**//pair1.cpp**

// make\_pair example

**#include** <utility> // std::pair

**#include** <iostream> // std::cout

**int** **main** () {

std::pair <**int**,**int**> first;

std::pair <**int**,**int**> second;

first= std::make\_pair (10,20);

second = std::make\_pair (10.5,'A'); // ok: implicit conversion from pair<double,char>

std::cout << "first: " << first.first << ", " << first.second << '\n';

std::cout << "second: " << second.first << ", " << second.second << '\n';

**return** 0;

}

**output**

first: 10, 20

second: 10, 65

//**map1.cpp**

**#include** <iostream>

**#include** <map>

**using** **namespace** std;

**int** **main** () {

map<string, **int**> myMap; // a (string,int) map

map<string, **int**>::iterator p; // an iterator to the map

myMap.insert(pair<string, **int**>("Rob", 28));// insert ("Rob",28)

myMap["Joe"] = 38; // insert("Joe",38)

myMap["Joe"] = 50; // change to ("Joe",50)

myMap["Sue"] = 75; // insert("Sue",75)

cout << "print after inserts" << **endl**;

**for** (p = myMap.begin(); p != myMap.end(); ++p) {// print all entries

cout << "(" << p->first << "," << p->second << ")\n";}

p = myMap.find("Joe"); // \*p = ("Joe",50)

myMap.erase(p); // remove ("Joe",50)

myMap.erase("Sue"); // remove ("Sue",75)

p = myMap.find("Joe");

**if** (p == myMap.end()) cout << "nonexistent\n"; // outputs: "nonexistent"

cout << "print after erases" << **endl**;

**for** (p = myMap.begin(); p != myMap.end(); ++p) {// print all entries

cout << "(" << p->first << "," << p->second << ")\n";

}

**return** 0;

}

**output**

print after inserts

(Joe,50)

(Rob,28)

(Sue,75)

nonexistent

print after erases

(Rob,28)

//**map2.cpp**

**#include** <iostream>

**#include** <map>

**using** **namespace** std;

**int** **main** ()

{

map<**char**,**int**> mymap;

// first insert function version (single parameter):

mymap.insert ( pair<**char**,**int**>('a',100) );

mymap.insert ( pair<**char**,**int**>('z',200) );

pair<map<**char**,**int**>::iterator,**bool**> ret;

ret = mymap.insert ( pair<**char**,**int**>('z',500) );

**if** (ret.second==**false**) {

cout << "element 'z' already existed";

cout << " with a value of " << ret.first->second << '\n';

}

// second insert function version (with hint position):

map<**char**,**int**>::iterator it = mymap.begin();

mymap.insert (it, pair<**char**,**int**>('b',300)); // max efficiency inserting

mymap.insert (it, pair<**char**,**int**>('c',400)); // no max efficiency inserting

// third insert function version (range insertion):

map<**char**,**int**> anothermap;

anothermap.insert(mymap.begin(),mymap.find('c'));

// showing contents:

cout << "mymap contains:\n";

**for** (it=mymap.begin(); it!=mymap.end(); ++it)

cout << it->first << " => " << it->second << '\n';

cout << "anothermap contains:\n";

**for** (it=anothermap.begin(); it!=anothermap.end(); ++it)

cout << it->first << " => " << it->second << '\n';

**return** 0;

}

**Output**

element 'z' already existed with a value of 200

mymap contains:

a => 100

b => 300

c => 400

z => 200

//**unordermap1.cpp**

*// unordered\_map::find*

*#include <iostream>*

*#include <string>*

*#include <unordered\_map>*

*int* main ()

{

std::unordered\_map<std::string,*double*> mymap = {

{"mom",5.4},

{"dad",6.1},

{"bro",5.9} };

std::string input;

std::cout << "who? ";

getline (std::cin,input);

std::unordered\_map<std::string,*double*>::const\_iterator got = mymap.find (input);

*if* ( got == mymap.end() )

std::cout << "not found";

*else*

std::cout << got->first << " is " << got->second;

std::cout << std::endl;

*return* 0;

}

**Output**

who? dad

dad is 6.1

## Code Fragment: Map

template <typename K, typename V>

class Map { // map interface

public:

class Entry; // a (key,value) pair

class Iterator; // an iterator (and position)

int size() const; // number of entries in the map

bool empty() const; // is the map empty?

