# C++标准库（Part2）

## 算法

- 算法与容器通过 `iterator` 沟通
- 算法是 `function template`，具有 `实参推导` 特性，所以不需要指明 `iterator` 的类型

```
template<typename Iterator>
Algorithm(Iterator it1,Iterator it2) {
    ...
}
```

## iterator_category

- 5种iterator：
  - `input_iterator_tag`
  - `forward_iterator_tag`：继承 `input_iterator_tag`
  - `bidirectional_iterator_tag`：继承 `forward_iterator_tag`
  - `random_access_iterator_tag`：继承 `bidirectional_iterator_tag`
  - `output_iterator_tag`
- 各种容器的iterator_category
  - `array`、`vector`、`deque`：`random_access_iterator_tag`
  - `list` 和底层是 `红黑树` 的容器：`bidirectional_iterator_tag`
  - `forward_list` 和底层是 `HashTable` 的容器：`forward_iterator_tag`
- `iterator_category` 对算法的影响

```
template<typename _InputIterator>
inline _GLIBCXX17_CONSTEXPR
typename iterator_traits<_InputIterator>::difference_type
    distance(_InputIterator __first, _InputIterator __last)
{
    // concept requirements -- taken care of in __distance
    return std::__distance(__first, __last,
                           std::__iterator_category(__first));
    //根据iterator_category调用不同的__distance
}

template<typename _Iter>
inline _GLIBCXX_CONSTEXPR
typename iterator_traits<_Iter>::iterator_category //返回iterator_category类型的对象
    __iterator_category(const _Iter&)
{ return typename iterator_traits<_Iter>::iterator_category(); } //最后的括号创建临时对象
```

```
template<typename _InputIterator>
inline _GLIBCXX14_CONSTEXPR
typename iterator_traits<_InputIterator>::difference_type
    __distance(_InputIterator __first, _InputIterator __last,
               input_iterator_tag) //当传入input/forward/bidirecitonal的iterator_tag对象时调用
{
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)

        typename iterator_traits<_InputIterator>::difference_type __n = 0;
    while (__first != __last)
    {
        ++__first;
        ++__n;
    }
    return __n;
}

template<typename _RandomAccessIterator>
inline _GLIBCXX14_CONSTEXPR
typename iterator_traits<_RandomAccessIterator>::difference_type
    __distance(_RandomAccessIterator __first, _RandomAccessIterator __last,
               random_access_iterator_tag) //当传入random_access的iterator_tag对象时调用
```

```
{
    // concept requirements
    __glibcxx_function_requires(_RandomAccessIteratorConcept<
                                _RandomAccessIterator>)
        return __last - __first;
}
```

# 算法实例学习

- Standard library header - cppreference.com
- `sort(rbegin,rend)`：逆序排序（默认从小到大，逆序后从大到小）
- `reverse_iterator`
  - rbegin指向end的前一个元素，`rbegin = reverse_iterator(end)`
  - rend指向begin的前一个元素，`rend=reverse_iterator(begin)`
- 算法的参数一般可以传入自定义操作，可以传入`仿函数`或`函数名`
  - 仿函数：`minus<int>()` minus为结构体的名字，int为模板参数，()创建临时对象
  - 仿函数实则为内部自定义操作符的结构体的对象

```
struct myclass{
    bool operator()(int x, int y){
        return x<y;
    }
} myobj;

bool myfunc(int x, int y){ return x<y; }

sort(vec.begin(),vec.end(),myfunc); //传入函数名
sort(vec.begin(),vec.end(),myobj); //传入仿函数
```

