

Fartash Faghri's Tricks

C++11 Cheat-sheet

31 Dec 2016

This post is a selection of changes introduced in C++11 that I found to be useful for me. All the quotations and examples are taken from the [Stroustrup's C++11 FAQ](#). Anything not quoted is my own notes. For more detailed explanations and additional resources, I strongly encourage you to read the FAQ.

C++11 is the ISO C++ standard ratified in 2011. The previous standard is often referred to as C++98 or C++03; the differences between C++98 and C++03 are so few and so technical that they ought not concern users.

Table of Contents

- [Must Knows](#)
 - `auto` – deduction of a type from an initializer
 - [Range-for statement](#)
 - [Right-angle brackets](#)
 - [Initializer lists](#)
 - [Uniform initialization syntax and semantics](#)
 - [Rvalue references](#)
 - [Algorithms improvements](#)
 - [Container improvements](#)
 - [Move operators](#)
 - [Improved push operations](#)
 - `std::array`
 - [Unordered containers](#)
 - `std::tuple`
 - `long long` – a longer integer
 - `nullptr` – a null pointer literal
- [Extras](#)
 - [Lambdas](#)
 - [Delegating constructors](#)
 - `decltype` – the type of an expression

- [Suffix return type syntax](#)
- [Raw string literals](#)
- [User-defined literals](#)
- [Explicit conversion operators](#)
- [control of defaults: default and delete](#)
- `std::function` and `std::bind`

Must Knows

`auto` – deduction of a type from an initializer

```
auto x = expression;
```

The old meaning of `auto` (“this is a local variable”) is now illegal.

Range-for statement

```
for (auto x : v) cout << x << '\n';
for (auto& x : v) ++x; // using a reference to allow us to change the value
for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

Right-angle brackets

There is no need to put spaces between double `>` s for nested templates.

```
list<vector<string>> lvs;
```

Initializer lists

```
void f(initializer_list<int>);
f({23,345,4567,56789});
vector<double> v2 = {9};
vector<double> v1{7};
v1 = {9};
int x0 {7};
```

Uniform initialization syntax and semantics

```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};
```

```

struct D : X {
D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
int a[3];
S(int x, int y, int z) :a{x,y,z} { /* ... */ }; // solution to old problem
};

```

Importantly, `X{a}` constructs the same value in every context, so that `{}` - initialization gives the same result in all places where it is legal. For example:

```

X x{a};
X* p = new X{a};
z = X{a};           // use as cast
f({a});             // function argument (of type X)
return {a};         // function return value (function returning X)

```

Rvalue references

The result of any expression is either an lvalue or an rvalue. Lvalue can be thought of as named values and rvalue as temporary unnamed values. Think of any expression as a function call, result of which is the return value of the function.

In C++98, you cannot keep an lvalue beyond its scope which means you had to copy the result returned from a function before you return from the function. This copying was a clear overhead, so to optimize it you had to either pass a reference to the function and write the result to it or go back to using C pointers and memory allocation.

C++11 solves this problem by giving you a way to keep that temporary value. You can think of it as renaming the memory and changing the scope of the value.

Compared to the solution using C pointers, you still have the benefit of a destructor which will destroy the value at the end of the scope.

The new main tools that you have are:

- `&&` (rvalue reference) to keep an rvalue, a temporary value
- `move(x)`, when x is a reference, invalidates the reference and moves the data. Lets us implement efficient swap with references.

```

template<class T> class vector {
// ...
vector(const vector&);           // copy constructor
vector(vector&&);               // move constructor
vector& operator=(const vector&); // copy assignment

```

```
vector& operator=(vector&&);    // move assignment
}; // note: move constructor and move assignment takes non-const &&
// they can, and usually do, write to their argument
```

Algorithms improvements

```
bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

T min(initializer_list<T> t);
T max(initializer_list<T> t);

pair<const T&, const T&> minmax(const T& a, const T& b);

OutIter move(InIter first, InIter last, OutIter result);
```

Effects of move: Moving can be much more efficient than copying (see Move semantics). For example, move-based `std::sort()` and `std::set::insert()` has been measured to be 15 times faster than copy based versions.

