C++ CheatSheet for NumCSE

Henrik Laxhuber

For proper, exhaustive C++ documentation, look no further than `cppreference.com`, or `devdocs.io` for a nice interface and offline storage&search.

1 Basics

Variable Declaration and Initialization

```
double foo; // foo now has _some_ (any) value

double foo = 0; // declare foo and assign 0

type name(args...); // declare name of type type and initialize
                    // it by invoking a constructor
Matrix A(rows, cols); // example

auto foo = Matrix(rows, cols); // let the compiler infer the
                               // type of foo. Don't do this
                               // (the type will often not be
                               // what you think it is),
                               // unless you really have to.

Matrix A = {rows, cols}; // initializer list, see below
```

The *initializer list* should be a bit surprising. From what we learned, the syntax `type name = expression;` should declare the variable `name` of type `type` and assign to it the value of `expression`. But what the heck is `{rows, cols}`? Is that even an expression?

The answer is yes, any expression of the form `{expr [, expr]*}` is called an *initializer list*. The initalizer list is a type itself, but will be implicitly converted to whatever we want. C++ does a lot of magic to support this implicit conversion by figuring out which constructor to call, or by creating an instance of a struct and initializing its members directly. In this case, because we assign the initializer list to a variable of type `Matrix`, the compiler figures out how to instantiate a Matrix using the provided arguments in the initializer list.

This is why initializer lists are cool:

```
struct S {
  int a;
  char b;
};
void foo(S const& bar) {
  for (int i = 0; i < bar.a; ++i) std::cout << bar.b;
  std::cout << std::endl;
}

int main() {
  foo({4, 'a'}); // cool! no explicit construction of an S!
}
```

References You might know pointers from C. C is a subset of C++, so C++ also has pointers, but there should be almost no reason to use them.

```
int i = 0, j = 1; // declare two ints
int& ref = i; // declare ref as reference to i
ref -= 1;
assert(i == -1);
ref = j;
ref += 1;
assert(i == 0 && i == ref && j == 1); // huh?
```

You cannot reassign the object that a reference points to. Hence, `ref = j` does *not* make ref point to `j`, but instead assigns to `i` the value of `j`. `ref` still points to `i`!

Loops C++ has the notion of "iterators":

```
std::vector<int> a = {1, 2, 3, 4, 5, 6};
for (int v : a) { // nice
  std::cout << v << ‘‘ ’’;
}
std::cout << std::endl;
```

The conventional C-style syntax `for(init; test; increment)` is also used.

Overloading

```
bool true_if_int(bool arg) { return false; }
bool true_if_int(int arg) { return true; }
int main() { assert(true_if_int((int) 1) && !true_if_int(false)); }
```

Note that overloading is a source for long compiler errors: if you try to call an "unavailable overload", the compiler (at least gcc) will helpfully tell you every overload that it tried, which can be a bit overwhelming.

Namespaces

```
using namespace Eigen; // bring name into scope
namespace eig = Eigen; // alias eig to Eigen
namespace MyNamespace {
  // members
}
std::cout // refers to member called 'cout' in namespace std
```

Class Definition

```
// 'class' is same as 'struct' except in class, all members are
// private unless specified otherwise.
class Matrix {
    public: // all public members:
    // constructor
    Matrix(size_t r, size_t c)
        : rows(r), cols(c) // 'initialization list', copies
                           // values into members. Only way
                           // to initialize const members.
    {
        // body; do stuff
    }
    // destructor, optional
    ~Matrix() {
        // do stuff
    }

    /* special constructors, optional
        * (auto-generated if not provided)
        */

    // default constructor
    Matrix() {
        // initialize to some 'default' value
    }

    // copy constructor
    Matrix(Matrix const& other) {
        // make this a copy of other
    }

    // move constructor
    Matrix(Matrix &&other) {
        // 'move' data from other to this. Can leave other in
        // some empty but valid state. Faster than copying other
        // into this when other is dropped afterwards anyway.
        //
        // whenever you see && in a type, this means 'construct
        // the thing using its move constructor'. E.g.
        //    void myfunc(Eigen::MatrixXd const&& mat)
        // means 'take the argument mat by stealing it from the
        // caller, who will be left with an empty matrix or so'
    }

    private:
    // all private members. Honestly, just make everything public.
    size_t rows, cols;
};
```

