SECTION 07

Library changes

© Feabhas Ltd 2012

Fixed-sized vector

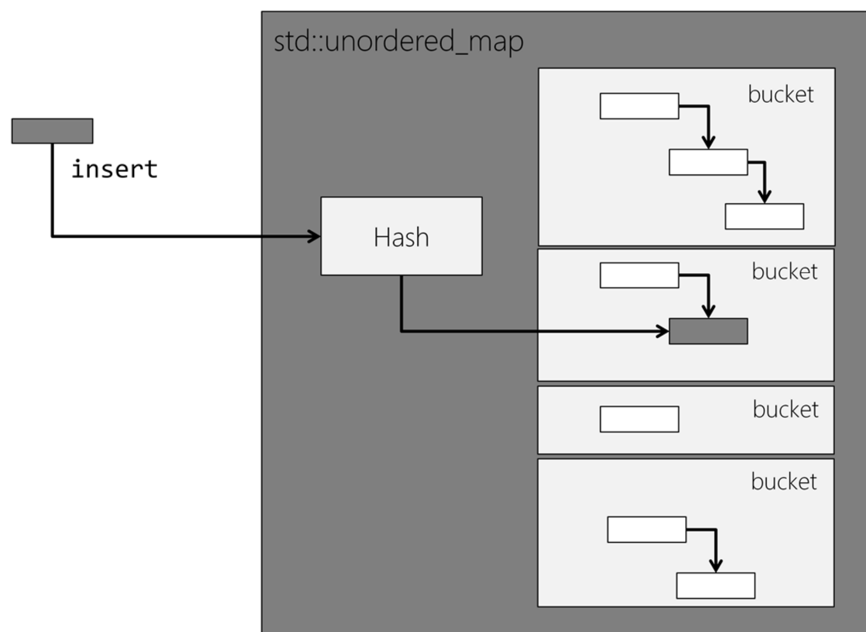
```
#include <array>

int main()
{
  array<int, 10> a;      // Created on the stack

  // arrays can be used just like other standard
  // containers.
  //
  iota(a.begin(), a.end(), 0);
  copy(a.begin(), a.end(), ostream_iterator<int>(cout, "\n"));

  cout << a.at(5) << endl;    // Bounds-checked access
  cout << a[5]    << endl;    // No bounds-checking
}
```

`std::array` is fixed-size vector; basically, a thin wrapper around C-style arrays.

This allows you to use (efficient) C-style arrays with algorithms using STL-style iterators rather than C-style pointer syntax. For example, you can write `a.end()`, rather than `&a[10]`.

© Feabhas Ltd 2012

std::unordered_map



`std::unordered_map` can be used as a hash table.

The `unordered_map` is implemented as a sequence split into ordered-subsequences, called *buckets*. The `unordered_map` uses a hashing algorithm to locate elements in an appropriate bucket.

If there is more than one element per bucket these elements can be iterated through.

© Feabhas Ltd 2012

Hash tables in operation

```
class Part
{
public:
  Part(unsigned int id = 0) : partNum(id) {}
  void show() const
  {
    cout << "Part Number: " << partNum << endl;
  }

private:
  unsigned int partNum;
};
```

Suppose we want to store `Part` objects by their type (identified by a `std::string`)

© Feabhas Ltd 2012

```
#include <unordered_map>
using namespace std;

int main()
{
  typedef unordered_map<string, Part> HashTable;
  HashTable partsList;

  partsList.insert(HashTable::value_type("board", Part(91011)));
  partsList.insert(HashTable::value_type("fastener", Part(121314)));
  partsList.insert(HashTable::value_type("device", Part(91011)));


  // Find a particular bucket, based on a key.
  //
  HashTable::size_type bucketNum = partsList.bucket("fastener");
  cout << "Fastener bucket size : " << partsList.bucket_size(bucketNum) << endl;


  // Iterate through the bucket, printing the values
  //
  for_each(partsList.begin(bucketNum), partsList.end(bucketNum),
    [](const pair<string, Part>& elem)
    {
      cout << elem.first << " : ";
      elem.second.show();
    });
}
```

`HashTable::value_type` is a `typedef` for a `std::pair<Key, value>`

The `bucket()` method retrieves the identifier for any particular bucket, based on a supplied key. Each bucket is a sequence of `std::pair` elements; which may be empty.

