A structured programming approach

Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.

Always try to choose the simplest algorithm. Computational speed can be improved upon later.

Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!

Getting Started

Compiling and linking, without QTcreator
In order to obtain an executable file for a C++ program, the following instructions under Linux/Unix can be used:

```bash
cc -o myprogram myprogram.o
```

where the compiler is called through the command `cc`. The compiler option `-o` is used for compilation only, where the `-c` option links the object file `myprogram.o` and produces the executable program.

For Fortran2008 we use the Intel compiler, replace `cc` with `ifort`

Also, to speed up the code use compile options like

```bash
c++ -O3 -c -Wall myprogram.cpp
```

Extremely useful tools, strongly recommended

Makefiles and simple scripts
Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands.

```bash
# Common lines:
# General makefile for c - choose MD or - name of given program
# Here we define compiler option, libraries and the target
CC= -Wall
FROG: myprogram
```

where the compiler is called through the command `cc` and the executable file is in this case `myprogram`. The option `-c` is for compilation only, where the program is translated into machine code, while the `-o` option links the produced object file `myprogram.o` and produces the executable program.

For Fortran2008 we use the Intel compiler, replace `cc` with `ifort`

Also, to speed up the code use compile options like

```bash
c++ -O3 -c -Wall myprogram.cpp
```

Unit tests

If you name your file for `makefile`, simply type the command `make` and Linux/Unix executes all of the statements in the above makefile. Note that C++ files have the extension `.cpp`.
Hello World

The C encounter

Here we present first the C version.

```c
#include <stdio.h> /* printf function */
#include <math.h> /* sine function */
#include <stdlib.h> /* atof function */
int main (int argc, char* argv[]) {
    double r, s;
    // declare variables
    r = atof(argv[1]); // convert the text argv[1] to double
    printf("Hello, World! sin(\%g)=%g\n", r, s);
    return 0;
}
```

Without namespace

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes. Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Here we present the C++ version without using namespace.

```cpp
int main (int argc, char* argv[]) {
    double r, s;
    // declare variables
    r = atof(argv[1]); // convert the text argv[1] to double
    printf("Hello, World! sin(\%g)=%g\n", r, s);
    return 0;
}
```

C++ Hello World

Dissection I

We have replaced the call to printf with the standard C++ function cout. The header file <iostream.h> is then needed. In addition, we don’t need to declare variables like r and at the beginning of the program. I personally prefer however to declare all variables at the beginning of a function, as this gives me a feeling of greater readability.

```cpp
#include <iostream> // input and output
#include <cmath> // math functions
#include <cstdlib> // stdlib functions
int main (int argc, char* argv[]) {
    double r, s;
    // declare variables
    r = atof(argv[1]); // convert the text argv[1] to double
    std::cout << "Hello, World! sin(\%g)=%g\n", r, s;
    return 0; // success execution of the program
}
```
C/C++ program
- A C/C++ program begins with include statements of header files (libraries, intrinsic functions, etc).
- Functions which are used are normally defined at top (details next week).
- The main program is set up as an integer; it returns 0 (everything correct) or 1 (something went wrong).
- Standard if, while and for statements as in Java, Fortran, Python...
- Integers have a very limited range.

Arrays
- A C/C++ array begins by indexing at 0!
- Array allocations are done by size, not by the final index value. If you allocate an array with 10 elements, you should index them from 0, 1, ..., 9.
- Initialize always an array before a computation.

