C++ tutorial

I will assume that you know some basics of C++:

```
#include <iostream>

int main(void) {
    std::cout << "Hello world!" << std::endl;
}
```

Introduction II

I hope that, by the end, you can tell what this does:

```
#include <iostream>
#include <vector>
template <typename T> struct hello {
    T operator*() { return "Hello"; }
    static const char e = '!';
    template <char c> char put_char() { return c; }
};
int main(void) {
    std::vector<char>  world = {'w','o','r','l','d'};
    hello<std::string> h;
    std::cout << *h << h.put_char<' '>();
    for(auto & word : world) { std::cout << word; }
    std::cout << hello<std::string>::e << "\n";
}
```

Contents

- Terminology
- Object Oriented Programming
- Template metaprogramming
- STL containers
- C++11
- Eigen

Some terminology

- Keywords: reserved syntactical elements
  e.g. `template`, `class`, `return` ...
- Build-in types: reserved for basic types
  e.g. `int`, `char`, `double`, ...
- Operators: basically functions but "better looking"
  *, +, &, :: , ?:, ...

Terminology: `lvalue` and `rvalue`

Variables can be placed in two categories.
We differentiate[1] between a `lvalue` and a `rvalue`:

- `lvalue` can be seen as a physical memory object;
- `rvalue` is just a temporary value.

Rule of thumb: can be put it on the left of `=` ?
Then `lvalue`, otherwise `rvalue`.

```
int incr(int a) { return a+1; }
int a;
a = 9; // a is lvalue, 9 is rvalue
9 = a; // invalid!
incr(a) = 3; // invalid, f(a) is rvalue
a = incr(a); // valid
```

[1] Actually things are a bit more complicated than this...

Passing by value and by reference

Consider a generic function:

```
void function(MatrixXd A);
```

All data from `A` copied to a new temporary.
We say that `A` is **passed by value**.

Passing by value and by reference

Consider a generic function:

```
void function(MatrixXd & A);
```

Now, `A` can be modified inside `function`.
We say that `A` is **passed by reference**.
We do not want that.

Passing by value and by reference

Consider a generic function:

```
void function(const MatrixXd & A);
```

Now, `A` cannot be modified inside `function`.
No copy is performed.
We say that `A` is **passed by constant reference**.
Passing by const. reference is very common in C++.
Remark: not needed for built-ins (e.g. `int`, `double`, ...).
Remark: can also return by reference, but it is dangerous.

Function and operator overload I

Let's say that I have functions:

```
int Id(int a) { std::cout << "int"; return a; }
int Id(double a) { std::cout << "double"; return a; }
int Id() { std::cout << "void"; return 1; }
```

This is valid in C++ and called **function overloading**.

```
Id(1); // prints "int"
Id(1.); // prints "double"
Id(); // prints "void"
```

Compiler must be able to pick the "best fit":

```
// error: ambiguating new declaration:
double Id() { return 1; }
double Id(int a = 1) { return a; }
```

Function and operator overload II

Most commonly used with operators.
Assume Complex is a type for complex numbers[2]:

```
bool operator<(Complex a, Complex b) {
  return a.real() < b.real();
}
```

overloads a lot of built-in operator< (undefined for $\mathbb{C}$).

```
Complex(3,4) < Complex(3,4); // now makes sense
```

Could also define a weird operator<:

```
   std::string operator<(Complex a, bool b);
```

$\Rightarrow$ completely new syntactical constructs.

[2] Here define Complex to be std::complex<double>, cf. later.

Pointers

Pointers refer to a specific memory location:

```
int * a; // ptr. to int, undefined memory location
int * b = new int; // ptr. to allocated int
a = b; // a points to the same as b
*a = 9; // value at the memory pointed by a is 9
std::cout << *b; // *b = 9
int c;
a = &c; // a points to the address of c
delete b;
```

Four important operators:

- `*` (unary) return value of pointer (dereference);
- `&` (unary) return address of variable (reference);
- `new` and `delete` allocate/deallocate memory for pointers.

Rule: for each `new` there must be a `delete`.

