Assignment 2: Number Limits

EC602 Design by Software

Fall 2017

Contents

1 Introduction

1.1 Assignment Goals

The assignment goals are to

- introduce binary, bytes, and basic principles of integer mathematics.
- introduce the capabilities and limitations of C++ integers
- introduce the capabilities and limitations of floating-point numbers
- provide practice on calculations with units
- provide practice with estimating error bounds

1.2 Group Size

For this assignment, the maximum group size is 3.

1.3 Due Date

This assignment is due 2017-09-19 at midnight

1.4 Late policy

Late assignments will be accepted through a grace period of H hours.

Usually, H will be the number of hours until the beginning of the lecture immediately following the due date (the normal H is 14.5).

The grade will be scaled on a linear scale from 100% at the deadline to 50% at the end of the grade period.

If the *natural grade* on the assignment is g, the number of hours late is h, and the grace period is H, then the grade is

```
grade = (h > H) ? 0 : g * (1- h/(2*H));
```

in C++ or

```
grade = 0 if h>H else g * (1- h/(2*H))
```

in python.

If the same assignment is submitted ontime *and* late, the grade for that component will be the maximum of the ontime submission grade and the scaled late submission grade.

1.5 Submission Link

You can submit here: week 2 submit link

1.6 Points

This assignment is worth 5 points.

2 Overview

2.1 Python integers

In Python, there is one base type for integers: `int`. It can represent any positive or negative integer no matter how large.

Here is an example:

```
>>>bignum=2**(2**30)-1
>>>print(bignum.bit_length()/8/1e6)
134.217728
```

The number `bignum` consists of $2^30 = 1073741824$ consecutive ones in binary, and so requires 134 MB (megabytes) of storage.

Do not try to print `bignum`, it will take too long. However, we can use `bin(bignum)` to take a look at its representation:

```
>>>bin(bignum)[:10]
'0b11111111'
```

2.2 C++ integers

In C++, there are many flavors of integer. The basic one is `int`, but there are also `long` and `short` integers, and signed and unsigned integers.

Read the following two documents:

- Fundamental types (at cppreference) and
- Variables and types (at cplusplus)

Since the actual size of the types `int` and `long int` are compiler and machine specific, when the number of bytes required is important, it is best to use the types defined in `<cstdint>` which are

```
int64_t
int32_t
```

```
int16_t
int8_t
uint64_t
uint32_t
uint16_t
uint8_t
```

2.3 Floating Point Numbers

Please read about floating point numbers here: IEEE floating point

In this assignment, we will focus on three floating point number representations:

- `half` / `binary16`
- `single` / `binary32`
- `double` / `binary64`

Please also see IEEE 754-1985. This standard is superseded, but there are some nice explanations at this page.

2.3.1 Floating Point Numbers in C++

In C++, the commonly used floating point number types are `float` which is normally a `binary32` or single-precision number, and `double` which is normally a `binary64` or double-precision number.

2.3.2 Floating Point Numbers in Python

In python, floating point numbers have the type `float` and are normally `binary64` or double-precision numbers.

It is possible when using the `numpy` library to define other kinds of floating-point numbers, but we won't deal with that in this assignment.

2.4 Timing in C++

It is often useful to measure the time between events, or the amount of time that a segment of code takes to execute. In C++, the `<ctime>` library provides the `clock` function, which is demonstrated below:

```
#include <ctime>

int main() {

  clock_t start_clock, end_clock;
```

```
  int i;

  // start timing
  start_clock = clock();

  for (i=0; i< 1'000'000'000; i++) { };

  // time ends here
  end_clock = clock();

  double seconds = static_cast<double>(end_clock - start_clock) / CLOCKS_PER_SEC;
}
```

Here is a link timed_example.cpp. Try it.

3 Examples

3.1 Effect of different precision

Here is a C++ program which shows how the value 1/3 is different depending on how you store it.

```
#include <iostream>
#include <iomanip>
#include <cassert>

using namespace std;


int main()
{
  float num_f = 1.0/3;
  double num_d = 1.0/3;

  long double num_ld = 1;
  num_ld /= 3;

  long double num_ld2 = (long double)1.0 /3;

  long double num_ld3 = 1.0 /3;

   cout << setprecision(22) << num_f << endl;
   cout << setprecision(22) << num_d << endl;
   cout << setprecision(22) << num_ld << endl;
```

```
   cout << num_d - num_f << endl;

   cout << num_ld - num_d << endl;

   cout << num_ld - num_ld2 << endl;

   cout << num_ld - num_ld3 << endl;

   cout << (long double) 1.0 - ((long double) num_f) * 3 << endl;
   cout << (long double) 1.0 - (num_f * 3) << endl;

}
```

Here is a link errors_in_floating.cpp. Try it.

