COMPUTATIONAL PHYSICS

Introduction to C++

Static and global variables

Preprocessor directives

Header files

Random and complex numbers

STL library

Static and global variables

- Static variables are instantiated only once ! Global variables are defined outside main and user functions and are....globally available.

```
int gvar=0; //if un-initialized it is set to 0
void foo(string name)
{
  static int n = 0;
  n++; cout << n << endl; cout <<
"Greetings " << name << endl; gvar++;
}
int main() {
  char tmp[21]; string name="";
  while (name != "chega") {
    scanf("%20s",&tmp);
    name=tmp;
    foo(name);
  }
cout << gvar;return 0;
}
```

- ***Local*** variables only "live" inside the function's scope ***{}*** (it is ***private*** and lost on exit). If non-initialised value is unknown…
- ***Global*** variables are ***public*** and avaliable everywhere. Since they have "static storage" (*persistent*) nature, if non-initialised they are set to 0.

Preprocessor directives

- We have seen numerous times many #include or #define. These are cast under the ***preprocessor directives***.
  - Directives the C++ compiler "*resolves*" before compilation starts...

#include	#include <**header**> or #include "**file**" to be inserted.
#define MACRO value	Define a macro (variable); MACRO is replaced in source code
#undef MACRO	Undefine the MACRO.
#ifdef MACRO…#endif	If the MACRO is defined execute statements
#ifndef MACRO…#endif	If the MACRO is ***not*** defined execute statements

- C++ also offers some "free" macros that are very handy ("__MACRO__")

__FUNC__	Function name where macro statement is.
__PRETTY_FUNC__	Function declaration (GNU extension)
__FILE__	String with source filename (with macro) being compiled.
__DATE__	String in "Mmm dd yyyy" format (month day year).
__TIME	String in "hh:mm:ss" with compilation start time.

Header files – why, content, code build

- Header files play crucial role in C++.
  - *Speeds up* compilation time → multi file project
  - Keeps you code *organized* → essential in large projects.
  - *Interface* and *implementation* are clearly separated
- Header files are used for
  - function prototypes
  - symbolic constants defined using #define or const
  - structure declarations
  - class declarations
  - inline functions
- In a nutshell, header files provide "object" awareness to all source codes that include them !
- *Let's check with an easy example...*

Separating interface and implementation

- Don't forget: Header files have the ***interface*** only !

Header file (functions.h)

```
int multiply_value(int a, int b);
void multiply_ref(int a, int b, int &c);
void multiply_ptr(int a, int b, int * c);
```

Souce file (functions.cpp)

```
#include "functions.h" //optional here...
int multiply_value(int a, int b) {
  int result; result=a*b; return result;}
void multiply_ref(int a, int b, int &c) {
  c=a*b;}
void multiply_ptr(int a, int b, int * c) {
  *c=a*b;}
```

Main file (main.cpp)

```
#include <iostream>
#include "functions.h"
using namespace std; //cout
int main() {
  int a=5; int b=50; int out; cout << multiply_value(a,b) << endl;
  multiply_ref(a,b,out); cout << out << endl;
  multiply_ptr(a,b,&out); cout << out << endl;
}
```

…a bit more complex now...

- Placing all function implementations in .cpp file becomes trickier...

Header file (functions.h)

```
#include <string>
struct country {
  std::string name;
  int population;
  void test(int &a);};
int multiply_value(int a, int b);
```

Souce file (functions.cpp)

```
#include "functions.h" //compulsory here...
void country::test(int &a) {a=2;};
int multiply_value(int a, int b) {
  int result; result=a*b; return result;}
```

Main file (main.cpp)

```
#include <iostream>
#include "functions.h"
using namespace std; //cout
int main() {
  int a=5; int b=50; int out; cout << multiply_value(a,b) << endl;
  country A; A.name="Wonderland"; A.population=120000;
  A.test(a); cout << multiply_value(a,b) << endl;
}
```

To compile...

gcc *.cpp -o main.exe

...more on multi file compilation later...