Iterator find(const K& k) const; // find entry with key k

Iterator put(const K& k, const V& v); // insert/replace pair (k,v)

void erase(const K& k) // remove entry with key k

throw(NonexistentElement);

void erase(const Iterator& p); // erase entry at p

Iterator begin(); // iterator to first entry

Iterator end(); // iterator to end entry

};

## Code Fragment: Entry

template <typename K, typename V>

class Entry { // a (key, value) pair

public: // public functions

Entry(const K& k = K(), const V& v = V()) // constructor

: \_key(k), \_value(v) { }

const K& key() const { return \_key; } // get key

const V& value() const { return \_value; } // get value

void setKey(const K& k) { \_key = k; } // set key

void setValue(const V& v) { \_value = v; } // set value

private: // private data

K \_key; // key

V \_value; // value

};

## Code Fragment: Class

template <typename K, typename V, typename H>

class HashDict : public HashMap<K,V,H> {

public: // public functions

typedef typename HashMap<K,V,H>::Iterator Iterator;

typedef typename HashMap<K,V,H>::Entry Entry;

// ...insert Range class declaration here

public: // public functions

HashDict(int capacity = 100); // constructor

Range findAll(const K& k); // find all entries with k

Iterator insert(const K& k, const V& v); // insert pair (k,v)

};

## Code Fragment: FindAll

template <typename K, typename V, typename H> // find all entries with k

typename HashDict<K,V,H>::Range HashDict<K,V,H>::findAll(const K& k) {

Iterator b = finder(k); // look up k

Iterator p = b;

while (!endOfBkt(p) && (\*p).key() == (\*b).key()) { // find next unequal key

++p;

}

return Range(b, p); // return range of positions

}

## Code Fragment: Insert

template <typename K, typename V, typename H> // insert pair (k,v)

typename HashDict<K,V,H>::Iterator HashDict<K,V,H>::insert(const K& k, const V& v) {

Iterator p = finder(k); // find key

Iterator q = inserter(p, Entry(k, v)); // insert it here

return q; // return its position

}

## Code Fragment: Range

class Range { // an iterator range

private:

Iterator \_begin; // front of range

Iterator \_end; // end of range

public:

Range(const Iterator& b, const Iterator& e) // constructor

: \_begin(b), \_end(e) { }

Iterator& begin() { return \_begin; } // get beginning

Iterator& end() { return \_end; } // get end

};

## Code Fragment: Simple

template <typename K, typename V, typename H> // constructor

HashDict<K,V,H>::HashDict(int capacity) : HashMap<K,V,H>(capacity) { }

## Code Fragment: BeginEnd

template <typename K, typename V, typename H> // iterator to end

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::end()

{ return Iterator(B, B.end()); }

template <typename K, typename V, typename H> // iterator to front

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::begin() {

if (empty()) return end(); // emtpty - return end

BItor bkt = B.begin(); // else search for an entry

while (bkt->empty()) ++bkt; // find nonempty bucket

return Iterator(B, bkt, bkt->begin()); // return first of bucket

}

## Code Fragment: Class

template <typename K, typename V, typename H>

class HashMap {

public: // public types

typedef Entry<const K,V> Entry; // a (key,value) pair

class Iterator; // a iterator/position

public: // public functions

HashMap(int capacity = 100); // constructor

int size() const; // number of entries

bool empty() const; // is the map empty?

Iterator find(const K& k); // find entry with key k

Iterator put(const K& k, const V& v); // insert/replace (k,v)

void erase(const K& k); // remove entry with key k

void erase(const Iterator& p); // erase entry at p

Iterator begin(); // iterator to first entry

Iterator end(); // iterator to end entry

protected: // protected types

typedef std::list<Entry> Bucket; // a bucket of entries

typedef std::vector<Bucket> BktArray; // a bucket array

// ...insert HashMap utilities here

private:

int n; // number of entries

H hash; // the hash comparator

BktArray B; // bucket array

public: // public types

// ...insert Iterator class declaration here

};

## Code Fragment: Erase

template <typename K, typename V, typename H> // remove utility

void HashMap<K,V,H>::eraser(const Iterator& p) {

p.bkt->erase(p.ent); // remove entry from bucket

n--; // one fewer entry

}

template <typename K, typename V, typename H> // remove entry at p

void HashMap<K,V,H>::erase(const Iterator& p)

{ eraser(p); }

template <typename K, typename V, typename H> // remove entry with key k

void HashMap<K,V,H>::erase(const K& k) {

Iterator p = finder(k); // find k

if (endOfBkt(p)) // not found?

throw NonexistentElement("Erase of nonexistent"); // ...error

eraser(p); // remove it

}

## Code Fragment: Find

template <typename K, typename V, typename H> // find utility

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::finder(const K& k) {

int i = hash(k) % B.size(); // get hash index i

BItor bkt = B.begin() + i; // the ith bucket

Iterator p(B, bkt, bkt->begin()); // start of ith bucket

while (!endOfBkt(p) && (\*p).key() != k) // search for k

nextEntry(p);

return p; // return final position

}

template <typename K, typename V, typename H> // find key

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::find(const K& k) {

Iterator p = finder(k); // look for k

if (endOfBkt(p)) // didn't find it?

return end(); // return end iterator

else

return p; // return its position

}

## Code Fragment: IteratorClass

class Iterator { // an iterator (& position)

private:

EItor ent; // which entry

BItor bkt; // which bucket

const BktArray\* ba; // which bucket array

public:

Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())

: ent(q), bkt(b), ba(&a) { }

Entry& operator\*() const; // get entry

bool operator==(const Iterator& p) const; // are iterators equal?

