std::pair

Storing two pieces of different data

Nattee Niparnan

Basic

```cpp
#include <iostream>

int main() {
    std::pair<int,float> x;
    std::pair<int,float> y;

    x.first = 10;
    x.second = 20.65;

    std::cout << x.first << " "
    std::cout << x.second << std::endl;

    y = x;

    std::cout << y.first << " "
    std::cout << y.second + 20 << std::endl;
}
```
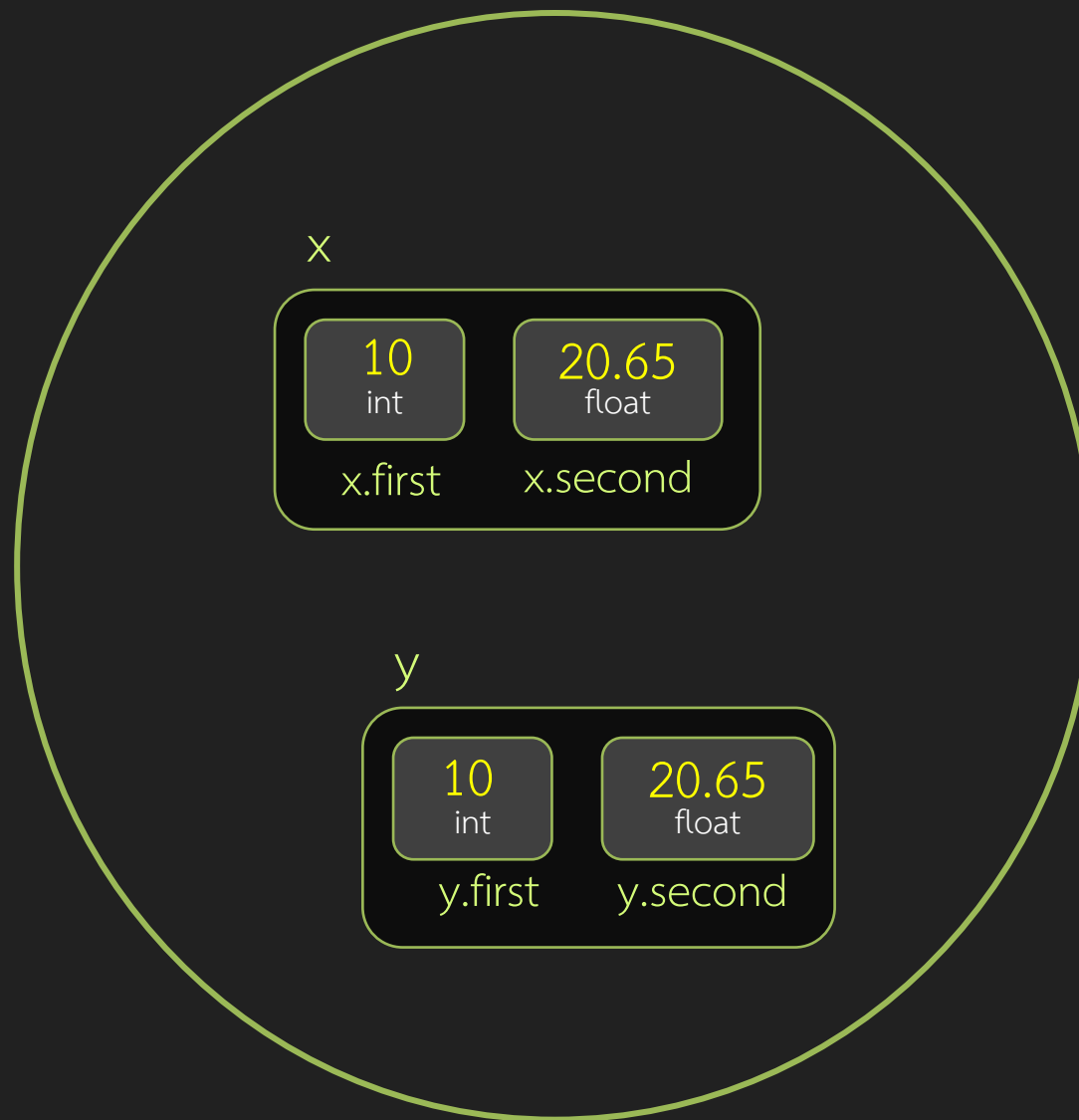
x

