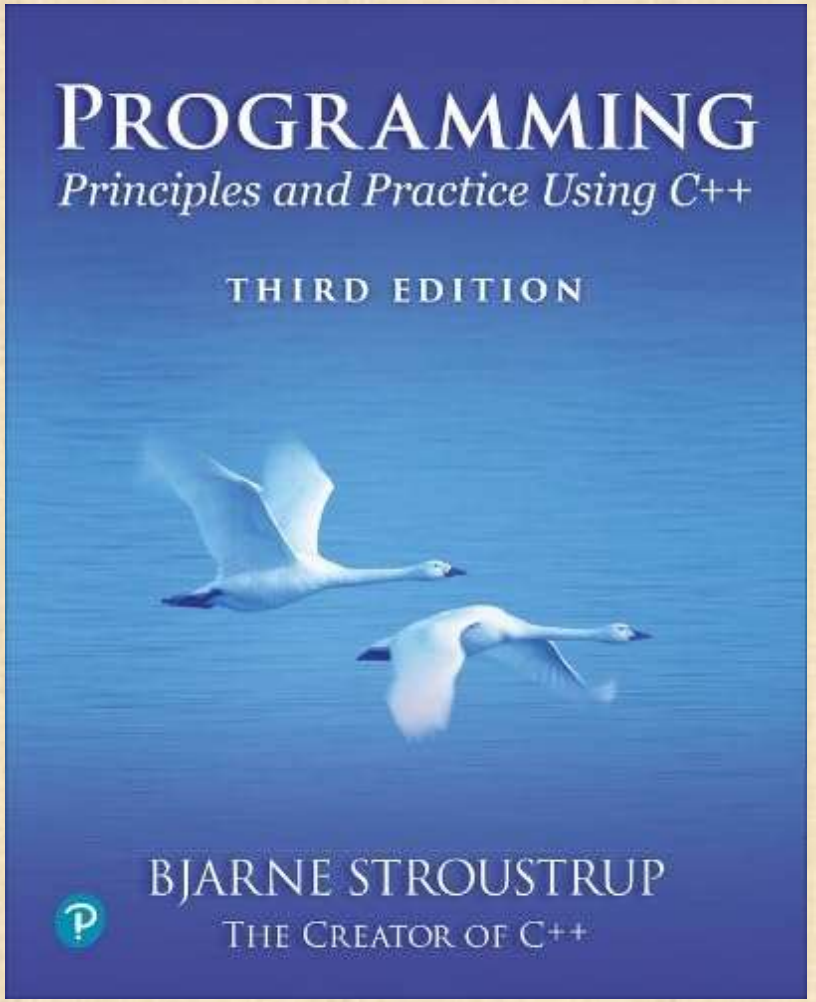
Chapter 20 – Maps and Sets



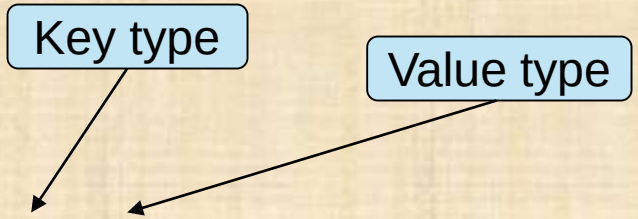
Write programs that do one thing
and do it well.
Write programs to work together.
– Doug McIlroy

Overview

- Maps and unordered_maps
  - Balanced trees and hash tables
- Timing
- Sets
- “Almost containers”
  - Adapting container-like data structures to the STL

Map (an associative array)

- For a **vector**, you subscript using an integer
- For a **map**, you can define the subscript to be any type with an order




```
int main()
{
    map<string,int> words;              // keep (word,frequency) pairs

    for (string s; cin>>s; )
            ++words[s];                 // words is subscripted by a string
                                        // string < (less than) determines the order
                                        // words[s] returns an int&
                                        // the int values are initialized to 0

    for (const auto& [key,value]   :  words)
            cout << key << ": " << value << "\n";
}
```

The words program (word frequencies)

- Input and output

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.

