Contents

1 The `Num<T>` Class

`Num<T>` is a multipurpose object for storing numbers in an easier to use wrapper. **In depth documentation is available through the doxygen folder, or can be generated with the included doxyfile in the source folder.** All classes and functions related to `Num<T>` are inside the namespace `cg`.

In this section, You will lean how to create a `Num<T>` with the MakeNum and NumType helpers that will deduce the proper template parameters, and directly with `Num<T>` type itself.

For the `Num<T>` class, the parameter of `T` will be the type of argument the constructor will take when creating the object. If `T=const int&` then the constructor will take a `const int&` and its data will be of type `int`. If `T=int&` then the constructor will take `int&` and its data will be `int&`. The constructor uses `T&&` parameter for proper forwarding based on the template parameter T. Using the helper `cg::MakeNum` in section 1.1 on the following page is recommended.

The following example code(Example 1) is a set of rval and lval test functions and a test class.

Example 1: Test Functions and Classes

```
/**ConstRef example*/
const uint16_t& testfunc(const uint16_t& t)
{
  return t;
}
/**Ref example*/
uint16_t& testfunc(uint16_t& t)
{
  return t;
}
/**Member ref example*/
struct testclass {
  uint16_t& Get()
  {
    return n;
  }
  const uint16_t& Get() const
  {
    return n;
  }
  uint16_t m_num = 5555;
};
```

1.1 The `cg::MakeNum` Helper

The first thing to know is the helper function `cg::MakeNum`. It will take any argument of a fundamental type or `Num<T>` type and make a reference or copy depending on the the deduction of T. See Example 2.

Example 2: Reference with cg::MakeNum

```
uint16_t n = 4;
auto a = cg::MakeNum(n); //‘a‘ is now REFERENCING ‘n‘
a.Set(89); //‘a‘ REFERENCES ‘n‘ which now equals 89.
bool check1 = a.Get() == 89; //true
check1 = n == 89; //true
```

The `Num<T>` Can also be used to create a regular wrapper that owns its own data and wont be invalidated before destruction.

Example 3: Copy with cg::MakeNum

```
const uint16_t m = 4;
auto b = cg::MakeNum(m); //‘b‘ is now a COPY of ‘m‘
b.Set(89); //‘b‘ does NOT reference ‘m‘.
bool check2 = b.Get() == 89; //true
check2 = m == 89; //false
```

`cg::MakeNumC` Will force the creating of a copy instead of a reference. See Example 4.

Example 4: cg::Forcing Copies

```
uint16_t n = 4;
auto a2 = cg::MakeNum((const uint16_t&) n); //will copy ‘n‘ instead
     of reference.
auto a3 = cg::MakeNumC(n); //Same as above line
```

Even other `Num<T>` can be referenced or copied. See Example 5.

Example 5: MakeNum with cg::MakeNum

```
uint16_t n = 4;
auto a = cg::MakeNum(n);        // a references n
auto b = cg::MakeNum(a);        //b references a
b.Set(89);                      //sets a,b, and n to 89.
bool check = n == b.Get();      //true
bool check2 = n == a.Get();     //true
bool check3 = &n == &b.Get();   //true
bool check4 = &n == &a.Get();   //true
```

A reference `Num<T>` can be copied by another `Num<T>` or it can be referenced by a reference `Num<T>` . See Example 7.

Example 6: Referencing a Copy

```
uint16_t n = 4;
auto a = cg::MakeNumC(n);       // a does not reference n
auto b = cg::MakeNum(a);        //b references a
b.Set(89);                      //sets a,b, to 89. n is unchanged.
bool check = n == b.Get();      //false
bool check2 = n == a.Get();     //false
bool check3 = &n == &b.Get();   //false
bool check4 = &n == &a.Get();   //false
bool check4
  = &a.Get() == &b.Get();     //true
```

Consider the above test functions in Example 1 on page 1. Rvalues and LValues returned by functions are valid parameters to the helper `cg::MakeNumC` and `cg::MakeNum`.

Example 7: Rval and Lval returns

```
uint16_t a = 555;
const uint16_t b = 999;
testclass c;
const testclass d;
auto r = cg::MakeNum(testfunc(a));    // r is a reference of a
auto s = cg::MakeNum(testfunc(b));    //s is a copy of b
auto u = cg::MakeNumC(testfunc(a));   //u is a copy of a
auto v = cg::MakeNum(c.Get());        //v is a reference of c.m_num
auto w = cg::MakeNumC(c.Get());       //w is a copy of c.m_num
auto x = cg::MakeNum(d.Get());        //x is a copy of d.m_num
```

1.2 Using `Num<T>` Directly

To use `Num<T>` directly, one should use `decltype` with either a **function call, or a parenthesized variable** so it will retain the type of references that the variable is.

The parameter `T` should be used with `decltype((var))` when `var` is going to be the primitive parameter of the constructor for the object of `Num<T>` (see Example 8 on the next page).

Example 8: Construct a `Num<T>` with a varaible

```
uint16_t a = 888;
const uint16_t b = 555;
cg::Num<decltype((a))> num(a);  //num references a
              //^^^^^ Notice the parenthesized variable name
cg::Num<decltype((b))> num2(b); //num copies b
              //^^^^^ Notice the parenthesized variable name
num.Set(5432);                  //set num AND a to the value.
num2.Set(44);                   //sets num2 ONLY.
bool check = a == 5432;         //true
bool check2 = &a == &num.Get()  //true
bool check3 = b == 44;          //false
bool check4 = &b == &num2.Get() //false
```

`Num<T>` may also be used as `decltype(testfunc(var))` where the result of the function will be made to be the parameter of the constructor (see Example 9).