Main program with arguments

```
#include <iostream>
using namespace std; //cout and stof

int main(int narg, char * args[]) {
  //inserts a new line and flushes the stream
  cout << "Number of arguments = " << narg <<  endl;
  cout << "Arg1+10.0=" << stof(args[1]) + 10.e0 << endl;
return 0;
}
```

Only the actual input arguments are given on code call

Non char input needs conversion…

- The code being called on the Terminal is ***ALWAYS*** considered to be the first argument → ./main.exe 1 3 5 7 yields 5 arguments

Random numbers

- Random numbers are ubiquitous in Science e.g. Statistics, Games, Brownian motion. We need:
  - Random number generators e.g. linear congruential generator (look it up !)
  - Control over reproducibility of the random sequence i.e. *repeat* or draw *new one*…
- What C++ has to offer:
  - rand() – “random” integer between 0 and RAND_MAX (cstdlib.h)
  - srand(int) – “seed” generator. Different argument sets different sequence
- Common practice to ensure always a different sequence: elapsed time in seconds since some date.
  - The time(NULL) function call (ctime.h) returns elapsed time since 00:00 UTC 01/01/1970
- *Let’s check with an easy example...*

Random number example

- Random generate sequence in [0,1] with 100000 samples and time the execution...

```
#include <iostream>  //cout
#include <ctime>  // time (), clock()
#include <cstdlib>  // rand()
using namespace std;

int main() {
  srand(time(NULL)); //set random number seed generator
  clock_t time1=clock(); //starting clock ticks
  double b;
  for (int i=0;i<100000;i++) {
    b=(double)rand()/(double)RAND_MAX; //uniform in [0,1]}
  clock_t time2=clock();
  double lap=((double)time2-(double)time1)/(double)CLOCKS_PER_SEC;
  cout << "Time elapsed in seconds = " << lap << endl;
}
```

Complex numbers

- Contrary to C, C++ features complex numbers (*and arithmetic*)
- Including complex.h header file (*#include<complex>*) we access:
  - real(complex<double>), imag(complex<double>), abs(complex<double>)
  - All usual arithmetics in complex numbers

```
#include <iostream>  //cout
#include <complex>  //complex
#include <cmath>  //sqrt
using namespace std;
typedef complex<double> dcomp;  //define a "custom" type (alias)
int main() {
  dcomp i(0.,1.);  // 0 + i*1  complex number !
  dcomp a=2.+i;  // a= 2 + i
  dcomp mod2_a=a*conj(a);  // modulus squared of a → still a complex !
  cout << sqrt(real(mod2_a)) << endl;  // now square root the real part...
  cout << abs(a) << endl;   // and use the abs() just to check...
}
```

The Standard Template Library (STL)

- The STL is a set of C++ template classes to provide common programming data structures and functions such as ***lists***, ***stacks***, ***vectors***, etc.
- It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.
- STL has four components:

Algorithms	***sorting***, searching, accumulate,...
Containers	***vector, list, map, stack***,…
Functions	functors
Iterators	***iterators***

- Let's examine the most commonly used ones with some simple examples

STL: vector

- Vectors are *sequence containers* representing arrays that can change in size.
- Vectors use contiguous storage locations for their elements → *Vec[index]* is perfectly legitimate ! For efficiency, the vector's *capacity* is slightly higher than the *size* requested/needed.
- Flexibility has it's cost: adding new elements halfway forces "pushing" all elements in front (*performance penalty*)
- Some of the existing "*methods*"…

begin(), end()	Return iterator to beginning/end of vector
size()	Returns size of vector
push_back(data)	Add data of given type to the end of the vector
front(), back()	Returns first and last vector element
insert(args)	Insert new elements on vector (3 options)

STL: vector examples

- Some of the usual vector manipulations…

```
#include <iostream>  //cout
#include <vector>
using namespace std;
int main() {
  vector<int> g1; int arr[]={11,4,-7};
  for (int i = 1; i <= 10; i++) {g1.push_back(i * 10);}
  cout << "\n Reference operator [g] : g1[2] = " << g1[2];
  cout << "\n at : g1.at(4) = " << g1.at(4);
  cout << "\n front() : g1.front() = " << g1.front();
  cout << "\n back() : g1.back() = " << g1.back();
  int* pos = g1.data(); // pointer to the first element
  cout << "\nThe first element is " << *pos << endl;
  vector<int>::iterator vecit = g1.begin(); //random access iterator to first element
  cout << "value in g1[0] using iterator=" << *vecit << endl;
  g1.insert(vecit+2,arr,arr+3); //new elements inserted at position vecit+2 (3rd in
                                //this case) using arr[0] up to arr[2] (last is excluded)
}
```