# Functor 仿函数

- functor需要`重载()`
- 参考资料：Standard library header - cppreference.com
- 不完全分类：
  - 算术类：minus, ...
  - 逻辑运算类：logical_and, ...
  - 相对关系类：less, ...
- stl的`functor`会继承`unary_function`或者`binary_function`
  - `binary_function`或`unary_function`告知`argument_type`，例如通过语句`typename Fn::first_argument_type`获取仿函数中参数类型【功效类似iterator中的5个typedef】

```
/**
*  This is one of the @link functors functor base classes@endlink.
*/
template<typename _Arg, typename _Result>
struct unary_function
{
/// @c argument_type is the type of the argument
typedef _Arg    argument_type;

/// @c result_type is the return type
typedef _Result     result_type;
};

/**
*  This is one of the @link functors functor base classes@endlink.
*/
template<typename _Arg1, typename _Arg2, typename _Result>
struct binary_function
{
/// @c first_argument_type is the type of the first argument
typedef _Arg1   first_argument_type;

/// @c second_argument_type is the type of the second argument
typedef _Arg2   second_argument_type;
```

```
/// @c result_type is the return type
typedef _Result     result_type;
};
```

- 以 `minus` 为例

```
/// One of the @link arithmetic_functors math functors@endlink.
template<typename _Tp>
struct minus : public binary_function<_Tp, _Tp, _Tp>
{
_GLIBCXX14_CONSTEXPR
_Tp
operator()(const _Tp& __x, const _Tp& __y) const
{ return __x - __y; }
};
```

# Adapter 适配器

- adapter：通过 `内含` 或 `继承` 实现
  - 包括 `container/iterator/functor adapter`，这三类都是通过 `内含` 实现的
- `container adapter`：例如：stack/queue
- `functor adapter`：例如：binder2nd（现已被bind替代）Standard library header - cppreference.com
- 下面代码简单体现了 `binder2nd` 作为一个 `functor adapter` 的用法

```
template< class Fn >
class binder2nd :
    public std::unary_function<typename Fn::first_argument_type,
                               typename Fn::result_type> //本身是functor
{
protected:
    Fn op;  //内含functor
    typename Fn::second_argument_type value;
public:
    binder2nd(const Fn& fn,
              const typename Fn::second_argument_type& value);

    typename Fn::result_type
        operator()(const typename Fn::first_argument_type& x) const;

    typename Fn::result_type
        operator()(typename Fn::first_argument_type& x) const;
};
```

```
template< class F, class T >
std::binder2nd<F> bind2nd( const F& f, const T& x ); //bind2nd是binder2nd的辅助函数
```

```
// 使用示例,辅助函数bind2nd是模板函数，可以进行实参推导，传入functor对象，返回functor adapter
count_if(vec.begin(),vec.end(), bind2nd(less<int>(),40));
```

## bind

- bind的用法：
  - 绑定function
  - 绑定member function/data member， `_1` 必须是对象地址
  - 绑定functor
- 绑定function

```
auto fn_bind = bind(func,para1,para2); //绑定函数，传入函数名func，func的实参para1和para2
fn_bind(); //使用绑定后的函数，2个参数已绑定为para1和para2

auto fn_bind = bind<int>(func,para1,para2); //将func的结果转型为int
fn_bind(); //结果为int
```

引入 `placeholders`

```
using namespace std::placeholders; //引入后可以使用_1,_2代替参数
auto fn_bind = bind(func, _1, para2);
fn_bind(10); //10代替_1

auto fn_invert = bind(func, _2, _1); //_2代替func的第一个参数，但是用传入的第二个参数(下面的5)
fn_invert(10,5);//10代替_1, 5代替_2
```

- 绑定member function/data member

```
auto mem_fn = bind(&MyClass::func,_1); //func函数没有参数，但作为成员函数，需要this指针
memfn(MyClassObject); //传入this指针

auto mem_data = bind(&MyClass::prop,_1);  //绑定成员数据同理
mem_data(MyClassObject); //传入this指针
```

- 绑定functor

```
bind2nd(less<int>(),50); //旧版本

bind(less<int>(),_1,50); //新版本
```

