Basic rules for programming in C++

Professors of Programming 1

September 1st 2015

1 Preamble

These rules were originally written for the subject “Programming 1”, and have been applied (with changes) since 2006-2007, but they are applicable in any course that involves learning to program in C++. The original version, made by Salvador Roura, can be found in http:www.lsi.upc/~roura/normes.pdf.

The suggested style of programming in this document looks for a difficult commitment: On one hand, an introductory course should not become a workshop about the many instructions and constructions a modern programming language can offer. Which means that, the student must not end up being dependent of the first programming language that he learnt, and he should be able to adapt to other programming languages. On the other hand, students should learn from the very beginning to make easy what’s easy, which sometimes requires the use of specific constructions of the chosen programming language.

These rules were written after searching a lot of on-line material, in particular the one found in the websites referenced at the end of this document. Furthermore, the author tried to include simple and concise rules, minimizing as much as possible all kinds of exceptions. As a result, the programming style is “neutral”, in the sense that following the rules listed below can produce code (almost) indistinguishable from the one produced by excellent programmers with lots of experience.

There are many more motives that justify the existence of strict rules in a basic programming subject. For instance, a fixed programming style avoids the fact that students have to think how they will write their code, and start living bad habits that are hard to remove later on. Additionally, a consistent style helps reducing the errors. Furthermore, with a common style, students can understand better the solutions of their exercises, either from teachers or from other students, because the presentation will be similar to theirs. Similarly, teachers can understand better what students do (or try to do) in their codes.

Finally, experience in Programming 1 shows that many students used this programming style throughout their student life, and found it helpful when they had to make group projects, for example.
2 Some general considerations

(All the statements must be considered in the context of a programming learning subject, not universally.)

- Programs written in C++ have the extension .cc.

- Each line of code has a basic instruction.

Example:

```c
i = 0;
j = 1;
```

is better than

```c
i = 0; j = 1;
```

But

```c
i = j = 0;
```

is better than

```c
i = 0;
j = 0;
```

- A variable is declared when it’s needed, not before, and is usually initialized in its declaration. In this case, you have to declare it individually.

Example:

```c
// Returns the euclidean distance between the points(x1,y1) and (x2,y2).
double distance(double x1, double y1, double x2, double y2) {
    double x = x1 - x2;
    double y = y1 - y2;
    return sqrt(x*x + y*y);
}
```

is better than

```c
// Returns the euclidean distance between the points(x1,y1) and (x2,y2).
double distance(double x1, double y1, double x2, double y2) {
    double x, y;
    x = x1 - x2;
    y = y1 - y2;
    return sqrt(x*x + y*y);
}
```

and that

```c
// Returns the euclidean distance between the points(x1,y1) and (x2,y2).
double distance(double x1, double y1, double x2, double y2) {
    double x = x1 - x2, y = y1 - y2;
    return sqrt(x*x + y*y);
}
```
• Some typical situations where you don’t have to initialize a variable in its declaration are:

  – When reading
    Example:
    
    ```
    int x, y;
    cin >> x >> y;
    ```

  – When the value of the variable depends on a condition.
    Example:
    
    ```
    double seconds;
    if (degrees) seconds = 3600*angle;
    else seconds = 60*angle;
    ```

  – When the value of the variable depends on a loop.
    Example:
    
    ```
    // We want to add up all the numbers and find out the position of
    // any negative number. We know that there's at least one.
    int position_negative;
    double sum = 0;
    for (int i = 0; i < v.size(); ++i) {
      sum += v[i];
      if (v[i] < 0) position_negative = i;
    }
    ```

  – When the variable will be immediately used as a return parameter.
    Example:
    
    ```
    vector<double> v;
    read_vector(v);
    double sum, multiplication;
    // Calculates the sum and multiplication of all the elements of v.
    sum_and_multiply(v, sum, multiplication);
    ```

  – In some extreme cases like:
    
    ```
    int i1 = 0;
    int i2 = 0;
    int i3 = 0;
    int i4 = 0;
    int i5 = 0;
    int i6 = 0;
    int i7 = 0;
    int i8 = 0;
    ```

    Here, maybe it would be better to do:
    
    ```
    int i1, i2, i3, i4, i5, i6, i7, i8;
    i1 = i2 = i3 = i4 = i5 = i6 = i7 = i8 = 0;
    ```
Variables should be declared in the most intern visibility scope possible.

