Додаткова функціональність в алгоритмах

1. Розширення STL функціями користувача
2. Предикати
3. Об'єкти-функції
4. Шаблони базових типів стандартних об'єктів-функцій
5. Стандартні об'єкти-функції
6. Адаптери об'єктів-функцій

Розширення STL функціями користувача

- Передача функції через аргумент – стандартний механізм розширення функціональності бібліотеки
- Надійність механізму
  - неформальний опис стандартних вимог до прототипу функції
  - строга перевірка при компілюванні правильності оголошення функції
- Гнучкість механізму
- Може буди зовнішньою функцією або перевантаженим оператором виклику функції

```
template<class In, class Funct> inline
Funct for_each(In first, In last, Funct f){
// виконання функції для кожного елемента
   for (; first != last; ++first)
      f(*first);
   return f;
}
```

Предикати

- Окрема категорія функцій, які передають у алгоритми для перевірки аргумента
  - повертає `bool`
  - результат залежить лише від аргументів, не залежить від послідовності виклику
  - може буди зовнішньою функцією або перевантаженим оператором виклику функції (константним)
- Унарні та бінарні предикати

```
template<class _InIt, class _Pr> inline
_InIt _Find_if(_InIt _First, _InIt _Last, _Pr _Pred){

   for (; _First != _Last; ++_First)
      if (_Pred(*_First))   break;
   return (_First);
}
```

Об'єкти-функції

- Об'єкти-функції – абстрактна категорія C++: все, що веде себе як функція, є функцією
- Крім виклику звичайних функцій – об'єкт типу, в якому перевантажено `operator()(...)`, у відповідному виразі зі списком аргументів для цього оператора

Переваги об'єктів-функцій

- Можливості типу не обмежуються наявністю `operator()(...)`, це може бути повноцінний тип C++ з додатковими даними та функціональністю
- Придатність до використання у ролі назви типу в аргументах функцій та шаблонів
- При передачі об'єктів-функцій вирази можуть бути оптимізованими, більшість операторів перевантажують як `inline`

Accumulate (сума елементів послідовності)

```
template<class In, class T>
T accumulate (In first, In last, T init)
{
    while (first!=last) {
     init = init + *first;
     ++first;
    }
    return init;
}
```

v:

1	2	3	4

```
int sum = accumulate(v.begin(),v.end(),0);     // sum = 10
```

Accumulate (сума елементів послідовності)

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0);
    // тип 3-о аргументу, ініціалізатора, визначає точність

    int si = accumulate(p, p+n, 0);    // небезпека переповнення

    long sl = accumulate(p, p+n, long(0));   // результат в long
    double s2 = accumulate(p, p+n, 0.0);     // результат в double

    // визначаєм змінну для результату і ініціалізації:

    double ss = 0;    //обов”язкова ініціалізація
    ss = accumulate(vd.begin(), vd.end(), ss);
}
```

Accumulate

(узагальнення: обробка елементів послідовності)

- Цим алгоритмом можна не тільки сумувати елементи послідовності, а здійснювати над ними довільну бінарну операцію (н.п., *)
- будь-яка функція, яка “оновлює величину **init** ” може бути використана:

```
template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
     init = op(init, *first);          // означає init op *first
     ++first;
    }
    return init;
}
```

Accumulate

```
// перемноження елементів послідовності:

#include <numeric>
void f(list<double>& ld)
{
    double product = accumulate(ld.begin(), ld.end(), 1.0,
    multiplies<double>());
    // …
}
```

множення *

ініціалізація 1.0

multiplies – стандартний бібліотечний функціональний об”єкт для множення

```
template<class Type>
struct multiplies : public binary_function <Type, Type, Type>
{
    Type operator()( const Type& _Left, const Type& _Right ) const;
};
```

Accumulate (дані є частиною об’єкта)

```
struct Tovar {
    int units;
    double unit_price;
    // …
};
```

Для ініціалізації та зберігання результату

```
double price (double v, const Tovar& r)
{
    return v + r.unit_price * r.units;
}
```

```
void f(const vector<Tovar>& vr)
{
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // …
}
```

Accumulate (добуток елементів на певне значення)

```
class NewPrice
{
	double zleva;
public:
	NewPrice(double v):zleva(v){};
	double operator()(double v,const Tovar& r){
		return v + r.unit_price * r.units*zleva;}
};
```

