A few C++11 tips

Bruce Merry

IOI Training Mar 2014

Outline

The `auto` Keyword

Variables declared `auto` take their type from the initializer:

```
map<int, int>::iterator it1 = m.begin();
auto it2 = f.begin();
```

The `auto` Keyword

Variables declared `auto` take their type from the initializer:

```
map<int, int>::iterator it1 = m.begin();
auto it2 = f.begin();
```

Can also force it to be a reference:

```
auto &value = vec[3];
value++; // modifies vec[3]
```

Loops With Auto

The `auto` keyword saves typing in loops:

```
for (map<int, int>::iterator it = m.begin();
     it != m.end(); ++it) {
  // Do stuff with *it
}
for (auto it = m.begin(); it != m.end(); ++it)
  // Do stuff with *it
}
```

For Each Loop

C++11 also supports Java-style for-each loops:

```
vector<int> v;
for (int value : v) {
  // value is a copy of the vector element
}
for (int &value : v) {
  // value references the vector element
}
for (auto &value : v) {
  // auto keyword works here too
}
```

Brace Initialization

In C++03, arrays and structures can be initialized with a brace-enclosed list:

```
struct S { int a; double b; };

S array[2] = { {3, 1.5}, {4, 2.5} };
```

But STL containers cannot.

Brace Initialization of Containers

In C++11, containers can be brace-initialized:

```
vector<int> v = {1, 2, 3};
vector<int> v{1, 2, 3}; // equivalent
```

Brace Initialization of Containers

In C++11, containers can be brace-initialized:

```
vector<int> v = {1, 2, 3};
vector<int> v{1, 2, 3}; // equivalent
```

One can also construct temporaries:

```
func_call(vector<int>{1, 2, 3});
```

Brace Initialization of Containers

In C++11, containers can be brace-initialized:

```
vector<int> v = {1, 2, 3};
vector<int> v{1, 2, 3}; // equivalent
```

One can also construct temporaries:

```
func_call(vector<int>{1, 2, 3});
```

In a number of cases, the type name can be omitted:

```
func_call({1, 2, 3});
```

In-class Initialization

As in Java, class members can have initializers:

```
class Foo
{
private:
    int a = 3;
    string name = "bob";
public:
    Foo(int a) : a(a) {}
};
```

Constructors can override the default.

Emplacing Elements

Can combine construction and insertion

```
vector<pair<int, int> > v;
// C++03
vec.push_back(pair<int, int>(3, 5));
// C++11
vec.emplace_back(3, 5);
```

Unordered Set and Map

- `unordered_set` is like `set`

Unordered Set and Map

- `unordered_set` is like `set`
- `unordered_map` is like `map`

Unordered Set and Map

- `unordered_set` is like `set`
- `unordered_map` is like `map`
- Keys are unordered

Unordered Set and Map

- `unordered_set` is like `set`
- `unordered_map` is like `map`
- Keys are unordered
- Operations are amortized $O(1)$ rather than $O(\log N)$

Unordered Set and Map

- `unordered_set` is like `set`
- `unordered_map` is like `map`
- Keys are unordered
- Operations are amortized $O(1)$ rather than $O(\log N)$
- Like `vector`, `reserve` can improve performance

Unordered Set and Map

- `unordered_set` is like `set`
- `unordered_map` is like `map`
- Keys are unordered
- Operations are amortized $O(1)$ rather than $O(\log N)$
- Like `vector`, `reserve` can improve performance
- Insertion invalidates iterators

Moving Containers

Make one container take over the storage from another:

```
vector<int> v1(10000);
vector<int> v2;

// C++03: copies elements, expensive
v2 = v1; v1.clear();
// C++03: cheap
v2.swap(v1); v1.clear();
// C++11: cheap, easier to read
v2 = move(v1);
```

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions
- `array` container: STL-compatible array wrapper

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions
- `array` container: STL-compatible array wrapper
- `tuple`: extends `pair`

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions
- `array` container: STL-compatible array wrapper
- `tuple`: extends `pair`
- `prev` and `next`

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions
- `array` container: STL-compatible array wrapper
- `tuple`: extends `pair`
- `prev` and `next`
- Random number generation

Other Stuff

These are possibly interesting

- Lambda functions: define anonymous functions
- `array` container: STL-compatible array wrapper
- `tuple`: extends `pair`
- `prev` and `next`
- Random number generation
- Regular expressions (not yet in GCC)