Essentials of C++ for numerical methods

alexander.dabrowski@sam.math.ethz.ch

Numerical methods for CSE, ETH Zürich

September 19, 2018
Goals

To pass the exam you should:

(1) understand the numerical methods presented during the lectures: why they work, their speed (in an asymptotic complexity sense), strengths and weaknesses;

(2) be able to write C++/Eigen algorithms to solve problems similar to the ones in the exercise classes.

For (2) practice is essential.

Disclaimer: this tutorial is a non-comprehensive introduction, focused on how to use C++ to solve this course’s exercises. If you’re new to C++, we recommend you first skim through A tour of C++ by Bjarne Stroustrup (1st or 2nd edition).
Goals

To pass the exam you should:

(1) understand the numerical methods presented during the lectures: why they work, their speed (in an asymptotic complexity sense), strengths and weaknesses;

(2) be able to write C++/Eigen algorithms to solve problems similar to the ones in the exercise classes.

For (2) practice is essential.

Disclaimer: this tutorial is a non-comprehensive introduction, focused on how to use C++ to solve this course’s exercises. If you’re new to C++, we recommend you first skim through *A tour of C++* by Bjarne Stroustrup (1st or 2nd edition).
Topics of this tutorial

1. Dissection of a “Hello World!” program
2. Some fundamental types
3. Life of a local variable
4. Functions
5. A picture for the RAM and the operator&
6. Pointers
7. Passing variables to functions
8. Classes
9. Templates
10. Essentials from the STL
11. Useful C++11 features: auto, lambdas, range-for
12. On dynamic memory management
13. Eigen tutorial
Dissection of a “Hello World!” program

What does each line mean/do in the following codes?

```cpp
// A simple hello world program
#include <iostream>
int main() {
    std::cout << "Hello World!\n";
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << "Hello World!" << endl;
}
```
Dissection of a “Hello World!” program

```
// A simple hello world program

#include <iostream>

int main() {
    std::cout << "Hello World!\n";
    return 0;
}

A successful program returns 0
Dissection of a “Hello World!” program

A precompiled header which contains all the functions from the standard library which we need (and more). Warning! This is non-standard and non-portable (works only with the gcc compiler). Is considered as bad practice in software engineering; however you are allowed to use it in the exercises to save time.

#include<bits/stdc++.h>

If the compiler doesn't find a name, tries to add "std::" in front of it. Warning! This is considered bad practice (as you might shadow your own functions and introduce bugs); however you are allowed to use it.

using namespace std;

int main() {

    cout << "Hello World!" << endl;
    "return 0;" is automatically inserted at the end of main() and can be safely omitted
}

Essentially equivalent to "\n"
Some fundamental types

The fundamental types we will use most often are:

- **int**, an integer between INT_MIN and INT_MAX (on most architectures today, INT_MIN = $-2^{31}$ and INT_MAX = $2^{31} - 1$);
- **double**, a floating point number (with 11 bits to represent the exponent, 53 bits for the significant digits);
- **char**, an ASCII character (which usually occupies 1 byte);
- **bool**, a boolean which can take only the values true or false.

When we append "[]" to a type T we are referring to an array of Ts, that is a sequence of elements of type T which are contiguous in memory. For example the following code

```c
bool x[5];
x[4] = 1;
```

allocates an array of 5 contiguous boolean values and sets the last one to true (the first 4 values remain undefined, and the compiler might not initialize them at all!).
Some fundamental types

The fundamental types we will use most often are:

- **int**, an integer between INT_MIN and INT_MAX (on most architectures today, INT_MIN = −2^{31} and INT_MAX = 2^{31} − 1);
- **double**, a floating point number (with 11 bits to represent the exponent, 53 bits for the significant digits);
- **char**, an ASCII character (which usually occupies 1 byte);
- **bool**, a boolean which can take only the values **true** or **false**.

When we append “[]” to a type T we are referring to an **array** of Ts, that is a sequence of elements of type T which are contiguous in memory. For example the following code

```plaintext
1  bool x[5];
2  x[4] = 1;
```

allocates an array of 5 contiguous boolean values and sets the last one to **true** (the first 4 values remain undefined, and the compiler might not initialize them at all!).
Life of a local variable

Consider the following code

```java
int main() {
    int x;
    x = 3;
}
```

It allocates memory for a variable called `x` and it sets it to the value 3. The variable `x` lives only in the **scope** (the space between the innermost curly braces) where it was declared. For example, the following code will fail at line 5, because after line 4 the variable `x` does not exist anymore.

