COMPUTATIONAL PHYSICS

Introduction to C++

Program structure

Data types, structures

Operators

Basic input/output

Program structure

- Nothing better than looking at the classical example…

```
// most boring though useful program in C++
#include <iostream>
using namespace std;

int main()
{
    /* Also a comment for more than one line…
    */
    cout << "Hello World!" << endl;
}
```

> One liner comment

> Pre-processor directive

> Namespace std library to shortcut function calls

> **Main code function** (returns optional integer, accepts optional arguments)

> Multi liner comment

> Redirect string of characters to standard output (and insert new line and flush data stream…)

- Commands, or *statements*, in C++ are strings of characters separated by blanks ("words", even a new line) and end with a semicolon (;) → ***All C++ statements must end with a semicolon character***

Program structure

```
// most boring though useful program in C++
#include <iostream>
using namespace std;

int main()
{
    /* Also a comment for more than one line…
    */
    cout << "Hello World!” << endl;
}
```

- Directive to include file content
  → crucial to include *header files*
  In this case standard C++ iostream header file.

- The { } define the scope or the body of the function and contain the statements to be executed when the function is called

- C++ does not have strict rules on *indentation* or on how to split instructions in different lines e.g. main function is equivalent to

```
int main() { cout << "Hello World!” << endl; }
```

```
int main() { cout << "Hello World!” <<
                                endl; }
```

- The main function can take arguments *(later…)*

```
int main(int narg, char * user[])
```

Program structure – compile&run

```
// most boring though useful program in C++
#include <iostream>
using namespace std;

int main()
{
   /* Also a comment for more than one line…
   */
   cout << "Hello World!” << endl;
}
```

→ **C++** code is designed as a *compiled language* i.e. code is translated into machine code that our computer understands, making it highly efficient !

→ Other languages e.g**. MATLAB or Python**, are *interpreted languages* i.e. code lines are interpreted line by line.

→ **Speed** *vs* **User friendliness**

- Compiling: *g++ -c Hello_world.cpp*
- Linking: *g++ Hello_world.o -o Hello_world*
- *Shortcut: g++ Hello_world.cpp -o Hello_world*

More on compilation later…

Variables and Data types

- In C++ one must declare and initialize variables to store a value of a given type e.g.

```
int A=23; int a=45; float b=23.1; char letter='w'
```

- The variable's identifier (*name*) must start with a letter or "_" and ***never*** with a number e.g. int 4season=34;
- C++ is case sensitive → int A=23; int a=45 is perfectly legitimate !
- There are "*C++ reserved*" identifiers e.g. and, auto, default, union that *cannot* be used.

Fundamental data types

❑ Character types: a single character, such as 'a' or '$'.

❑ Numerical integer types: store a whole number value, such as 7 or -12.

❑ Floating-point types: represent "real values", such as 3.14 or -0.7, with different levels of precision.

❑ Boolean type: can only represent one of two states, *true* or *false*.

Variables and Data types

```
// integer
int a; a=45;
int a(3); //a=3 alternative way
unsigned int a=123u; //positive integer
long int a=1234567891234L; //long integer

// reals (approximations of it…)
float b=23.1; //single precision
double b=23.1; //double precision
float b=23.1e0; //scientific notation

// integer constants
const c=112; //cannot be modified…

// characters
char d='q'; //single character…
char d=101; //int code for "e"
```

```
// "strings" (from C++ std library)
#include <string>
string course="Physics";

// Character sequence
char name[15]; //15 chars including for
null-termination ('\0')
name[0]='M'; name[1]='A';
name[2]='R'; name[3]='I'; name[4]='O';

char job[]="plumber";

// boolean
bool question=true; //...or false...
```

→ *More on character sequences later…*

Variables and Data types

Type specifier	Description	Size (bits) in 64 bit machine
short	Shortest integers	16
short int	Range: -32768 to 32767	
signed short int	same	
unsigned short int	Range: 0 to 65535	
int		32
signed int	Range: -2147483648 to 2147483647	
unsigned int	Range: 0 to 4294967295	
long	Range: -9.22...e18 to 9.22...e18	64
float	Single precision	32
double	Double precision	64
char	Single character	8
signed char	Int code from -128 to 127	
unsigned char	Int code from 0 to 255	
boolean	Boolean true or false	8

Structures

- A structure (aka *"public class"*) is a group of data type elements grouped together under one name. These *members* can have different types and different lengths.

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

```
struct country {
string name;
int population;
} ;
```

```
struct country {
string name;
int population;
};

int main() {
  country C1={"Portugal",10000000};
  country C2;
  C2.name="Spain";
  C2.population=46000000;
}
```

Operators

Assignment

y=x; x=y=z=5;

Arithmetic

+ sum

- subtraction

* multiplication

/ division (int x = 7/2 → 3)

% modulo (x = *11 % 2 → 1*)

Compound assignment

y+=x; (same as y=y+x;)

z-=10; (same as z=z-10;)

x/=3.0; (same as x=x/3.0;)

y*=4+c; (same as y=y*(4+c);)

Increment/decrement assignment

x++ (same as x=x+1;)

x-- (same as x=x-1;)

x=1; y=++x; (same as x=2; y=2;)

x=1; y=x++; (same as x=2; y=1;)

Relational and comparison operators

(4 == 8) (false)

(3 < 5) (false)

(12 != 3) (true)

(7 >= 1) (true)

((f = 34) == 35) (false)

Logical operators

!(4 == 3) (true since ! is bool for NOT)

&& and || (same as AND and OR)

Conditional ternary operator

a>b ? c : d (*if* a>b return c *else* return d

Bitwise

int a=5; a << 2; (000101 → 010100 = 20)

& or | (bit AND or OR)

Operators

- Some Honourable mentions…

sizeof

- Accepts one parameter, *type* or *variable* and returns by size (*but be careful when using pointers*…) e.g. sizeof(int) returns 4 !

