20-numbers-headaches

June 2, 2024

1 Numbers headaches

C++ benefits for scientific computing: * performance and precision is the primary priority * compatibility with C

Drawbacks: * portability is not the primary priority * some C inherited features, such as implicit conversions, may lead to tricky numerical errors.

1.1 Uninitialized variables

First thing to be very careful about: if a basic variable is defined without any initial value, there is no guarantee that its value is set to 0. The C designers probably thought it was a waste of precious runtime…

Recommendation : **always give an initial value to your variables**.

If by mistake you use an uninitialized variable, no doubt that the compiler will give you a warning.

Recommendation : **compiler's warnings should be taken with the utmost seriousness**.

1.2 Unportable numeric types

The size of numeric variable types in C++ depends on the implementation. This may impede the portability of the code.

For example, the following rules are imposed on integer types by the C++ standards: * `short` : a width of at least 16 bits. * `int` : a width of at least 16 bits. * `long` : a width of at least 32 bits. * `sizeof(short)` <= `sizeof(int)` <= `sizeof(long)`

The rules on floating point types are not strict either: * `sizeof(float)` <= `sizeof(double)` <= `sizeof(long double)` * `float` : typically 32 bits *(IEEE 754, 6-9 significant digits, typically 7)*. * `double` : typically 64 bits *(IEEE 754, 15-18 significant digits, typically 16)*. * `long double` : 80 to 128 bits *(18-36 significant digits)*.

For a given platform, the standard class `numeric_limits` can help to check the sizes:

[6]:
```
#include <iostream>
```

[7]:
```
#include <limits>

std::cout
 << "type\tbits\tmin\t\tmax\n"
```

```
<< "int\t" << sizeof(int)*8 << "\t"
<< std::numeric_limits<int>::min() << '\t'
<< std::numeric_limits<int>::max() << '\n'
<< "float\t" << sizeof(float)*8 << "\t"
<< std::numeric_limits<float>::min() << '\t'
<< std::numeric_limits<float>::max() << '\n'
<< "double\t" << sizeof(double)*8 << "\t"
<< std::numeric_limits<double>::min() << '\t'
<< std::numeric_limits<double>::max() << '\n' ;
```

```
type    bits    min             max
int     32      -2147483648     2147483647
float   32      1.17549e-38     3.40282e+38
double  64      2.22507e-308    1.79769e+308
```

1.3 New in C++11: fixed size integer types

C++11 borrows from C99 a bunch of integer types following different rules: - good for memory, the smallest with at least N bits: `int_least8_t`, `int_least16_t`, `int_least34_t`, `int_least64_t` ; - good for performance, the fastest with at least N bits: `int_fast8_t`, `int_fast16_t`, `int_fast34_t`, `int_fast64_t` ; - good for portability, with exactly N bits: `int8_t`, `int16_t`, `int34_t`, `int64_t`.

They are availables when including `<cstdint>`.

1.4 Hardly predictable precision

When an operation mix variables with different precisions, the compiler casts all the operands to the ***best*** precision.

[6]:
```
float f = std::numeric_limits<float>::max() ;
std::cout<<( f * 10.f )<<std::endl ;
std::cout<<( f * 10.  )<<std::endl ;
```

```
inf
3.40282e+39
```

Floating point types are considered better than integer types.

[10]:
```
float f = std::numeric_limits<float>::max() ;
std::cout<<( f * 10 )<<std::endl ;
```

```
inf
```

`unsigned` types are considered better than their `signed` flavor…

[11]:
```
unsigned int i = 1 ;
int j = -1 ;
std::cout<<( i * j )<<std::endl ;
```

```
4294967295
```

Sometimes, computation is performed with a precision higher than the original variables. For example, a `short` based operation will always be evaluated as an `int` (because `int` is the same size as the hardware registers).

[12]:
```
short s1 = std::numeric_limits<short>::max() ;
short s2 = 1 ;
std::cout << (s1+s2) << std::endl ;
```

```
32768
```

[13]:
```
short s3 = (s1+s2) ;
std::cout << s3 << std::endl ;
```

```
-32768
```

BEWARE: Intel processors were typically computing their `double` operations with extraneous digits (80). One may think **the higher the precision, the better**. But it implies a **portability issue**, because the results will differ when running your code on a different processor, when vectorizing, when porting the code to GPU…

1.5 Unnoticed conversions

For ease of writing code, C compilers and thus, C++ compilers as well, are authorized to perform multiple conversions between predefined numeric types. These so-called **implicit conversions** are automatic and often unnoticed by the developer.

Some of these allowed implicit conversions can introduce a loss of precision: for example a transformation from a floating-point number to an integer. The compiler assumes that such ***narrowing*** is done on purpose. Is it still a reasonable assumption for a code made of thousands of lines?

[15]:
```
double pi = 3.1416 ;
int i = pi ;
std::cout<<"double "<<pi<<" => int "<<i<<std::endl ;
```

```
double 3.1416 => int 3
```

[16]:
```
long lmax = std::numeric_limits<long>::max() ;
short s = lmax ;
std::cout<<"long "<<lmax<<" => short "<<s<<std::endl ;
```

```
long 9223372036854775807 => short -1
```

[17]:
```
double dmax = std::numeric_limits<double>::max() ;
float f = dmax ;
std::cout<<"double "<<dmax<<" => float "<<f<<std::endl ;
```

```
double 1.79769e+308 => float inf
```

Even worse, the compiler can transform any signed/unsigned integer into unsigned/signed !