Example:

```cpp
int x, y;
cin >> x >> y;
if (x > y) {
    int aux = x;
    x = y;
    y = aux;
}
cout << "sorted elements: " << x << " " << y << endl;
```

is better (and also different) than

```cpp
int x, y, aux;
cin >> x >> y;
if (x > y) {
    aux = x;
    x = y;
    y = aux;
}
cout << "sorted elements: " << x << " " << y << endl;
```

In the second case, the auxiliary variable `aux` keeps existing after the printed message. That could create name collision, if later on we would like to use another variable with the same name `aux`.

The past rule has a very important case: the control variable of a `for`, when its content is not necessary anymore after the end of the loop. In this case, the declaration of a variable must be done inside the `for`.

Example:

```cpp
for (int i = 0; i < 10; ++i) cout << i*i << endl;
```

is better (and also different) than

```cpp
int i;
for (i = 0; i < 10; ++i) cout << i*i << endl;
```

Only in the second case the variable `i` keeps existing after the `for`. If some lines of code below we wanted to create another loop using the variable `i`, we should have to remember that it had already been declared, in order not to declare it again, generating a compilation error. Instead, this code compiles with no errors:

```cpp
for (int i = 0; i < 10; ++i) cout << i*i << endl;
for (int i = 0; i < 10; ++i) cout << i*i*i << endl;
```

You should avoid variables in more intern scopes hiding other variables in more extern scopes. Even if the code is correct, it’s still confusing.

Example:
// Prints an nxn square.
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) cout << "*";
    cout << endl;
}

is better than

// Prints an nxn square
for (int i = 0; i < n; ++i) {
    for (int i = 0; i < n; ++i) cout << "*";
    cout << endl;
}

• You should not use “magic numbers”. Instead, you have to use constants.

Example:

const int MAX_PEOPLE = 700;
...
if (n > MAX_PEOPLE/2) cout << "It’s quite crowded." << endl;

is better than

if (n > 350) cout << "It’s quite crowded." << endl;

• You have to use carefully comparisons between real numbers, specially if they’re the result of a series of operations, because small round off errors can make the program crash.

• It is better not to compare a boolean with true or with false.

Example:

if (found) ...
if (not repeated) ...

is better than

if (found == true) ...
if (repeated == false) ...

• Boolean expressions in C++ are evaluated left to right and stopping when the result is known. This can be useful to simplify your code.

Example:

// Returns the first position of v where x is found, or -1 if x is not found.
int position(double x, const vector<double>& v) {
    int i = 0;
    while (i < v.size() and v[i] != x) ++i;
    if (i < v.size()) return i;
    else return -1;
}
is as correct as

```cpp
// Returns the first position of v where x is found, or -1 if x is not found.
int position(double x, const vector<double>& v) {
    bool found = false;
    int i = 0;
    while (not found and i < v.size()) {
        if (v[i] == x) found = true;
        else ++i;
    }
    if (trobat) return i;
    else return -1;
}
```

• The advised formats of the instruction for are:

```cpp
for (t var = expr1; var < expr2; ++var) ...
for (t var = expr1; var > expr2; --var) ...
```

and the analogous with <= and >=, respectively, where var is the control variable, t is its type (that can be omitted if var is already declared) and expr1, expr2 are the expressions that don’t contain var.

△ It is prohibited to change the value of a control variable of a for inside its body.

• The instructions while and for may contain return instructions inside their body when this does not affect to their readability. There are some case where writing a return inside a loop is the easy way of programming, and produces a more efficient and readable code.