Iterator& operator++(); // advance to next entry

friend class HashMap; // give HashMap access

};

## Code Fragment: IteratorEquality

template <typename K, typename V, typename H> // are iterators equal?

bool HashMap<K,V,H>::Iterator::operator==(const Iterator& p) const {

if (ba != p.ba || bkt != p.bkt) return false; // ba or bkt differ?

else if (bkt == ba->end()) return true; // both at the end?

else return (ent == p.ent); // else use entry to decide

}

## Code Fragment: IteratorIncrement

template <typename K, typename V, typename H> // advance to next entry

typename HashMap<K,V,H>::Iterator& HashMap<K,V,H>::Iterator::operator++() {

++ent; // next entry in bucket

if (endOfBkt(\*this)) { // at end of bucket?

++bkt; // go to next bucket

while (bkt != ba->end() && bkt->empty()) // find nonempty bucket

++bkt;

if (bkt == ba->end()) return \*this; // end of bucket array?

ent = bkt->begin(); // first nonempty entry

}

return \*this; // return self

}

## Code Fragment: IteratorStar1

template <typename K, typename V, typename H> // get entry

typename HashMap<K,V,H>::Entry&

HashMap<K,V,H>::Iterator::operator\*() const

{ return \*ent; }

## Code Fragment: IteratorStar2

template <typename K, typename V, typename H> // get entry

typename HashMap<K,V,H>::Entry& HashMap<K,V,H>::Iterator::operator\*() const

{ return \*ent; }

## Code Fragment: Put

template <typename K, typename V, typename H> // insert utility

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::inserter(const Iterator& p, const Entry& e) {

EItor ins = p.bkt->insert(p.ent, e); // insert before p

n++; // one more entry

return Iterator(B, p.bkt, ins); // return this position

}

template <typename K, typename V, typename H> // insert/replace (v,k)

typename HashMap<K,V,H>::Iterator HashMap<K,V,H>::put(const K& k, const V& v) {

Iterator p = finder(k); // search for k

if (endOfBkt(p)) { // k not found?

return inserter(p, Entry(k, v)); // insert at end of bucket

}

else { // found it?

p.ent->setValue(v); // replace value with v

return p; // return this position

}

}

## Code Fragment: Simple

template <typename K, typename V, typename H> // constructor

HashMap<K,V,H>::HashMap(int capacity) : n(0), B(capacity) { }

template <typename K, typename V, typename H> // number of entries

int HashMap<K,V,H>::size() const { return n; }

template <typename K, typename V, typename H> // is the map empty?

bool HashMap<K,V,H>::empty() const { return size() == 0; }

## Code Fragment: UtilitiesClass

Iterator finder(const K& k); // find utility

Iterator inserter(const Iterator& p, const Entry& e); // insert utility

void eraser(const Iterator& p); // remove utility

typedef typename BktArray::iterator BItor; // bucket iterator

typedef typename Bucket::iterator EItor; // entry iterator

static void nextEntry(Iterator& p) // bucket's next entry

{ ++p.ent; }

static bool endOfBkt(const Iterator& p) // end of bucket?

{ return p.ent == p.bkt->end(); }

## Code Fragment: Main

map<string, int> myMap; // a (string,int) map

map<string, int>::iterator p; // an iterator to the map

myMap.insert(pair<string, int>("Rob", 28)); // insert ("Rob",28)

myMap["Joe"] = 38; // insert("Joe",38)

myMap["Joe"] = 50; // change to ("Joe",50)

myMap["Sue"] = 75; // insert("Sue",75)

p = myMap.find("Joe"); // \*p = ("Joe",50)

myMap.erase(p); // remove ("Joe",50)

myMap.erase("Sue"); // remove ("Sue",75)

p = myMap.find("Joe");

if (p == myMap.end()) cout << "nonexistent\n"; // outputs: "nonexistent"

for (p = myMap.begin(); p != myMap.end(); ++p) { // print all entries

cout << "(" << p->first << "," << p->second << ")\n";

}