[18]:
```
void display_signed( short v )
 { std::cout<<v<<std::endl ; }
```

```
unsigned short us = 42000 ;
display_signed(us) ;
```

```
-23536
```

[19]:
```
void display_unsigned( unsigned short v )
 { std::cout<<v<<std::endl ; }

short s = -42 ;
display_unsigned(s) ;
```

```
65494
```

BEWARE: when mixing signed and unsigned integers in an expression, the compiler will consider the unsigned flavor as the most accurate, and transform all the integers accordingly. One more time, **paying attention to compiler warnings will help you out**.

1.6 Disputed practice: never use unsigned numbers

…if you can ! - There are some contexts (embedded computing) where every bits is worth saving. - The standard library designers made the choice of unsigned integers for the size and indexes of all the containers :(

1.7 Good old-fashioned practice: make all conversions explicit

In a program of a large size, implicit conversions are more of a hindrance than a help. It is advised to: - set the warning level to maximum, - scrutinize carefully all compiler warnings, - make explicit any conversion you identify in the code.

The C way, for explicit conversions, is to use the type name as a function:

[17]:
```
unsigned short i = 42000 ;
short j = short(i) ;
std::cout<<j<<std::endl ;
```

```
-23536
```

But this does not catch the eye. Better, C++ comes with a set of explicit type casting operators. The one to be used by default is `static_cast`:

[10]:
```
unsigned short i = 42000 ;
short j = static_cast<short>(i) ;
std::cout<<j<<std::endl ;
```

```
-23536
```

Three other type casting operators are available, for rare specific use-cases: * `const_cast` : in rare cases, when one wants to get rid of the constness of a variable; * `dynamic_cast` : to goes down an inheritance tree; * `reinterpret_cast` : in very rare cases, when one wants to change the way a memory chunk is interpreted.

1.8 User-defined numerical types

If you are daredevil enough to develop your own numerical type, you may want to provide a constructor and/or a conversion operator, so to ease interaction with functions which generate and/or require doubles.

restart kernel

[13]:
```
#include <iostream>
```

[14]:
```
class Number
 {
  public :
    Number( double f ) { std::cout<<"double_to_number"<<std::endl ; }
    operator double() { std::cout<<"number_to_double"<<std::endl ; return 0 ; }
 } ;
```

[15]:
```
Number number_pi(3.14) ;
```

```
double_to_number
```

[16]:
```
void foo_double( double ) {}
```

[17]:
```
foo_double(number_pi) ;
```

```
number_to_double
```

[18]:
```
void foo_number( Number ) {}
```

[19]:
```
foo_number(3.14) ;
```

```
double_to_number
```

BEWARE: the unary constructor, and the conversion operator, opens the door for **implicit conversions**. The compiler can even chain several ones, such as `short` => `double` => `Number` below, or the contrary.

[20]:
```
short sh = 42 ;
foo_number(sh) ;
```

```
double_to_number
```

[21]:
```
void foo_short( short ) {}
```

[22]:
```
display_short(number_pi) ;
```

```
number_to_double
```

1.9 Good old-fashioned practice: make unary constructors explicit

The implicit conversions problem does not only apply to numerical classes. It is true for any class which has a unary constructor (constructor with only one argument).

The keyword **explicit** forbids the use of those unary constructors for implicit conversions. It should be used almost always.

restart kernel

[1]:
```
#include <iostream>
```

[2]:
```
class Number
 {
  public :
    explicit Number( double f ) { std::cout<<"double_to_number"<<std::endl ; }
    operator double() { std::cout<<"number_to_double"<<std::endl ; return 0 ; }
 } ;
```

[3]:
```
void foo_number( Number ) {}
```

[4]:
```
foo_number(42) ;
```

```
input_line_10:2:2: error: no matching function for call

to 'foo_number'
 foo_number(42) ;
 ^~~~~~~~~~

input_line_9:1:6: note: candidate function not viable:
no known conversion from 'int' to '__cling_N52::Number' for 1st argument
void foo_number( Number ) {}
     ^
```

```
Interpreter Error:
```

Yet the implicit use of the conversion operator is still ok.

[5]:
```
Number number_pi(3.14) ;
```

```
double_to_number
```

[6]:
```
void foo_double( double ) {}
```

[7]:
```
foo_double(number_pi) ;
```

```
number_to_double
```

1.10 New in C++11 : explicit conversion operators

The keyword **explicit** has been generalized to conversion operators within C++11.

restart kernel

[1]:
```
#include <iostream>
```

[2]:
```
class Number
 {
  public :
    explicit Number( double f ) { std::cout<<"double_to_number"<<std::endl ; }
    explicit operator double() { std::cout<<"number_to_double"<<std::endl ;␣
 ↪return 0 ; }
 } ;
```

[3]:
```
void display_double( double ) {}
```

[4]:
```
Number value(3.14) ;
```

```
double_to_number
```

[5]:
```
display_double(value) ;
```

```
input_line_11:2:2: error: no matching function for call

to 'display_double'
 display_double(value) ;
 ^~~~~~~~~~~~~~

input_line_9:1:6: note: candidate function not viable:
no known conversion from '__cling_N52::Number' to 'double' for 1st argument
void display_double( double ) {}
     ^
```

```
Interpreter Error:
```

2 Take Away

- I use *unsigned integers* only for sizes and indexes.
- Implicit conversions are a major source of bugs in ancient C++.
- Modern C++ provides different ways to take the control back:
  - explicit constructors and converters,
  - forbidden overloads (to be seen later),
  - **universal initialization**.

3 Questions ?

4 References

Types * Cpp Reference * Fixed size integer types

Implicit conversions - Cpp Reference - LearnCPP - Nothing But