2 Lambda Expression

A *lambda expression* allows you to construct a function directly in the source code and *assign it [the function] to a variable.* Because why not! Using functions as objects is very useful, you will see much more of this in the FMFP course.

Basic Syntax (explanation follows):

```
auto const&& foo = [capture-list](args) -> ret-type {
  body
}
```

Example:

```
size_t i = 0;
double const d = 2.0;
auto const&& predicate = [d, &i](double const& v) -> bool {
  i += 1;
  d += 1;
  return v == d;
}
assert(predicate(3) == true);
```

Capture List Bring stuff from the surrounding scope (i.e. the scope where foo is declared) into the scope of the lambda body. The syntax `&name` means "bring `name` into scope *by reference*". The syntax `name` of course then means "bring `name` into scope *by copying*". There is a special syntax `[&](args) -> ... {...}` which means "bring everything into scope by reference".

Advanced use: If the lambda lives longer than the scope it was declared in, e.g. if it is returned from a function, make sure that any referenced captures live long enough as well!

`auto`-what? `auto` instructs the compiler to infer the type of the lambda. This is necessary, because the type of the lambda itself is generated by the compiler. To understand this, see the section on lambda implementation below.

Lambda Implementation (unimportant)

What is a lambda under the hood? It turns out that lambdas are just nice syntax for (i) generating a new struct/class with an overload for `operator()(args...)`, (ii) instantiating the struct by copying the captures into the struct members. Here's what the compiler is doing[1] in the `predicate` example:

```
// in ``sneaky_compiler_generated_stuff.hpp''
struct PredicateLambdaStructAutoGeneratedBlaBlaLongName {
  size_t& i;
  double d;

  PredicateLambdaStructAutoGeneratedBlaBlaLongName
    (size_t &i, double const& d) : i(i), d(d) {}
```

[1]at least you can think of it like that.

```
  bool operator()(double const& v) {
    i += 1;
    d += 1;
    return v == d;
  }
};
// then in the code we saw:
size_t i = 0;
double const d = 2.0;
PredicateLambdaStructAutoGeneratedBlaBlaLongName predicate(i, d);
assert(predicate(3) == true);
```

3 Templates

What if we want to write a function that takes a lambda? The solution is to make the function take *any* type, using generics. Generics in C++ can be achieved using templates, which are strictly much more powerful than cheap java generics, but that doesn't matter here.

```
  template<typename T, typename P> // T, P are 'template argument'
                                   // whose "type" (kind) is a type itself.
std::vector<T> filter(std::vector<T> const& v, P const& predicate) {
  std::vector<T> result;
  result.reserve(v.size()); // allocate space. Does not change result.size()

  for (size_t i = 0; i < v.size(); ++i) {
    if (predicate(v[i])) {
      result.push_back(v[i]);
    }
  }
  return result;
}
```

Note `template<class T>` is an alternative syntax for exactly the same thing. Why do we need to say that `T` is a `typename`? Because a template can take any kind of argument, we could also template over an `int` etc.

Also note that templates can be overloaded, e.g. we could also have a function template `filter` taking 3 different arguments etc. This can be a frequent source of veeeeeery looooooooong compiler messages, especially in advanced C++ code like Eigen or the standard library (historically sometimes called the *STL*, the standard *template* library), and even more so when combined with function overloading. The compiler will tell you every overload that it tried. It can help to scroll to the top of the error message, see where the error is, think about what you though the compiler would do, and then go read through what the compiler tried to do.