C++: `map`

Collection of *keys*, each with an associated *value*

- Like Java's `java.util.HashMap` or `TreeMap`
- Like Python's `dict` (dictionary) type

Value can be any type you wish

Key can be any type for which < can compare two values

- All numeric types, `char`, `std::string`, etc

C++: map

Declare a map:

```
map<int, string> id_to_name;
```

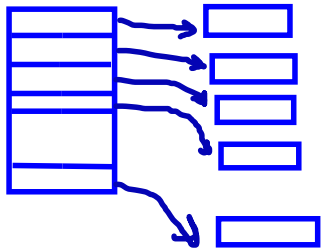
Add a key + value to a map:

```
id_to_name[92394] = "Alex Hamilton";
```

Print a key and associated value:

```
const int k = 92394;
cout << "Key=" << k << ", Value=" << id_to_name[k] << endl;
```

C++: map

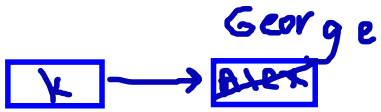


A map can only associate 1 value with a key

```
const int k = 92394;
id_to_name[k] = "Alex Hamilton";
id_to_name[k] = "George Washington"; // Alex is replaced
cout << k << ": " << id_to_name[k] << endl;
```

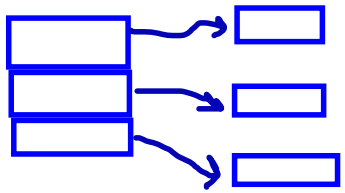
C++: map

```cpp
#include <iostream>
#include <string>
#include <map>

using std::cout;    using std::endl;
using std::string;  using std::map;

int main() {
    map<int, string> id_to_name;
    const int k = 92394;
    id_to_name[k] = "Alex Hamilton";
    id_to_name[k] = "George Washington";
    cout << k << ": " << id_to_name[k] << endl;
    return 0;
}
```




```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c id_map_0.cpp
$ g++ -o id_map_0 id_map_0.o
$ ./id_map_0
92394: George Washington
```

C++: map



Get number of keys:

```
id_to_name.size();
```

Check if map contains (has a value for) a given key:

```
if(id_to_name.find(92394) != id_to_name.end()) {
    cout << "Found it" << endl;
} else {
    cout << "Didn't find it" << endl;
}
```

C++: map

To visit all the elements of the map, use an *iterator*:

```
for(map<int, string>::iterator it = id_to_name.begin();
    it != id_to_name.end();
    ++it) {
    cout << "  " << it->first << ": " << it->second << endl;
}
```

Iterator type: `map<int, string>::iterator`

Loop is similar to the loop for `vector`

Iterator moves over the keys *in ascending order* (increasing id in this case)

C++: map

Looking at the body:

```
cout << "  " << it->first << ": " << it->second << endl;
```

Dereferenced map iterator type is `std::pair<key_type, value_type>`

- `it->first` is the key (`int` here)
- `it->second` is the value (`string` here)

C++: map

id_map.cpp:

```
#include <iostream>
#include <map>
#include <string>
using std::cout;   using std::endl;
using std::string; using std::map;

int main() {
    map<int, string> id_to_name;
    id_to_name[92394] = "Alex Hamilton";
    id_to_name[13522] = "Ben Franklin";
    id_to_name[42345] = "George Washington";
    cout << "size of id_to_name " << id_to_name.size() << endl;
    cout << "id_to_name[92394] " << id_to_name[92394] << endl;
    for(map<int, string>::iterator it = id_to_name.begin();
        it != id_to_name.end();
        ++it) {
        cout << "  " << it->first << ": " << it->second << endl;
    }
    return 0;
}
```

C++: map

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c id_map.cpp
$ g++ -o id_map id_map.o
$ ./id_map
size of id_to_name 3
id_to_name[92394] Alex Hamilton
  13522: Ben Franklin
  42345: George Washington
  92394: Alex Hamilton
```

Note again: the keys are printed *in ascending order*

Can use `reverse_iterator`, `.rbegin()` and `.rend()` to get keys in *descending* order

Functions that return multiple values

We want a function that takes integers `a` and `b` and returns both `a/b` (quotient) and `a%b` (remainder)

- `divmod(10, 5)` returns `2, 0`
- `divmod(10, 3)` returns `3, 1`

Functions only return *one* thing

Functions that return multiple values

One solution: pass-by-pointer arguments:

```
void divmod(int a, int b, int *quo, int *rem) {
    *quo = a / b;
    *rem = a % b;
}
```

Another: define `struct` for `divmod`'s return type:

```
struct quo_rem {
    int quotient;
    int remainder;
};

struct quo_rem divmod(int a, int b) { ... }
```

Functions that return multiple values

```
#include <iostream>

using std::cout;    using std::endl;

struct quo_rem {
    int quotient;
    int remainder;
};

quo_rem divmod(int a, int b) {
    quo_rem result = {a/b, a%b};
    return result;
}

int main() {
    quo_rem qr_10_5 = divmod(10, 5);
    quo_rem qr_10_3 = divmod(10, 3);
    cout << "10/5 quotient=" << qr_10_5.quotient
         << ", remainder=" << qr_10_5.remainder << endl;
    cout << "10/3 quotient=" << qr_10_3.quotient
         << ", remainder=" << qr_10_3.remainder << endl;
    return 0;
}
```

C++: `pair` to return multiple values

STL solution: return `pair<int, int>`, a pair with items of types `int` and `int`:

quo_rem_3.cpp:

```
#include <iostream>
#include <utility> // where pair and make_pair are defined

using std::pair;
using std::make_pair;
using std::cout;
using std::endl;

pair<int, int> divmod(int a, int b) {
    return make_pair(a/b, a%b);
}

int main() {
    pair<int, int> qr_10_5 = divmod(10, 5);
    pair<int, int> qr_10_3 = divmod(10, 3);
    cout << "10/5 quotient=" << qr_10_5.first
         << ", remainder=" << qr_10_5.second << endl;
    cout << "10/3 quotient=" << qr_10_3.first
         << ", remainder=" << qr_10_3.second << endl;
    return 0;
}
```

C++: pair

We've used `pair` already; dereferenced `map` iterator is a key, value pair:

```
for(map<int, string>::iterator it = id_to_name.begin();
    it != id_to_name.end();
    ++it) {
    cout << "  " << it->first << ": " << it->second << endl;
}
```

C++: `pair`

Relational operators for `pair` work as expected:

- Compares first field first...
- ...if there's a tie, compares second field

`make_pair(2, 3)` < `make_pair(3, 2)` is true

More on `pair`:

- www.cplusplus.com/reference/utility/pair/

C++: `typedef` with STL containers

Iterator type can be complex

- `map<int, string>::iterator` – iterator over a `map`
- `map<string, map<string, int>>::iterator` – iterator over a `map` *where the values are themselves maps*

`typedef` can help by:

- Reducing clutter
- Bringing related type declarations closer together in your code:

```
typedef map<int, string> TMap;      // map type
typedef TMap::iterator TMapItr;     // map iterator type
```

C++: Iterators

With `iterator` (or `reverse_iterator`) you can modify the data structure via the dereferenced iterator:

```
typedef vector<int>::iterator TItr;

void prefix_sum(TItr begin, TItr end) {
    int sum = 0;
    for(TItr it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

C++: Iterators

```cpp
#include <iostream>
#include <vector>
#include <string>
using std::cout;   using std::endl;
using std::vector; using std::string;

typedef vector<int>::iterator TItr;
typedef vector<int>::const_iterator TConstItr;

void prefix_sum(TItr begin, TItr end) {
    int sum = 0;
    for(TItr it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

```cpp
int main() {
    vector<int> ones = {1, 1, 1, 1};
    cout << "Before: ";
    for(TConstItr it = ones.cbegin();
        it != ones.cend(); ++it) {
        cout << *it << ' ';
    }
    prefix_sum(ones.begin(), ones.end());
    cout << endl << "After: ";
    for(TConstItr it = ones.cbegin();
        it != ones.cend(); ++it) {
        cout << *it << ' ';
    }
    return 0;
}
```

```
$ g++ -std=c++11 -pedantic -Wall -Wextra prefix_sum_iter.cpp
$ ./a.out
Before: 1 1 1 1
After: 1 2 4 8
```

C++: Iterators

`const_iterator` *does not* allow modifications

