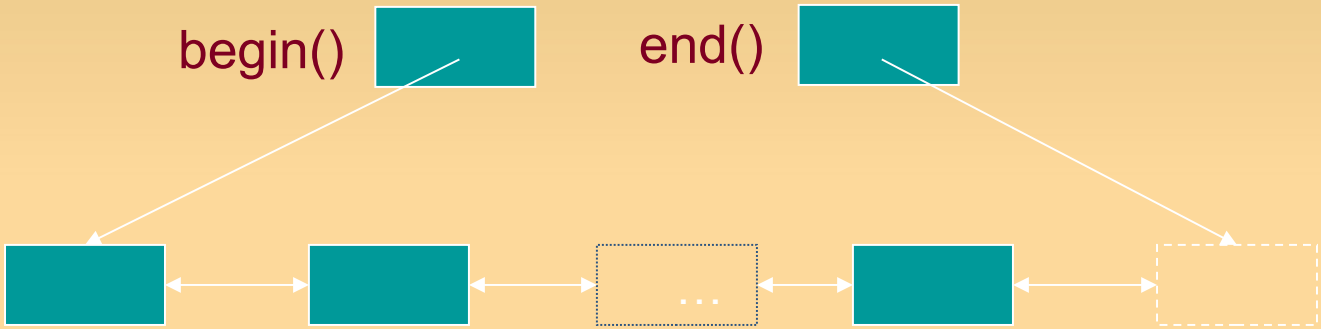
The STL
(maps and algorithms)

Overview

- Common tasks and ideals
- Containers, algorithms, and iterators
- The simplest algorithm: find()
- Parameterization of algorithms
  - find_if() and function objects
- Sequence containers
  - vector and list
- Associative containers
  - map, set
- Standard algorithms
  - copy, sort, …
  - Input iterators and output iterators
- List of useful facilities
  - Headers, algorithms, containers, function objects

Basic model (reminder)

- A pair of iterators define a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



- An iterator is a class that supports (at least) following operations:
  - **++** - go to next element (iterates)
  - * - get element's value (dereference)
  - ==, != - does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. **--**, **+**, and **[ ]**)

Accumulate algorithm
(sum the elements of a sequence)

```
// one of many generic algorithms
template<class It, class T> T accumulate(It first, It last, T init)
{
    while (first != last) {
        init = init + (*first);      // dereferenced iterator is element value
        ++first;                     // "running iterator" of input sequence
    }
    return init;
}
```

v:


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

Accumulate
(examples of use)

```
void f(vector<double>& vd)
{
    // sum up elements of vector vd
    double sum = accumulate(vd.begin(), vd.end(), 0.0);

    // 3rd argument is called the "initializer"
    // Popular idiom: use the variable for the result as the initializer.
    double sum = 0.0;
    sum = accumulate(vd.begin(), vd.end(), sum);
}
```

Accumulate
with binary operator as 4-th parameter

This generic algorithm works with any binary (2 operands) operation ***, -, +** etc. I.e. any (user-defined or library's) binary function (functor) can be used.

```
template<class It, class T, class BinOp>
T accumulate(It first, It last, T init, BinOp op) // operation is a parameter !
{
    while (first != last) {
        init = op(init, (*first)); // means "perform binary operation op
                                   // with init and (*first)"
        ++first;                   // go to next element of the sequence
    }
    return init; // return result of type T
}
```

Accumulate
(with binary operation from library as parameter)

```
// often, we need multiplication rather than addition:
#include <numeric>
#include <functional>     // for library functors
double f(list<double>& ld)
{
    double product = accumulate( ld.begin(), ld.end(),
                                 1.0, multiplies<double>() );
    return product;
}
```

initializer 1.0

multiplies for double

*// **multiplies** is (one of many) standard library templated functors.*

*// This functor do multiplication (similar to **BinOp** on previous slide)*

Accumulate
(what to do if the data is part of an complex object?)