For example, this code

```cpp
// Returns the first position of v where x is found, or -1 if x is not found.
int position(double x, const vector<double>& v) {
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

is an alternative to the past two codes. The use of return is only accepted to stop a loop when it’s clearly better to use it than not to use it.

△ We advise not to use do-while, continue, break, switch, operator, enum, implicit conversion from boolean to integer, conditional expressions, and references a part from the pass of parameters.

△ Pointers are forbidden.

△ Key words new, delete, goto, try, throw, catch, template and class are forbidden.

△ It is forbidden to make system calls.
3 Constructions not recommended in an introductory course

△ Preincrements and postincrements with lateral effects.
   For example, this code is worse than the last recommended one:

   ```c++
   int i = -1;
   while (++i < v.size() and v[i] != x);
   if (i < v.size()) cout << x << " is found in " << i << endl;
   ```

△ The construction `do while`.

△ Bit operators `&`, `|`, `<<`, `>>`, `^`.

△ The construction `boolean_condition ? expression1 : expression2`.
   Instead, you should use `and`, `or`, `not`.

△ Casting in C.
   In the few cases where it’s necessary to use a casting between different types,
   you should use the modern notation in C++.

   ```c++
   int i = int(M_PI);
   cout << char('a' + i);
   ```
   is better than

   ```c++
   int i = (int)M_PI;
   cout << (char)('a' + i);
   ```

△ The implicit comparison with the nul value of a type (except with booleans).
   Example:

   ```c++
   int x, y;
   cin >> x >> y;
   if (x == 0) cout << "the first one is zero" << endl;
   if (y != 0) cout << "the second one is not zero" << endl;
   ```
   is better than

   ```c++
   ... 
   if (!x) cout << "the first one is zero" << endl;
   if (y) cout << "the second one is not zero" << endl;
   ```

△ Prototypes, which means declaring function headers before implementing them.
   They’re not necessary because there’s not crossed recursion.

¹Some of this constructions are not a drawback for expert programmers.
# 4 Program example

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Maximum number of paths that can be read.
const int MAX_PATHS = 100;

// A point of the plane has a component x and a component y.
struct Point {
    double x, y;
};

// A path is a sequence of points.
typedef vector<Point> Path;

// Reads and returns a point.
Point read_point() {
    Point p;
    cin >> p.x >> p.y;
    return p;
}

// Reads a path, by reading its number of point n and reading
// the n points. n==0 indicates end of input, and in such
// a case end is set to true.
void read_path(Path& p, bool& end) {
    int n;
    cin >> n;
    if (n == 0) end = true;
    else {
        p = Path(n);
        for (int i = 0; i < n; ++i) p[i] = read_point();
    }
}

...  

// Reads at most MAX_PATHS paths.
// Next, ...
int main() {
    vector<Path> v(MAX_PATHS);
    bool end = false;
    int n = 0;
    read_path(v[n], end);
    while (not end) {
        ++n;
        read_path(v[n], end);
    }
    ...
}
```
5 Structure of a program

As you can see in the last example, most programs have five parts:

1. Inclusions and name space
2. Definition of constants—it can be empty
3. Definition of types (vectors and structs)—it can be empty
4. Procedures—it can be empty
5. Main program (main)

5.1 Inclusions and name space

Only write the necessary inclusions, starting with \texttt{#include}.

\textit{Except \texttt{#include}, all the other directives are prohibited.}

The admitted inclusions are:

\begin{itemize}
  \item \texttt{	extless iostream\textgreater}: you can only use the construction

  \begin{verbatim}
  cin >> variable1 >> variable2 ... ;
  \end{verbatim}

  to read, and the construction

  \begin{verbatim}
  cout << expression1 << expression2 ... ;
  \end{verbatim}

  to write.

  If you have to write any real number, then you have to previously choose the number of decimals \( d \). After that, you have to write the following lines of code at the beginning of your \texttt{main}:

  \begin{verbatim}
  cout.setf(ios::fixed);
  cout.precision(\( d \));
  \end{verbatim}

  If you want to write an end of line, you should use \texttt{endl} (and not `\texttt{\textbackslash n}`), or anything similar).

  There are three typical schemes for a reading loop:

  \begin{enumerate}
    \item Knowledge of the number of elements,
    \item until finding a sentinel,
    \item until there's no more input information.
  \end{enumerate}

  The good way of writing the third scheme is using the implicit conversion of \texttt{cin} to boolean.
Example:

```cpp
int sum = 0;
int x;
while (cin >> x) sum += x;
cout << "the total sum is " << suma << endl;
```

We understand “while (cin >> x)” as “while we can keep reading x”.
Using `cin.eof()` is a possible source of errors.

△ *We advise not to use any other input/output operations like getline().*

△ *It is forbidden reading or writting in C style, with scanf(), printf() or similar.*

• `<string>`: They can be declared (with no initial value, or assigning one literal value), assign, compare, read and write.

Examples:

```cpp
const string GREETING = "Hola!!!";
...
cout << GREETING << endl;
string s, t;
cin >> s >> t;
if (s > t) {
    string aux = s;
    s = t;
    t = aux;
}
cout << s << " is smaller or equal than " << t << endl;
```

Furthermore, strings can be declared giving the size and initial value in each position, you can get their size, and you can access each and every one of its positions.