```java
int main() {
    {
        int x;
        x = 3;
    }
    x = 3;
}
```
Functions

Usually an exercise will ask you to implement a function. Here is an example of a function which calculates the factorial of a non-negative integer.

```cpp
int f(int n) {
    if (n <= 1)
        return 1;
    return n * f(n-1);
}
```

Line 1 is called signature of the function; it starts with the type it returns, then its name, then in parentheses the arguments it takes.

Any function in C++ can be overloaded by changing the signature in such a way that the compiler can distinguish which function it should run. For instance if we define `double f(double n)` with the same body in the previous lines 2-4 and run `f(3.4)`, the compiler knows that it should run the latter version.

Since operators like `+`, `<<`, ... are functions, they can be overloaded too.
Functions

Usually an exercise will ask you to implement a function. Here is an example of a function which calculates the factorial of a non-negative integer.

```c
int f(int n) {
    if (n <= 1)
        return 1;
    return n * f(n-1);
}
```

Line 1 is called signature of the function; it starts with the type it returns, then its name, then in parentheses the arguments it takes.

Any function in C++ can be overloaded by changing the signature in such a way that the compiler can distinguish which function it should run. For instance if we define `double f(double n)` with the same body in the previous lines 2-4 and run `f(3.4)`, the compiler knows that it should run the latter version.

Since operators like +, <<, ... are functions, they can be overloaded too.
A picture for the RAM and the operator &

It is sometimes useful to have a schematic picture of the Random Access Memory as a sequence of named cells which contain a number.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

When we want to obtain the address of a variable we use the reference operator &.

For example when we execute

```plaintext
1 int x;
2 x = 3;
```

the cell 2 might be allocated to `x` and its value set to 3. In this case `&x` would return 2.
A picture for the RAM and the operator &

It is sometimes useful to have a schematic picture of the Random Access Memory as a sequence of named cells which contain a number.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

When we want to obtain the address of a variable we use the `reference` operator &.

For example when we execute

```c
1 int x;
2 x = 3;
```

the cell 2 might be allocated to x and its value set to 3. In this case &x would return 2.
Pointers

A memory cell might contain the address of another cell. This is the case of a **pointer**. The syntax to declare a pointer in C++ is `T* x` (or equivalently `T *x`), where `T` is the type of the variable `x` points to. To access the value `x` points to, we can use the **dereference** operator `*`.

How does the following program manage the memory and what does it print?

```
1    char t = 'a';
2    char *y;
3    y = &t;
4    cout << *y;
```

In line 1 we allocate some memory for `t` (for example at address 2), and set it to the ASCII value of ‘a’. In line 2 we allocate memory for a pointer to a `char` (for example at address 4), and in line 3 we set its value to the address of `t`. Finally we print the value of the variable `y` points to, that is ‘a’.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0</td>
<td>97</td>
<td>0</td>
<td>2</td>
<td>…</td>
</tr>
</tbody>
</table>
Pointers

A memory cell might contain the address of another cell. This is the case of a **pointer**. The syntax to declare a pointer in C++ is T* x (or equivalently T *x), where T is the type of the variable x points to. To access the value x points to, we can use the dereference operator *.

How does the following program manage the memory and what does it print?