Serious problems and representation of numbers
- Integer and Real Numbers
  - Overflow
  - Underflow
  - Roundoff errors
  - Loss of precision

Limits, you must declare variables
- C++ and Fortran declarations
  - int/INTEGER
    - 16: -32768 to 32767
  - unsigned int
    - 16: 0 to 65535
  - signed int
    - 16: -32768 to 32767
  - short int
    - 16: -32768 to 32767
  - unsigned short int
    - 16: 0 to 65535
  - signed short int
    - 16: -32768 to 32767
  - int/long int/INTEGER
    - 32: -2147483648 to 2147483647
  - signed long int
    - 32: -2147483648 to 2147483647
  - float/REAL(4)
    - 32: \(3.4 \times 10^{-44}\) to \(3.4 \times 10^{38}\)
  - double/REAL(8)
    - 64: \(1.7 \times 10^{-322}\) to \(1.7 \times 10^{308}\)
  - long double
    - 64: \(1.7 \times 10^{-322}\) to \(1.7 \times 10^{308}\)

From decimal to binary representation
- How to do it
  \[a_n2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_02^0.\]
- In binary notation we have thus \((417)_{10} = (110110001)_{2}\) since we have
  \[(110100001)_{2} = 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^0.\]

From decimal to binary representation, the actual operation
- To see this, we have performed the following divisions by 2:
  \[
  \begin{align*}
  417/2 &= 208 \text{ remainder } 1 \text{ coefficient of } 2^0 = 1 \\
  208/2 &= 104 \text{ remainder } 0 \text{ coefficient of } 2^1 = 0 \\
  104/2 &= 52 \text{ remainder } 0 \text{ coefficient of } 2^2 = 0 \\
  52/2 &= 26 \text{ remainder } 0 \text{ coefficient of } 2^3 = 0 \\
  26/2 &= 13 \text{ remainder } 0 \text{ coefficient of } 2^4 = 0 \\
  13/2 &= 6 \text{ remainder } 1 \text{ coefficient of } 2^5 = 1 \\
  6/2 &= 3 \text{ remainder } 0 \text{ coefficient of } 2^6 = 0 \\
  3/2 &= 1 \text{ remainder } 1 \text{ coefficient of } 2^7 = 1 \\
  1/2 &= 0 \text{ remainder } 1 \text{ coefficient of } 2^8 = 1
  \end{align*}
  \]
### From decimal to binary representation

**Integer numbers**

```cpp
#include <iostream>
#include <cmath>
#include <cstdio>

IMPLICIT NONE

INTEGER i, number, terms(0:31)

! Fortran takes only integer loop variables
DO k=1, 31
  number = MOD(number, 2)
  terms(i) = number
  number /= 2;
ENDDO
```

```
# A comment line begins like this in C++ programs
// A comment line begins like this in C++ programs
END PROGRAM binary_integer
```

**Machine Numbers**

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa $q$ would be $(1.f_2)$ and $1 \leq q < 2$. This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitly that a binary number with 23 bits for the mantissa reads

```
(0.1.a−1a−2 ... a−23)2 = 2−1 \times 2−2 \times 2−3 \times 2−4 \times \ldots \times 2−23.
```

### Representing Integer Numbers

**Possible Overflow for Integers**

```cpp
// A comment line begins like this in C++ programs
// ... << " 2^N * 2^N = " << int1*int1 << endl;
return 0;
// End: program main()
```

```cpp
for (i=0; i < 32 ; i++){
  cout << " Term nr: " << i << "Value= " << terms[i];
  cout << endl;
}
return 0;
}
```

### Loss of Precision

**Machine Numbers**

In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation.

```
x = 9.90625 = 9.90625 \times 10^1.
```

and a real non-zero number could be generalized as

```
x = a \times 10^m.
```

with $r$ a number in the range $1/10 \leq r < 1$. In a similar way we can use a binary number in scientific notation as

```
x = a \times 2^m.
```

with $q$ a number in the range $1/2 \leq q < 1$. This means that the mantissa of a binary number would be represented by the general formula

```
(0.1.a−1a−2 ... a−n)2 = a−1 \times 2−1 + a−2 \times 2−2 + \ldots + a−n \times 2−n. (3)
```

### Loss of Precision

**Possible Overflow for Integers**

```cpp
// A comment line begins like this in C++ programs
// ... << " 2^N*(2^N - 1) = " << int1 * int3 << endl;
cout << " 2^N- 1 = " << int3 << endl;
return 0;
// End: program main()
```

```
for (i=0; i < 32 ; i++){
  cout << " Term nr: " << i << "Value= " << terms[i];
  cout << endl;
}
return 0;
}
```