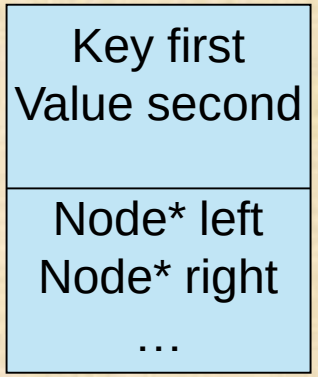
Except for minor details, C++ is a superset of the C programming language.

In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

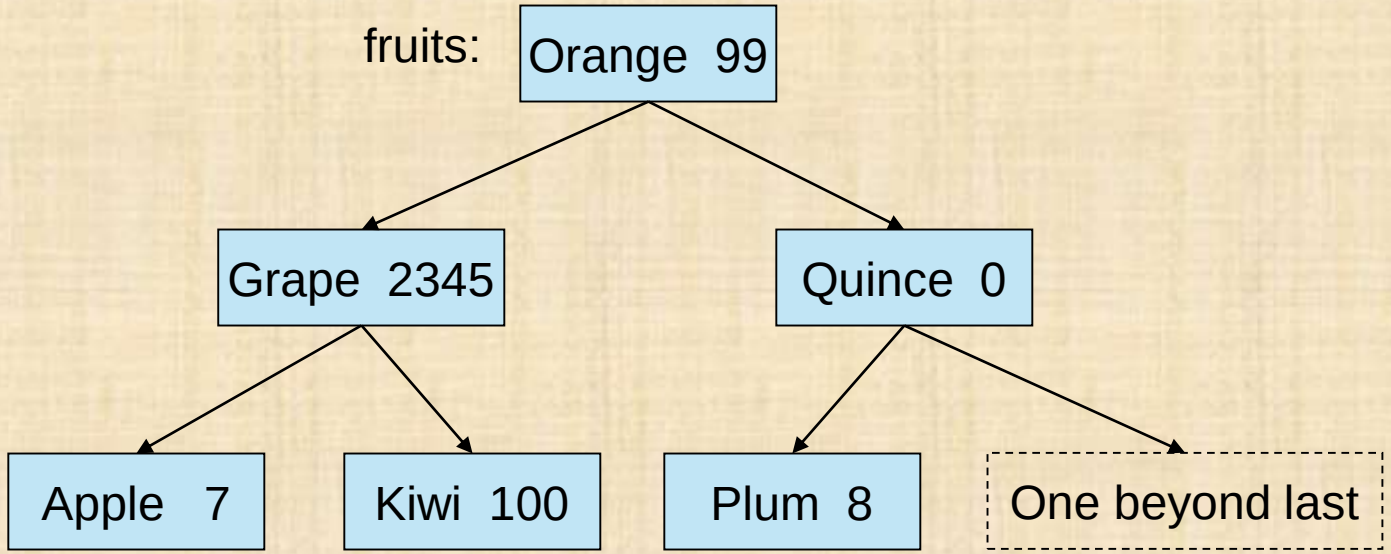
C: 1
C++: 3
C,: 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
more: 1
is: 2
language: 1
language.: 1
make: 1
minor: 1
new: 1
of: 1
programmer.: 1
programming: 3
provided: 1
provides: 1
purpose: 1
serious: 1
superset: 1
the: 3
to: 2
types.: 1

Map

Map node:

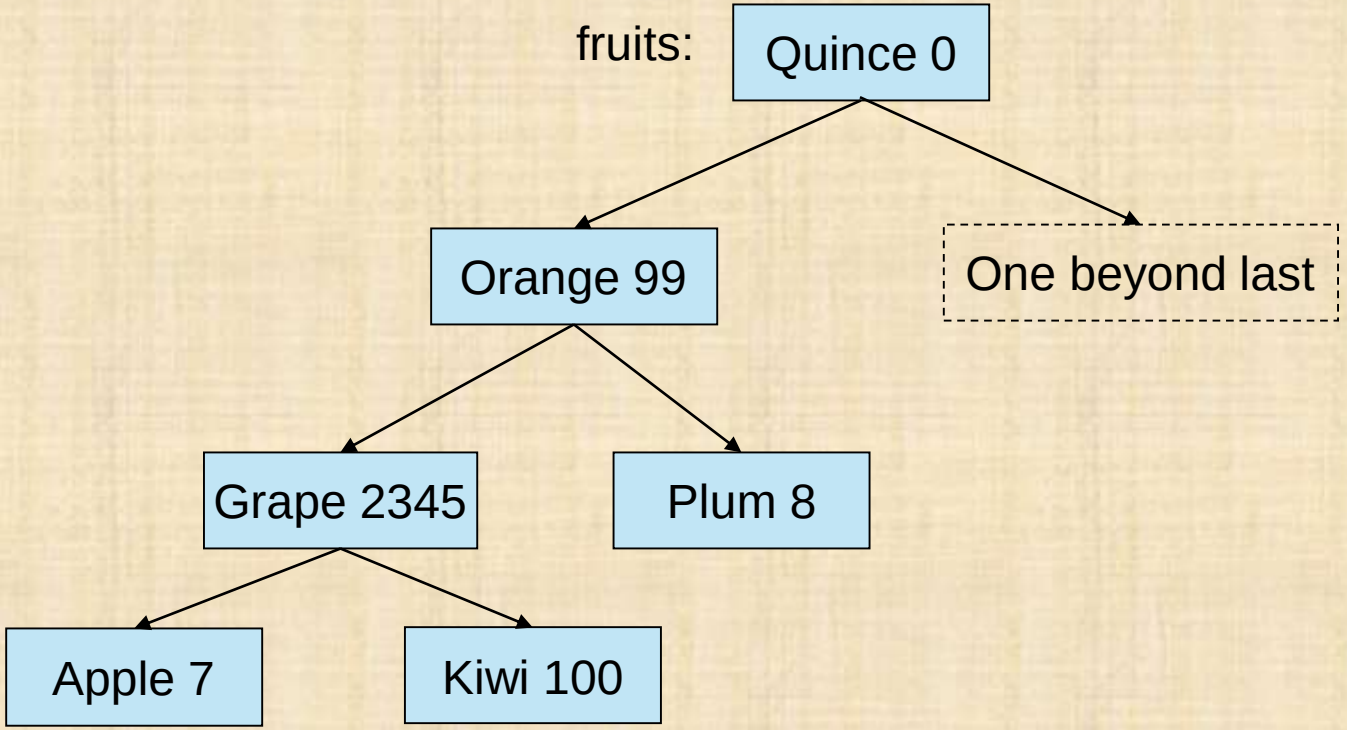


- After **vector**, **map** is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A **map** is really an ordered balanced binary tree
  - By default, ordered by **<** (less than)
  - **map<Fruit,int> fruits = { {Kiwi,100}, {Quince,0}, {Plum,8}, {Apple,7}, {Grape,2345}, {Orange,99} };**



Map

- Maps are balanced
  - Giving on average Log2(N) indirections to reach a node
- Unbalanced trees often require more indirections
  - **`map<Fruit,int> fruits = { {Kiwi,100}, {Quince,0}, {Plum,8}, {Apple,7}, {Grape,2345}, {Orange,99} };`**



Some implementation defined type

Map

```
template<class Key, class Value>
class map {   // note the similarity to vector and  list
  // …
  using value_type = pair<Key,Value>;    // a map deals in (Key,Value) pairs

  using iterator = ???;                          // probably a pointer to a tree node
  using const_iterator = ???;

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

  Value& operator[ ](const Key&); // get Value for Key; creates pair if
                                  // necessary, using Value( )
  iterator find(const Key& k);    // is there an entry for k?

  void erase(iterator p);         // remove element pointed to by p
  pair<iterator, bool> insert(const value_type&);      // insert new (Key,Value) pair ; the bool is false if insert failed
```

Map example (build some maps)

```
// Values from  www.djindexes.com


map<string,double> dow_price = {          // Dow Jones Industrial index
  (symbol,price)
  {"MMM",104.48}, {"AAPL",165.02}, {"MSFT",285.76},
  // ...
};
map<string,double> dow_weight = { // Dow (symbol,weight)
  {"MMM", 2.41}, {"AAPL",2.84}, {"MSFT",4.88},
  // ...
};
map<string,string> dow_name = {           // Dow (symbol,name)
  {"MMM","3M"}, {"AAPL","Apple"}, {"MSFT","Microsoft"},
  // ...
};
```

Map example (some uses)

```
double caterpillar = dow_price ["CAT"];                    // read values from
  a map
double boeing_price = dow_price ["BA"];


if (dow_price.find("INTC") != dow_price.end())              // find an entry in
  a map
  cout << "Intel is in the Dow\n";


// Iterating through a map is easy:
for (const auto& [symbol,price] : dow_price)     // output names in
  alphabetic order of symbols
  cout << symbol << '\t' << price << '\t' << dow_name[symbol] << '\n';
```

