COMPUTATIONAL PHYSICS

Introduction to C++

Statements and flow control

Functions, overloading

Namespaces

Arrays

Pointers, dynamic memory

Statements and flow control

- In C++ we are never restricted to simple statements e.g.

```
int A=23; string s="hurray !"
```

- In numerus occasions we want to execute multiple commands pending some flow logic e.g.

```
if (condition) statement1;
```

```
if (condition) {
  statement1;
  statement2; }
```

- Note that the *compound* **{…}** statement does ***NOT*** end in **";"**
- Identation and line breaks are irrelevant…

```
if (condition) {statement1; statement2;}
```

- What about "if-then-else" ?

```
if (condition) {
  statement1;}
else {
 statement2;}
```

```
if (condition1)
  statement1;
else if (condition2)
  statement2;
else
  statement3;
```

Iteration statements

- We also have iteration statements…

while loop

```
while (condition) {
  statement1;
  statement2; }
```

do-while loop

```
do {
  statement1;
  statement2;
} while (condition);
```

N.B. statement is evaluated at least once

for loop

```
for (initialization; condition; increment) {
  statement1;
  statement2; }
```

Example 1

```
for (int i=0; i<10; i++) {
  statement1;
  statement2; }
```

Example 2

```
for (int i=10; i>1; i--) {
  statement1;
  statement2; }
```

Example 3

```
for (int i=1, j=6; i<j; i++, j--) {
  statement1;
  statement2; }
```

Jump and selection statements

- Who never wished to drop out of a loop...

break

```
for (i=1;i<11;i++) {
  cout << i << ", ";
  if (i==7) {
      cout << "I hate 7 ! I'm leaving…";
      break;} //drop out from loop…
}
```

continue

```
for (i=1;i<11;i++) {
   if (i==7) {
      cout << "I hate this number !, ";
      continue;} //jump to next iterator !
   cout << i << ", ";
}
```

- Or thought about choosing different options ...?

switch

```
switch (variable) {
  case value1:
    statement1; break;
  case value2:
    statement2; break;
  default:
    statement3;
  }
```

Functions

- In C++ we can define functions other than *main()*....
- The general syntax is

```
type function_name(arg1, arg2,…) {statement1, statement2,…}
```

but it *hides* some neat even if obvious *features*:

- **A function must be *declared* before being used...**
- **The function *arguments* must be given a type...**
- **A function need not have an explicit *return* type or statement...yet still return data !**
- **A maximum of *one expression* can be included in a *return statement*.**
- **The function *arguments* can be given *default values* and can be set to *constant*.**
- **Variables defined inside the *scope of the function* are only local.**

- Let's see how this comes into play…

Functions – declaration and specification

- A function must be declared before being used, arguments given a type.

```
int multiply(int a, int b) {
  int result; result=a*b; return result;
}
int main() {
  int x=23; int y=45;
  cout << multiply(x,y) << endl;
}
```

AND

- Variable *result* is only valid within the function's scope **{…}** !
- Impractical with too many functions → ***PROTOTYPING***

```
int multiply(int, int); // function prototype

int main() {
  int x=23; int y=45;
  cout << multiply(x,y) << endl;
}
int multiply(int a, int b) {
  int result; result=a*b; return result;
}
```

- Function prototype appears before *main()* and ends with ";"
- Argument's name are irrelevant.
- Function is defined after *main()*.
- *Layout will become way simpler once we introduce header files*

Functions – void and passing arguments

- A function without an explicit return statement

```
void salute(string a) {
  cout << "Good morning " << a;
}
int main() {
  string x="John";
  salute(x);
}
```

OR

```
void sad() {
  cout << "I am a function !";
}
int main() {
  sad(); // sad; would compile fail !
}
```

- What about returning values from a function without an explicit return ?
  - Is that even possible ???....***YES !***
  - It is enough that one of the input arguments is actually an output one !
- Passing arguments to a function can be done by ***value*** or by ***reference*** so let's see how it's done...

Functions – passing by value and reference

- Passing by *value*, the argument's value is *copied* → changes are local !

```
int test(int a) {
  a++; return a;
}
int main() {
  int x=23;
  cout << test(x) << " , " << x << endl;
}
```

→ **24 , 23**

Why: Memory addresses are different !

- Passing by *reference*, the actual argument variables are *given* → changes are global since same memory address is used !

```
int test(int &a) {
  a++; return a;
}
int main() {
  int x=23;
  cout << test(x) << " , " << x << endl;
}
```

→ **24 , 24**

Why: Memory addresses are the same !