- `bind`和`placeholders`部分代码

```
// FUNCTION TEMPLATE bind (implicit return type)
template <class _Fx, class... _Types>
_NODISCARD _CONSTEXPR20 _Binder<_Unforced, _Fx, _Types...> bind(_Fx&& _Func, _Types&&... _Args) {
    return _Binder<_Unforced, _Fx, _Types...>(_STD forward<_Fx>(_Func), _STD forward<_Types>(_Args)...);
}

// FUNCTION TEMPLATE bind (explicit return type)
template <class _Ret, class _Fx, class... _Types>
_NODISCARD _CONSTEXPR20 _Binder<_Ret, _Fx, _Types...> bind(_Fx&& _Func, _Types&&... _Args) {
    return _Binder<_Ret, _Fx, _Types...>(_STD forward<_Fx>(_Func), _STD forward<_Types>(_Args)...);
}

// PLACEHOLDER ARGUMENTS
namespace placeholders {
    _INLINE_VAR constexpr _Ph<1> _1{};
    _INLINE_VAR constexpr _Ph<2> _2{};
    _INLINE_VAR constexpr _Ph<3> _3{};
    _INLINE_VAR constexpr _Ph<4> _4{};
    _INLINE_VAR constexpr _Ph<5> _5{};
    _INLINE_VAR constexpr _Ph<6> _6{};
    _INLINE_VAR constexpr _Ph<7> _7{};
    _INLINE_VAR constexpr _Ph<8> _8{};
    _INLINE_VAR constexpr _Ph<9> _9{};
    _INLINE_VAR constexpr _Ph<10> _10{};
    _INLINE_VAR constexpr _Ph<11> _11{};
    _INLINE_VAR constexpr _Ph<12> _12{};
    _INLINE_VAR constexpr _Ph<13> _13{};
    _INLINE_VAR constexpr _Ph<14> _14{};
    _INLINE_VAR constexpr _Ph<15> _15{};
    _INLINE_VAR constexpr _Ph<16> _16{};
    _INLINE_VAR constexpr _Ph<17> _17{};
    _INLINE_VAR constexpr _Ph<18> _18{};
    _INLINE_VAR constexpr _Ph<19> _19{};
    _INLINE_VAR constexpr _Ph<20> _20{};
} // namespace placeholders
```

# iterator adapter

- 内含其他的iterator，例如reverse_iterator，inserter
- 内含其他类，例如ostream_iterator，istream_iterator
- iterator_adapter保证返回的是iterator，在算法中可以使用该种iterator完成不同操作，例如在copy算法中，调用inserter作为第三参数，将进行插入操作【原本：将it1到it2的内容拷贝到it3开始的地方；现在：将it1到it2的内容用插入的方式拷贝到it3开始的地方】

## reverse iterator

- 逆向迭代器，需要对相应的正向迭代器退一位取值

```
reverse_iterator rbegin() { return reverse_iterator(end())};

class reverse_iterator{
    reference operator*() const {
        Iterator tmp = current;
        return *--tmp;
    }
}
```

## inserter

- `inserter(container,iterator)`返回`insert_iterator`

```
vector<int> result{1,2,3,6,7};
vector<int> vec{4,5};
vector<int>::iterator it = result.begin()+3;
// 如果是非random_access的迭代器需要用advance(result.begin(),3)
copy(vec.begin(),vec.end(),inserter(result,it)); //result变为{1,2,3,4,5,6,7}
```

## ostream_iterator

```
std::ostream_iterator<int> out_it(std::cout,","); //第二个参数为分割符号
std::copy(vec.begin(),vec.end(),out_it); //执行输出至cout
```

## istream_iterator

```
std::istream_iterator<double> eos;//end of stream
std::istream_iterator<double> iit(std::cin); //执行此处，cin中的值已经进入iit中
if(iit≠eos) value1 = *iit; //获取第一个输入

++iit;
if(iit≠eos) value2 =*iit; //获取第二个输入

copy(iit,eos,inserter(vec,vec.begin())); //将输入插入到vec开始的地方
```

