3-configure-precision

June 3, 2024

1 Configuring the precision

Many scientists choose the double precision type (`double`) by default, out of habit. They suspect that simple precision (`float`) might be insufficient, and higher precision (`long double`) might be slow and bulky. Do not rely on such fuzzy intuition: instead, write your code with a configurable precision, and **try out** different ones.

1.0.1 First pre-requisite : accuracy control

Before trying different floating-point types, one must wonder how to **validate how accurate is the final result**, or at least if it accurate enough.

1.0.2 Second-prequisite : execution time control

When monitoring the execution time, there are a few rules to known about, especially when playing with small/toy code: * run your program many times and compute a mean execution time ; * your program must run long enough, so that the processor pipelines get filled and you go well beyond the initial *computing latency* ; * if the size of your data become too big, you may fill out the processor cache memory, become I/O bound, and the results will not any more express the CPU performance, but the bandwidth of the memory bus.

1.1 First step with `using`

If you are starting from a code written with `double`, you can simply search and replace all `double` with, for example, `real`, and add `using real = double` at the beginning of all your `*.cc` body files, or at the beginning of some `*.h` header file included everywhere.

From there on, you can change your alias into `using real = float`, and recompile everything. Then check if the results are still accurate enough, and measure how faster is execution.

[1]:
```
%%file tmp.precision.h

using real = double ;
```

```
Writing tmp.precision.h
```

[2]:
```
%%file tmp.precision.cpp

#include "tmp.precision.h"
#include <iostream>
```

```
#include <vector>

void reduce( std::vector<real> const & collection )
 {
  real res {static_cast<real>(1.)} ;
  for ( auto element : collection )
   { res *= element ; }
  std::cout.precision(18) ;
  std::cout << res << std::endl ;
 }

int main()
 {
  reduce({ 1.1, 2.2, 3.3, 4.4, 5.5 }) ;
 }
```

```
Writing tmp.precision.cpp
```

[3]:
```
!rm -f tmp.precision.exe && g++ -std=c++17 tmp.precision.cpp -o tmp.precision.
 ↪exe
```

[4]:
```
!./tmp.precision.exe
```

```
193.261200000000031
```

[5]:
```
%%file tmp.precision.h

using real = float ;
```

```
Overwriting tmp.precision.h
```

[6]:
```
!rm -f tmp.precision.exe && g++ -std=c++17 tmp.precision.cpp -o tmp.precision.
 ↪exe
```

[7]:
```
!./tmp.precision.exe
```

```
193.261199951171875
```

This approach requires a complete recompilation and works in an "all or nothing" mode. We can improve it a bit by using a collection of `real1`,`real2`,... type aliases, to be used in different sub-sections of the code.

BEWARE: in order not to stumble on unattended side effects, it is adviced to **track and remove from the original code any form of implicit conversion** between floating-point types.

1.2 Go further with templates

For each portion of your code that can be configured separately, you can create a template, taking as a parameter the floating-point type to be used.

This flexibility comes with a price: the function bodies must be completely moved into header, which causes the usual lengthening of compilation and bloating of executables… until C++20 provide more efficient solutions using *Modules*.

Make all and every class a template will often imply adaptations: * some nested types may become "dependent", and therefore require an additional `typename` keyword ; * some inherited members may become unreachable when the base class become a template, and therefore require the addition of `this->` or `using` ; * some implicit conversion may become dependent on a template parameter, not any more inferable by the compiler, and therefore require to be made explicit.

[8]:
```
%%file tmp.precision.cpp

#include <iostream>
#include <vector>

template< typename Real >
void reduce( std::vector<Real> const & collection )
 {
  Real res = 1. ;
  for ( auto element : collection )
   { res *= element ; }
  std::cout.precision(18) ;
  std::cout << res << std::endl ;
 }

int main()
 {
  reduce(std::vector<double>{ 1.1, 2.2, 3.3, 4.4, 5.5 }) ;
  reduce<float>({ 1.1, 2.2, 3.3, 4.4, 5.5 }) ;
 }
```

```
Overwriting tmp.precision.cpp
```

[9]:
```
!rm -f tmp.precision.exe && g++ -std=c++17 tmp.precision.cpp -o tmp.precision.
 ↪exe
```

[10]:
```
!./tmp.precision.exe
```

```
193.261200000000031
193.261199951171875
```

2 Questions?

3 Exercise

3.1 Initial code

In the example below, we make a big array of unit complex numbers, them make the power of each of them (through multiplication), then reduce the array to a single number (through multiplication).

Depending on the array size and the power degree, we can make the computation either cpu-bound or io-bound. Understand this code and run it.