Functions – passing by value and reference

- ***In practice***, we can easily have a void return function with the output as an argument passed by reference !

```
void multiply(int a, int b, int &c) {
  c=a*b;
}
int main() {
  int x=6; int y=7; int z;
  multiply(x,y,z);
  cout << x << " x " << y << " = " << z << endl;
}
```

- Some side notes:
  - The stream extraction with multiple "<<" is horrible → **printf** *formatted output preferred*
  - Caller routine is passing the variable's *value* and not the variable's *address*. There is a alternative way where one passes the variable's address as argument → *pointers*
  - If one adds a **const** qualifier to an argument passed by reference it ***cannot*** be changed.

Functions – inline functions

- Calling function is not trivial for the compiler: compiled function lives in different memory space and jumping back and forth is inevitable.
- C++ comes to the rescue with *inline* keyword: function is actually inserted by the compiler where used ! Only feasible for small functions…

```
inline void multiply(int a, int b, int &c) {
  c=a*b;
}
int main() {
  int x=6; int y=7; int z;
  multiply(x,y,z);
  cout << x << " x " << y << " = " << z << endl;
}
```

N.B. : inline functions MUST be fully defined before the compiler "sees" their usage → PROTOTYPING is not enough (*they must be included in header files*)

Functions – default values

- A function can have default argument values in case only first or no arguments are given.
- ***Be careful with prototyped functions...***

```
int multiply(int x=1, int y=1); // function prototype

int main() {
   int x=23; int y=45;
   cout << multiply(x,y) << endl;  // prints 1035
   cout << multiply(4) << endl;  // prints 4
   cout << multiply() << endl;  // prints 1
}
int multiply(int a, int b) {
   int result; result=a*b; return result;
}
```

➔ ***Do not set defaults on definition, only on prototype !***

Functions – overloading

- Motivation is obvious: one same "operation" has meaning for different data types so…why not using the same function name e.g. SUM

```
int SUM(int a, int b) {
  int c; c=a+b; return c;}
double SUM(double a, double b) {
  double c; c=a+b; return c;}
string SUM (string a, string b) {                // Luckily "someone" defined what
  string result; result=a+b; return result;}    // "+" means for strings...concatenation !

int main() {
  int a=23; int b=45;
  cout << SUM(a,b) << endl;  // 68
  double c=2.5; double d=4.0;                  // Beware: overloading MUST mean different
  cout << SUM(c,d) << endl; // 6.5             // input types. Different return types WILL NOT DO !
  string e="Polly "; string f="Dolly";
  cout << SUM(e,f) << endl;
}
```

Namespaces

- We have seen several times the statement *using namespace std*;
- A ***namespace*** groups entities into the namespace's scope only.
- Practical example to see how it works...

```
namespace human {
  int weight=70; string speak() {return "Hello";}
}
namespace dog {
  int weight=12; string speak() {return "Uauf";}
}
int main() {
  cout << human::speak() <<"\n"; cout << dog::weight << "\n";
  using namespace dog; //namespace entities always seen in "interior" scopes...careful!
  cout <<"Dogs do " << speak() <<"\n";
  cout << human::speak()<< " , humans avg weight is " << weight << " quilos !" << endl;
}
```

Which one comes out ?

Arrays

- In numerous applications, we deal with multiple data values with the same characteristics e.g. position of projectile over time, phone number list, number sequence in Euromilhões. → **ARRAYS**
- All array's elements have the same ***type*** and an associated ***subscript*** (always starting at 0 – *unless with pointers to halfway the array*).
- Consecutive array elements are stored *contiguous* in memory (*handy…*).
- Multidimensional arrays can also be defined. Storage in memory is in row-major order i.e. A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{33}.