STL: vector examples

- To initialize a vector there are several choices...

```
#include <iostream>  //cout
#include <vector>
using namespace std;
int main() {
  vector<int> g1; //empty vector size=0
  vector<int> g2(3); //vector of 3 int….filled with 0
  vector<int> g2(5,10); //vector of 5 int….filled with 10
  vector<int> g3[2]; //array of 2 vectors and each vector can be sized differently !
  vector<int> g4(g3); //g4 is a copy of g3
  vector<int> g4(g3.begin()+1,g3.begin()+3); //g4 has g3[1],g3[2]. Note last excluded
  int arr[]={2,4,6,8,10,12};
  vector<int> g5(arr+1,arr+4); //g5 has arr[1],..,arr[3]. Note last excluded
  vector<int> g6(&arr[1],&arr[1]+3); //g6 has arr[1],..,arr[3]. Note last excluded
  vector<int> g7; g7.assign(arr+1, arr+4); // similar to vector<int> g5(arr+1,arr+4)
  vector<vector<double> > Array2D; // a vector of vectors...how convenient !
}
```

STL: pair

- An easy way (*a class*) to ***bundle*** together a ***pair of values as a single unit***, which may be of different types (T1 and T2).
- To access the elements, we use variable name followed by dot operator followed by the keyword ***first*** or ***second***.

```
#include <iostream>  //cout
#include <utility>  //container including "pair"
using namespace std;
int main() {
  pair<double,string> g1;        //default (0.0,"")
  pair<int,string> g2(1, "atlas");  //initialized,  different data type
  pair<int,int> g3(1, 10);   //initialized,  same data type
  pair<int,int> g4(g3);    //copy of g3. Same outcome as pair<int,int> g4;g4=g3;
  pair<int,string> g5; g5=make_pair(1, "atlas");  //same as g2
  g5.first=121; g5.second="safari";  //either access or change the elements.
  vector<pair<int,int> > human;  /vector of pairs e.g. age and height. Note "> >" identation
  human.push_back(make_pair(12,35));
  human.push_back(make_pair(30,75));
}
```

STL: list

- Similar in intent to ***vector*** but elements not stored contiguous and impact is noticeable
  - ***Arithmetic*** on iterators is ***NOT*** allowed.
  - Indexing with **[]** is ***NOT*** allowed…

```
#include <iostream>  //cout
#include <utility>  //container including “pair”
using namespace std;
int main() {
  list<int> g2;
  list<int>:: iterator listit;
  for (int i = 1; i <= 6; i++)
    g2.push_back(i * 6);
  int a3[]={-5,0,5}; g2.insert(g2.end(),a3,a3+3); // insert a3 after iterator g2.end()
  listit=g2.begin() ;
  while ( listit != g2.end()) {
    cout << "value in g2=" << *listit << endl; //g2[i] fails !
    listit++; //luckily this operator applies...}
}
```

STL: map

- Maps are containers that store elements formed by a combination of a *key* value and a *mapped* value (can be any type…), following a specific order.

```
#include <iostream>  //cout
#include <map>
using namespace std;
int main() {
  map<char,int> mymap;
  mymap['a']=97; mymap['b']=98; mymap['c']=99; mymap['r']=100;
  map<char,int> mycopy(mymap.begin(), mymap.end());
  map<char,int>::iterator it; it = mymap.find('b'); cout << it->first << "->" << it->second; << endl;
  map<char,int>::iterator it2;
  it2=mymap.insert(it,pair<char,int>('r',119)); cout << it2->first << "->" << it2->second<<endl;
  it2=mymap.insert(it,pair<char,int>('h',235)); cout << it2->first << "->" << it2->second<<endl;
  mymap.erase('b');
}
```

- There cannot be 2 elements with same *key*: first come-first served ! But one can always change the value of a given key a posteriori.

STL: sort

- Sort methods exist for either random access iterators e.g. ***Vector*** or bidirectional iterators e.g. ***List***. It is also valid for arrays.

```
{
   int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
   int n = sizeof(arr)/sizeof(arr[0]);
   sort(arr, arr+n, greater<int>()); //std::greater, we could use less() for "lower than”

   int arr2[] = {-1, 5, 8, -9, 6, 7, 13, 4, 2, 0};
   vector<int> myvector (arr2, arr2+n);
   sort (myvector.begin(), myvector.begin()+n);

   list<int> mylist (arr2, arr2+n);
   list<int>::iterator itlist;
   mylist.sort(); //no random access iterator so "ordered iterator" meaningless
}
```

- For decreasing order, one can use a dedicated method (*extra argument*)