```
void iota(Iter first, Iter last, T value);
/* For each element referred to by the iterator i in the range [first,last),
assigns *i = value and increments value as if by ++value
```

`iota` can be used to create a range of values:

```
// Warning, written by Fartash
vector<int> v(n);
iota(v.begin(), v.end(), 1);
// v={1,2,...,n}
```

Container improvements

- `array` (a fixed-sized container)
- `forward_list` (a singly-linked list)
- unordered containers (the hash tables)

Move operators

The most important implication of this is that we can efficiently return a container from a function.

Because it is an rvalue, and `operator=` called with an rvalue performs a move.

Improved push operations

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

`std::array`

```
array<int,6> a = { 1, 2, 3 };
```

Unordered containers

A unordered container is a kind of hash table. C++11 offers four standard ones:

- `unordered_map`
- `unordered_set`
- `unordered_multimap`
- `unordered_multiset`

They should have been called `hash_map` etc., but there are so many incompatible uses of those names that the committee had to choose new names and the `unordered_map`, etc. were the least bad we could find.

`std::tuple`

Generalization of `std::pair`

```
tuple<string,int> t2("Kylling",123);
auto t = make_tuple(string("Herring"),10, 1.23); // t will be of type tuple<string,int,
double>
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

long long – a longer integer

```
long long x = 9223372036854775807LL;
```

nullptr – a null pointer literal

`nullptr` is a literal denoting the null pointer; it is not an integer

Extras

Lambdas

A lambda expression is a mechanism for specifying a function object.

```
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a)<abs(b); });
```

A lambda expression can access local variables in the scope in which it is used. For example:

```
std::sort(indices.begin(), indices.end(), [&](int a, int b) { return v[a].name<v[b].name; });
```

- `[]` capture nothing
- `[&]` capture all by reference
- `[=]` all by value
- `[&v]` v by reference
- `[=v]` v by value

If an action is neither common nor simple, I recommend using a named function object or function.

```
struct Cmp_names {
    const vector<Record>& vr;
    Cmp_names(const vector<Record>& r) :vr(r) { }
    bool operator()(int a, int b) const { return vr[a].name<vr[b].name; }
};

// sort indices in the order determined by the name field of the records:
std::sort(indices.begin(), indices.end(), Cmp_names(v));
```

Delegating constructors

```
X() :X{42} { }
```

`decltype` – the type of an expression

```
typedef decltype(a[0]*b[0]) Tmp;
```

Suffix return type syntax

```
auto mul(T x, U y) -> decltype(x*y)
```

Raw string literals

`R"(...)"` broken into pieces means:

- `R` : prefix
- `" (` : sequence representing the open quotation
- `) "` : sequence representing the closing quotation

Inside the literal, a single `"` is not special. If you need to have `) "` in the string, you can change to sequences to for example `R"*** (...) ***"` by changing the opening and closing sequences by starting with `*** (` or anything else in between `"` and `(`, and repeating it for the closing sequence.

User-defined literals

The basic (implementation) idea is that after parsing what could be a literal, the compiler always check for a suffix

```
constexpr complex<double> operator "" i(long double d) // imaginary literal
{
    return {0,d}; // complex is a literal type
}
std::string operator ""s (const char* p, size_t n) // std::string literal
{
    return string(p,n); // requires free store allocation
}
f("Hello"); // pass pointer to const char*
f("Hello"s); // pass (5-character) string object
f("Hello\n"s); // pass (6-character) string object
auto z = 2+1i; // complex(2,1)
```

Explicit conversion operators

Adding a constructor to a previous type that converts from a new type by writing an operator inside the new type.

```
struct S { S(int) { } };

struct SS {
    int m;
```

```

    SS(int x) :m(x) { }
    explicit operator S() { return S(m); } // because S don't have S(SS)
};

SS ss(1);
S s1 = ss; // error; like an explicit constructor
S s2(ss); // ok ; like an explicit constructor
void f(S);
f(ss); // error; like an explicit constructor

```

control of defaults: default and delete

```

X& operator=(const X&) = delete/default; // Disallow copying / Default copying
X(const X&) = delete/default;

```

`std::function` and `std::bind`


Generalized function pointers

```

int f(int, char, double);
auto frev = bind(f, _3, _2, _1); // reverse argument order
int x = frev(1.2, 'c', 7); // f(7, 'c', 1.2);

function<float (int x, int y)> f; // make a function object
struct int_div { // take something you can call using ()
    float operator()(int x, int y) const { return ((float)x)/y; };
};
f = int_div(); // assign
cout << f(5, 3) << endl; // call through the function object

```


0 Comments**Fartash****1 Login** ▼ **Recommend** **Share****Sort by Best** ▼

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Be the first to comment.

 **Subscribe** **Add Disqus to your site**

Add DisqusAdd

 **Disqus' Privacy Policy**

Privacy PolicyPrivacy