string s2{"bbb"};
s2 = s1; // s1 == "aaa", s2 == "aaa"
// move semantics
std::string s1{"aaa"};
std::string s2{"bbb"};
s2 = std::move(s1); // s1 is undefined, s2 == "aaa"
```

- `std::move(Type)` returns a `Type&&` (rvalue reference).
- It is undefined behaviour referencing a moved variable.
- Sometimes it is okay to do that.
- Good practice is not to touch moved things: don't take a chance.

Classes

- `struct` and `class` are quite the same:
  - `struct` defaults to public members;
  - `class` defaults to private members.
- Let's focus on `class`.

```
class Person{
  std::string name_;
  unsigned int age_;
public:
  Person(std::string name, unsigned int age) : name_{name},
                                               age_{age}
  {}
  const std::string& name() const{
    return name_;
  }
  unsigned int age() const{
    return age_;
  }
};
```

Classes

- A `class A` always has a (unless deleted):
  - default constructor: `A a{};`
  - copy constructor: `A a1{args...}; A a2{a1};`
  - move constructor: `A a1{args...}; A a2{std::move(a1)};`
  - copy assignment:
    ```
    A a1{args...}, a2{args...};
    a2 = a1;
    ```
  - move assignment:
    ```
    A a1{args...}, a2{args...};
    a2 = std::move(a1);
    ```
  - destructor:
    ```
    {
      A a1{args...};
      ...
    } // ~A() is called here
    ```
- If not provided, a implicitly declared implementation may be used:
  - for each member variable, it calls its counterpart.

Classes

`Person p1{...}; Person p2{...};`

- `Person` has a implicitly declared:
  - default constructor:
    - `Person()` calls `name_{}` and `age_{}`;
    - it can be called with `Person p{};`
  - copy constructor:
    - `Person(const Person& p)` calls `name_{p.name_}` and `age_{p.age_};`
    - it can be called with `p2{p1};`
  - move constructor:
    - `Person(Person&& p)` calls `name_{std::move(p.name_)}` and `age_{std::move(p.age_)};`
    - it can be called with `p2{std::move(p1)}`.

Classes

- `Person` has a implicitly declared:
  - copy assignment:
    - `Person& operator=(const Person& p)` calls `name_ = p.name_` and `age_ = p.age_;`
    - it can be called with `p1 = p2;`
  - move assignment:
    - `Person& operator=(Person&& p)` calls `name_ = std::move(p.name_)` and `age_ = std::move(p.age_);`
    - it can be called with `p1 = std::move(p2)`.

Classes

On {default, copy, move} constructors, {copy, move} assignments and destructor:

- It is okay for a `class A` to have all of them implicitly declared:
  - if all members of `A` are language defined or;
  - all user defined members of `A` are not special:
    - does `A` deal with files or network connections in a special manner?
- if you need to write one of them, write all of them.

Object Oriented Programming

- Multiple inheritance:
  - more powerful than single inheritance with interfaces;
  - also more dangerous (inheritance conflicts);
  - no `super` pointer.
- Virtual classes: can't be instantiated.
- Virtual methods:
  - interface-ish;
  - a class with virtual methods is virtual as well.
- `this` pointer.
- `super` pointer makes no sense with multiple inheritance.

Templates

- C++ metaprogramming device.
- Seen a lot of them so far (`std::vector<int>`).

```
template<typename Type>
Type square(const Type& n){
  return n*n;
}

int x{2};
std::cout << square(x) << std::endl; // Type is deduced to be int
std::cout << square<int>(x) << std::endl; // Type is set explicitly

Matrix M{}; // suppose Matrix defines operator*
Matrix A{square(M)}; // Type is deduced to be Matrix
```

Templates

Templates can have default values:

```
template<typename Type = double>
class Point{
  Type x_;
  Type y_;
public:
  Point(Type x, Type y) : x_{x},
                          y_{y}
  {}
  Point operator+(const Point& p){
    return {x_ + p.x_, y_ + p.y_};
  }
};

Point x{0.0, 0.0};
Point y{1.0, 1.0};
Point z{x + y};
```

Templates

Beyond `typename`:

```
template<typename Type = double, unsigned int dimension = 3>
class Point{
  std::array<Type, dimension> arr_;
public:
  // ...
};
```

Templates

- We have seen template functions and template classes.
- What about template variables?

