Introduction to Modern C++

Lab Session 2
Math, Arrays, Vectors, and the Quake III Arena trick

This lab session is dedicated to fundamental types, and introducing some important dependent types.

1 A simple example: Computing π

Task 1. We will now compute a rough approximation of $\pi \approx 3.14$. A fast way to compute this approximation is to use a Bailey–Borwein–Plouffe formula:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]. \tag{1}$$

This series converges quickly. Let's write:

$$a_k = \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \tag{2}$$

So that now, $\pi = a_0 + a_1 + a_2 + \cdots$. Complete the following C++ program to compute an approximation of π up to a_2:

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

int main() {
  double a0, a1, a2;
  double pi;
  int k;

  k = 0;
  a0 = 1.0/pow(16, k) * (4.0/(8*k+1) - 2.0/(8*k+4) - 1.0/(8*k+5) - 1.0/(8*k+6));
  k = 1;
  a1 = 1.0/pow(16, k) * (4.0/(8*k+1) - 2.0/(8*k+4) - 1.0/(8*k+5) - 1.0/(8*k+6));

  // Your code here

  pi = a0 + a1 + a2;

  std::cout.precision(17);
  std::cout << "Pi is around " << pi << std::endl;
}
```

What is the purpose of `std::cout.precision(17)`? Try augmenting the precision. Is there a maximum value? You can modify the program to compute a better approximation, up to a_5 for instance. What is the relative error, knowing that the first decimals of π are 3.14159265358979323 84?

● **Task 2.** Translate into C++ the following fomulae:

$$x = \cos(\sin(\cos(\sin(42)))), \qquad y = 2^x + 42, \qquad z = \operatorname{Arccos}\left(\frac{\sqrt{y}}{y}\right) \tag{3}$$

Hint: The arccos function is called `acos` in C++. If everything went well, you should find $z = 1.4191$.

2 Some limits of fundamental types

Remember: `int` $\neq \mathbb{N}, \mathbb{Z}$ and `float`, `double` $\neq \mathbb{R}$. In this section we experiment with the `int` and `float` types to see where they differ from their mathematical counterparts.

Task 3. Try the following program. What happens and why?

```
#include <iostream>

int main() {
  int num;
  num = 16777126 * 16777;
  std::cout << num << std::endl ;
}
```

Try to find the large value of `num` such that this program displays `0`.

Task 4. We already saw that `double` has a limited precision. But there are other things to be cautious about. Consider the following code:

```
#include <iostream>

int main() {
  float x, y, z;

  x = 1e32;
  y = 1.0/x;
  z = x/y;
  z = z * 0;

  std::cout << "x + y - x = " << x + y - x << std::endl;
  std::cout << "x / y = "      << x/y       << std::endl;

  std::cout << "x == x ? " << (x == x) << std::endl;
  std::cout << "y == y ? " << (y == y) << std::endl;
  std::cout << "z == z ? " << (z == z) << std::endl;
  std::cout << "z == 0 ? " << (z == 0) << std::endl;
}
```

What happens and why? If you replace `float` by `double` the above code behaves “normally”. Can you find a value of `x` so that the problems appear again? Hint: Choose a larger value of `x`.

Remark 1. There are many other quirks and gotchas with floating-point numbers, for the most part documented in the IEEE 451 document. Amongst other examples, in general :

$$x * (y + z) \neq x * y + x * z, \qquad (x * y) * z \neq x * (y * z), \qquad (x + y) + z \neq x + (y + z)$$

We will ignore this for the rest of the course, but *please keep it somewhere in your mind*.

3 Arrays and Vectors

An `array` is a sequence of n values, where n is fixed in advance. A `vector` is a sequence of n values, but n may be changed at any time. Other than that they are used identically. They are important examples of *dependent types*.

The following program creates an `array` and displays its contents:

```
#include <iostream>
#include <array>

int main() {
  std::array<int, 6>  myarray = {3, 1, 4, 1, 5, 9};

  std::cout << "myarray : ";
  for(int element : myarray) {
    std::cout << element << ", ";
  }
  std::cout << std::endl;
}
```

Task 5. Play with this program: What does the 6 mean? What does the int mean? Modify the program so that myarray contains the sequence : 3, 1, 4, 1, 5, 9, 2, 6.

Vectors are like arrays, but we don't need to specify the size. Example :

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> myvector;

  myvector.push_back(3);
  myvector.push_back(1);
  myvector.push_back(4);

  std::cout << "myvector : ";
  for(int element : myvector) {
    std::cout << element << ", ";
  }
  std::cout << std::endl;
}
```

Task 6. Again, play with this program. What does push_back do?

Task 7. You can access the i-th element of an array (or a vector) using the syntax myarray[i-1]. Display the first element of myarray. What is its type? Change the second element of myvector to 73.

4 ● Type representations and the Quake III Arena trick

Remember that types tell us how to interpret data. The same data can be interpreted in two different ways, e.g. as an integer and as a string.

Task 8. Type-safety is essential in C++, and the language usually prevents you from doing silly things. But you can always insist: Try this program for instance

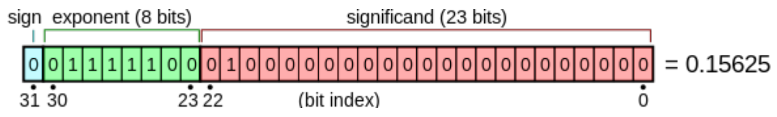
```
#include <iostream>

int main(){
  int a = 1042284544;
  float b = *(float*)&a; // BAD! Type-unsafe conversion

  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;
}
```

Don't worry too much about the strange syntax for a type-unsafe conversion; Just note that although a and b have the *same value* 1042284544, this is not interpreted the same way.

Indeed, a floating-point number is interpreted as follows, differently of an integer:



In 1999, the game Quake III Arena used this to compute quickly the function $1/\sqrt{x}$ *without* computing a square root! You can find all the information on the Internet[1].

The key reason is that interpreting a float x as an integer gives a value that is (almost) proportional to $\log_2 x$. Can you see why?

[1]See the Wikipedia article: https://en.wikipedia.org/wiki/Fast_inverse_square_root.