Examples:

```cpp
int n;
char c;
cin >> n >> c;
string t(n, c);
cout << t << " has " << n << " caracter(s)" << endl;
cout << "all of them equal to " << c << endl;
```
You have to be careful with arithmetic operations that use multiple calls to `size()` (either with strings or with vectors), because `size()` returns unsigned integer numbers, which combined (particularly, subtracted) can give incorrect results.

For example:

```cpp
string x, y;
cin >> x >> y;
if (x.size() - y.size() > 0) cout << "x is longer than y" << endl;
```

This code is *not* correct. (Try what happens when `x` is shorter than `y`.) Fortunately, usually you can re-write the code and make it right:

```cpp
string x, y;
cin >> x >> y;
if (x.size() > y.size()) cout << "x is longer than y" << endl;
```

We can also concatenate strings with the operator `+`, or with `+=`.

```cpp
string x = "hello", y = "how are you?";
x += " " + y;
cout << x << endl;
```

⚠️ *You can’t use any other operation on strings.*

- `<vector>`: They can be declared (with or without initial size, and in the first case, with or without initial values for their positions) and assigned. Furthermore, you can get their size and access each and every one of their positions.

Examples:

```cpp
void read_vector(vector<double>& v) {
    int n;
cin >> n;
v = vector<double>(n);
    for (int i = 0; i < n; ++i) cin >> v[i];
}

void doubles(vector<int>& v) {
    for (int i = 0; i < v.size(); ++i) v[i] *= 2;
}
```

```
vector<double> v;
read_vector(v);
vector<int> u(100, -4);
doubles(u); // now u is equal to a vector<int>(100, -8)
```

You can use the instruction `push_back` for constructing a vector only in the cases where the size is not known previously, or when its use gives rise to a much simpler code. For example,
void choose_positives(const vector<int>& v, vector<int>& r)
{
    r = vector<int>();
    for (int i = 0; i < int(v.size()); ++i)
        if (v[i] > 0)
            r.push_back(v[i]);
    return r;
}

is better than

void choose_positives(const vector<int>& v, vector<int>& r)
{
    int numpositives=0;
    for (int i = 0; i < int(v.size()); ++i)
        if (v[i] > 0)
            ++numpositives;
    int ir = 0;
    r = vector<int>(numpositives);
    for (int i = 0; i < int(v.size()); ++i) {
        if (v[i] > 0) {
            r[ir] = v[i];
            ++ir;
        }
    }
    return r;
}

• Don’t use vectors unless is strictly necessary, or not using them produces a
  more inefficient solution or a more complex code.

△ It is forbidden to use vectors before they’re explained in class.

△ You can’t use any other operation on vectors.

• <cmath>: You can only use the predefined constant M_PI (π with the best precision
  the machine allows), the functions maximum max() and minimum min(),
  the trigonometric functions sin(), cos() and tan(), the exponentiation (only
  for double) pow(), and the square root sqrt().

△ You can’t use any other mathematical operation.
• <algorithm>: In the final part of the course and for the exercises that don’t state the opposite, if you have to sort a vector you can use the standard procedure sort().

Example:

```cpp
vector<int> v(MAX);
for (int i = 0; i < MAX; ++i) cin >> v[i];
sort(v.begin(), v.end());
cout << "sorted:";
for (int i = 0; i < MAX; ++i) cout << " " << v[i];
cout << endl;
```

If you want to sort the elements of a vector, and they’re not basic elements, or you don’t want them to be sorted from smaller to greater, then you can create a function to compare the elements and determine which one is smaller than the other.

Example:

Consider that the members of a club are defined with multiple fields, two of which are the name and the age:

```cpp
struct Member {
    string name;
    int age;
    ...
};
```

If we want to sort the members first by their age, from older to younger, and if there’s a tie, by their name, then we can implement this function:

```cpp
bool comp(const Member& a, const Member& b) {
    if (a.age != b.age) return a.age > b.age;
    return a.name < b.name;
}
```

Now, to sort a members vector `v` you just have to do:

```cpp
sort(v.begin(), v.end(), comp);
```