```
template<typename Type>
constexpr const Type pi{3.14159265358979323846};

pi<float>; // a less precise pi
pi<double>; // a more precise pi
```

- This is not the point of template variables.

Templates

Templates can be specialized:

```
template<typename Type>
void read(Type& var){
  std::cin >> var;
}

template<>
void read(std::string& var){
  std::cin >> std::ws;
  getline(std::cin, var);
}
```

Devise better implementations: `std::vector<bool>` is implemented bitwise.

Templates

Template variables can be specialized as well.

```
template<typename A, typename B>
constexpr const bool equal{false};

template<typename Type>
constexpr const bool equal<Type, Type>{true};
```

Now we can specialize `read` using `equal` and `if constexpr`.

```
template<typename Type>
void read(Type& var){
  if constexpr (equal<Type, std::string>){
    std::cin >> std::ws;
    getline(std::cin, var);
  }
  else{
    std::cin >> var;
  }
}
```

Memory Management

- C uses `malloc` and `free` for dynamic allocation.
- Old C++ uses `new` and `delete`.
- Modern C++ uses smart pointers.
- We need to `#include <memory>` to use them.

Memory Management

When used in a proper manner:

- `unique_ptr<Type>`: no other `unique_ptr<Type>` shares its resource.
- `shared_ptr<Type>`: other `shared_ptr<Type>` may share its resource (reference counting).
- `weak_ptr<Type>`: offers access to the (possibly dead) resource of a `shared_ptr<Type>` (no reference counting).

Memory Management

Forget about `new`:

- `unique_ptr<Type>`: `make_unique<Type>(args...)`.
- `shared_ptr<Type>`: initialize it with a moved `unique_ptr<Type>`.
- `weak_ptr<Type>`: initialize it with a `shared_ptr<Type>`.

Forget about `delete`:

- `unique_ptr<Type>`: when it dies or is assigned, resource is freed.
- `shared_ptr<Type>`: a resource is freed when its last `shared_ptr` is dead or assigned.
- `weak_ptr<Type>`: unless locked, it does not prevent a resource from being freed (`lock` produces a `shared_ptr` from a `weak_ptr`).

Memory Management

```
class A{
  // ...
};

// C style
A* obj = malloc(sizeof(A));
// obj still needs to be initialized
free(obj);
// Old C++ style (at least calls a constructor)
A* obj = new A(...);
delete obj;
// C++11 style
std::unique_ptr<A> obj{new A(...)};
// deletion is automatic (end of scope)
// C++14 style
std::unique_ptr<A> obj{std::make_unique(...)}; // or
auto obj{std::make_unique<A>(...)};
// deletion is automatic (end of scope)
// shared pointer
std::shared_ptr<A> shared_obj{std::move(obj)};
// deletion is automatic (reference counting)
// weak pointer
std::weak_ptr<A> weak_obj{obj};
// weak_ptr cannot delete obj
```

Memory Management

Allocating contiguous memory

```
class A{
  // ...
};

// C style stack memory (size already known)
A arrA[100];

// C style heap memory (size not known)
int n = // ...
A* arrA = malloc(n * sizeof(A));

// C++ style stack memory
std::array<A, 100> arrA{};

// C++ style heap memory
int n{...};
std::vector<A> arrA{};
arrA.reserve(n);
```

- GCC `push_back` doubles `vector` capacity when it is full.
- n `push_back` operations on empty `vector`:
  - $\frac{n}{2} + \frac{n}{4} + \cdots + 1 = 2n - 1$;
  - after `reserve(n)`, they cost n.

Lambda Functions

Anonymous functions.

```
const auto f{[](int& x) {x += 3;}};
std::vector<int> vec{1, 2, 3, 4, 5};
for (auto& e : vec){
  f(e);
}
// {4, 5, 6, 7, 8}
int sum{0};
const auto g{[&sum](int x) {sum += x;}};
for (auto e : vec){
  g(e);
}
// sum == 30
```

Iterators

Let's perform a traversal using a pointer.

```
// C style
int arr[10];

int* begin = arr;
int* end = arr + 10; // pointer arithmetic: arr + sizeof(int) * 10

int* it;
for(it = begin; it != end; ++it){
  // do something with *it
  *it += 3;
}
```

Iterators provide a high level pointer arithmetic abstraction.

Iterators

We use `std::begin` and `std::end` to control a traversal.