Unordered_map (aka hash table)

- An unordered_map is like a map, except
  - Lookup is by a "hash function"
  - An **unordered_map**'s elements are not in order

```
unordered_map<string,double> dow_price=  {      // Dow Jones Industrial index (symbol,price)
  {"MMM",104.48}, {"AAPL",165.02}, {"MSFT",285.76},
  // ...
};


for (const auto& [symbol,price] : dow_price)    // output names in no defined order
        cout << symbol << '\t' << price << '\t' << dow_name[symbol] << '\n';
```

Lookup: vector, map, and unordered_map

- Linear search in a vector is proportional to N, the number of elements
  - But each lookup is very cheap
- Map Lookup cost is proportional to log2(N)
  - Indirections through pointers
- Unordered_map lookup cost is proportional to 1 (constant time)
  - One
    vector:
      clas
    map:
unordered_map:

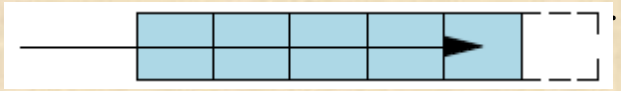
ue

N	16	128	1024	16*1024	1024*1024	1024*1024*1024
Log2(N)	4	7	10	14	20	30
1	1	1	1	1	1	1

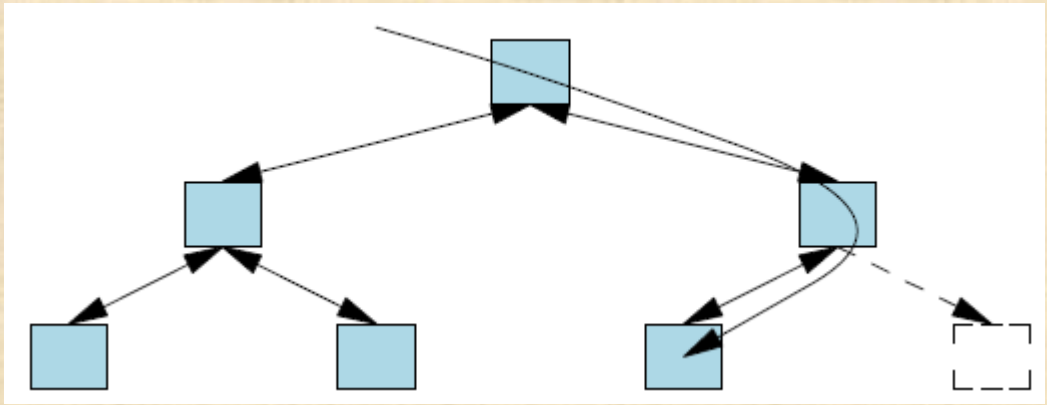
- The performance of an unordered_map is critically dependent on the hash function