Remark: use pointers with extreme care.

Casting

Why does this make sense?

```
double i = 9; // <=> double i = (double) 9.
```

The `int 9` is *casted* implicitly to a `double`.
Most of the times you do not have to worry about it.
If you **really need** you can use the operators:
`static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast`.

Namespaces

Namespaces are like boxes with labels.

- Access with `::` *operator*
- or import them with the *keyword* `using`.

```
namespace MySpace {
  int i = 9;
}
std::cout << i; // error: 'i' was not declared ...
std::cout << MySpace::i << "\n"; // ok
using namespace MySpace;
std::cout << i << "\n"; // now it works ok
int i = 7;
std::cout << i << "\n"; // i = 7 here
```

`Eigen` and `std` are examples of namespaces.

Terminology: if it contains `::` is called `qualified`.

Keyword: typedef

This keyword allows to rename types for convenience:

```
typedef std::vector<double> custom_vector;
custom_vector cv; // cv is std::vector<double>
```

or, with using (in C++11):

```
using custom_vector = std::vector<double> ;
custom_vector cv1; // cv is std::vector<double>
```

using can be used with templates.

OOP: struct and class in short

Both struct and class keywords to define an *Object*.
Mainly, an Object is a collection of:

- other Objects or types (called members);
- functions (methods).

```
struct myStruct {
    int      some_int;
    double   some_double;
    void     incr() { some_int++; } // function
};
class myClass {
    int          I;
    myStruct     S; // contains a struct
};
```

OOP: constructors

Constructors: called to build an Object

```
class myClass {
  public:
    myClass() { I = 9; }; // default constructor
    myClass(int I_) : I(I_) {} // any constructor
    myClass(const myClass & other) // copy constructor
      : I(other.I) {} // special syntax I = other.I
    int          I;
};
```

The `:  I(I_)` is like writing `I = I_;` in the first line. Usage:

```
myClass m; // default construct
myClass n(1); // construct with ints
myClass o(n); // copy m to o
myClass p(); // NOT what you think
int a(); // function declaration like this
```

OOP: operators

Operators: a fancier way to define special function

```
class myClass {
  int operator*(const myClass & other) const {
    return 5*I + other.I;
  }
  int weird_mult(const myClass & other) const {
    return 5*I + other.I;
  }
}
```

Calling sequence:

```
myClass m, n;
int ret = m * n; // a custom uncommon operator
int ret2 = m.weird_mult(n); // is exactly the same
```

OOP Keyword: this

Inside a class, this is a pointer to the class itself.

```
struct T {
  int i = 9; // syntax sugar to initialize i = 9
  T & operator=(const T & other) {
    this->i = other.i; // this not required (implicit)
    // this->i <=> *this.
    return *this;
  }
};
```

Then I can write:

```
  T e1, e2, e3;
  e1 = e2 = e3; // because (e2 = e3) returns e2
```

OOP: class Namespaces

Each class and struct comes with a "free" namespace.
Functions, other objects and statics members are included in it.

```
struct N {
  class N_c { // nested class
    // stuff ...
  };
  typedef double N_t;
  static int N_i;
  void N_f(); // function declaration
};
```

They are accessed with `::`:

```
N::N_c n_c; // <=> n_c is a class N_c
N::N_t n_d; // <=> double N_d
N::N_i = 9; // variable of type int
void N::N_f() { ++N_i; } // definition of function N_f
```

OOP: const, some special syntax

A const member function promises not to modify the class:

```
struct myClass {
    int nonconst_function() {
        // something
        I = 8; // fine
    }
    int const_function() const {
        // something
        I = 8; // is illegal: modifies myClass
    }
};
```

Then you can use this function also on constant classes:

```
const myClass mc;
mc.nonconst_function(); // illegal
mc.const_function(); // legal
```

OOP: static members

A static member is shared amongst all instances:

```
class myClass {
  static int static_I;
  void incr() { static_I++; }
}
```