Example 9: `Num<T>` and function decyltype

```
uint16_t a = 888;
cg::Num<decltype(testfunc(a))> num(testfunc(a));  // num references
     a
num.Get() = 5432;                                 // num == a ==
    5432

const uint16_t b = 888;
cg::Num<decltype(testfunc(b))> num2(testfunc(b)); // num2 copies b
num2.Get() = 5432;                                // num2 == 5432,
    b == 888
```

1.3 The `cg::NumType<T,bool>` Helper

The developer may want to create an appropriate type easily without actually creating an object. This might be useful for inheritance or making lists or using the `Num<T>` as a template parameter in some other fashion. Thats where `cg::NumType<T,bool>` comes into play. The `cg::NumType<T,bool>` helper is designed to get a type that is appropriate to the developers need quickly. It will create the type automatically based on the desired reference status and type. See Example 10.

Example 10: NumType Helper

```
int a = 999;
const int b = 999;
cg::NumType<const int, true> W(a); //W is a copy of a
cg::NumType<int, true> X(a);       //X is a reference of a
cg::NumType<const int, true> Y(b); //Y is a copy of b
cg::NumType<int, false> Z(b);      //Z is a copy of b
```

`cg::NumType<T,bool>` helper is also usable with `decltype`. See Example 15 on page 7.

Example 11: NumType Helper and decltype

```cpp
int a = 999;
const int b = 999;
cg::NumType<decltype(a), true> W(a);  //W is a reference to a
cg::NumType<decltype(a), false> X(a); //X is a copy of a
cg::NumType<decltype(b), true> Y(b);  //Y is a copy of b
cg::NumType<decltype(b), false> Z(b); //Z is a copy of b
```

An example of the `cg::NumType<T,bool>` helper in action in Example 12

Example 12: NumType Helper Useful Use

```cpp

#include <vector>

int main(int argc, char ** argv)
{
  int a = 999;
  int b = 999;
  int c = 999;
  int d = 999;
  int e = 999;
  /*All are references*/
  cg::NumType<int, true> W1(a);
  cg::NumType<int, true> W2(b);
  cg::NumType<int, true> W3(c);
  cg::NumType<int, true> W4(d);
  cg::NumType<int, true> W5(e);

  std::vector<cg::NumType<int, true>> nums;
  /*All are copied, but remain as references. nums.emplace_back(a)
    would work as well.*/
  nums.push_back(W1);
  nums.push_back(W2);
  nums.push_back(W3);
  nums.push_back(W4);
  nums.push_back(W5);

  auto sz = nums.size();
  for (std::size_t i = 0; i < sz; ++i)
    nums[i].Get() = i+1;

  /*Result: (a to b) == (1 to 5)*/
  return 0;
}
```

1.4 Splitters

There are helper functions `Hi()` and `Lo()` that will do the same thing. The benefit of the helpers is that they are overloaded for multiple other types to make the use of `Num<T>` and others objects such as `BigNum<T,S>`. See Example 13 and Example 14.

Example 13: Hi members

```
uint16_t a = 256 + 5;                // a = 0000 0001 0000 0101
cg::Num<decltype((a))> num(a);      //references a. If a was const, it would be a copy
auto& X = cg::Hi(num).Get();          //Store a reference to the HI part of num in X, also references the HI part of a (for this example).  If a was const, it would not reference the Hi part of a, only num
X = 0;                               // would be compile error IF num was const.  If num
bool check = a == 5;                 //check a == 0000 0000 0000 0101 -- true
```

Example 14: Lo members

```
uint16_t y = 256 + 5;                // a = 0000 0001 0000 0101
cg::Num<decltype((y))> num(y);      //references a. If a was const, it would be a copy
auto& D = cg::Lo(num).Get();          //Store a reference to the LO part of num in X, also references the Lo part of a (for this example).  If a was const, it would not reference the Lo part of a, only num
D = 0;                               // would be compile error IF num was const.  If num
bool check = a == 256;                //check a == 0000 0001 0000 0000 -- true
```

1.5 Special Members of `Num<T>`

`Num<T>` has various special members to help it act appropriate in certain environments. They wont be discussed in too much detail. Refer to the Doxygen documentation for in depth information about them. The special members are available to allow the number to be treated as a list of size 1 in place of any ol' list.

Example 15: Special Members

```cpp
const static bool IAmConst
  = std::is_const<std::remove_reference_t<_Internal_T>>::value;
using StoreType = std::conditional_t<
  std::is_const<std::remove_reference_t<_Internal_T>>::value,
  std::remove_const_t<std::remove_reference_t<_Internal_T>>,
  _Internal_T
>;
using BasicStoreType
  = std::remove_const_t<std::remove_reference_t<StoreType>>;
using RefSelf = Num<std::remove_reference_t<StoreType>&>;
using NonRefSelf = Num<const std::remove_reference_t<StoreType>&>;
using Self = Num<_Internal_T>;
using DemotedBaseType = typename cg::DemoteType<BasicStoreType>::
    Type;
BasicStoreType& Get();
const BasicStoreType& Get() const;
RefSelf GetReference();
void Set(const BasicStoreType& n);
void Set(BasicStoreType&& n);
auto Size() const;
bool IsZero() const;
NonRefSelf HardCopy() const;
template<typename U>
void Swap(Num<U>& other);
```