Syntax

```
int a[3]={1,2,3};
int a[]={1,2,3}; //same as before
int a[3]={};  //all elements are 0.
int a[3];  //unknown but memory is allocated !
int a[3]; a={1,2,3};  //COMPILE FAIL
int a[3]; a[0]=1; a[1]=2; a[2]=3;  //fine !
```

```
int a[3][2]={{1,2},{4,1},{-3,20}};
int b[3][2];
for (int i=0;i<3;i++)
  for (int j=0;j<2;j++)
    b[i][j]=i*j;
```

Pointers – concept and declaration

- ***Variables*** are handy: given the identifier (name) and value, one need not care about where the data lies in memory space (*address*).
- What if, rather than a value, we store an ***address*** ? If the value being stored at that address changes…we may "track change" it → ***POINTERS***

a	identifier	
112	value	
0x7fff516d75e0	address	→ &a

- One can define a ***pointer p*** that ***points*** to variable ***a***, holding address of ***a***. By *dereferencing* the pointer, we gain access to the value in ***a.***

```
int a=112;  // &a=0x7fff516d75e0
int * p=NULL;  // p as pointer to integer
p = &a;  // p=0x7fff516d75e0
cout << "a=" << *p;  // * is dereference op.
```

```
a++; // a=113
cout << "a=" << *p;  // *p is also updated !
*p=125;
cout << "a=" << a;  // can you guess it ?
```

Pointers – connection with arrays

- Arrays are very closely related to pointers → removing the subscript the array is basically the address to it's first element !

```
int a[2]={1,2};
cout << a;      //prints 0x7fff51d275d0
cout << a+1;  //prints 0x7fff51d275d4
```

a <=> &a[0]

a+1 <=> &a[1]

- If one defines a pointer ***p*** that points to array ***a***, interesting things emerge:

```
int a[3]={1,2,3};
int * p=a; // a means address of a[0]
*p=10;
++p; *p=20;
p=a+2; *p=30;
...can you guess what a[] now has ?
```

```
int a[3]={1,2,3};
*a=10;  // same as a[0]=10 !!!
*(a+1)=20;  // same as a[1]=20 !!!
*(a+2)=30;  // same as a[2]=30 !!!
...yes...we can dereference an array !
```

N.B.1 Assigning an array with subscript higher than size does ***NOT*** raise an exception → unknown behaviour !!! e.g. `p=a+3; *p=666;` would compile & run

N.B.2 You can use *(p+i) or p[i] but be **ALWAYS** sure of where **p** is pointing to…

Pointers – …the "devilish" const qualifyer…

- ***Basic idea***: if we want some variable/object *not to be changed* → qualify it with a ***const*** qualifier !

```
int a=5; a=14;  // no problem…
const int b=5;
int const c=5; c=14;  // error: assignment of read-only variable 'c'
```

- Likewise for pointers but there is a catch: what is constant ?
  - **The address stored by the pointer ?**
  - **The value the pointer is pointing to ?**

```
int x=10;
int * p1=&x; *p1=23; // p1 is a pointer to a int
int const * p2=&x;     // p2 is a pointer to a const int…*p2=66 → ERROR !
int * const p3=&x;     // p3 is a const pointer to an int….p3=p1 → ERROR ! *p3=212 → OK !
int const * const p4=&x;  // p4 is a const pointer to a const int…*p4= or p4= → ERROR !
```

N.B. Constant variables/pointers ***MUST*** be assigned immediately when declared !

Pointers – const, vars, references, pointers

- ***References***: a quick "shortcut" to variables but to handle with care…
  - ***MUST*** be assigned immediately when declared !

```
int a=5;
a=14;  // change a to 14
int &refa=a;  // refa has same memory address of a !
int *ptra=&a;  // prta points to a.
refa=57; // now both a and refa store 57 ! And *ptra=57 too !
```

```
int const c=5;
int &refc=c; //FAIL ! The int is const
int const &refc=c;  //OK
int const *ptrc=&c;  //OK

int d=5;
int const &refd=d;  //OK
int const *ptrd=&d;  //OK
refd=57; *ptrd=57;  //FAIL, ref/ptr to const int !
d=57; //OK, refd and ptrd both point to new value !
```

Pointers – strings and char sequences

- In C++, a string is a sequence of characters → natural to consider arrays of char → termination by “\0” char in both cases.
- Power of “string” lies on size/memory management...

```
string name=“Jose”;
name=“Bernardo”; // OK !
```

```
char name[]=“Jose”;
name=“Miguel”; //disallowed !!! ERROR
name[4]=“!”; // no compile error but we mustn’t !
```

- And now with pointers...maybe we bypass the char[] limitation...

```
// 2 Compiler warning !!!
char * name=“Jose”;
name=“Susana”;
```

String literal is a const !