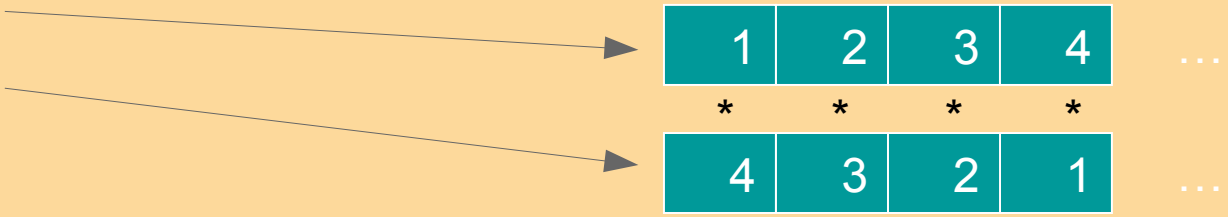
```
struct Record {  // our class to keep record of sold units
    int n_units;            // number of units sold
    double unit_price;      // price of one unit
    string unit_name;       // sold units' name
};  // we need binary operator function to deal with some of data members

// our binary (2 arguments) function to add price of one Record object:
double price(double v, const Record& r)      // sum up v and price of r
{
    return (v  +  r.unit_price * r.n_units);
}

// Here we use our (user-defined) binary operator in "accumulate"
double total (const vector<Record>& vr) {    // calculate total price
    return (accumulate(vr.begin(), vr.end(), 0.0, price));
}
```

Inner product algorithm
(only for sequences of the same size)

```
// Multiply element by element
// (could be used, for example, to multiply two 4-vectors
// to get a scalar product)
template<class It1, class It2, class T>
T inner_product(It1 first1, It1 last1, It2 first2, T init)
{
    while(first1 != last1) {
        init  = init + (*first1) * (*first2);     // multiply pairs of elements and
        add
        ++first1;
        ++first2;
    }
    return init;
}
```



Inner product example

```
// calculate the Dow Jones industrial index:

vector<double> dow_price;     // share price for each company
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);
// …
vector<double> dow_weight;   // weight index for each company
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
// …
double dj_index = inner_product ( // multiply (price,weight) pairs and sum
          dow_price.begin(), dow_price.end(),
          dow_weight.begin(), 0.0 );
```

Inner product
(even more generalized)

```
// we can provide our own operations for combining pairs
// of elements' values with templated initializer "T init":
template<class It1, class It2, class T, class BinOp1, class BinOp2 >
T inner_product (It1 first1,  It1 last1,  It2 first2,  T init,
                 BinOp1 op1, BinOp2 op2)     // 6 templated parameters
{
    while( first1 != last1 ) {
        init  = op1(init, op2(*first1, *first2));
        ++first1;
        ++first2;
    }
    return init;
}
// for Dow Jones calculation: op1 is + and op2 is * binary operators
```

An associative container map.

- for a **vector**, you subscript is an integer (element number
- for a **map**, you can define the subscript to be any type (!)

map<Tkey, Tval> m; *// this map will keep (Tkey, Tval) pairs*

map<string, int> m; *// will keep (string, int) pairs*

m["abc"] *will add pair ("abc",0) to map* **m**

An associative container map.

Use of **map** in simple words counting program.

```
map<string, int> words; // this map will keep ('string', 'int') pairs

string s;
// read in loop strings from cin ('till Ctrl-D), use it as Key in the map
// and  increment  Value (int) every time string s is found in the map
while (cin>>s) ++words[s];  // words are "subscripted" by a string
                           // and words[s] returns an int&

// define my_iter simply as short name for map<string,int>::const_iterator
typedef map<string, int>::const_iterator  my_iter;

// loop over map and print "word" and "counter"
for (my_iter  p = words.begin(); p != words.end(); ++p) {     // iterate
   cout << (*p).first << ": " <<(*p).second << endl;
}
```

map
(important remarks)

- If the key **s** is not yet in the map,
  attempts to access **words[s]** creates the key (!) and call default constructor for value type (e.g. set **int** to 0)

- Map's iterator (**p**) points to pair<Tkey,Tval> object (**pair<string, int>** in our example)
- To get **key**: p->first or (*p).first
- To get **value**: p->second or (*p).second

- To check if a key **s** is in the map one has to use **find()**:
  p = words.find(s);
  if (p != words.end()) { *// key is found ...***}**