### Machine Numbers

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on $q$ and $m$ imposed by the available word length. In the machine, our number $x$ is represented as

```
x = (-1)^s \times \text{mantissa} \times 2^\text{exponent} \quad \quad \quad (4)
```

where $s$ is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa.
As an example, consider the 32 bits binary number

\[ 10111101110100000000000000000000 \_2 \]

where the first bit is reserved for the sign, 1 in this case yielding a negative sign. The exponent \( m \) is given by the next 8 binary numbers \( 01111101 \) resulting in 125 in the decimal system.

If our number \( x \) can be exactly represented in the machine, we call \( x \) a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a computation, an inevitable error will arise in representing it as accurately as possible by a machine number.

In the machine a number is represented as

\[ f_l(x) = x(1 + \epsilon) \quad (7) \]

where \( |\epsilon| \leq \epsilon_M \) and \( \epsilon \) is given by the specified precision, \( 10^{-7} \) for single and \( 10^{-16} \) for double precision, respectively. \( \epsilon_M \) is the given precision. In case of a subtraction \( a - b \), we have

\[ f_l(a) - f_l(b) = f_l(c) = a(1 + \epsilon_a) \quad (8) \]

or

\[ f_l(a) = b(1 + \epsilon_b) - c(1 + \epsilon_c) \quad (9) \]

The above means that

\[ f_l(a) / a = 1 + \epsilon_a b^-1 - \epsilon_c c^-1 \quad (10) \]

and if \( b = c \) we see that there is a potential for an increased error in \( f_l(a) \).
Loss of numerical precision

Suppose we wish to evaluate the function
\[ f(x) = \frac{1 - \cos(x)}{\sin(x)} \]
for small values of \( x \). Five leading digits. If we multiply the
denominator and numerator with \( 1 + \cos(x) \) we obtain the
equivalent expression
\[ f(x) = \frac{\sin(x)}{1 + \cos(x)} \]
If we now choose \( x = 0.007 \) (in radians) our choice of precision
results in
\[ \sin(0.007) \approx 0.69999 \times 10^{-2}, \]
and
\[ \cos(0.007) \approx 0.99998. \]

Loss of precision can cause serious problems

Real Numbers

- **Overflow**: When the positive exponent exceeds the max
  value, e.g., \( 308 \) for DOUBLE PRECISION (64 bits). Under such
circumstances the program will terminate and some compilers
may give you the warning OVERFLOW.
- **Underflow**: When the negative exponent becomes smaller
  than the min value, e.g., \( -308 \) for DOUBLE PRECISION.
  Normally, the variable is then set to zero and the program
  continues. Other compilers (or compiler options) may warn
  you with the UNDERFLOW message and the program terminates.
Loss of precision, real numbers

Roundoff errors. A floating point number like

\[ x = 1.234567891123 \times 10^9 \]

may be stored in the following way. The exponent is small and is stored in full precision. However, the mantissa is not stored fully. In double precision (64 bits), digits beyond the 15th are lost since the mantissa is normally stored in two words, one which is the most significant one representing 123456 and the least significant one containing 789111213. The digits beyond 3 are lost. Clearly, if we are summing alternating series with large numbers, subtractions containing in 10^{-11} may lead to roundoff errors, since not all relevant digits are kept. This leads eventually to the next problem, namely

A problematic case

Three ways of computing \( e^{-x} \)

**Brute force:**

Program to compute \( e^{-x} \)

Still Brute Force

Program to compute \( e^{-x} \)

Oh, it never ends!
Most used formula for derivatives

3 point formulae

First derivative \( f'_0 = f(x_0), f_{-h} = f(x_0 - h) \) and \( f_{+h} = f(x_0 + h) \)

\[
     f'_0 = \frac{f_{+h} - f_{-h}}{2h} = \tilde{f}'_0 + \sum_{j=1}^{\infty} \frac{c_j}{(2j + 1)!} h^{2j}.
\]