```
void f(const list<Tovar>& vr)
{
	double total = accumulate(vr.begin(), vr.end(), 0.0, NewPrice(0.1));
	// …
}
```

Використання предикатів

```
//предикат – функція
// переміщення в b елементів не більших за 15

bool gt15(int x) { return 15 < x; };

int main() {
    int a[]={10,20,30};
    vector<int> b;
    remove_copy_if(a,a+3,back_inserter(b), gt15);
           //b has 10 only
    ...
}
```

Використання предикатів

```
// предикат – функціональний об"єкт
// переміщення в b елементів не більших за val

class gt_n {
    int value;
public:
    gt_n(int val) : value(val) {}
    bool operator()(int n) { return n > value; }
};

int main() {
    int a[]={1, 2, 3, 4, 5, 6, 7, 8,9 };
    vector<int> b;
    gt_n f(4);
    remove_copy_if(a, a+3, back_inserter(b), f);
    remove_copy_if(a, a+3, back_inserter(b), gt_n(6));
```

стандартні об'єкти-функції

<functional>

Класифікація функціональних об”єктів бібліотеки STL здійснюється:

- за кількістю аргументів їх оператора **operator( )**
- за типом значення, що повертається

- Генератор (Generator) - тип функціонального об”єкта, що не має аргументів і повертає величину довільного типу

Приклад: функція **rand( )** (визначений в **<cstdlib>),** застосовний до алгоритму **generate_n( )**

- Унарна функція – має один аргумент деякого типу і повертає величину різного типу (може бути **void**).

- Бінарна функція – має два аргументи деяких типів і повертає величину різного типу (може бути **void**).

- Унарний предикат – унарна функція, що повертає **bool**.

- Бінарний предикат – бінарна функція, що повертає **bool**.

Шаблони базових типів стандартних об'єктів-функцій

```
template<class _Arg,  class _Result>
struct unary_function
{// base class for unary functions
   typedef _Arg argument_type;
   typedef _Result result_type;
};
template<class _Arg1,class _Arg2,class _Result>
struct binary_function
{// base class for binary functions
   typedef _Arg1 first_argument_type;
   typedef _Arg2 second_argument_type;
   typedef _Result result_type;
};
```

Стандартні об'єкти-функції

- Всі визначені в бібліотеці функціональні об’єкти є шаблонними типами і представляють всі вбудовані знаки арифметичних операцій, операції порівняння і логічні операції.
- Параметрами шаблонів є типи елементів, над якими виконуються операції, які при використанні в алгоритмах повинні співпадати з типами елементів у контейнерах.
- Параметри шаблону - довільні вбудовані типи C++ або типи, створені користувачем, для яких перевантажені відповідні оператори.

Стандартні об'єкти-функції

plus	Left + Right
minus	Left - Right
multiplies	Left * Right
divides	Left / Right
modulus	Left % Right
negate	-Left
equal_to	Left == Right
not_equal_to	Left != Right
greater	Left > Right
less	Left < Right
greater_equal	Left >= Right
less_equal	Left <= Right
logical_and	Left && Right
logical_or	Left || Right
logical_not	!Left

negate

```
template<class _Ty>
struct negate
   : public unary_function<_Ty, _Ty>
{// functor for unary operator-
   _Ty operator()(const _Ty& _Left)const
   {// apply operator- to operand
      return (-_Left);
   }
};
```

plus

```
template<class _Ty>
struct plus
:public binary_function<_Ty,_Ty,_Ty>
{// functor for operator+
  _Ty operator()(const _Ty& _Left,
       const _Ty& _Right) const
  {// apply operator+ to operands
    return (_Left + _Right);
  }
};
```

logical_or

```
template<class _Ty>
struct logical_or
 :public binary_function<_Ty,_Ty, bool>
{// functor for operator||
   bool operator()(const _Ty& _Left,
                   const _Ty& _Right) const
  {// apply operator|| to operands
    return (_Left || _Right);
  }
};
```

Використання стандартних об”єктів-функцій

// копіює елементи послідовності [first, last) у result, попередньо застосувавши до кожного елемента функціональний об’єкт op:

```
template <class In, class Out, class UnaryOper >
    Out transform(In first, In last, Out result, UnaryOper  op)
```

// застосовує функціональний об’єкт op до кожної пари з вказаних діапазонів.