An input for our “words counting ” program

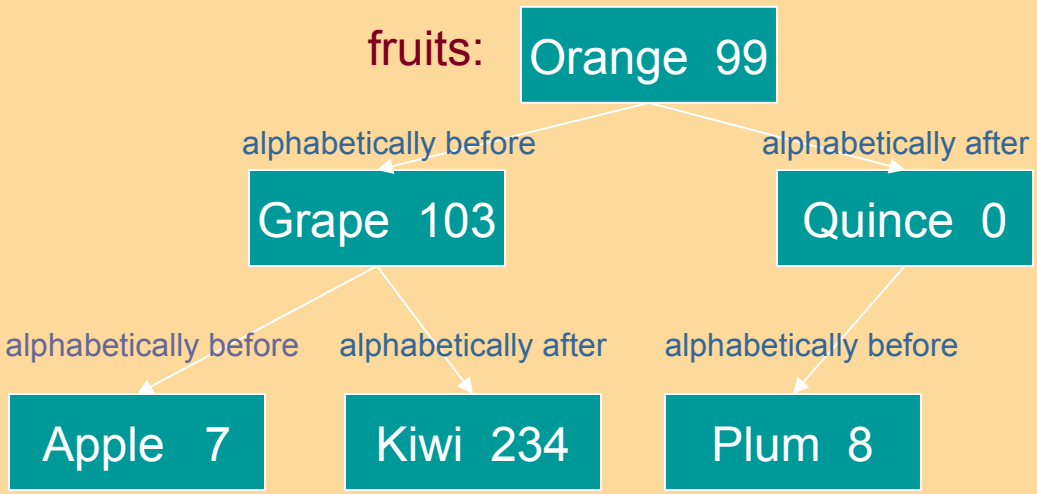
This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

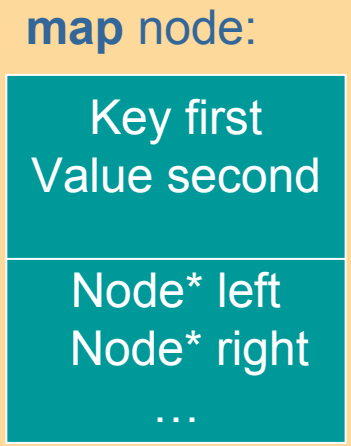
Output of words counting program

(data): 1
(processing): 1
(the: 1
C++: 2
First,: 1
Function: 1
I: 1
It: 1
STL: 1
The: 1
This: 1
a: 1
algorithms: 3
algorithms.: 1
an: 1
and: 5
are: 2
concepts,: 1
containers: 3
data: 1
dealing: 1
examples: 1
extensible: 1
finally: 1
framework: 1
fundamental: 1
general: 1
ideal,: 1
in: 1
is: 1
iterator: 1
key: 1
lecture: 1
library).: 1
next: 1
notions: 1
objects: 1
of: 3
parameterize: 1
part: 1
present: 1
presented.: 1
presents: 1
program.: 1
sequence: 1
standard: 1
the: 5
then: 1
tie: 1
to: 2
together: 1
used: 2
with: 3
“policies”.: 1

Map
(internal organization)

- after **vector**, **map** is the most useful standard library container
  - Maps (or Dictionaries and/or hash tables) are the backbone of scripting languages
- a **map***'s elements are ordered* in balanced binary tree
  - By default they are ordered by **<** (less than) operator of **Key***s*
  - For example, **map<string, int> fruits**; (e.g.fruit→ pieces)





Map (declaration)

```
// note the similarity to vector and  list

template<class Key, class Value> class map {
    // ...
    class iterator {....};          // a pointer to a tree node

    iterator begin();               // points to first element
    iterator end();                 // points to one beyond the last element

    Value& operator[ ] (const Key&);    // accessor (as in vector)

    iterator find(const Key& k);        // is there is an entry for k?

    void erase(iterator p);             // remove element pointed to by p

    // returns iterator to inserted element and true/false for new/old Key
    pair<iterator, bool> insert(const pair<Key,Value>& v);  // insert pair

    // ...
};
```

Map examples (let's build some maps)

```
map<string,double> dow;     // Dow Jones industrial index (symbol → price map)
// here we fill our map dow
dow["MMM"] = 81.86;
dow["AA"]    = 34.69;
dow["MO"]    = 54.45;

// alternative (less recommended) way to fill the map
map<string,double> dow_weight;  // Dow  (symbol → weight)
dow_weight.insert(make_pair("MMM", 5.8549));
dow_weight.insert(make_pair("AA",    2.4563))
dow_weight.insert(make_pair("MO",    3.8940));

map<string,string> dow_name;     // Dow   (symbol → full name map)
dow_name["MMM"] = "3M Co.";
dow_name["AA"]    = "Alcoa Inc.";
dow_name["MO"]    = "Altria Group Inc.";
```