Explicit type casting operator

- To convert a value of a given type to another type !

```
int a;
float b=2.76;
a=(int) b;
```

Operator precedence

- Increment/decrement, bitwise NOT, casting…have higher prcedence over the fundamental arithmetic ones.....and of course / and * precede + and - !

Basic input/output : output

- In C++, input and output operations from the screen, the keyboard or a file, are made using *data streams*.
- The standard C++ library defines some stream objects to facilitate data input/output namely ***cin*** and ***cout***.

Standard output (**cout**)

```
cout << “Good morning class !”;  //print the message on the screen
cout << 787;  //print the integer on the screen
cout << x;  //print value of variable x on the screen
cout << “This prints ” << “as a single ” << “sentence”;
cout << “This prints x=” << x << “ value \n and this line on a newline”;
cout << “This also prints and adds a newline at the very end” << endl;
```

- Difference between */n* and *endl* is on flushing the data stream → only endl does it i.e. *cout << endl ⇔ cout << “/n” << flush; (more relevant for file writing – wait for it...)*

Basic input/output: input

Standard input (cin)

- Using the extraction operator (>>), formatted input reading is trivial

```
int c;
cin >> c;  //reads an integer from std input and waits for ENTER.
```

- *First caveat*: if a user enters abc4567 and not an integer, the extracted variable is unset and the code won't crash...*poor programming* !
- *Second caveat*: when using strings, only a single "word" is extracted (spaces, tabs or ENTER terminate the extraction) → *getline* to the rescue

```
string name;
cout << "Name: ";
cin >> name;  //input "Road Runner"
cout << name; //only "Road" comes out…
getline(cin,name);  //user input ?!
cout << name; // guess what comes out...?
```

```
string name;
cout << "Name: ";
getline(cin,name);  //input "James Bond"
cout << name; //"James Bond" comes out.
```

Basic input/output: using files

- C++ standard library provides simple *methods* to perform output and input of characters to/from files.
- The following libraries *(...classes, but more on that later...)* come to the rescue:

ofstream	Stream *class* to write on files
ifstream	Stream *class* to read from files
fstream	Stream *class* to both read and write from/to files

```
// basic file write
#include <fstream>  //for std::ofstream
using namespace std;

int main () {
  ofstream file1;
  file1.open ("lemma.txt");
  file1 << "I love studying...\n";
  file1.close();
  return 0;
}
```

```
// basic file read
#include <fstream>  //for std::ifstream
using namespace std;

int main () {
  string mymojo;
  ifstream file2("lemma.txt");
  getline(file2,mymojo);
  file2.close();
  cout << "My mojo is : " << mymojo << endl;
return 0;
}
```

Basic input/output: using files (multi lines)

```
// multi line file write
#include <ostream>   // for std::flush
#include <fstream>    // for std::ofstream
#include <unistd.h>   // to use sleep(time)
using namespace std;

int main () {
 ofstream outfile (”count.txt");
  for (int n=0; n<20; n++) {
    outfile << n << "\n" << flush;
    //outfile << n << "\n";
    sleep(1);
}
  outfile.close();
 return 0;
}
```

```
// multi line file read
#include <fstream>  //for std::ifstream
using namespace std;

int main () {
  ifstream infile ("count.txt");
  int count=0; int array[200]; // let's exagerate,
unknow file size...
  while (infile >> array[count] && count < 200) {
     cout << "n=" << array[count] << endl;
     count++;
  }
  infile.close();
  return 0;
}
```

→ **Flush** is essential to flush data to file while we cycle in the loop !
→ A simple “\n” ***won’t do !!!***

→ **Extract** from the stream while you can !
→ Get number of lines as a bonus…

Basic input/output: formatted (in)output

- C++ standard library provides simple *methods* to perform formatted input and output of data using ***scanf*** and ***printf*** but also building on ***streams***.

Basic formatting

```
#include <iomanip>     // for setw, setprecision
#include <iostream>    // for cout
using namespace std;
int main() {
  double M;
  /* Read in value of M */
  printf("\nM = ");
  scanf(”%lf", &M); //must be a reference to M
  //watch out for setw() if width not large enough
  cout << setprecision(9); cout << fixed;
  cout << “I read “ << setw(12) << M << endl;
  printf(“I read %.9lf\n”,M);
}
```

Multiple columns

```
#include <math.h>     // for sin, atan
#define PI=3.14159;

ofstream out3file ("3columns.txt");
double tmp1,tmp2;
for (int n=0; n<11; n++) {
  tmp1=sin( PI*(double)n/10. );
  tmp2=cos( PI*(double)n/10. );
  out3file << setprecision(9);
  out3file << fixed;
  out3file << setw(13) << (double) n <<
  setw(13) << tmp1 << setw(13) <<
  tmp2 << "\n" << std::flush;
}
out3file.close();
```