```
template <class In1, class In2, class Out, class BinOper>
    Out transform(In1 first1,In1 last1,In2 first2,Out result,
                  BinOper  op)
```

```
vector<int> v,v1,v2;
// всі елементи з v запишуться в out з протилежним знаком
transform(v.begin(), v.end(), out,  negate<int>());
// додавання двох векторів поелементно
transform(v1.begin(), v1.end(), v2.begin(), out,  plus<int>());
```

```
Vector <Tovar> sklad1, sklad2;
// заповнення товаром sklad1, sklad2;

transform(sklad1.begin(), sklad1.end(), sklad2.begin(),
    sklad1,  minus<Tovar>());
// для класу Tovar повинен бути перевантажений operator-
//  Tovar& operator-(Tovar& t1, Tovar & t2)
```

Адаптери об'єктів-функцій

- Як використати бінарний функціональний об”єкт `divides` для ділення вектора на число? Як використати предикат для оберненої операції?
- Адаптери конвертують функціональний об”єкт до вимог алгоритмів, змінюючи кількість аргументів чи тип, що повертається
- Функціональний об”єкт може бути адаптований завдяки заданню типу аргументів і значення, що повертається через **typedef (argument_type** і **result_type)**.
- Визначені в заголовковому файлі **<functional>**

Адаптери стандартних об'єктів-функції

`bind1st(op,val)`	`op(val,param)`
`bind2nd(op,val)`	`op(param,val)`
`not1(op)`	`!op(param)`
`not2(op)`	`!op(param1,param2)`
`ptr_fun(op)`	`*op(param)` `*op(param1,param2)`
`mem_fun(op)`	`param ->* op()` `param1 ->* op(param2)`
`mem_fun_ref(op)`	`param .>* op()` `param1 .>* op(param2)`

Адаптери **binder1st** і **binder2nd**

- Класи-адптери **binder1st** і **binder2nd**, конвертують адаптовані бінарні функції в адаптовані унарні функції, фіксуючи та зберігаючи перший (другий) аргумент як поле адаптера.
- Функції-адаптери **binder1st()** і **bind2nd()** повертають об”єкти однойменних класів-адаптерів.

class binder2nd

```
template<class _Fn2>// functor adapter _Func(left, stored)
class binder2nd
: public unary_function<typename _Fn2::first_argument_type,
                                typename _Fn2::result_type>{
public:
   typedef unary_function<typename _Fn2::first_argument_type,
       typename _Fn2::result_type> _Base;
   typedef typename _Base::argument_type argument_type;
   typedef typename _Base::result_type result_type;
    . . .
protected:
   _Fn2 op; // the functor to apply
   typename _Fn2::second_argument_type value; // the right operand
};
```

class binder2nd-2

```
template<class _Fn2>
class binder2nd : public unary_function ....
   binder2nd(const _Fn2& _Func, const typename
                          _Fn2::second_argument_type& _Right)
       : op(_Func), value(_Right) {}
   result_type operator()(const argument_type& _Left) const {
                 return (op(_Left, value)); }
   result_type operator()(argument_type& _Left) const{
                  return (op(_Left, value));}
};
```

bind2nd()

```
template<class Fn2, class T> inline
binder2nd<Fn2> bind2nd(const Fn2& func,
                       const T& right)
{// return a binder2nd functor adapter

  typename Fn2::second_argument_type
            val(right);

  return binder2nd<Fn2>(func, val);
}
```

class binder1st

```
template<class _Fn2>
class binder1st: public unary_function<typename
_Fn2::second_argument_type, typename _Fn2::result_type>
{  // functor adapter _Func(stored, right)
public:
  typedef unary_function<typename _Fn2::second_argument_type,
  typename _Fn2::result_type> _Base;
  typedef typename _Base::argument_type argument_type;
  typedef typename _Base::result_type result_type;
      . . . . .
protected:
  _Fn2 op;  // the functor to apply
  typename _Fn2::first_argument_type value;// the left operand
};
```

class binder1st-2

```
template<class _Fn2>
class binder1st : public unary_function
. . .

  binder1st(const _Fn2& _Func,
       const typename _Fn2::first_argument_type& _Left)
            : op(_Func), value(_Left) {}
  result_type operator()(const argument_type& _Right)
           const {return (op(value, _Right));}
  result_type operator()(argument_type& _Right) const
             {return (op(value, _Right));}