The comparison function must implement a strict order, which means, that two calls to `comp(a, b)` and `comp(b, a)` can’t both return true. The following function would not be correct:

```cpp
bool comp(const Member& a, const Member& b) {
    if (a.age != b.age) return a.age > b.age;
    return a.name <= b.name;
}
```

Alternatively, we can define the comparison operator `<` for the type `Member`:

```cpp
bool operator<(const Member& a, const Member& b) {
    if (a.age != b.age) return a.age > b.age;
    return a.name <= b.name;
}
```
and next use \texttt{sort} without indicating any comparison method, that implicitly it will be the one defined for $<$. 

\begin{verbatim}
sort(v.begin(), v.end());
\end{verbatim}

\begin{itemize}
\item You can’t use any other operation from the library \texttt{<algorithm>}. \end{itemize}

After the inclusions, you always have to add the line

\begin{verbatim}
using namespace std;
\end{verbatim}

\begin{itemize}
\item It is forbidden to use any other name space, or any other inclusion rather than \texttt{<iostream>}, \texttt{<string>}, \texttt{<vector>}, \texttt{<cmath>} or \texttt{<algorithm>}. \end{itemize}

5.2 Definition of constants

Every constant must be defined in one line, using the syntax 

\begin{verbatim}
const name_of_type name_of_constant = value;
\end{verbatim}

The key word \texttt{const} creates a constant and not a global variable. One variable is a global variable if it is defined outside any procedure, including the \texttt{main}. 

\begin{itemize}
\item Global variables are forbidden. \end{itemize}

5.3 Definition of new types

The basic types that can be used are: \texttt{int}, \texttt{double}, \texttt{bool} and \texttt{char}. The \texttt{string} type is not a basic type; you have to include the \texttt{string} library to use it. 

\begin{itemize}
\item It is forbidden to use any other basic type or modifier. For example, you can’t use \texttt{short} (int), \texttt{long} (int), \texttt{long long} (int), \texttt{unsigned} \texttt{int} or \texttt{float}. \end{itemize}

\begin{itemize}
\item We advise to not use the enumerated types (\texttt{enum}). \end{itemize}

\begin{itemize}
\item C styled arrays (declared with \texttt{[]} \texttt{)} are prohibited. \end{itemize}

\begin{itemize}
\item It is forbidden to use \texttt{class} to define a new type. \end{itemize}

5.4 Procedures

Input parameters are passed by value (which means, by copy) if they have basic or string types, and by constant reference otherwise (vectors and structs) in order to obtain a more efficient code, unless being able to manipulate them gives rise to a simpler and more readable code, in which case they may be passed by value.\footnote{Here, we’re considering that the strings used in the exercises are “small enough”. Otherwise, you should pass them by constant reference.} Output parameters or input/output parameters are passed by reference.

There are two kinds of procedures: actions and functions.
• Actions: Actions don’t return anything, which is specified by `void`.

Example:

```c++
// Returns the sum and the multiplication of the elements of v.
void sum_and_multiply(const vector<double>& v, double& sum,
                      double& multiplication) {
    sum = 0;
    multiplication = 1;
    for (int i = 0; i < v.size(); ++i) {
        sum += v[i];
        multiplication *= v[i];
    }
}
```

Usually it’s helpful to put the parameters in this order: input, input/output, output.

• Functions: Functions return any kind of type (preferably basic or string types, for efficiency), and all of their parameters are input parameters.

Example:

```c++
// Returns the multiplication of the elements of v.
double multiplication(const vector<double>& v) {
    double prod = 1;
    for (int i = 0; i < v.size(); ++i) prod *= v[i];
    return prod;
}
```

As you can see in the past example, to avoid overwriting names, you should give variables a different name from the one you gave to the procedure. That’s very important for recursive procedures.

From some examples, you can see that operators `+=`, `-=` , `*=` , `/=` , `%=` are allowed. Usually, their use allow a more natural a succinct way to express what we want.

### 5.5 Main program

The main program will always have the header

```c++
int main()
```

Although the `main` must return an integer number that indicates the final state of the program (0, if the program ran correctly), the compiler implicitly understands that there’s the instruction `return 0;` at the end of the program. Which means that you never have to write it down. Additionally, note that our `main` has no parameters.