[2]:
```
%%file tmp.precision.cpp

#include <iostream>
#include <cassert> // for assert
#include <cstdlib> // for rand
#include <valarray>
#include <stdfloat>
#include <complex>
#include <cmath>

// SoA of complex numbers
class Complexes {
  public :

    Complexes( std::size_t size ) : m_rs(size), m_is(size) {}
    std::size_t size() const { return m_rs.size() ; }

    std::complex<double> operator[]( std::size_t index ) const
     { return { m_rs[index], m_is[index] } ; }
    void real( std::size_t index, double value ) { m_rs[index] = value ; }
    void imag( std::size_t index, double value ) { m_is[index] = value ; }

    friend Complexes operator*( Complexes const & lhs, Complexes const & rhs ) {
      Complexes res {lhs.size()} ;
      res.m_rs = lhs.m_rs*rhs.m_rs - lhs.m_is*rhs.m_is ;
      res.m_is = rhs.m_rs*lhs.m_is + lhs.m_rs*rhs.m_is ;
      return res ;
    }

  private :
    std::valarray<double> m_rs, m_is ;
 } ;

Complexes random( std::size_t size )
 {
  srand(1) ;
  Complexes cplxs {size} ;
  for ( std::size_t i = 0 ; i < cplxs.size() ; ++i )
   {
    double e = 2*M_PI*(static_cast<double>(std::rand())/RAND_MAX) ;
    cplxs.real(i,cos(e)) ;
    cplxs.imag(i,sin(e)) ;
   }
  return cplxs ;
```

```
 }

Complexes pow( Complexes const & cplxs, long long degree )
 {
  Complexes res {cplxs} ;
  for ( long long d = 1 ; d < degree ; ++d ) {
    res = res*cplxs ;
  }
  return res ;
 }

std::complex<double> reduce( Complexes const & cplxs )
 {
  std::complex<double> res { 1., 0. } ;
  for ( std::size_t i = 0 ; i < cplxs.size() ; ++i )
   { res *= cplxs[i] ; }
  return res ;
 }

int main( int argc, char * argv[] )
 {
  assert(argc==3) ;
  std::size_t size = atoi(argv[1]) ;
  long long degree = atoll(argv[2]) ;
  std::cout.precision(18) ;

  std::complex<double> res = reduce(pow(random(size),degree)) ;
  double re = res.real(), im = res.imag() ;
  auto r = std::sqrt(static_cast<long double>(re*re+im*im)) ;
  auto n = std::atan(static_cast<long double>(im/re)) ;
  std::cout<<r<<" "<<n<<std::endl ;
 }
```

```
Writing tmp.precision.cpp
```

The main program receives two arguments: * the first argument sets up the size of the collections, and will affect the occupied RAM as well as the I/O volume ; * the second argument sets up the degree which we raise the numbers, and will affect the volume of calculations performed.

By fiddling with these two arguments we can modify the balance between calculation and data volume. Also, with low values, we can have a reference output, that we can quickly check when refactoring the code. With high values, we can profile the performances.

[5]:
```
%%file tmp.precision.sh
echo

opt=${1}
shift
```

```
rm -f tmp.precision.exe
g++ -O${opt} -std=c++17 tmp.precision.cpp -o tmp.precision.exe

./tmp.precision.exe $*

rm -f tmp.precision.py
echo "s = 0 ; m = 0" >> tmp.precision.py
for i in 0 1 2 3 4 5 6 7 8 9 ; do
    \time -f "s += %U ; m += %M" -a -o ./tmp.precision.py ./tmp.precision.exe␣
  ↪$*  >> /dev/null
done
echo "print(\"{:.4} s. {} kB.\".format(s/10.,m/10.))" >> tmp.precision.py
python3 tmp.precision.py

echo
```

```
Overwriting tmp.precision.sh
```

The script receives three arguments: * the first one sets the level of optimization requested from the compiler, * the next two are passed as is to the executable.

[6]:
```
! bash -l tmp.precision.sh 2 1024 100000
```

```
1.00000000022830138 -1.20400202884309004
0.236 s. 3979.6 kB.
```

3.2 To do

Modify the code so that it can turn either in `float`, `double`, `long double`, `std::float16_t`, `std::float128_t`, etc. Of course you can just recompile the code replacing all types each time. Rather try to template the code to the float type used and add an additional command line option in order to choose precision at execution.

Fill in the table below, which summarizes the computation times and the results significant digits, for different accuracies with the arguments `1024 10000`.

Type	time (s)	significant digits
std::float16_t		
float		
double		
long		
std::float128_t		

Try the same with `g++ -O0`, `g++ -O1`, `g++ -O3`, `g++ -Ofast`…