Supplying a bucket ID to the `std::unordered_map`'s `begin()` and `end()` functions returns an iterator to the specified bucket. That bucket can then be processed using any of the standard algorithms.

© Feabhas Ltd 2012

Custom hashing algorithms

```
class SimpleHash
{
public:
  size_t operator()(const string& str)
  {
    return (str.size() % 8);
  }
};

typedef unordered_map<string, Part, SimpleHash> HashTable;


int main()
{
  HashTable partsList;

  // As before…
}
```

Hashing algorithms exist for all the built-in types; although you can supply your own if you need. The hashing algorithm is a functor that takes an appropriate type (the key type)

(Note, there is already a supplied hashing algorithm for strings)

© Feabhas Ltd 2012

Other containers

`std::forward_list`

basically, a singly-linked list

Unordered containers

`unordered_multimap`

`unordered_multiset`

C++11 adds a number of other (previously missing) containers.

© Feabhas Ltd 2012

Binary functor adapters

```
using namespace std;

int main()
{
  list <double> vals;
  double val;

  while (cin >> val) vals.push_back(val);

  transform(vals.begin(), vals.end(), vals.begin(),
            bind2nd(divides<double>(), 3.0) );

  cout << "Divided contents : " << endl;

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

`std::bind1st` and `std::bind2nd` are template adapter functions that convert binary functors into unary functors by 'binding' either the first or second argument parameter to a supplied argument.

© Feabhas Ltd 2012

std::bind

```
using namespace std;
using namespace std::placeholders // For _1

int main()
{
  list <double> vals;
  double val;

  while (cin >> val) vals.push_back(val);

  transform (vals.begin(), vals.end(), vals.begin(),
            bind(divides<double>(), _1, 3.0));

  cout << "Divided contents : " << endl;

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

`std::bind` replaces `std::bind1st` and `std::bind2nd`

`std::bind` takes a function as its first parameter. The subsequent parameters are either placeholders or values. Placeholders, of the form `_1`, `_2, etc.` are filled in at run-time. Note placeholders live in their own namespace (`std::placeholders`)

The code shown is the equivalent of `std::bind2nd(divides<double>, 3.0)`

`std::bind` is not limited to 2 parameters; it can support up to ten.

© Feabhas Ltd 2012

Binding member functions

```
class X
{
public:
  X(int val = 0) : value(val){}
  void oneParamMethod(int i);
  void twoParamMethod(int i, int j);

private:
  int value;
};
```

```
void X::oneParamMethod(int i)
{
  cout << "X::oneParamMethod()" << endl;
  cout << "X::value : " << value << endl;
  cout << "i        : " << i << endl;
  cout << endl;
}

void X::twoParamMethod(int i, int j)
{
  cout << "X::oneParamMethod()" << endl;
  cout << "X::value : " << value << endl;
  cout << "i        : " << i << endl;
  cout << "j        : " << j << endl;
  cout << endl;
}
```

Here we have a simple class with one- and two-parameter member functions.

© Feabhas Ltd 2012

Binding member functions

```
int main()
{
  vector<X> v(10);
  int i = 0;
  generate(v.begin(), v.end(), [&i]() -> X { return *new X(i++); });

  cout << "USING BIND..." << endl;
  for_each(v.begin(), v.end(), bind(&X::oneParamMethod, _1, 100));
  for_each(v.begin(), v.end(), bind(&X::twoParamMethod, _1, 100, 200));


  cout << "USING LAMBDAS..." << endl;
  for_each(v.begin(), v.end(), [](X& theX){ theX.oneParamMethod(100); });
  for_each(v.begin(), v.end(), [](X& theX){ theX.twoParamMethod(100, 200); });

}
```

`std::bind` can be used to access member functions on objects in a more readable way than C++98. In C++98 the equivalent code for the unary procedure would be

```
std::bind2nd(std::mem_fun_ref(&X::twoParamMethod), 100);
```

For member functions remember the first parameter is the `this` pointer.