```
template<typename Collection>
void increment_all(Collection& col){
  for (auto it{std::begin(col)}; it != std::end(col); ++it){
    *it = *it + 1;
  }
}
```

`increment_all` can be used with any container that provides iterators.

```
std::array<int, 5> arr{1, 2, 3, 4, 5};
increment_all(arr); // {2, 3, 4, 5, 6}
std::vector<int> vec{1, 2, 3, 4, 5};
increment_all(vec); // {2, 3, 4, 5, 6}
std::string str{"apple"};
increment_all(str); // "bqqmf"
// ugly C arrays
int ugly[5]{1, 2, 3, 4, 5};
increment_all(ugly); // {2, 3, 4, 5, 6}
```

Also works with `std::list`, `std::deque` and some others.

Iterators

Another example.

```
template<typename Iterator, typename Function>
void apply_to(Iterator begin, Iterator end, Function f){
  for (auto it{begin}; it != end; ++it){
    *it = f(*it);
  }
}

std::array<int, 5> arr{1, 2, 3, 4, 5};
apply_to(std::begin(arr) + 1, std::begin(arr) + 3,
         [](int x) {return x + 3;});
// {1, 5, 6, 4, 5}
```

Iterators

Template variables love lambda functions.

```
template<typename Type, Type t>
constexpr const auto add{[](Type x) {return x + t;}};

std::array<int, 5> arr{1, 2, 3, 4, 5};
apply_to(std::begin(arr) + 2, std::end(arr), add<int, 5>);
// {1, 2, 8, 9, 10}
```

Iterators

`std::`	Read-only	Reverse
`begin`, `end`	no	no
`cbegin`, `cend`	yes	no
`rbegin`, `rend`	no	yes
`crbegin`, `crend`	yes	yes

- Read about iterator adaptors.
  - We are going to use `back_inserter` here.
- Why bother with iterators?
  - generic algorithms; STL.

Iterators

Example of a (silly) iterator adaptor.

```
template<typename Collection>
class Reverse{
  Collection& col_;
public:
  Reverse(Collection& col) : col_{col} {}
  using iterator          = typename Collection::reverse_iterator;
  using traits            = std::iterator_traits<iterator>;
  using difference_type   = typename traits::difference_type;
  using value_type        = typename traits::value_type;
  using reference         = typename traits::reference;
  using pointer           = typename traits::pointer;
  using iterator_category = typename traits::iterator_category;
  iterator begin() {return std::rbegin(col_);}
  iterator end()   {return std::rend(col_);}
};
```

Iterators

```
template<typename Type>
class Range{
  Type from_, to_, step_;
public:
  Range(Type from, Type to, Type step) : from_{from}, to_{to}, step_{step}
  Range(Type from, Type to) : Range{from, to, 1} {}
  struct iterator{
    Type x_; Range& parent_;
    using difference_type = Type; using value_type = Type;
    using reference = Type&; using pointer = Type*;
    using iterator_category = std::input_iterator_tag;
    iterator(Type x, Range& parent) : x_{x}, parent_{parent} {}
    bool operator!=(const iterator& it) { return x_ != it.x_;}
    bool operator==(const iterator& it) {return x_ == it.x_;}
    reference operator*() {return x_;}
    difference_type operator-(const iterator& it) {return x_ - it.x_;}
    iterator& operator++(){
      Type next{x_ + parent_.step_}, to{parent_.to_};
      x_ = next < to ? possible_step : to;
      return *this;
    }
  };
  iterator begin() {return iterator{from_, *this};}
  iterator end() {return iterator{to_, *this};}
};
```

Standard Tenplate Library

- C++ Standard Library.
- Containers, algorithms and other facilities.
  - General interfaces, some specialized implementations.
- Let's see some of them by example.

Standard Template Library

```
#include <algorithm>

template<typename Type>
constexpr const auto is_odd{[](Type x) {return x % 2 == 1;}};

template<typename Type, Type ub>
constexpr const auto leq{[](Type x) {return x <= ub;}};

std::vector<int> vec{1, 3, 5, 7, 9};
std::all_of(std::cbegin(vec), std::cend(vec), is_odd<int>); // 1
std::vector<int> vec{1, 3, 6, 7, 9};
std::all_of(std::cbegin(vec), std::cend(vec), is_odd<int>); // 0

std::any_of(std::cbegin(vec), std::cend(vec), leq<int, 1>); // 1
std::any_of(std::cbegin(vec), std::cend(vec), leq<int, 0>); // 0
```

Standard Template Library