### 6 Name of the constants, types, procedures and variables

If a name contains different words, we advise to separate them by ‘_’. The rest of letters are lower and upper case letters (and sometimes digits), with the following format:
• The names of the constants are advised to be exclusively written with upper case letters:

    const double DIVINE_PROPORTION = (1 + sqrt(5.0))/2;
    const int MAX_PEOPLE = 1000;

• We advise each word of the name of a type to start with an upper case letter, followed by lower case letters:

    typedef vector<int> Row;
    typedef vector<Row> Matrix;

    struct Tridimensional_Point {
        double x, y, z;
    };

• We advise to write the names of procedures and variables with lower case letters.

You have to choose significant names for constants, types and procedures. Particularly, not being able to find a name for a constant or a procedure is a sign that indicates that they might not be right for your program.

For some variables it’s sometimes better to use shorter names to make the code more readable (that’s particularly true for vector indices).

Furthermore, you have to avoid giving denied names to boolean variables. Example:

    bool found = false;

or

    bool keep_going = true;

is better than

    bool not_found = true;

It’s convenient to use these rules for the names of procedures:

• The names of actions are (or include) imperative verbs.

    Examples:
    - void sort(vector<double>& v);
    - void remove_repeated(vector<string>& v);
    - void merge(vector<int>& v1, vector<int>& v2);

• Functions that return a boolean usually begin with "is_".

    Examples:
    - bool is_prime(int n);
    - bool is_digit(char c);
    - bool is_sorted(const vector<string>& v);
But there’s also other possibilities:

- bool belongs(int a, const vector<int>& v);

- The name of the functions that return non boolean types is usually a substantive that indicates what the function returns.

Examples:

- int factorial(int n);
- int number_of_repeated(const vector<double>& v);
- string max(const vector<string>& v);

But also:

- char to_lowercase(char c);

7 Page breaks, indentation, spaces, parentheses

7.1 Page breaks

It is convenient to separate with (two, for example) blank lines the different fields of your program (inclusions, constants, types, procedures, and main). Inside every field, we advise to separate every subfield with the same number of blank lines, with two exceptions: all the declarations of constants and all the definitions of types with the key word typedef are usually written with no blank spaces (at the beginning of the section 6 there’s an example of each).

Sometimes, when a piece of code is large, you should write a blank line that separates the two conceptual blocks. If the piece of code is too large, then you should make smaller pieces of it using procedures.

7.2 Indentation

- We advise to indent with two or four (always the same number) spaces per level.

- It is convenient to avoid tabs, by configuring your editor correctly.

- It is convenient to truncate lines that contain more than 78 characters, because they can’t usually be correctly printed.

- If a piece of code has too much indentations, then you have to make it more readable by using procedures.

7.3 Spaces

As a general rule, we advise to leave exactly one space between key words, instructions, etcetera. The most important exceptions are:

- Semicolons are written at the end of every final word.

- Parentheses should be hooked to words.

- The left side parentheses of a procedure should be written hooked to its name.
• The same rules for parentheses of procedures apply for parentheses of vectors or strings.

• Binary operators \* (multiplication), / (division) and \% (residue), and the unary operator - (sign change) are written without spaces to emphasize their high precedence.

Example:

\[ \text{int } a = x + y*z; \]

is better than

\[ \text{int } a = x + y * z; \]

and that

\[ \text{int } a = x*y*z; \]

Also:

\[ \text{int } a = -b; \]

is better than

\[ \text{int } a = - b; \]

• Operators ++ (preincrement) and -- (predecrement) are written on the left of the variable they modify, without spaces.

• Operators ++ (postincrement) and -- (postdecrement) are written on the right of the variable they modify, without spaces.

• We advise to write character & (to indicate that a parameter is passed by reference) immediately at the right of the type it is associated with.

This example includes many past ideas:

```cpp
void who_knows_who_would_do_that(const vector<int>& v) {
    for (int i = v.size() - 5; i >= -4; --i) {
        cout << -i << " " << factorial(i*i + 3)/v[i + 4] - 7 << endl;
    }
}
```

7.4 Curly-brackets

There are many ways and variants to write curly-brackets. The recommended one is:

```cpp
if (condition) {
    ...
}
```
if (condition) {
    ...
} else {
    ...
}

while (condition) {
    ...
}

def (initialize; condition; "increment") {
    ...
}

Sometimes you have many exclusive conditions that you have to evaluate one after another. This creates an if else if else if ... structure, which we advise to write like this:

    if (condition1) {
        ...
    } else if (condition2) {
        ...
    } else if (condition3) {
        ...
    } else {
        ...
    }