3.2 Rounding Errors

If floating-point numbers were perfect, the following program would not print anything:

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

int main()
{
  double one_third,zeroish;

   one_third = 1.0/3;

   for (int i=1;i<100;i++)
   {
      zeroish = 1.0 - 3.0 * (i * one_third)/ i;
      if (zeroish != 0)
         cout << i << " " << zeroish << endl;
   }

   return 0;
}
```

However, it actually prints:

```
7 1.11022e-16
14 1.11022e-16
25 1.11022e-16
28 1.11022e-16
```

```
31 1.11022e-16
50 1.11022e-16
53 1.11022e-16
56 1.11022e-16
59 1.11022e-16
62 1.11022e-16
97 1.11022e-16
```

Here is a link rounding_errors.cpp. Try it.

3.3 Overflow and underflow.

Floating point number representations have quirks, such as

- a smallest number greater than zero
- a largest number

```
f = 1
while f > 0:
    f = f/2
    print(f)

g, f = 0, 1.0
while g < f:
    print(f)
    g, f = f, f*2
```

4 The Assignment

The assignment is:

- `integer_limits.py`
- `overflow_times.cpp`
- `number_limits.cpp`
- `earth_electrons.cpp`
- `earth_electrons.py`

Each component is worth 1 point, for a total of 5 for this assignment.

4.1 Integer Limit Calculations

Using python, calculate and print a table for the capability of integers using 1 to 8 bytes of storage

Use the following format string:

```
Table = "{:<6} {:<22} {:<22} {:<22}"
```

to print both the header and the data. This string can be used to create a format: the braces `{}` indicate fields, `<` means left justify, and the number indicates the width of the field to use for this data.

The command to print the header is:

print(Table.format('Bytes','Largest Unsigned Int','Minimum Signed Int','Maximum Signed Int'))

The first two lines of the table should be

```
Bytes  Largest Unsigned Int   Minimum Signed Int     Maximum Signed Int
1      255                    -128                   127
2      65535                  -32768                 32767
```

The filename of the program submitted must be `integer_limits.py`

4.2 Limitations of C++ integers

Consider the following code segment:

```
int m=1;

while (m>0)
  m++;
```

Although *logically*, this is an infinite loop, in practice what happens is that eventually `m` will be represented in memory (binary) as all ones, i.e. it will be `0b111...111`. When it is incremented again, the result is `0b1000...000` but the `1` does not fit into the storage allocated for `m` and so `m` will become zero. This is called "wrap around".

Write a C++ program that will:

- estimate how long an `uint8_t` takes to "wrap around" from a starting value of 1 (this time is too short to measure)
- measure how long an `uint16_t` takes to "wrap around" from a starting value of 1
- estimate how long a `uint32_t` takes to "wrap around" from a starting value of 1 (this time is too long to measure)
- estimate how long a `uint64_t` takes to "wrap around" from a starting value of 1 (this time is too long to measure)

Your program should print out the following:

```
estimated int8 time (nanoseconds): 4564.4
measured int16 time (microseconds): 674
estimated int32 time (seconds): 3.42342
estimated int64 time (years): 45.234
```

except the numbers should be calculated by your code. The measured times will vary: this is natural and ok. Do not do any averaging.

Note that the units must match the ones shown: nanoseconds, microseconds, seconds, and years.

Use the following code as a starting point:

```
std::cout << "estimated int8 time (nanoseconds): "
          << 0 << std::endl;
std::cout << "measured int16 time (microseconds): "
          << 0 << std::endl;
std::cout << "estimated int32 time (seconds): "
          << 0 << std::endl;
std::cout << "estimated int64 time (years): "
          << 0 << std::endl;
```

The filename of the program submitted must be `overflow_times.cpp`

4.3 Number limits

Floating point representation of numbers is superior to integer representation in that numbers "close to zero" like 0.01213 can be represented and used for calculations. It also allows for the representation of very large numbers.

The tradeoff, however, is that floating point numbers cannot represent as many integers accurately (i.e. without round off error) as the integer of the same size.

For this exercise, we define three ratios

```
Rs = factor by which float is better than int at representing small numbers

Rm = factor by which float is better than int at representing large numbers

Ri = factor by which int is better than float at representing integers
```

Here are the formulas for these values

```
Rs = 1 / smallest_float_greater_than_zero

Rm = maximum_float_value / maximum_int_value

Ri = maximum_int_value / N
```

where N is the largest integer such that all integers 1,2,3,. . . ,N can be represented without loss of accuracy by a float of this size.

Repeat this exercise for 16, 32, and 64 bit number representations.

The exact format for the output is specified in the starter program number_ limits_start.cpp

The filename of the program submitted must be `number_limits.cpp`

For this problem, do not consider *denormal* or *subnormal* number formats. Use the *normal* formats only.

4.4 Earth as supercomputer

In this part, we get practice with units and doing floating-point calculations.

Please complete this part in both python and C++. Your goal will be to get identical results using both programs.

Suppose that the Earth is actually a giant supercomputer, and each electron represents a bit of storage (see Hitchhikers Guide to the Galaxy for "historical" reference) Estimate how many electrons are on the earth, and convert this number to an equivalent number of terabytes (TB).

Your program should print out three numbers:

- your estimate in TB
- a lower bound (in TB)
- an upper bound (in TB)

So, for example, a valid print out would be

```
4.5e6
1.0e6
9.0e6
```

which means this group estimated that the Earth has a memory capacity of 4.5 million terabytes, with a lower limit of 1.0 and an upper limit of 9 million terabytes.

The filenames of the program submitted must be `earth_electrons.py` and `earth_electrons.cpp`

5 Assignment checker

For this assignment, it is not practical to distribute a checker, as the checker would necessarily give away the answers.

The checker will be available online, most likely by 9/13.