Second derivative

\[
     f''_0 = \frac{f_{+2h} - 2f_{+h} + f_{-h}}{2h^2} = \tilde{f}''_0 + \sum_{j=1}^{\infty} \frac{c_j (2j+1)}{(2j + 1)!} h^{2j}.
\]

Error Analysis

\[
\epsilon = \log_{10} \left( \frac{\text{computed} - \text{exact}}{\text{exact}} \right)
\]

where \( \epsilon_{\text{approx}} \approx f^{(4)}(12h)/12 \).

For the computed second derivative we have

\[
    c_{2j} = \frac{f_{+2h} - 2f_{+h} + f_{-h}}{2h^2} = \sum_{j=1}^{\infty} \frac{c_j (2j+1)}{(2j + 1)!} h^{2j},
\]

and the truncation or approximation error goes like

\[
    \epsilon_{\text{approx}} \approx \frac{c_4}{12} h^2.
\]
Error Analysis

If we were not to worry about loss of precision, we could in principle make \( h \) as small as possible. However, due to the computed expression in the above program example

\[
\frac{f''_0}{h^2} = \frac{f_h - f - f_{-h} + f(2f - f_h + f_{-h})}{h^2}
\]

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial. If \( (f_h - f_{-h}) \) are very close, we have \( |\epsilon_M| \approx 10^{-15} \) for single and \( |\epsilon_M| \approx 10^{-15} \) for double precision, respectively.

We have then

\[
|\epsilon_t^2| = \left| \frac{(f_h - f_{-h})}{h^2} + \frac{(f_{-h} - f) + (f - f_h)}{h^2} \right| \leq 2|\epsilon_M|
\]

It is then natural to ask which value of \( h \) yields the smallest total error. Taking the derivative of \( |\epsilon_t^2| \) with respect to \( h \) results in

\[
h = \left( \frac{24|\epsilon_M|}{|\epsilon_t^2|} \right)^{1/4}.
\]

With double precision and \( x = 10 \) we obtain

\[
h \approx 10^{-4}.
\]

Beyond this value, it is essentially the loss of numerical precision which takes over.

---

Error Analysis

Due to the subtractive cancellation in the expression for \( f'' \) there is a pronounced deterioration in accuracy as \( h \) is made smaller and smaller. It is instructive in this analysis to rewrite the numerator of the computed derivative as

\[
(h_f - h_l) + (f_{-h} - f_0) = \epsilon^h - \epsilon^0 + (\epsilon^h - \epsilon^0),
\]

as

\[
(h_f - h_l) + (f_{-h} - f_0) = \epsilon^h + \epsilon^0 - 2,
\]

since it is the difference \( (\epsilon^h + \epsilon^0 - 2) \) which causes the loss of precision.

---

Error Analysis

The results for \( x = 10 \) are shown in the Table.

<table>
<thead>
<tr>
<th>( h )</th>
<th>( \epsilon^0 )</th>
<th>( \epsilon^h )</th>
<th>( \epsilon^h + \epsilon^0 - 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^-6</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
<tr>
<td>10^-5</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
<tr>
<td>10^-4</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
<tr>
<td>10^-3</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
<tr>
<td>10^-2</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
<tr>
<td>10^-1</td>
<td>2.0000000000000000</td>
<td>1.9999999999999990</td>
<td>1.9999999999999990</td>
</tr>
</tbody>
</table>

---

Error Analysis

Our total error becomes

\[
|\epsilon_t| \leq 2|\epsilon_M| + \frac{f(4)}{12} h^2.
\]

---

Program to compute derivative

We list here the program to compute the above derivative.

```cpp
#include <iostream>
#include <iomanip>

// Begin of main program
int main(int argc, char* argv[]) {
    char * ofilename = argv[1];
    // Open output file
    ofstream outfilenm(ofilename);
    // End of main program
    return 0;
}
```

---
A pointer specifies where a value resides in the computer's memory (like a house number specifies where a particular family resides on a street). A pointer points to an address not to a data container of any kind!