Example:

    // Returns "lowercase", "uppercase" or "digit" when c is one of them, 
    // and "unknown" in any other case.
    string type(char c) {
        if (c >= 'a' and c <= 'z') {
            return "lowercase";
        } else if (c >= 'A' and c <= 'Z') {
            return "uppercase";
        } else if (c >= '0' and c <= '9') {
            return "digit";
        } else {
            return "unknown";
        }
    }

    When inside the body of an if, a while or a for there's only one action executed, and this one fits in the same line, you can avoid writing curly-braces to make a more compact piece of code:

    for (int i = 0; i < v.size(); ++i) prod *= v[i];

    Even return make some else statements unnecessary:

    // Returns "lowercase", "uppercase" or "digit" when c is one of them, 
    // and "unknown" in any other case.
    string type(char c) {
        if (c >= 'a' and c <= 'z') return "lowercase";
        if (c >= 'A' and c <= 'Z') return "uppercase";
        if (c >= '0' and c <= '9') return "digit";
        return "unknown";
    }
7.5 Parentheses

- As already stated, parentheses are separated from key words, the internal part of a parentheses is not separated from the statement which it contains, and the left side of a parentheses from a procedure is written immediately after the name of the procedure.

- We advise to not write parentheses containing the returning expression of a function.
  
  Example:

  ```
  return found;
  ```

  is better than

  ```
  return (found);
  ```

- As a general rule, you should only write the necessary parentheses to change the precedence between operators.
  
  Example:

  ```
  int x = a + b*c;
  ```

  is better than

  ```
  int x = a + (b*c);
  ```

- In that cases where the precedence between operators is not obvious (for example, when you have two operators with the same priority and you don’t remember which one gets evaluated first), it’s convinient to add parentheses to make the code more readable.
  
  Example:

  ```
  int a = (x/y)*z;
  ```

  or (if necessary)

  ```
  int a = x/(y*z);
  ```

  is better than

  ```
  int a = x/y*z;
  ```

- Exceptionally, if an expression is too complicated to read with no redundant parentheses, then you can add them. Nevertheless, in this cases it’s usually better to use auxiliary variables to simplify the expressions.
You have to document your code with the right amount of comments. Writing too many comments can be as bad as not commenting important things. Particularly, you have to explain what every procedure does. This should not be confused with how is it done, which can be explained also if it is particularly non-trivial. The documentation of a procedure can be made by adding a precondition and/or a postcondition, or with an explanation. It’s helpful to make sure that your documentation explains every parameter, and that it’s enough to be able to call the function without knowing how it’s implemented inside.

You don’t have to mistake explaining what with how it does it. Sometimes it can be interesting to explain how a procedure is implemented if it is necessary to know what it does inside, like knowing how to “sort $v$ using the insertion method” or “sort $v$ with the quicksort method” But usually, the clarifications are more understandable inside the body of the procedure than in the header.

Examples:

```c
// Returns n!.
// Pre: n >= 0.
int factorial(int n) {
    if (n == 0) return 1;
    else return n*factorial(n - 1);
}
```

```c
// Swaps the values of a and b.
void swap(int& a, int& b) {
    int aux = a;
    a = b;
    b = aux;
}
```

```c
// Determines if n is a palindrome
// Pre: n >= 0, p is the power of 10 such that p <= n < 10*p,
//     or any value if n = 0.
bool is_palindrome(int n, int p) {
    // Explanation:
    // One number with only one digit is a palindrome.
    // Otherwise, to be a palindrome, the first and last digits must be the same,
    // and the rest of the number must also be a palindrome.
    // We divide p between 100 to mantain the precondition on p.
    if (n <= 9) return true;
    int primer_digit = n/p;
    int ultim_digit = n%10;
    return primer_digit == ultim_digit
        and es_capicua((n - primer_digit*p)/10, p/100);
}
```

\(\triangle\) It will be punished in exams to declare a procedure without any kind of documentation.

As you can see, to comment you have to use // . We use /* */ for the debugging phase. That way you can compile only some pieces of code, erasing logically (but not physically) the pieces of code that you don’t want to compile. We advise not to include commented code with /* */ in the final program.

One program can’t contain code which can never be reached.
References

- http://www.spelman.edu/~anderson/teaching/resources/style/
- http://geosoft.no/development/cppstyle.html
- http://geosoft.no/development/cpppractice.html