# 一个万用的HashFunction

- 注意以下4个函数的顺序，小心因为找不到函数定义编译出错

```
//计算种子数值
template<typename T>
inline void hash_combine(size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}

//递归调用出口
template<typename T>
inline void hash_val(size_t& seed, const T& val)
{
    hash_combine(seed, val);
}

template<typename T, typename... Types>
inline void hash_val(size_t& seed, const T& val, const Types&... args)
{
    //重新计算种子值
    hash_combine(seed, val);
    //递归调用
    hash_val(seed, args...);
}

template<typename... Types>
inline size_t hash_val(const Types&... args)
{
    size_t seed = 0;
    hash_val(seed, args...);
    return seed;
}
```

- 自定义的类需要实现自定义的hash，以及自定义的equal函数

```
// 自定义的类
class Custom {
 public:
    string str_FirstName;
    string str_LastName;
    long l_ID;
};

// 自定义hash
struct CustomHash
{
```

```
    std::size_t operator()(const Custom& custom) const
    {
        //传入所有的数据成员
        return hash_val(custom.str_FirstName, custom.str_LastName, custom.l_ID);
    }
};

// 自定义equal函数，必须存在，否则编译报错
struct CustomEqualTo
{
    bool operator()(const Custom& c1, const Custom& c2) const
    {
        return c1.str_FirstName == c2.str_FirstName &&
            c1.str_LastName == c2.str_LastName &&
            c1.l_ID == c2.l_ID;
    }
};
```

- 测试

```
//测试
void test()
{
    std::unordered_set<Custom, CustomHash, CustomEqualTo> hash_set;

    hash_set.insert(Custom{ "san", "Zhang", 1l });
    hash_set.insert(Custom{ "si", "Li", 2l });
    hash_set.insert(Custom{ "er", "Wang", 3l });
    hash_set.insert(Custom{ "wu", "Zhao", 4l });
    hash_set.insert(Custom{ "liu", "Guan", 5l });
    hash_set.insert(Custom{ "qi", "Wu", 6l });
    hash_set.insert(Custom{ "ba", "Wei", 7l });

    std::cout << "bucket size: " << hash_set.bucket_count() << std::endl;
    for (int i = 0; i < hash_set.bucket_count(); i++)
        std::cout << "bucket #" << i << " has " << hash_set.bucket_size(i) << " items." << std::endl;
}
```

- hash可以只写一个函数，但是用起来比较麻烦

```
size_t  customer_hash_func(const Customer  & c){
    return hash_val(c.age);
}

//函数的调用
unordered_set<Customer, size_t(*) (const Customer &)>  custset(20, customer_hash_func);
```

- 也可以用stuct hash偏特化形式实现

```
//1、自己定义hash 函数的方式
namespace std   //必须放在 std 内
{
    template<>
    struct hash<MyString>   //這是為了 unordered containers
    {
        size_t  operator()(const MyString& s) const  // noexcept
        { return hash<string>()(string(s.get())); }
        //借用現有的 hash<string> (in  ...\include\c++\bits\basic_string.h)
    };
}

unordered_set<MyString, hash<MyString>>  set;  //函数的调用
```

# tuple

使用示例（https://cplusplus.com/reference/tuple/tuple/）

```
// tuple example
#include <iostream>     // std::cout
#include <tuple>        // std::tuple, std::get, std::tie, std::ignore

int main ()
{
  std::tuple<int,char> foo (10,'x'); //构造
```

```
  auto bar = std::make_tuple ("test", 3.1, 14, 'y'); //make_tuple 构造

  std::get<2>(bar) = 100;                                     // access element

  int myint; char mychar;

  std::tie (myint, mychar) = foo;                             // unpack elements
  std::tie (std::ignore, std::ignore, myint, mychar) = bar;   // unpack (with ignore)

  mychar = std::get<3>(bar);  //get

  std::get<0>(foo) = std::get<2>(bar);  //通过get赋值
  std::get<1>(foo) = mychar;

  std::cout << "foo contains: ";
  std::cout << std::get<0>(foo) << ' ';
  std::cout << std::get<1>(foo) << '\n';

  return 0;
}
```

tuple通过`variadic template`实现，将存储的数据分为`head(第一个参数)`和`tail(剩余的参数构成的tuple)`

# type_traits

- 文档：https://cplusplus.com/reference/type_traits/
- 使用示例

```
// is_integral example
#include <iostream>
#include <type_traits>

int main() {
  std::cout << std::boolalpha;
  std::cout << "is_integral:" << std::endl;
  std::cout << "char: " << std::is_integral<char>::value << std::endl;
  std::cout << "int: " << std::is_integral<int>::value << std::endl;
  std::cout << "float: " << std::is_integral<float>::value << std::endl;
  return 0;
}
```