A static member is linked more to myClass than to m or n.
No need for an instance of myClass to access static_I.

```
int myClass::static_I = 9;
int main() {
    myClass m,n;
    std::cout << m.static_I << n.static_I; // 9  9
    m.incr();
    std::cout << m.static_I << n.static_I; // 10 10
}
```

OOP: static functions

A static member function is shared amongst all instances:

```
class myClass {
  static int function() {
      int a = static_I; // valid
      int b = I; // is illegal: cannot see non static
      return 8;
  }
}
```

No need for an instance of myClass to use function:

```
int y = myClass::function();
```

OOP: member access: private, protected and public

Until now, we never worried about the "privacy" of class members. Each member of an Object has a "privacy setting":

```
class C {
  // Default: private, accessed only within class
  int a; // C.a invalid outside
public: // can be accessed only by derived class
  int b; // C.b allowed only in child
protected: // can be accessed by anyone
  int c; // C.c always valid
}
```

Finally, we can tell the difference between struct and class:

- struct: members are public by default;
- class: members are private by default.

Template metaprogramming

The idea behind templates is: let the compiler write code for you.
Idea: write function and classes with generic types.
Templates are resolved at compile time

Function templates

I want a function:

```
double AXPY(double X, double Y, double A)
  { return A*X+Y; }
```

What if I want the same for `int`? `complex`? `MatrixXd`?
Define a "skeleton" function:

```
template <class whatever>
whatever AXPY(const whatever & X,
              const whatever & Y, double A)
              { return A*X+Y; }
```

as long as `*` and `+` are well defined for `whatever`.
The compiler defines the functions for us.
Remark: `template <typename whatever>` exactly the same.

Function templates: example I

We define the templated function myMin:
taking 2 arguments and returning the minimum.

```
template <class T1, class T2> // s.t. can mix T1/T2
T1 myMin(const T1 & a, const T2 & b) {
  return a < b ? a : b; // ternary operator
}

int a = myMin(8.,9.); // T1 = T2 = double
double b = myMin(8,9); // T1 = T2 = int
double c = myMin(8,9.); // T1 = int, T2 = double
// a = b = c = 8
```

Remark: myMin uses operator< from int and double.
Remark: operator expr1 ? expr2 : expr3 returns:
expr2 if expr1 otherwise returns expr3

Function templates: example II

This way, it can be used with custom types:

```
struct S {
  int i;
  S(int i) : i(i) { }; // constructor
  operator int() const { return i; } // S to int
  bool operator<(const S & other) const {
    return i > other.i; // swap < for >
  }
};

// Uses operator< for S, which is actually >
double d = myMin(S(8),S(9)); // d = 9
```

Class template

Just as with functions, we can template any class:

```
template <class T>
struct myComplex {
  T Re, Im; // real and imaginary part

  myComplex(T Re, T Im) : Re(Re), Im(Im) {};
};
```

Which has to be instantiated by specifying the type:

```
myComplex<int>  cmplx1(8,9); // cmplx = 8*i+9
myComplex<double>  cmplx2(2.3,3); // cmplx = 2.3*i+3
```

You can then go crazy and template everything:

```
template <class T1, class T2 = int> // T2 default int
class myComplex2 {
  T1 Re; // real part has type T1
  T2 Im; // imaginary part part has type T2
public:
  template <class U>
  U sum() { return Re + Im; }
};
```

Usage involve lots of <>:

```
myComplex2<double> cmplx3; // Re = double, Im = int
myComplex2<double, double> cmplx4; // Re = Im = double

cmplx3.sum<int>(); // must specify U above
double s1 = cmplx3.sum<double>(); // must specify U
```

Keyword: typename alternative meaning

```
template <class T>
void templated_function() {
   T::t * x; // multiplication or pointer def.?
}
```

What does this mean?

Is `T::t` a type or a (static) variable?