Lookup: vector, map, and unordered_map

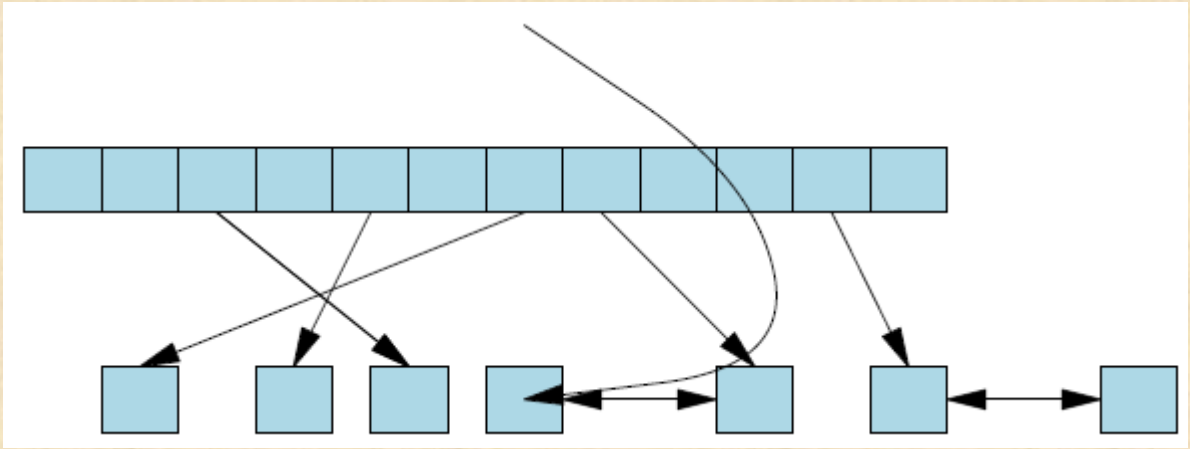
- Vector (linear sear



- Map (tree walk):



- Unordered_map (hash lookup):



Timing

- Never talk about "efficiency" without measurement
  - Space
  - Runtime
  - Compile time
- Beware of
  - Tiny examples
  - Useful complexities that only make sense for huge examples
- Don't tryst your "intuition"/guesses
  - Even experts can guess wrong by a factor of 1000
- Always run a test at least three times
  - And test for consistency
- Be suspicious of your results
  - Always try to explain them

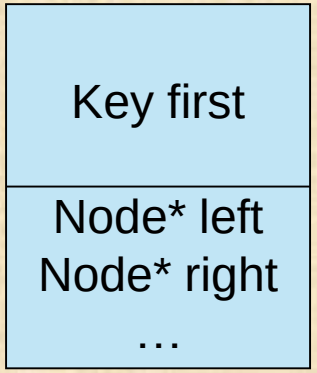
Timing

- Simple tests are much better than mere guesses

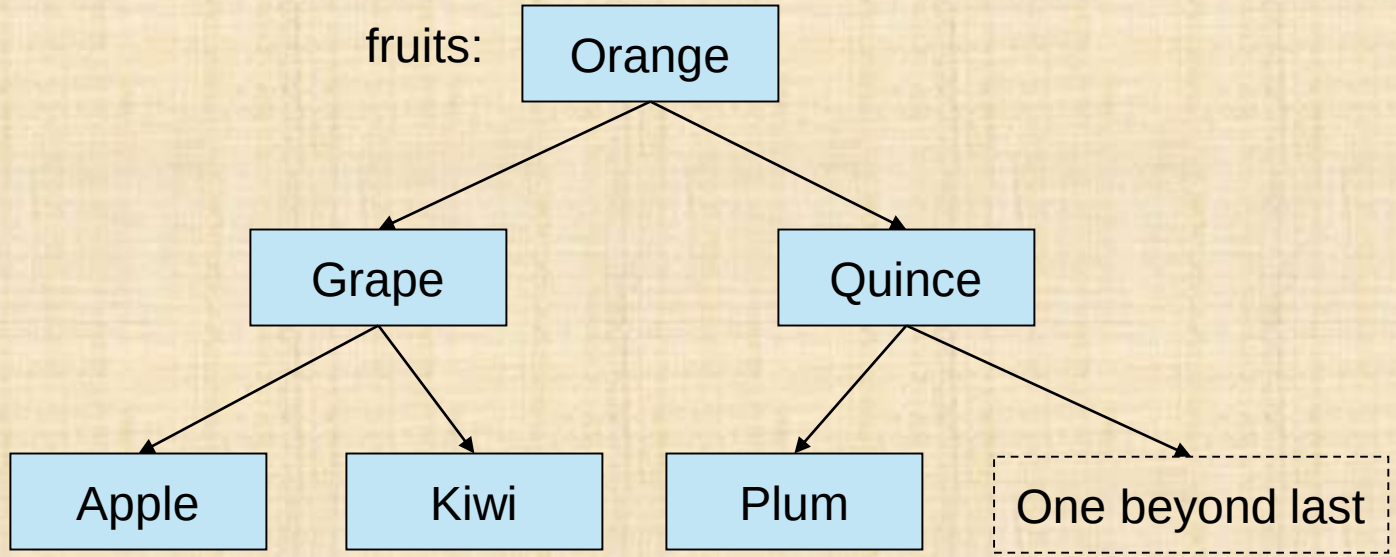
```
using namespace chrono;                    // that's where the timing support is

auto t0 = system_clock::now();                  // the point of time of the call
auto x = do_something();
auto t1 = system_clock::now();
cout << "res: " << x <<'\n';
cout << t1-t0 << '\n';                     // that's how long it took (defalt
  unit)
cout << duration_cast<microseconds>(t1-t0).count() << "us\n";  // that's
  how long it took in μseconds
```