};
```

Використання адаптерів

```
vector<double> v1(4,4);
ostream_iterator<double> out(cout," ");
transform(v1.begin(), v1.end(), out,
        bind2nd(divides <double>(), 2));
// 2 2 2 2

transform(v1.begin(), v1.end(), out,
        bind1st(divides <double>(), 2));
// ?
```

- Крім бібліотечних функціональних об”єктів можна конвертувати з допомогою алаптерів й функціональні об’єкти, створені користувачем.
- Для цього утворюють класи функціональних об’єктів, похідні від класів **unary_function** чи **binary_function**.
- В якості параметрів шаблонів передають типи, що будуть типами аргументів і типом результату

```
class mult: public binary_function<double, double, double>
{
public:
double operator()(const double &x, const double &y) const
    {return x* y;}
};
void main(){
   vector<double> v1(4,4);
   ostream_iterator<double> out(cout," ");
   transform(v1.begin(), v1.end(), out, bind2nd(mult(), 2));
```

class unary_negate

//Шаблонний клас забезпечує функцію, яка заперечує значення, яке повертає унарна функція

```
template<class Fn1>
class unary_negate
   : public unary_function
      <typename Fn1::argument_type, bool>{
public:
      explicit unary_negate(const Fn1& func):
             functor( func) {}
      bool operator()(const typename
            Fn1::argument_type& left)const
   {return ( !functor( left)); }
protected:
      Fn1 functor; // the functor to apply
};
```

not1()

```
template<class Fn1> inline
unary_negate<Fn1> not1(const Fn1&  func)
{ // return a unary_negate functor
adapter
  return std::unary_negate<Fn1> (func);
}
```

class binary_negate

```
template<class Fn2>
class binary_negate: public binary_function
    <typename Fn2::first_argument_type,
   typename Fn2::second_argument_type, bool> {
public:
  explicit binary_negate(const Fn2& func):
                                  functor(func){}
  bool operator()(
  const typename Fn2::first_argument_type& left,
  const typename Fn2::second_argument_type& right
                                      ) const {
    return (!functor(left, right));
  }
protected:
  Fn2  functor;
};
```

not2()

```
template<class Fn2> inline
binary_negate<Fn2> not2(const Fn2& func)
{
  return std::binary_negate<Fn2>(func);
}
```

class pointer_to_unary_function

//адаптер отримує вказівник на функцію і перетворює її (повертає) у функціональний об’єкт. Працює лише для унарних та бінарних функцій (не для функцій без аргументів)

```
template<class Arg, class Result,
   class Fn = Result (*)(Arg)>
class pointer_to_unary_function
      : public unary_function<Arg, Result>
{
public:
   explicit pointer_to_unary_function(Fn left)
      : pfun(left) {}
   Result operator()(Arg left) const
   {  // call function with operand
      return (pfun(left));
   }
protected:
   Fn pfun; // the function pointer
};
```

ptr_fun()

```
template<class Arg, class Result> inline
pointer_to_unary_function<Arg, Result,
                  Result (__cdecl *)(Arg)>
ptr_fun( Result(__cdecl * left)(Arg) ){
// return pointer_to_unary_function functor adapter
 return
 std::pointer_to_unary_function
   <Arg, Result, Result (__cdecl *)(Arg)>
   (left);
}
```

ptr_fun()-2

```
template< class Arg1, class Arg2,
                class Result> inline
pointer_to_binary_function< Arg1, Arg2,
Result, Result (__cdecl *)(Arg1, Arg2)>
    ptr_fun
( Result(__cdecl * left)(Arg1, Arg2) ){
// return pointer_to_binary_function functor adapter
 return
 pointer_to_binary_function
   <Arg1, Arg2, Result,
Result(__cdecl*) (Arg1, Arg2)> (left);
}
```

```
int d[ ] = { 123, 94, 10, 314, 315 };

bool isEven(int x) { return x % 2 = = 0; }

int main() {
  vector<bool> vb;
  transform(d, d + 5, back_inserter(vb),
                not1(ptr_fun(isEven)));
    // vb: 1 0 0 0 1
}
```