Keyword: typename alternative meaning

```
template <class T>
void templated_function() {
   T::t * x; // C++: multiplication
}
```

C++ interprets `T::t` and `x` as variables, `*` as multiplication, as if:

```
struct T {
  static int t; // t is a variable
};
```

Keyword: `typename` alternative meaning

```
template <class T>
void templated_function() {
  T::t * x; // C++: multiplication illegal!
}
```

But what if we meant `x` to be a pointer of type `T::t` instead?

```
struct T {
  class t { // t is a class name
    // stuff...
  };
};
```

Keyword: typename alternative meaning

```
template <class T>
void templated_function() {
   typename T::t * x; // now is a pointer def.
}
```

Keyword typename declares that a qualified name is a type.

```
struct T {
  class t { // t is a class name
    // stuff...
  };
};
```

Keyword: template alternative meaning

Besides the normal use `template <class T> ...`, there is a uncommon usage of the keyword template:

```
template <typename T>
struct tStruct {
    template <typename U>
    U tMemberFunc(void) { return U(); }
};

template <typename T>
void tFunc() {
    T s;
    int a = s.tMemberFunc<int>(); // error
}

int main() {
    tFunc<tStruct<int>>();
}
```

Keyword: template alternative meaning

You must tell the compiler that myMemberFunc is a template function and not something to compare $<$ with int?

```
template <typename T>
struct tStruct {
    template <typename U>
    U tMemberFunc(void) { return U(); }
};

template <typename T>
void tFunc() {
    T s;
    int a = s.template tMemberFunc<int>(); // OK
}

int main() {
    tFunc<tStruct<int>>();
}
```

Recap on templates

Templates are very useful:

- allow focusing on algorithms rather than types;
- allow writing efficient programs;
- reduce code duplication.

However, when using templates:

- your program may become unreadable;
- you could get very complicated errors;
- in general, it takes longer to compile.

STL containers

Containers let you store collections of elements (objects).
All have different underlying data structure and complexities.

name	descr.	push back	iterator insert	lookup
vector	dyn. array	O(1)[3]	O(n)	O(1)
list	linked list	O(1)	O(1)	O(n)
map	hash map	N/A	O(log(n))	O(log(n))

For instance:

- vector[4] is good for lookup;
- list is good for inserts;
- maps allows different type of indices.

⇒ choose the appropriate container for your needs

[3] amortized amongst many `push_back`.

[4] `std::vector` ≠ `Eigen::VectorXd`.

Example: std::vector

A std::vector is a dynamic array like

- a C like array with variable size or
- a Java ArrayList

Needs (at least) a template argument: type of item contained.
Usage:

```
#include <vector>
#include <complex>
std::vector<int> vec; // int vec., similar to int *
std::vector< std::complex<double> > cmplx_vec;
```

Guarantees data is sequential in memory (i.e. array), hence:

- changing size is expensive (reallocate everything);
- looking up an element is cheap.

Iterators I

Iterators are something that facilitates the iteration over a loop.
A container `c` of type `C` has:

- an iterator type `C::iterator`, pointing at an element of `C`:
  - `*it` return the element pointed by `it`;
  - `++it` return the iterator pointing to the next element of `c`;
- a `begin()` and an `end()` (both are interators `C::iterator`).

Instead of:

```
std::vector<int> v;
// ... fill v
for(int i = 0; i < v.size(); ++i) {
    v.at(i) = i+1;
    // loop over all v
}
```

Used to loop easily (and efficiently):

Iterators II

```
for(std::vector<int>::iterator it = v.begin(); it  !=
    v.end(); ++it) {
     *it = i+1;
     // loop over all v, *it is like v.at(i)
}
// range based alternative:
for(int & c: v) {
     c = i+1;
     // loop over all v, c is like v.at(i)
}
```

Keyword: decltype (C++11)

Copy and paste the type of a variable:

```
template <typename T>
struct declClass {
    T val;
};

declClass<int> m_i;
declClass<double> m_d;
decltype(m_i.val) i_type; // <=> int i_type;
decltype(m_d.val) d_type; // <=> double d_type;
```

Useful to infer the type of a variable automatically.

Keyword: auto (C++11)

In short: avoid specifying the type (let the **compiler decide**)

```
std::vector< Eigen::MatrixXd > some_function(int a);
```

instead of calling:

```
std::vector< Eigen::MatrixXd > v = some_function(8);
for(std::vector< Eigen::MatrixXd >::iterator it = v.begin();
    it < v.end(); ++it) {
  // some code
}
```

Remark: auto cannot be used everywhere.
Very useful, but do not abuse this feature.