Map example (usage of maps)

```
double alcoa_price   = dow["AA"];  // read values from a map
double boeing_price = dow["BO"];  // create entry with default value 0.0
                                  // as "BO" was not in the map

if (dow.find("INTC") != dow.end())        // look in a map for key "INTC"
    cout << "Intel is in the Dow"<<endl;  // if found, print

// iterate through a map
typedef map<string,double>::const_iterator  Dow_iterator; // shortcut

for (Dow_iterator p = dow.begin(); p!=dow.end(); ++p) {
    const string& symbol = (*p).first;              // get ref. to symbol (Key)
    cout << symbol << '\t' << (*p).second << '\t'  // print symbol and value
         << dow_name[symbol] << endl;               // print full name
}
```

Map example (calculate the DJ index)

```
// function to extract price and weight and multiply
// (will be  used as binary operator)
double value_product (
    const pair<string,double>&  dow,          // 1-st argument
    const pair<string,double>&  dow_weight)   // 2-d argument
{
    return (dow.second * dow_weight.second); // price*weight
}

// generalized form of inner_product algorithm is used (was shown before)
double dj_index =
    inner_product (dow.begin(), dow.end(),   // all companies in index
                   dow_weight.begin(), // their weights
                   0.0,                      // initial value
                   plus<double>(),     // + operator for doubles (library)
                   value_product       // extract prices and weights
                );                     // and multiply (binary operator 2)
```

Associative containers overview

- Associative containers:
  - **map**
  - **multimap**
  - **set**
  - **multiset**
  - **unordered_map**
  - **unordered_multimap**
  - **unordered_set**
  - **unordered_multiset**

- The backbone of text manipulation
  - Find a word
  - See if you have already seen a word
  - Find information that correspond to a word

Algorithms. An interface.

- An STL-style algorithm ...
  - … takes one or more sequences
    - Usually as pairs of iterators
  - … takes one or more operations
    - Usually as function objects (Ordinary functions also work)
  - Usually reports “failure”, “not found” by returning the **end()** of a sequence iterator

Some useful standard algorithms

- **i = find(b,e,v)** i points to the first occurrence of v in [b,e)
- **i = find_if(b,e,p)** i points to the first element in [b,e) so that p is true
- **x = count(b,e,v)** x is the number of occurrences of v in [b,e)
- **x = count_if(b,e,p)** x is the number of elements in [b,e) for which p is true
- **sort(b,e)** sort [b,e) using < *)
- **sort(b,e,p)** sort [b,e) using p *)
- **copy(b,e,t)** copy [b,e) to [t, t+(e-b))
- **merge(b,e,b2,e2,t)** merge two sorted sequence [b2,e2) and [b,e) into [r, r+(e-b)+(e2-b2))
- **r = equal_range(b,e,v)** r is the subsequence of [b,e) with range r: [i1-> first elem !< v ; i2 -> first elem > v]
- **equal(b,e,b2)** does all elements of [b,e) and [b2,b2+(e-b)) are equal?

Here: **i –** iterator, **b** – begin() iterator, **e** – end() iterator, **p** – predicate, **t** – target sequence **b** iterator, **r** – range (pair of iterators), **v** – value, **x** – integer

*) generic **sort** needs random access (i.e. operator[]). It's why it doesn't work e.g. for **list**. Corresponding member function **sort()** of the class **list** has to be used.

Generic copy algorithm

```
// definition (In and In_out are iterators here)
template<class In, class In_out> In_out copy(In first, In last, In_out res)
{
    while (first != last) (*res)++ = (*first)++;  // conventional shorthand  for:
                                                  //  *res = *first; ++res; ++first
    return res;
}
// function to copy from li to vd and sort
void f (list<int>& li, vector<double> vd)
{
    if (vd.size() < li.size()) error ("target container too small");
    copy(li.begin(), li.end(), vd.begin()); // generic copy.  Note:
                                            // different container types
                                            // different element types
                                            // implicit type conversion int → double
    // another generic algorithm (sort)
    sort(vd.begin(), vd.end());           // sort according to operator <
    // NB: for list container on has to use list::sort() member function: li.sort();
}
```

“Erase-remove”
(popular idiom)

The erase–remove idiom is a common C++ technique to eliminate elements that fulfill a certain criterion from a C++ Standard Library container (Wikipedia)