- 实现示例

```
// 以is_void为例，is_void<xxx>::value需要回答xxx是否为void类型
// is_void需要去除type的const和volatile修饰符

// __remove_cv_t (std::remove_cv_t for C++11).
  template<typename _Tp>
    using __remove_cv_t = typename remove_cv<_Tp>::type;  //remove_cv实现见下一段代码

template<typename>
  struct __is_void_helper    //泛化版本，默认返回false_type
  : public false_type { };

template<>
  struct __is_void_helper<void>  //特化，当传入的类型是void时返回true_type
  : public true_type { };

/// is_void
template<typename _Tp>
  struct is_void
  : public __is_void_helper<__remove_cv_t<_Tp>>::type
  { };
```

```
/// remove_cv
template<typename _Tp>
  struct remove_cv
  { using type = _Tp; };

template<typename _Tp>
  struct remove_cv<const _Tp>  //特化，若是const类型，去除const
  { using type = _Tp; };

template<typename _Tp>
```

```
    struct remove_cv<volatile _Tp>
    { using type = _Tp; };

  template<typename _Tp>
    struct remove_cv<const volatile _Tp> //特化，若是volatile类型，去除volatile
    { using type = _Tp; };
```

## cout

- cout属于一种ostream(https://cplusplus.com/reference/iostream/cout/)

```
extern istream cin;     /// Linked to standard input
extern ostream cout;    /// Linked to standard output
```

- ostream是basic_ostream(https://cplusplus.com/reference/ostream/ostream/)

```
typedef basic_ostream<char> ostream;

template<typename _CharT, typename _Traits>
class basic_ostream : virtual public basic_ios<_CharT, _Traits>
```

- 重载 `operator<<`

```
class Point3d
{
private:
    double x;
    double y;
    double z;
public:
    friend ostream&
    operator <<(ostream& os, const Point3d &pt);
};

ostream&
operator <<(ostream& os, const Point3d &pt)
{
    os<<"friend method, "<<"x: "<<pt.x<<", y: "<<pt.y<<", z: "<<pt.z;
    return os;
}
```

## move

- 可移动的类型，需要实现 `移动构造` 和 `移动赋值`

```
  // copy ctor
MyString(const MyString& str) : _len(str._len) {
  ++CCtor;
  _init_data(str._data);  // COPY
}

// move ctor, with "noexcept"
MyString(MyString&& str) noexcept : _data(str._data), _len(str._len) {
  ++MCtor;
  str._len = 0;
  str._data = NULL;  // 避免 delete (in dtor)
}

// copy assignment
MyString& operator=(const MyString& str) {
  ++CAsgn;
  if (this != &str) {
    if (_data) delete _data;
    _len = str._len;
    _init_data(str._data);  // COPY!
  } else {
    // Self Assignment, Nothing to do.
  }
  return *this;
}
```

```
// move assignment
MyString& operator=(MyString&& str) noexcept {
  ++MAsgn;
  if (this != &str) {
    if (_data) delete _data;
    _len = str._len;
    _data = str._data;  // MOVE!
    str._len = 0;
    str._data = NULL;  // 避免 deleted in dtor
  }
  return *this;
}
```

- 实现了移动构造的类，插入到容器中会自动调用`移动构造函数`和`构造函数`
- 没有实现移动构造的类，插入到容器中自动调用`拷贝构造函数`和`构造函数`，速度比移动构造函数慢
- 无论有没有实现移动构造，都可以通过`std::move(object)`来调用移动构造

```
NoMoveClass c21(c2); //拷贝构造
NoMoveClass c22(std::move(c2)); //移动构造
```