```
#include <algorithm>

template<typename Type>
class Leq{
  Type ub_;
public:
  Leq(Type ub) : ub_{ub} {}
  bool operator()(Type x){
    return x <= ub_;
  }
};

int n{1};
std::any_of(std::cbegin(vec), std::cend(vec), Leq{n}); // 1
n = 0;
std::any_of(std::cbegin(vec), std::cend(vec), Leq{n}); // 0
```

Standard Template Library

```
#include <algorithm>
```

Function	Description
`none_of`	1 iff no element of iterator range satisfies predicate
`for_each`	applies function to each element of iterator range
`find`	iterator to first occurrence of element in range
`find_if`	first element of range satisfying predicate
`find_end`	searches `range1` for last occurrence of `range2`
`find_first_of`	first element in `range1` occurring in `range2`
`adjacent_find`	first adjacent elements satisfying binary predicate
`count`	number of occurrences of element in range
`count_if`	number of elements in range satisfying predicate
`mismatch`	pair of first mismatching positions of ranges
`equal`	1 iff two ranges are equal
`is_permutation`	1 iff `range1` is permutation of `range2`
`search`	searches `range1` for first occurrence of `range2`

Standard Template Library

```
std::vector<int> vec1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<int> vec2{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

std::count_if(std::cbegin(vec1), std::cend(vec1), is_odd<int>); // 5

auto begin1{std::cbegin(vec1)};
auto end1{std::cend(vec1)};
auto begin2{std::cbegin(vec2)};
auto end2{std::cend(vec2)};

std::distance(begin1, end1) == std::distance(begin2, end2) &&
  std::is_permutation(begin1, end1, begin2);  // 1
```

Standard Template Library

```
bool is_prime(unsigned x){
  if (x == 1){
    return false;
  }
  if (x == 2 || x == 3){
    return true;
  }
  if (x % 2 == 0){
    return false;
  }
  else{
    unsigned ub{static_cast<unsigned>(floor(sqrt(x)))};
    Range<unsigned> divisor_candidates{3, ub + 1, 2};

    return std::none_of(std::cbegin(divisor_candidates),
                        std::cend(divisor_candidates),
                        [x](unsigned d) {return x % d == 0;});
  }
}
```

Standard Template Library

Function	Description
`copy`	copies range to output iterator
`copy_if`	copies range elements satisfying predicate to output
`swap`	exchanges contents of objects
`swap_ranges`	applies `swap` to two ranges
`transform`	operator applied to one (two) range(s) goes to output
`replace`	overwrites `value1` with `value2` in range
`replace_if`	assigns value to elements of range satisfying predicate
`generate`	fills range with return values of function
`remove`	removes value from range
`remove_if`	removes elements from range satisfying predicate
`unique`	removes consecutive duplicates from range

Note: `remove` and `remove_if` do not realloacte memory.

Standard Template Library

```cpp
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// this does not alter vec capacity
auto new_end{std::remove_if(std::begin(vec), std::end(vec), is_prime)};
// now, vec.erase(new_end, std::end(vec)) would alter vec capacity

std::vector<int> vec2{};
vec2.reserve(std::distance(std::begin(vec), new_end));

std::copy(std::begin(vec), new_end, std::back_inserter(vec2));

for (auto e : Reverse{vec2}){
  std::cout << e << ' ';
}
// 10 9 8 6 4 1
```

Standard Template Library

<algorithm> also has functions related to:

- partition;
- sorting;
- binary search (on sorted ranges);
- merging (of sorted ranges);
- heap adaptors;
- minimum and maximum values;

Also take a look at:

- containers: <set>, <map>, <bitset>;
- container adaptors: <stack>, <queue>;
- utilities: <optional>, <functional>, <tuple>, <type_traits>.

Further Reading

- A Tour of C++, by Bjarne Stroustrup.
- C++ Core Guidelines (link), by Bjarne Stroustrup and Herb Sutter.
- C++17 STL Cookbook, by Jacek Galowicz.

Thank you

Thank you!