Task: remove all not-a-letter character (such as !@#$%^12) from a string

```
// create predicate
bool NotLetter(char c) {return ! isalpha(c);}    // true if c is not a letter

// create and fill a string
string s = “(ab_cd+ef&123)”;

// remove all not letter characters from the string using generic remove_if
s.erase (remove_if (s.begin(), s.end(), NotLetter), s.end());
```

“Erase-remove”
(popular idiom)

- **remove_if** **do not remove elements** from the container, **but move** all elements that do not fit the removal criteria to the front of the range. (The tail of the container has a length equal to the number of "removed" items)
- **remove_if** **returns an iterator** pointing to the first of these tail elements
- **erase** – **erase** this tail

- before **remove_if** **s** is **(ab_cd+ef&123)**
- after **remove_if** **s** become **abcdef+ef&123)** and iterator points to ‘**+**’
- after **erase** **s** become **abcdef**

This idiom uses generic algorithm and allows to remove elements in one pass over sequence.

Input and output iterators

Stream iterator *it's where concept of "streams" and concept of "iterators for sequences" meet each other*

*// **STL** provides iterators for output and input streams:*

```
ostream_iterator<string> oo(cout); // assigning to *oo = write to cout

(*oo) = "Hello, ";        // means cout << "Hello, "
++oo;                     // "get ready for next output operation"
(*oo) = "world! \n";      // means cout << "world! \n"

istream_iterator<string> ii(cin); // reading *ii = read from cin

string s1 = (*ii);        // means cin>>s1
++ii;                     // "get ready for the next input operation"
string s2 = (*ii);        // means cin>>s2
```

Make a simple dictionary
(now using vector and I/O iterators)

```
int main()
{
   string from, to;
   cin >> from >> to;                  // get source and target file names

   ifstream is(from.c_str());          // open input stream   (file to read)
   ofstream os(to.c_str());            // open output stream (file to write)

   istream_iterator<string> ii(is);          // input iterator for stream is
   istream_iterator<string> eos;             // end-marker (default is EOF)
   ostream_iterator<string> oo(os,"\n");  // output iterator for stream os
                                             // (append "\n" each time)
   vector<string> b(ii, eos);   // create vector b filled from input ('till EOF)
   sort(b.begin(), b.end());                 // sort the vector (generic algo.)
   unique_copy(b.begin(), b.end(), oo);  // copy vector to output,
                                             // discarding adjacent duplicates
}
```

Note: not all compilers support this constructor: template <class InputIterator>
vector(InputIterator, InputIterator)

An input for simple dictionary
(the same as for “words counting” program)

This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

Output of simple dictionary

(data)
(processing)
(the
C++
First,
Function
I
It
STL
The
This
a
algorithms
algorithms.
an
and
are
concepts,
containers
data
dealing
examples
extensible
finally
Framework
fundamental
general
ideal,
in
is
iterator
key
lecture
library).
next
notions
objects
of
parameterize
part
present
presented.
presents
program.
sequence
standard
the
then
tie
to
together
used
with
“policies”.

Make a simple dictionary
(now using a set and I/O iterators)

- We are doing a lot of work that we don't really need
  - Why store all the duplicates? (in the vector)
  - Why sort?
  - Why suppress all the duplicates on output?

- Why not just put each word in the right place in a dictionary in the process of inserting data?

  It's possible: just use a **set**

Make a simple dictionary
(using a set and I/O iterators)

```
int main()
{
   string from, to;
   cin >> from >> to;                  // get source and target file names

   ifstream is(from.c_str());          // make input stream
   ofstream os(to.c_str());            // make output stream

   istream_iterator<string> ii(is);          // input iterator for stream is
   istream_iterator<string> eos;             // end-marker (default is EOF)
   ostream_iterator<string> oo(os,"\n");     // output iterator for stream os
                                             // append "\n" each time
   set<string> b(ii,eos);                    // create set b filled from input
   copy(b.begin() ,b.end() ,oo);             // copy set to output
}
```