Yet compiler allows reassignement

```
// No warning...
const char * name=“Jose”;
```

```
string job1=“Pilot”;
string * job2=“Pilot”; // ERROR
string * job2=&job1;  // OK
*job2=“Teacher”; // OK
```

Strings are best dealt with
string objects (std lib or classes)

Pointers – argument to functions

- In C++, one can pass ***arguments*** by ***value***, by ***reference*** but also by ***pointer*** !

```
void multiply_by_ref(int a, int b, int &c) {
  c=a*b;
}
void multiply_by_ptr(int a, int b, int *c) {
  *c=(a*b);
}
int main() {
  int x=6; int y=7; int z;
  multiply_by_ref(x,y,z);
  cout << x << " x " << y << " = " << z << endl;
  multiply_by_ptr(x,y,&z);
  cout << x << " x " << y << " = " << z << endl;
}
```

- Also, unlike "scalar" variables, ***arrays*** must be passed by ***reference*** or ***pointer*** *(together with array size...)* !

Pointers – pointers to functions

- In C++, it is sometimes flexible to have pointers to functions !

```
float method1 (int a, int b){ ...some algebra here...
return result; }
float method2 (int a, int b){ ...some algebra here...
return result; }
float ghost_op (int x, int y, int (*funct)(int,int)) { //funct is ptr receiving 2 int and outing an int
   int g;
   g = (*funct)(x,y);
   return g;
}
int main () {
   float n;
   float (*method)(int,int); //a pointer to a function of 2 int that returns a float
   method = &method1;
   n = ghost_op (2, 4, method);
   method= &method2;
   n = ghost_op (2, 4, method);
}
```

Pointers – pointer to pointer…

- ***Basic idea***: if a pointer can point to an array...can a pointer to a pointer cast a 2D array ?!....well...much like an “array of arrays” ?!

```
int f[3][2]={{1,2},{4,1},{-3,20}};
```

f is a 2D array….ok...though not yet there...

```
int f[3];
f[0]=a1;  //where int a1[]={3,5,6};
```

ERROR...f[0] should hold an int and not an array !

```
int (*f)[3]; //a pointer to array of 3 int
```

Nope...(*f)[i] is still an int....

```
int * f[3]; //array of 3 pointers to int
f[0]=a1;  //yes! f[0] points to a1
f[1]=a2;  // and f[1] points to a2
f[2]=a3;  // and f[2] points to a3
```

Already some improvement...

```
int ** f; //a pointer to a pointer to int
```

But now what ?

Dynamic allocation to the rescue !

Pointers – dynamic memory

- Before pushing for "2D arrays", let's see what dynamic memory allocation is and how it's used...
- Fundamentally 2 different memory spaces: ***Stack*** and ***Heap***
  - ***Stack memory*** is allocated automatically, in contiguous blocks but limited and fixed size.
  - ***Heap memory*** is user allocated, random order (can fragment) and sized at runtime !
- What this means in practice

```
int * d;  //pointer to int
d = new int[3]; //actually.....point it to array of 3 int (-:
for (int i=0;i<3;i++) {
  d[i]=i*2; // we cannot assign d=arrayX → delete[] fails
  cout << "d[" << i << "]=" << d[i] << endl;
}
cout << endl;
delete[] d;
```

→ Allocate memory

→ Deallocate memory

- Syntax to free memory: **delete[]** for array and **delete** for value

Pointers – dynamic memory

- Back to the “2D arrays”...we need to allocate twice (2 levels) !

```
int arr1[]={1,2};…; int arr3[]={14,21};
int ** e;  //pointer to a pointer to int
e = new int * [3]; //actually.....point it an array of 3 pointers to int (-:
e[0]=new int [2]; //well, actually e[0] points to array of 2 int !
for (int j=0;j<2;j++)
   e[0][j]=arr1[j];
…
e[2]=new int [2]; //well, actually e[2] points to array of 2 int !
for (int j=0;j<2;j++)
   e[2][j]=arr3[j];
//print the data if you wish…
for (int i=0;i<3;i++) {
   delete[] e[i];
}
delete[] e;
```

N.B. What *can/cannot* be done when *const* qualifier steps in becomes “*elaborate*”….

Pointers – exception handling

- If RAM is limited...memory allocation can fail. How to check it ?
- Simple: avoid the inevitable ***Exception*** and check for Null pointer...

```
#include <stdio.h>  //for printf
#include <stdlib.h> //for exit(1) call
#include <new>      //for nothrow
using namespace std;
int main() {
        float* gv = new (nothrow) float [5000000000000000];
        if (gv != NULL) { // check for null pointer
          printf(”Wow....enough memory!\n");}
        else {
          printf("Failed to allocate pointer but ends gracefully !!!\n");
          exit(1);
        }
}
```