1 Mid-term preparation

1.1 Variables, types, scope, default initialization

A variable is a small area of memory which is associated to an **identifier** and a **type**. The **scope** of a variable (or other identifier) is the area of the source code where the variable can be referred to, most of the time the smallest enclosing {}. Note that a variable is **not initialized by default.**

```
1  #include <iostream>
2
3  int main(int argc, char **argv) {
4    int a;
5    a = a+1;                    // ouch!
6    int b = 3;                  // good
7    if(b == 3) { int b = 5; int c = 4; } // ouch!
8    cout << "b=" << b << '\n'; // here b = 3
9    cout << "c=" << c << '\n'; // here can't compile : out of scope
10 }
```

1.2 Variables, pointers, dynamic allocation

A pointer is an **address in memory**. Its type depends with the type of the variable it refers to. The * operator allow to denote not the pointer's value but the pointed variable's value. The `new` operator allows to create a variable of a given type and to get its address. The `delete` operator (resp. `delete[]`) indicates to the computer a variable (resp. array) located at a given address is not used anymore. A variable created with `new` is called a **dynamically allocated variable**, while a *normal* variable is called **static**. The [] operator allow to access either an element in a static or dynamically allocated array.

```
1  #include <iostream>
2
3  double *definitelyStupid() {
4    double a[10];
5    return a;     // ouch !!! *NEVER* do that!!!
6  }
7
8  int main(int argc, char **argv) {
9    double *a, *b;
10   a = definitelyStupid();
11   delete[] a;                          // ouch!
12   b = new double[10];
13   for(int i = 1; i<100; i++) b[i] = i; // ouch!
14   double *c;
15   c[10] = 9.0                          // ouch!
16 }
```

1.3 Expressions, operators, implicit conversion, precedence

An expression is a sequence of one or more **operands**, and zero or more **operators**, that when combined, produce a value.

Operators are *most of the time* defined for two operands of same type. The compiler can automatically convert a numerical type into another one with no loss of precision, so that the operator exists.

Arithmetic computations can lead to **arithmetic exceptions**, either because the computation can not be done mathematically, or because the used type can not carry the resulting value. In that case the result is either a wrong value or a non-numerical value.

The precedence of operators is the order used to evaluate them during the evaluation of the complete expression. To be compliant with the usual mathematical notations, the evaluation is not left-to-right.