Keyword: auto (C++11)

In short: avoid specifying the type (let the **compiler decide**)

```
std::vector< Eigen::MatrixXd > some_function(int a);
```

just call:

```
auto v = some_function(8);
for(auto it = v.begin();
     it < v.end(); ++it) {
  // some code
}
```

Remark: auto cannot be used everywhere.

Very useful, but do not abuse this feature.

Lambda functions (C++11)

Lambda functions are syntax sugar for C++.
Are unnamed functions similar to Matlab function handles.
Syntax:

```
[] (arguments) { function_body; }
```

return value is deducted by the compiler. If it can't:

```
[] (arguments) -> return_type { function_body; }
```

You can also store the lambda function in a variable:

```
#include <functional>
auto f = [] (int i) { return i++; }; // using auto
std::function<int (int)> g = [] (int i)
       { return i++; }; // manual type
```

Lambda functions (C++11): variable capture

Lambda functions can capture the outer scope:

```
int j = 9, i = 8;
[] () { std::cout << j; }; // illegal!
// legal, j captured by reference:
auto f = [&] () { std::cout << j; };
f();
[i,&j] () { i++; j++ }; // illegal for i, legal for j
```

Inside `[]` you can put many "capture modes":

- `[]`: doesn't perform any capture;
- `[&]`: captures any variable by reference;
- `[=]`: captures any variable by making a copy;
- `[a,&b]`: captures `a` by copy, `b` by reference;
- `[this]` captures pointer to `this`;

Aliasing

What is wrong with the following code?

```
Eigen::VectorXd A(4);
A << 1,2,3,4;
A.tail(2) = A.head(2);
std::cout << A;
```

Should print:

```
1,1,2,3
```

instead prints

```
1,1,1,3
```

Sometimes Eigen warns you about that, but not always:

```
A = A.transpose(); // aborts during execution
```

Aliasing

What is wrong with the following code?

```
Eigen::VectorXd A(4);
A << 1,2,3,4;
A[3] = A[2]; A[1] = A[0]; A[2] = A[1];
std::cout << A;
```

Should print:

```
1,1,2,3
```

instead prints

```
1,1,1,3
```

Answer: *aliasing*, Eigen overrides the values before it used them. Sometimes Eigen warns you about that, but not always:

```
A = A.transpose(); // aborts during execution
```

Alignment issues in Eigen

A word of notice if you use **fixed size** Eigen Matrix or Vectors inside:

- structs;
- STL containers.

Alignment issues in Eigen mean that you could get **wrong** code.
Read: `http://eigen.tuxfamily.org/dox/group__DenseMatrixManipulation__Alignement.html`.

Debugging C++

To debug your cose you can use two tools:

- `std::cout` is always fine;
- gdb (or any other debugger): run a program with

```
gdb ./myprogram
```

  and you have step by step execution.

Remark: you should always test your code for correctness.

Efficient loops I

What is the difference between:

```
Eigen::MatrixXd A(n,n); // A is Col. Major format
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < n; ++j) {
    A(i,j) = i*j;
  }
}
```

and

```
for(int j = 0; j < n; ++j) {
  for(int i = 0; i < n; ++i) {
    A(i,j) = i*j;
  }
}
```

Efficient loops II

The second: it is **not jumping** all over the memory.

i.e. it has better memory locality

$\Rightarrow$ better use of the cache

$\Rightarrow$ faster execution

Rule: fastest index should loop over the closest memory locations.

Final remarks

Some final remark:

- The compiler is smarter than you think.
- Most of the time is also smarter than you.
- Think of cleanliness and correctness before efficiency.
- Do not avoid for loops: they are not evil as in Matlab.

Further help

- Eigen doc:
  http://eigen.tuxfamily.org/
- Matlab/Eigen dictionary:
  http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt
- C++ reference:
  - http://www.cplusplus.com/
  - http://en.cppreference.com/