Note: Lambdas will typically be more efficient than `std::bind`.

© Feabhas Ltd 2012

C++98 std::pair

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
  pair<string, int> p1("Key1", 1234);
  pair<string, int> p2 = make_pair("Key2", 5678);

  cout << "p1 : " << p1.first << " " << p1.second << endl;
  cout << "p2 : " << p2.first << " " << p2.second << endl;
}
```

The C++98 `std::pair` encapsulated a pair of values, which could be of different types.

© Feabhas Ltd 2012

General *n*-tuple - std::tuple

```
#include <string>
#include <iostream>
#include <tuple>

using namespace std;

int main()
{
  tuple<string, int> t1("Key1", 1234);  // Same as std::pair.

  auto t = make_tuple("Key1", 1234, "Key2", 17.678);

  // Use get<> to access elements
  //
  cout << "t : " << get<0>(t) << " ";
  cout << get<1>(t) << " ";
  cout << get<2>(t) << " ";
  cout << get<3>(t) << endl;
}
```

C++11 supplements `std::pair` with `std::tuple`, which represents a general *n*-tuple structure.

`std::make_tuple` constructs a `std::tuple` by deducing the tuple arguments from the supplied parameters.

tuple elements are retrieved using the template `std::get<>()` function

© Feabhas Ltd 2012

std::tie links objects with tuple values

```
int main()
{
  auto t = make_tuple("Key1", 1234, "Key2", 17.678);

  int i;
  string key1;
  string key2;
  double d;

  tie(key1, i, key2, d) = t;  // Tie variables to tuple values

  cout << "t: " << key1;
  cout << " " << i;
  cout << " " << key2;
  cout << " " << d << endl;
}
```

`std::tie` links objects with each of the tuple values.

© Feabhas Ltd 2012

Regular expressions

A regular expression is a match pattern expressed as a string.

For example, the regular expression

```
[[:digit:]]+:[[:digit:]]+(am|pm)
```

will match strings like

```
12:01am
1:20pm
```

Regular expression parsing is a key tool in many programming problems.

© Feabhas Ltd 2012

Regular expression syntax

There are many different regular expression syntaxes.

C++11 supports the following:

- ECMAScript
- POSIX (basic and extended)
- awk
- grep
- egrep

ECMAScript is the default (and is used by Perl); and is used for these notes.

© Feabhas Ltd 2012

ECMAScript syntax (subset)

Wildcards and repetition

```
.          match any character
?          match 0 or 1 times
*          match 0 or more times
+          match 1 or more times
{n}        match exactly n times
{n, }      match at least n times
{n, m}     match n to m times
```

Wildcards and repetition allow you to match any character, or set of characters.

© Feabhas Ltd 2012

ECMAScript syntax (cont'd)

Character sets

```
[abc]     match only a, b or c
[a-z]     match any lowercase character
[A-Z]     match any uppercase character
[^a-z]    match anything NOT a lowercase character
```

Sets allow you to match ranges of values. Both inclusive and exclusive ranges are supported.

© Feabhas Ltd 2012

ECMAScript syntax (cont'd)

Character classes

```
alpha            lowercase and uppercase letters
digit            digits
alnum            characters from either alpha or digit
classes
space            whitespace characters
blank            space or tab
cntrl            file format escape characters (\n, \r
etc.)
punct            punctuation characters
lower            lowercase letters
upper            uppercase letters
graph            characters from lower, upper, digit or
punct
print            characters from either graph or space
xdigit           hexadecimal digits
```

Regular expression classes provide convenient shorthand notations for commonly-used ranges.

© Feabhas Ltd 2012

std::regex_match