Simple example declarations:

```cpp
using namespace std;

int main()
{
    // what are the differences?
    int var;
    cin >> var;
    int *p, q;
    int *s, *t;
    int * a new[var]; // dynamic memory allocation
    delete [] a;
}
```

Dissection: Pointer example I

Discussion

```cpp
int main()
{
    int var; // Define an integer variable var
    int *p; // Define a pointer to an integer
    p = &var; // Extract the address of var
    var = 421; // Change content of var
    printf("Address of integer variable var : %p\n", &var);
    printf("Its value: %d\n", var); // 421
    printf("Value of integer pointer p : %p\n", p); // = &var
    printf("The value p points at : %d\n", *p);
    printf("Address of the pointer p : %p\n", &p);
    return 0;
}
```

Dissection: Pointer example II

```cpp
int matr[2]; // Define integer array with two elements
int *p; // Define pointer to integer
p = &matr[0]; // Point to the address of the first element in matr
matr[0] = 321; // Change the first element
matr[1] = 322; // Change the second element
printf("Address of matrix element matr[1]: %p\n", &matr[0]);
printf("Value of the matrix element matr[1]; %d\n", matr[0]);
printf("Address of matrix element matr[2]: %p\n", &matr[1]);
printf("Value of the matrix element matr[2]; %d\n", matr[1]);
printf("Value of the pointer p: %p\n", p);
printf("The value p points to: %d\n", *p);
printf("The value that (p+1) points to %d\n", *(p+1));
printf("Address of pointer p : %p\n", &p);
```

Output of Pointer example II

```
Address of the matrix element matr[1]: 0xbfffef70
Value of the matrix element matr[1]; 321
Address of the matrix element matr[2]: 0xbfffef74
Value of the matrix element matr[2]; 322
Value of the pointer: 0xbfffef70
The value pointer points at: 321
The value that (pointer+1) points at: 322
Address of the pointer variable : 0xbfffef6c
```
File handling; C-way

```c
using namespace std;
#include <iostream>
int main(int argc, char *argv[])
{
    FILE *in_file, *out_file;
    if( argc < 3) {
        printf("The program has the following structure:
                write in the name of the input and output files
            ");
        exit(0);
    }
    in_file = fopen( argv[1], "r" );
    if( in_file == NULL ) {
        printf("Can't find the input file %s
", argv[1]);
        exit(0);
    }
    out_file = fopen( argv[2], "w" );
    if( out_file == NULL ) {
        printf("Can't find the output file %s
", argv[2]);
        exit(0);
    }
    fclose(in_file);
    fclose(out_file);
    return 0;
}
```

File handling, C++-way

```cpp
#include <fstream>
// input and output file as global variables
ofstream ofile;
ifstream ifile;
```

```cpp
int main(int argc, char* argv[])
{
    char *outfilename;
    // Read in output file, abort if there are too
    // few command-line arguments
    if( argc <= 1 ) {
        cout << "Bad Usage: " << argv[0] << " read also output file on same line" << endl;
        exit(1);
    }
    else{
        outfilename=argv[1];
    }
    ofile.open(outfilename);
    ....
    ofile.close(); // close output file
}
```

```cpp
void output(double r_min , double r_max, int max_step,
double *d)
{
    int i;
    ofile << "RESULTS" << endl;
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << "R_min = " << setw(15) << setprecision(8) << r_min <<endl;
    ofile << "R_max = " << setw(15) << setprecision(8) << r_max <<endl;
    ofile << "Number of steps = " << setw(15) << max_step << endl;
    for(i = 0; i < 5; i++) {
        ofile << setw(15) << setprecision(8) << d[i] << endl;
    }
}
```

```cpp
int main(int argc, char* argv[])
{
    char *infilename;
    // Read in input file, abort if there are too
    // few command-line arguments
    if( argc <= 1 ) {
        cout << "Bad Usage: " << argv[0] << " read also input file on same line" << endl;
        exit(1);
    }
    else{
        infilename=argv[1];
    }
    ifile.open(infilename);
    ....
    ifile.close(); // close input file
}
```
C++ allows the programmer to use solely call by reference (note that call by reference is implemented as pointers). To see the difference between C and C++, consider the following simple examples. In C we would write

```c
int main(int argc, char* argv[]) {
    int n; n = 8;
    func(n); // just transfer n itself
    ....
}

void func(int *i)
{
    *i = 10; // n is changed to 10
    ....
}
```