Set

set node:



- A **set** is really an ordered balanced binary tree
  - By default, ordered by **<**
  - **set<string> fruits = { Kiwi, Quince, Plum, Apple, Grape, Orange };**



Set

- We can use a set to order (sort) values

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

    ifstream is {from};             // make input stream
    ofstream os {to};           // make output stream

    set<string> b {istream_iterator<string>{is},
istream_iterator<string>{}};      // read into the set
    copy(b.begin() ,b.end() , ostream_iterator<string>{os, "\n"});
    // copy to output
}
```

The sorting program
(using a set)

- Input and output

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.

Except for minor details, C++ is a superset of the C programming language.

In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

C
C++
C,
Except
In
a
addition
and
by
defining
designed
details,
efficient
enjoyable
facilities
flexible
for
general

more
is
language
language.
make
minor
new
of
programmer.
programming
provided
provides
purpose
serious
superset
the
to
types.

A container (incomplete overview)

- An STL container is characterized by having a set of useful types and operations
- Has a sequence of elements (**begin()**:**end()**)
- Provides copy operations that copy all elements.
- Provides move operations that move all elements
- Names its element type **value_type**
- Has iterator types called **iterator** and **const_iterator**
- Iterators provide *****, **++** (both prefix and postfix), **==**, and **=** with the appropriate semantics
- The iterators for **list** also provide **--** for moving backward; that's called a bidirectional iterator
- The iterators for vector also provide **--**, **[ ]**, **+**, and **-** and are called random-access iterators
- Provides **insert()** and **erase()**, **front()**, and **back()**, **push_back()**, and **pop_back()**, **size()**, **swap()**, etc.
- **vector** and **map** also provide subscripting (e.g., operator **[ ]**)
- Provides comparison operators (**==**, **!=**, **<**, **<=**, **>**, and **>=**) that compare the

Standard containers

- **vector** a contiguously allocated sequence of elements; use it as the default container
- **list** a doubly-linked list; use to insert and delete elements without moving existing elements
- **forward_list** a singly-linked list; use for lists that are mostly empty
- **deque** a cross between a **list** and a **vector**
  don't use until you have expert-level knowledge of algorithms and machine architecture
- **map** a balanced ordered tree; use it when you need to access elements by value (§ 20.2)
- **multimap** a balanced ordered tree where there can be multiple copies of a key
- **set** a balanced ordered tree; use it when you need to keep track of individual values (§ 20.5)
- **multiset** a balanced ordered tree where there can be multiple copies of a key
- **unordered_map** a hash table; an optimized version of **map**;
  use for large maps when you need high performance
  and can devise a good hash function (§20.3)

“Almost containers”

- Data types that provide much of what is required from a standard container, but not all
  - There are many because needs of data representations are many
  - Most are not in the standard (but those may be essential for an application)

- **T[n]** a built-in array; no **size()** or other member functions; prefer a container, such as **vector**, **string**, or **array**, over a built-in array when you have a choice
- **array** a fixed-size array that doesn't suffer most of the problems related to the built-in arrays
- **string** holds only characters but provides operations useful for text manipulation, such as concatenation (**+**and **+=**); prefer the standard string to other strings
- **valarray** a numerical vector with mathematical vector operations, but with many restrictions to encourage high-performance implementations; use only if you do a lot of vector arithmetic

Built-in array and **array**

- You can use **std::array** much like a container
  - A built-in array doesn't carry its number of elements out of its scop; a **std::array** does
  - A built-in implicitly converts to a pointer ("forgetting its size); a **std::array** does not
  - Identical size and speed

```
void array_test(auto& a1, auto a2)   // or f(array<double,6>& , double arr2[] )
{
    ranges::sort(a1);                // OK
    sort(begin(a1), end(a1));        // OK

    ranges::sort(a2);                // error: we don't know where the end of the array is
    sort(begin(a2), end(a2));        // error: we don't know where the end of the array is
}
```



```
array<double,6> arr1 = { 0.0, 1.1, 3.3, 5.5, 2.2, 4.4 };
```

Input and output iterators

- we can provide iterators for **istream**s and **ostream**s
  - Thus, make them usable like containers