```
double d[] = { 01.23, 91.370, 56.661,023.230, 19.959,
  1.0, 3.14159 };
int main() {
vector<double> vd;

transform(d, d + 7,back_inserter(vd),
bind2nd(ptr_fun<double, double, double>(pow), 2.0));

copy(vd.begin(),vd.end(),ostream_iterator<double>(cout
  , " "));
}
```

Адаптери `mem_fun()` та `mem_fun_ref_t`

- Адаптер **mem_fun( )** утворює функціональний об”єкт, який викликається, використовуючи вказівник на об”єкт, для якого треба викликати функцію-член (метод)
- Адаптер **mem_fun_ref( )** викликає метод прямо для об”єкта.

class mem_fun_t

```
template<class Result, class T>
class mem_fun_t
          : public unary_function< T*, Result> {
// functor adapter (*p->*pfunc)(), non-const
public:
    explicit mem_fun_t(Result (T::* pm)())
     : pmemfun(pm){}
    Result operator()(T * pleft) const
    { // call function
     return ((pleft->*pmemfun)());
    }
private:
    Result(T::*pmemfun)();// the member function pntr
};
```

mem_fun()

```
template<class Result, class T> inline
mem_fun_t<Result, T>
   mem_fun(Result (T::* pm)())
{  // return a mem_fun_t functor adapter
   return mem_fun_t<Result, T>(pm);
}

template<class Result, class T> inline
const_mem_fun_t<Result, T>
   mem_fun(Result (T::* pm)() const)
{ // return a const_mem_fun_t functor adapter
return const_mem_fun_t<Result, T>(pm);
}
```

```
class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  virtual void draw() { cout << "Circle::Draw()" << endl; }
  ~Circle() { cout << "Circle::~Circle()" << endl; }
};

class Square : public Shape {
public:
  virtual void draw() { cout << "Square::Draw()" << endl; }
  ~Square() { cout << "Square::~Square()" << endl; }
};

int main() {
  vector<Shape*> vs;
  vs.push_back(new Circle);
  vs.push_back(new Square);
  for_each(vs.begin(), vs.end(), mem_fun(&Shape::draw));

} ///:~
```

class mem_fun_ref_t

```
template<class Result, class T>
class mem_fun_ref_t
      : public unary_function<T, Result> {
   // functor adapter (*left.*pfunc)(), non-const *pfunc
public:
   explicit mem_fun_ref_t(Result (T::* pm)())
      : pmemfun(pm){}
   Result operator()(T&  left) const {
      return (left.*pmemfun)();
   }

private:
   Result(T::*memfun)();// the member function pointer
};
```

mem_fun_ref()

```
template<class Result, class _T> inline
mem_fun_ref_t<Result, T>
     mem_fun_ref(Result (T::* pm)())
{  // return a mem_fun_ref_t functor adapter
   return mem_fun_ref_t<Result, T>(pm);
}
```

```
class Angle {
  int degrees;
public:
  Angle(int deg) : degrees(deg) {}
  int mul(int times) { return degrees *= times; }
};

int main() {
  vector<Angle> va;
  for(int i = 0; i < 50; i += 10)
    va.push_back(Angle(i));
  int x[] = { 1, 2, 3, 4, 5 };
  transform(va.begin(), va.end(), x,
     ostream_iterator<int>(cout, " "),
     mem_fun_ref(&Angle::mul));
  cout << endl;
  // Output: 0 20 60 120 200
} ///:~
```

mem_fun1_t

```
template<class Result, class T, class Arg>
class mem_fun1_t
   : public binary_function<T *, Arg, Result> {
   // functor adapter (*p->*pfunc)(val), non-const *pfunc
public:
      explicit mem_fun1_t(Result (T::* pm)(Arg))
           : pmemfun(pm){}
   Result operator()(T *pleft, Arg right) const
   {  // call function with operand
         return ( pleft->*pmemfun)(right);
   }
private:
   Result(T::* pmemfun)(Arg);//the member function pntr
};
```

mem_fun()-2

```
template<class Result, class T, class Arg> inline
mem_fun1_t<Result, T, Arg>
    mem_fun(Result (T::*pm)(Arg))
{  // return a mem_fun1_t functor adapter
   return mem_fun1_t<Result, T, Arg>(pm);
}


template<class Result, class T, class Arg> inline
const_mem_fun1_t<Result, T, Arg>
    mem_fun( Result (T::*pm)(Arg) const)
{  // return a const_mem_fun1_t functor adapter
   return const_mem_fun1_t<Result, T, Arg> (pm);

}
```