whereas in C++ we would write

```cpp
int n; n = 8;
func(&n); // just transfer n itself
....
void func(int& i)
{
    i = 10; // n is changed to 10
    ....
}
```

The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code.

In Fortran we can use `INTENT(IN), INTENT(OUT), INTENT(INOUT)` to let the program know which values should or should not be changed.

```fortran
SUBROUTINE coulomb_integral(np,lp,n,l,coulomb)
    IMPLICIT NONE
    INTEGER :: i
    REAL(KIND=8), INTENT(INOUT) :: coulomb
    ....
    *n is changed to 10*
    ....

double *a = new double[i];
// You can use double *a = new double[i]; or
int i = atoi(argv[2]);
// to check for memory leaks, use the software called -valgrind-
return 0; /* success execution of the program */
```

This hinders unwanted changes and increases readability.
# Example codes in C++: transfer of data using call by value and call by reference

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

// Declare functions before main
void func(int, int*);

int main(int argc, char *argv[])
{
    int a;
    int b[10];
    for(int i = 0; i < 10; i++)
    {
        b[i] = i;
        cout << b[i] << endl;
    }
    // the variable a is transferred by call by value. This means
    // that the function func cannot change a in the calling function
    func(a, b);
    delete [] b;
    return 0;
}
```

```cpp
// End: function main()
```

```cpp
// Function definition

void func(int x, int *y)
{
    // a becomes locally x and it can be changed locally
    x += 7;
    // func gets the address of the first element of y (b)
    // it changes y[0] to 10 and when returning control to main
    // it changes also b[0]. Call by reference
    *y += 10;
    // *y = *y + 10;
    y[6] += 10;
    // in this function y[0] and y[6] have been changed and when returning
    // control to main this means that b[0] and b[6] are changed.
    return;
}
```

```cpp
// End: function func()
```

---

# Example codes in C++: operating on several arrays and printing time used

```cpp
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include <time.h>

using namespace std;

int main(int argc, char* argv[])
{
    int i = atoi(argv[1]);
    double *a, *b, *c;
    a = new double[i];
    b = new double[i];
    c = new double[i];
    clock_t start, finish;
    start = clock();
    for (int j = 0; j < i; j++)
    {
        a[j] = cos(j*1.0);
        b[j] = sin(j+3.0);
        c[j] = 0.0;
    }
    for (int j = 0; j < i; j++)
    {
        c[j] = a[j] + b[j];
    }
    finish = clock();
    double timeused = (double) (finish - start)/(CLOCKS_PER_SEC);
    cout << setiosflags(ios::showpoint | ios::uppercase);
    cout << setprecision(10) << setw(20) << "Time used for vector addition=" << timeused << endl;
    delete [] a;
    delete [] b;
    delete [] c;
    return 0; /* success execution of the program */
}
```

---

```cpp
// Function definition

void func(int x, int *y)
{
    // a becomes locally x and it can be changed locally
    x += 7;
    // func gets the address of the first element of y (b)
    // it changes y[0] to 10 and when returning control to main
    // it changes also b[0]. Call by reference
    *y += 10;
    // *y = *y + 10;
    y[6] += 10;
    // in this function y[0] and y[6] have been changed and when returning
    // control to main this means that b[0] and b[6] are changed.
    return;
}
```

```cpp
// End: function func()
```

---

```cpp
// End: function main()
```