```
typedef vector<int>::const_iterator TItr;

void prefix_sum(TItr begin, TItr end) {
    int sum = 0;
    for(TItr it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

C++: Iterators

```
#include <iostream>
#include <vector>
#include <string>
using std::cout;   using std::endl;
using std::vector; using std::string;

typedef vector<int>::const_iterator TItr;

void prefix_sum(TItr begin, TItr end) {
    int sum = 0;
    for(TItr it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

```
int main() {
    vector<int> ones = {1, 1, 1, 1};
    cout << "Before: ";
    for(TItr it = ones.cbegin();
        it != ones.cend(); ++it) {
        cout << *it << ' ';
    }
    prefix_sum(ones.begin(), ones.end());
    cout << endl << "After: ";
    for(TItr it = ones.cbegin();
        it != ones.cend(); ++it) {
        cout << *it << ' ';
    }
    return 0;
}
```

C++: Iterators

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c prefix_sum_iter
prefix_sum_iter.cpp: In function ‘void prefix_sum(TItr, TIt
prefix_sum_iter.cpp:12:13: error: assignment of read-only l
   12 |             *it += sum;
      |             ~~~~^~~~~
```

C++: Iterators

Type	`++it`	`--it`	Get with	*it type
`iterator`	forward	back	`.begin()/end()`	-
`const_iterator`	forward	back	`.cbegin()/cend()`	`const`
`reverse_iterator`	back	forward	`.rbegin()/rend()`	-
`const_reverse_iterator`	back	forward	`.crbegin()/crend()`	`const`

C++: tuple

`tuple` is like `pair` but with as many fields as you like

```
#include <tuple>

using std::tuple; using std::make_tuple;

tuple<int, int, float> divmod(int a, int b) {
    return make_tuple(a/b, a%b, (float)a/b);
}
```

`get<N>(tup)` gets the Nth field of variable `tup`:

C++: tuple

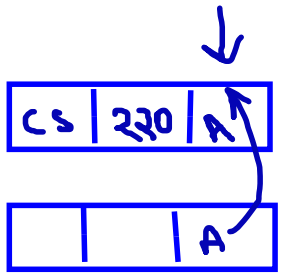
```cpp
#include <iostream>
#include <tuple>

using std::cout;  using std::endl;
using std::tuple; using std::make_tuple;
using std::get;

tuple<int, int, float> divmod(int a, int b) {
    return make_tuple(a/b, a%b, (float)a/b);
}

int main() {
    tuple<int, int, float> qr_10_3 = divmod(10, 3);
    cout << "10/3 quotient="      << get<0>(qr_10_3)
         << ", remainder="        << get<1>(qr_10_3)
         << ", decimal quotient=" << get<2>(qr_10_3) << endl;
    return 0;
}
```

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c quo_rem_tuple.cpp
$ g++ -o quo_rem_tuple quo_rem_tuple.o
$ ./quo_rem_tuple
10/3 quotient=3, remainder=1, decimal quotient=3.33333
```

quiz!

What output would be printed?

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

void main(void) {
  using myData = std::tuple<std::string, int , char>;
  myData data1 = std::make_tuple("CS", 220, 'B');
  myData data2 = std::make_tuple("CS", 220, 'A');
  if (data1 > data2) std::get<2>(data1) = std::get<2>(data2);
  std::cout << "I got " << std::get<2>(data1) << " in " << std::get<0>(data1)
    << std::get<1>(data1) << "!" << std::endl;

  return 0;
}
```



A. I got A in CS220!

B. I got B in CS220!

C. I got CS in 220A!

D. I got CS in 220B!

E. Compilation error!

C++: STL

Some STL classes (click for links):

- `array` – fixed-length array
- `vector` – dynamically-sized array
- `set` – set; an element can appear at most once
- `list` – linked list!
- `map` – associative list, i.e. dictionary
- `stack` – last-in first-out (LIFO)
- `deque` – double-ended queue, flexible combo of LIFO/FIFO
- `unordered_map` – another map, more like a hash table
- `pair` – pair template
- `tuple` – like pair, but can have >2 fields