```
ostream_iterator<string> oo(cout); // assigning to *oo is to write to cout
*oo = "Hello, ";                   // meaning cout << "Hello, "
++oo;                              // "get ready for next output operation"
*oo = "world!\n";                  // meaning cout << "world!\n"

istream_iterator<string> ii(cin);  // reading *ii is to read a string from cin
string s1 = *ii;                   // meaning cin>>s1
++ii;                              // "get ready for the next input operation"
string s2 = *ii;                   // meaning cin>>s2
```

Make a quick dictionary (using a vector)

- Now we can treat **istream**s and **ostream**s as if they were containers
  - As we say: "copy from input" and "copy to output"

```
istream_iterator<string> ii(cin);                    // make input iterator for cin
istream_iterator<string> eos;                        // input sentinel (defaults to EOF)
ostream_iterator<string> oo(cout,"\n");              // make output iterator for cout; append "\n" each time

vector<string> buf {ii,eos};                         // buf is a vector initialized from input
ranges::sort(buf);                                   // sort the buffer
ranges::unique_copy(buf,oo);                         // copy buffer cout; discard replicated values
```

The sorting program
(using a vector)

- Input and output

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.

Except for minor details, C++ is a superset of the C programming language.

In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

```
C
C++
C,
Except
In
a
addition
and
by
defining
designed
details,
efficient
enjoyable
facilities
flexible
for
general
more
is
language
language.
make
minor
new
of
programmer.
programming
provided
provides
purpose
serious
superset
the
to
types.
```

Which version do you think is faster?
The one using a map or the one using a vector?
Why?

Range checking

- But what if we were writing into a fixed-sized buffer?
  - Buffer overflows are nasty
  - Consider

```
vector<int> v = {0,1,2,3,4,5};
vector<int> v1(10);
vector<int> v2(5);
```

- #1 check the size

```
If (v.size() < v1.size()) Ranges::copy(v,v1); else throw Range_error{};
If (v.size() < v2.size()) Ranges::copy(v,v2); else throw Range_error{};
```

- #2 use a checked iterator

```
ranges::copy(v,Output_range{v1});          // copies v into v1
ranges::copy(v,Output_range{v2});          // throws Range_error
```

Range checking

- But what if we were writing into a fixed-sized buffer?

```
template<ranges::range R>
class Output_range {
public:
    using value_type = ranges::range_value_t<R>;
    using difference_type = int;
    Output_range(R r) : b{ r.begin() }, e{ r.end() }, p{ b } {}
    Output_range& operator++() { check_end(); ++p; return *this;
}
    Output_range operator++(int) { check_end(); auto t{ *this };
++p; return t; }
    value_type& operator*() const { check_end();  return *p; }
private:
    void check_end() const { if (p == e) throw Range_error{}; }
    ranges::iterator_t<R> b;
    ranges::iterator_t<R> e;
    ranges::iterator_t<R> p;
```

Ranges and iterators

- Iterators can be a source of errors

```
sort(v.end(),v.start());          // Oops!
sort(v1.start(),v2.end());        // Oops!
```

- Prefer ranges
  - They more precisely say what's intended

```
ranges::sort(v);                  // if this doesn't make sense the compiler will tell us
```

  - we can define a range in three ways:
    - *{begin,end}* a pair of iterators
    - *{begin,length}* an iterator and a number of elements
    - *{begin,predicate}* an iterator and predicate to determine if the end has been reached

Ranges and iterators

- Iterator categories
  - **input iterator** Can iterate forward using **++** and read element values using *****.
    This is the kind of iterator that **istream** offers.
    If **(*p).m** is valid, **p->m** can be used as a shorthand.
  - **output iterator** Can iterate forward using **++** and write element values using *****.
    This is the kind of iterator that **ostream** offers.
  - **forward iterator** An input iterator that can iterate repeatedly over a sequence
    and repeatedly read from or write to an element.
    This is the kind of iterator that **forward_list** offers.
  - **bidirectional iterator** A forward iterator that can move backward (using **--**).
    This is the kind of iterator that **list**, **map**, and **set** offer.
  - **random-access iterator** A bidirectional iterator that can move forward

Next lecture

- Algorithms!