```
1 char t = 'a';
2 char *y;
3 y = &t;
4 cout << *y;
```

In line 1 we allocate some memory for t (for example at address 2), and set it to the ASCII value of ‘a’. In line 2 we allocate memory for a pointer to a char (for example at address 4), and in line 3 we set its value to the address of t. Finally we print the value of the variable y points to, that is ‘a’.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0</td>
<td>97</td>
<td>0</td>
<td>2</td>
<td>...</td>
</tr>
</tbody>
</table>
Passing variables to functions

A variable \( x \) can be passed to a function \( f \) by **value**, that is the value of \( x \) is copied into a new memory location with which \( f \) works, or by **reference**, that is we pass the address of \( x \) and \( f \) works directly with \( x \).

For example consider the following program

```plaintext
void f1(int x) { x++; }
void f2(int &x) { x++; }

int main() {
    int a = 7, b = 7;
    f1(a);
    f2(b);
    cout << a << " " << b;
}
```

It will print “7 8”, since \( f1 \) works on a copy of \( a \), while \( f2 \) works directly on \( b \).
Passing variables to functions

A variable \( x \) can be passed to a function \( f \) by **value**, that is the value of \( x \) is copied into a new memory location with which \( f \) works, or by **reference**, that is we pass the address of \( x \) and \( f \) works directly with \( x \).

For example consider the following program

```c
void f1(int x) { x++; }
void f2(int &x) { x++; }

int main() {
    int a = 7, b = 7;
    f1(a);
    f2(b);
    cout << a << " " << b;
}
```

It will print “7 8”, since \( f1 \) works on a copy of \( a \), while \( f2 \) works directly on \( b \).
Classes collect variables and functions in a convenient way and give the possibility to create new objects. To access members of a class use the dot operator.

For example, the following class represents a Point in the plane and permits to calculate its squared euclidean distance from the origin by calling `p.squared_norm()`.

```cpp
class Point {
    double x, y;

public:
    Point(double _x, double _y){ // constructor
        x = _x;
        y = _y;
    }
    double squared_norm() {
        return x*x + y*y;
    }
};
```
**Templates**

Templates are a very powerful idea which allows to re-use the same code for different types. For example we can define a template function to calculate the squared norm of any pair of objects (for which multiplication and addition are defined) as follows:

```cpp
template<class T> T squared_norm(T x, T y){
    return x*x + y*y;
}
```

Similarly one can also define template classes:

```cpp
template<class T> class Point {
    T x, y;
public:
    Point(T x, T y){ // constructor
        ...
    }
}
```

If then we want to instantiate an object of this class we must explicitly specify its template type; for example if we want it to be a `double` we will call `Point<double> p;`.
Essentials from the STL

The Standard Template Library contains many useful collections and algorithms. We recall here only the two most important ones.

- **std::vector** is an array which resizes dynamically, guaranteeing $O(1)$ access time and amortized-$O(1)$ appending of elements. The method `size()` returns the number of elements.
- **std::sort** sorts in $O(n \log n)$ any iterable collection in a range given by two iterators.

Consider the following code:

```cpp
#include<vector>
#include<algorithm>
...
vector<int> v;
for (int i=0; i<n; i++)
    v.push_back(rand()); // append a random int to v
sort(v.begin(), v.end()); // sort all of v
```

Lines 5-6 run in $O(n)$ and line 7 runs in $O(n \log n)$.
The Standard Template Library contains many useful collections and algorithms. We recall here only the two most important ones.

- **std::vector** is an array which resizes dynamically, guaranteeing $O(1)$ access time and amortized-$O(1)$ appending of elements. The method `size()` returns the number of elements.

- **std::sort** sorts in $O(n \log n)$ any iterable collection in a range given by two iterators.

Consider the following code:

```cpp
#include<vector>
#include<algorithm>
...
vector<int> v;
for (int i=0; i<n; i++)
    v.push_back(rand()); // append a random int to v
sort(v.begin(), v.end()); // sort all of v
```

Lines 5-6 run in $O(n)$ and line 7 runs in $O(n \log n)$. 
Useful C++11 features

- The compiler can deduce automatically the type of a returned variable with the `auto` keyword. For instance, you can write `auto x = 3` instead of `int x = 3`.

- **Lambdas** are anonymous functions which can be defined anywhere. For instance, you can sort in reverse order a `vector<int>` `v` by passing a custom comparator as follows:

```cpp
sort(v.begin(), v.end(),
    [] (int x, int y) { return x > y; } );
```

- You can iterate directly on the elements of an iterable collection `v` with a range-for. For example, you can print in order all the elements of a vector as follows:

```cpp
for (auto el : v) {
    cout << el << " ";
}
```
Useful C++11 features

• The compiler can deduce automatically the type of a returned variable with the `auto` keyword. For instance, you can write `auto x = 3` instead of `int x = 3`.

• **Lambdas** are anonymous functions which can be defined anywhere. For instance you can sort in reverse order a `vector<int> v` by passing a custom comparator as follows:

```cpp
sort(v.begin(), v.end(),
     [] (int x, int y) { return x > y; } );
```

• You can iterate directly on the elements of an iterable collection `v` with a range-for. For example you can print in order all the elements of a vector as follows:

```cpp
for (auto el : v) {
    cout << el << " ";
}
```
Useful C++11 features

• The compiler can deduce automatically the type of a returned variable with the `auto` keyword. For instance, you can write `auto x = 3` instead of `int x = 3`.

• **Lambdas** are anonymous functions which can be defined anywhere. For instance you can sort in reverse order a `vector<int> v` by passing a custom comparator as follows:

```cpp
sort(v.begin(), v.end(),
     [] (int x, int y) { return x > y; } );
```

• You can iterate directly on the elements of an iterable collection `v` with a range-for. For example you can print in order all the elements of a vector as follows:

```cpp
for (auto el : v) {
    cout << el << " ";
}
```
On dynamic memory management

The memory for local variables must be already known at compile-time. It might happen that you know how much memory a variable needs only at run-time (consider for example an array with variable number of elements). You can manage memory dynamically with the `new` and `delete` commands (`new[]` and `delete[]` for arrays).

Consider for example the following code:

```c++
int n;
cin >> n;
int *array;
array = new int[n];
delete[] array;
```

Line 2 asks the user to input an integer `n`, which in line 4 is used to allocate an array of `n` integers. Line 5 deallocates the memory used by the array, letting the operating system know that it is free to use.

Always prefer local variables or wrapper classes (such as `vector`). We will not manage memory directly with `new/delete` in this course.
On dynamic memory management

The memory for local variables must be already known at compile-time. It might happen that you know how much memory a variable needs only at run-time (consider for example an array with variable number of elements). You can manage memory dynamically with the `new` and `delete` commands (`new[]` and `delete[]` for arrays).

Consider for example the following code

```cpp
int n;
cin >> n;
int *array;
array = new int[n];
delete[] array;
```

Line 2 asks the user to input an integer \( n \), which in line 4 is used to allocate an array of \( n \) integers. Line 5 deallocates the memory used by the array, letting the operating system know that it is free to use.

Always prefer local variables or wrapper classes (such as `vector`). We will not manage memory directly with `new/delete` in this course.
On dynamic memory management

The memory for local variables must be already known at compile-time. It might happen that you know how much memory a variable needs only at run-time (consider for example an array with variable number of elements). You can manage memory dynamically with the `new` and `delete` commands (`new[]` and `delete[]` for arrays).

Consider for example the following code

```cpp
int n;
cin >> n;
int *array;
array = new int[n];
delete[] array;
```

Line 2 asks the user to input an integer `n`, which in line 4 is used to allocate an array of `n` integers. Line 5 deallocates the memory used by the array, letting the operating system know that it is free to use.

_ALWAYS prefer local variables or wrapper classes (such as `vector`). We will not manage memory directly with `new/delete` in this course._
Eigen tutorial

Eigen is a pure template library for fast linear algebra computations which we will use throughout the course. We strongly suggest you take a close look at the getting started page of the project as soon as possible, and follow the rest of the tutorial as you encounter new topics during the lectures.

http://eigen.tuxfamily.org/dox/GettingStarted.html