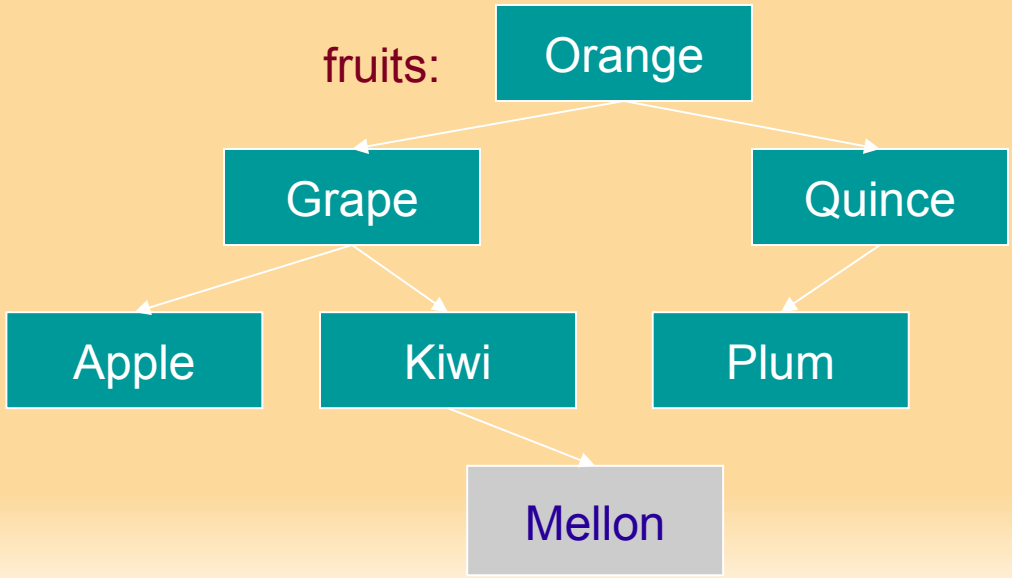
Set

- A **set** *is sorted container without duplicates*

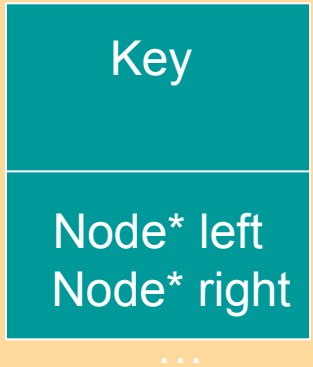
  (one can also consider **set** *as a* **map** *with no values, only keys)*
- A **set** is organized as balanced binary tree
  - By default ordered according "**<**" *operator*

    Example: **set<string> fruits**;
- Conventional way to fill it: **insert(T&)** member function:

  e.g. fruits.insert ("Mellon")



set node:



copy_if()

```
// Let's write useful algorithm for conditional copy (missing in STL):

template<class It, class It_out, class Pred>
   It_out copy_if(It first, It last, It_out res, Pred p)

// copy elements that fulfill the predicate p passed as parameter
{
    while (first != last) {
        if (p(*first)) *res++ = *first; // first, last, res are iterators
        ++first;
    }
    return res;
}

// Now we need templated class for predicate p
// (to be provided by user). E.g. less_then (was already presented)
```

copy_if()
(templated predicate with data for this algorithm)

```
// Implementation of templated predicate with data (functor)
// "data" is T val in this example
// This is what you can't do simply / elegantly with a function operator

template<class T> class less_than {
  public:
    T val;   // data member of parametrized type T
    less_than(const T& v): val(v) { }  // constructor to initialize val
    bool operator()(const T& v) const { return v < val; } // overloaded ()
};

// Example of use of predicate with data:
// Function to copy all elements of v1 to v2 with a value less than M
void f (const vector<int>& v1, vector<int>& v2, int M) {
    copy_if(v1.begin(), v1.end(), v2.begin(), less_than<int>(M));
}
```

Some standard *function objects* (or functors, or predicates)

- From <functional> header of STL:
  - Binary predicates
    - plus, minus, multiplies, divides, modulus
    - equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or
  - Unary predicates
    - negate
    - logical_not

- Unary predicates with data (such as shown **less_then**) are not a part of STL. Write yourself if it's needed.

  "Unary" as they accept one argument. 2-d one is "data" (T val;).

Remark

- STL library contains much more than had been presented.
- It may takes some time to became really familiar with (some of) containers, iterators, algorithms, functors **but**
- this will save a lot of you time in future as you do not need to write yourself a code for many various operations which your software has to perform with your data. Just look into any C++ reference manual (e.g. here https://en.cppreference.com/w/cpp/language), and (with high probability) you will find what you need.

That’s all

Thanks for your attention!

Practical part for today

- Create a **list** of **double**s, sort it and print sorted list.
- Do the same for **list** of **string**s.
- Modify your code to use **set** container for sorting **double**s and **string**s elements.

- Create “telephone book”, i.e. “name → telephone number” pairs. Use **map** container for this.
  - fill it and print content