```
#include <regex>

using namespace std;

int main()
{
  string inputA = "Flight arrival: SQ521 ETA 12:35 Actual 12:33, Terminal 3";
  string inputB = "SQ521 ETA 12:35";

  regex searchPattern
    ("[[:alpha:]][[:alpha:]][[:digit:]]+ ETA [:digit:]]+:[[:digit:]]+");

  if (regex_match(inputA, searchPattern)) cout << "Input A MATCH" << endl;
  else cout << "Input A NO MATCH" << endl;

  if (regex_match(inputB, searchPattern)) cout << "Input B MATCH" << endl;
  else cout << "Input B NO MATCH" << endl;
}
```

A `std::regex` is initialised with string containing the regular expression.

`std::regex_match` looks for an *exact* match of the search pattern to the input string. It returns *true* if there is a match.

All the regular expression behaviour is found in the header `<regex>`

Input B will match the search pattern on this example.

© Feabhas Ltd 2012

std::regex_search

```
int main()
{
  string inputA = "Flight arrival: BA51 ETA 01:37 AR77 ETA 15:29";

  regex searchPattern ("([[:alpha:]][[:alpha:]][[:digit:]]+) ETA ([[:digit:]]+:[[:digit:]]+)");


  smatch results;                                    // match_results<string::const_iterator>



  regex_search(inputA, results, searchPattern);       // Find a match.

  cout << results.size() << " results:" << endl;
  cout << "Match string : " << results[0] << endl;    // results[0] holds the matched string
  cout << "Sub-matches :" << endl;

  for_each(results.begin() + 1, results.end(),        // Subsequent entries hold the submatches
      [](const string& str) { cout << str << endl;});

}
```

`std::regex_search` looks for a partial match to the search pattern.

`std::regex` requires the search pattern (`std::regex`) and the input string. The output is held in a `std::match_result<string::const_iterator>` object. For convenience, `<regex>` contains a typedef, `std::smatch` (string *match).*

A submatch can be defined in the regex. Submatches are specified by parentheses in the search string.

The std::smatch object will contain a list of submatch strings. The first element (0) holds the matched string (if any); the subsequent entries hold any submatches.

© Feabhas Ltd 2012

Regular expression iterators

```
int main()
{
  string input = "Flight arrivals: BA51 ETA 01:37 AR77 ETA 15:29 BA51 ETA 13:45";

  regex searchPattern ("([[:alpha:]][[:alpha:]][[:digit:]]+)");


  map<string, int> results;


  for (sregex_iterator iter(input.begin(), input.end(), searchPattern);
       iter != sregex_iterator();
       ++iter)
  {
    string str = (*iter)[1];  // first sub-match
    ++results[str];
  }


  for_each(results.begin(), results.end(),
    [](const pair<string, int>& p) { cout << p.first << " " << p.second << endl; });
}
```

A regular expression iterator allows you to iterate through a set of search matches

In this example we are searching through the input string counting unique flight numbers

The `std::regex_iterator` points to the next `smatch` that matches the search pattern. Thus, dereferencing the iterator will give a `smatch` object.

We use the `smatch` sub-match (that is, the second element - the first element gives the entire match string) to index into a map. If the key (match string) is in the map it is incremented; if it doesn't exist it is added (then incremented)

The results are then output using a lambda.

© Feabhas Ltd 2012

Key points

New containers have been added, most notably,

`std::array`
`std::forward_list`
`std::unordered_map`

`std::bind` supercedes `std::bind1st` and `std::bind2nd` as a general-case mechanism for binding parameters to functors in algorithms.

`std::pair` is supplemented by `std::tuple`, for modelling *n*-tuples.

The STL now supports regular expressions through the `regex` library

© Feabhas Ltd 2012