swap

1- 最通用的模板交换函数模式：创建临时对象，调用对象的赋值操作符

在标准头文件 `<algorithm>` 中，C++ 11中定义如下。

```
template <class T> void swap ( T& a, T& b )
{
  T c(a); a=b; b=c; /// 构建临时对象，一次拷贝构造，两次赋值操作
}
```

上面开销很大，在很多STL容器中，进行了优化。 只需要内部的指针交换，防止大量的赋值操作，这样交换在常数时间内。

在相同的命名空间 `std` 中自定义重载swap函数，提高效率。

以下举例：

交换两个vector容器。

```
// swap algorithm example (C++98)
#include <iostream>     // std::cout
#include <algorithm>    // std::swap
#include <vector>       // std::vector

int main () {

  int x=10, y=20;                                 // x:10 y:20
  std::swap(x,y);                                 // x:20 y:10

  std::vector<int> foo (4,x), bar (6,y);          // foo:4x20 bar:6x10
  std::swap(foo,bar);                             // foo:6x10 bar:4x20

  std::cout << "foo contains:";
  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```
// foo contains: 10 10 10 10 10 10
```

2- 针对int型优化— 异或

无需构造临时对象，异或

```
#include <iostream>
#include <ctime>
using namespace std;
void swap(int & __restrict a, int & __restrict b)   // restrict 关键字?
{
    a ^= b;
    b ^= a;
    a ^= b;
}

int main()
{
    int a =10, b= 20;
    swap(a,b);
    cout<<a<< " " << b <<endl;      // 20 10
    return 0;
}
```

3- 交换指针

因为指针是int，所以基于这个思路可以优化1：

```
template <typename T> void Swap(T & obj1,T & obj2)
{
    /// 对每个对象的指针进行操作
    unsigned char * pObj1 = reinterpret_cast<unsigned char *>(&obj1);
    unsigned char * pObj2 = reinterpret_cast<unsigned char *>(&obj2);
    for (unsigned long x = 0; x < sizeof(T); ++x)
    {
        pObj1[x] ^= pObj2[x];
        pObj2[x] ^= pObj1[x];
        pObj1[x] ^= pObj2[x];
    }
}
```

4- 针对内建类型进行优化

内建类型的优化： int, flaot, double 等，甚至重载运算符的用户自定义类型：向量，矩阵，图像等。。。

```
type  a; -- e.g 10
type  b; -- e.g 5

a = a+b ; -- a=15,b=5
b = a-b ; -- a=15,b=10
a= a -b ; -- a= 5,b=10
```

// 无需构造临时变量。使用基本运算操作符。

```
Ok, let's see.
a = a + b;
b = a - b;
a = a - b;
Let's introduce new names, 引入临时变量进行证明
c = a + b;
d = c - b;
e = c - d;
And we want to prove that d == a and e == b.
d = (a + b) - b = a, proved.
e = (a + b) - ((a + b) - b) = (a + b) - a = b, proved.
For all real numbers
```

5- swap的一些特化

std::string, std::vector各自实现了swap函数，

5.1- string

第二个swap(Right)进行判断，如果使用了相同的分配器，则直接交换控制信息，否则调用string::operator=进行拷贝赋值。。。所以建议优先使用swap函数，而不是赋值操作符。

```
emplate<class _Elem,
    class _Traits,
    class _Alloc> inline
    void __CLRCALL_OR_CDECL swap(basic_string<_Elem, _Traits, _Alloc>&
_Left,
        basic_string<_Elem, _Traits, _Alloc>& _Right)
    {   // swap _Left and _Right strings
    _Left.swap(_Right);
```

```
    }
    void __CLR_OR_THIS_CALL swap(_Myt& _Right)
        {   // exchange contents with _Right
        if (this == &_Right)
            ;   // same object, do nothing
        else if (_Mybase::_Alval == _Right._Alval)
            {   // same allocator, swap control information
 #if _HAS_ITERATOR_DEBUGGING
            this->_Swap_all(_Right);
 #endif /* _HAS_ITERATOR_DEBUGGING */
            _Bxty _Tbx = _Bx;
            _Bx = _Right._Bx, _Right._Bx = _Tbx;
            size_type _Tlen = _Mysize;
            _Mysize = _Right._Mysize, _Right._Mysize = _Tlen;
            size_type _Tres = _Myres;
            _Myres = _Right._Myres, _Right._Myres = _Tres;
            }
        else
            {   // different allocator, do multiple assigns
            _Myt _Tmp = *this;
            *this = _Right;
            _Right = _Tmp;
            }
        }
```

5.2 vector 中

vector的swap原理跟string完全一致，只有当当使用了不同分配器才进行字节拷贝。其余情况直接交换控制信息。

```
template<class _Ty,
    class _Alloc> inline
    void swap(vector<_Ty, _Alloc>& _Left, vector<_Ty, _Alloc>& _Right)
    {   // swap _Left and _Right vectors
    _Left.swap(_Right);
    }
    void swap(_Myt& _Right)
        {   // exchange contents with _Right
        if (this == &_Right)
            ;   // same object, do nothing
        else if (this->_Alval == _Right._Alval)
            {   // same allocator, swap control information
 #if _HAS_ITERATOR_DEBUGGING
            this->_Swap_all(_Right);
 #endif /* _HAS_ITERATOR_DEBUGGING */
            this->_Swap_aux(_Right);
```

```
            _STD swap(_Myfirst, _Right._Myfirst);
            _STD swap(_Mylast, _Right._Mylast);
            _STD swap(_Myend, _Right._Myend);
            }
        else
            {   // different allocator, do multiple assigns
            this->_Swap_aux(_Right);
            _Myt _Ts = *this;
            *this = _Right;
            _Right = _Ts;
            }
        }
```

6- smart_ptr

C++异常有三个级别：基本，强，没有异常。通过创建临时对象然后交换，能够实现重载赋值操作符的强异常安全的执行。

Loki中智能指针 临时变量跟this交换，临时变量自动销毁~

```
SmartPtr& operator=(SmartPtr<T1, OP1, CP1, KP1, SP1, CNP1 >& rhs)
{
    SmartPtr temp(rhs);
    temp.Swap(*this);
    return *this;
}
```

7- shared_ptr

boost::shared_ptr，share_ptr定义了自己的swap函数。

```
shared_ptr & operator=( shared_ptr const & r ) // never throws
{
    this_type(r).swap(*this);
    return *this;
}

void swap(shared_ptr<T> & other) // never throws
{
    std::swap(px, other.px);
    pn.swap(other.pn);
}
```