10 int	20.65 float
x.first	x.second

y

10 int	20.65 float
y.first	y.second

Initialize

```cpp
#include <iostream>

int main() {
    //default constructor
    std::pair<std::string,bool> p;
    std::cout << "default [" << p.first << "] [" << p.second << "]" << std::endl;

    //initialize by { }
    std::pair<std::string,bool> p1 = { "somchai", true };

    //create pair without specifying type by "make_pair"
    std::pair<bool,int> p2;
    p2 = std::make_pair(false,10);

    std::pair<bool,int> p3(p2);

    //more complex pair
    std::pair< std::pair<float, int>, std::string >  p4 = { {20.5, -3}, "abc"};
    std::cout << p4.first.first << " " << p4.first.second << " " << p4.second << std::endl;
}
```

Template

- Template allows "same code, different data type"

- std::pair is a "class template"
  - In generic term, pair is defined as
    - pair<T1, T2>
    - "first" member is of type T1
    - "second" member is of type T2

```
std::pair<int, float>
```

Class Template

What "first" shoul be

What "second" shoul be

- To use std::pair, we must supply "Type Information" to the template
  - What T1 and T2 should be.

STL and Namespace

- The "Fullname" of cout is

  std::cout

- std is a namespace

- We need to use fullname

  - Too lazy? use "using" keywords

```cpp
#include <iostream>

//this tell C++ that when we say cout, we mean std::cout
using std::cout;

int main() {
    //we still need to use std::endl
    cout << "Yes" << std::endl;
}
```

```cpp
#include <iostream>

//this tell C++ that when we say something
// that it does not understand, C++ should
// try to use std as its namespace

//this is VERY BAD PRACTICE in real world.
//10/10 not recommend
//.... but it's ok for this class
using namespace std;

int main() {
    cout << "Yes" << endl;
}
```

std::vector

A linear storage of a single data type

Nattee Niparnan

Basic

v2

v2[0] v2[1] v2[2]

| 2 | 3 | 4 |

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v1;
    cout << "Size of v1 is " << v1.size() << endl;

    vector<int> v2 = {2,3,4};

    cout << v2[1] << endl;
    v1 = v2;
    v1[0] = 20;

    cout << v1[0] << ", " << v1[1] << ", " << v1[2] << endl;

    v1.push_back(99);
    cout << v1.size() << endl;
}
```

- Vector start with empty element

- Use `size()` to get the number of element

- Use `empty()` to check if a vector has any element

```
Size of v1 is 0
v1 is empty
3
20, 3, 4
4
```

Initialize

- With specific size

- With specific size and starting value

```cpp
#include <iostream>
#include <vector>

using namespace std;

void print_vector(vector<float> v) {
    for (size_t i = 0;i < v.size(); i++) {
        cout << v[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<float> v1(10);
    print_vector(v1);

    vector<float> v2(5,3.55);
    print_vector(v2);

    vector<float> v3(v2);
    print_vector(v3);
}
```

Access

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<float> v1(2);
    vector<float> v2(2);

    cout << "-- v1 --" << endl;
    for (int i = 0;i < 7; i++) {
        v1[i] = i;
        cout << i << ": " << v1[i] << endl;
    }
    cout << "-- v2 --" << endl;
    for (int i = 0;i < 7; i++) {
        cout << i << ": " << v2[i] << endl;
    }
    //cout << "using at" << endl;
    //v2.at(1) = 99;
    //this will cause exception
    //for (int i = 0;i < 7; i++) {
    //    cout << i << ": " << v2.at(i) << endl;
    //}
}
```

- Operator [ ] won't check for 'out-of-range'
  - Reading, writing beyond size is undefined
  - Might crash
  - Grader will give 'x'
- at() method will check bound
  - But slower

Resizing

- Resize change the size
  - Enlarge will fill with default

```cpp
#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> v) {
    cout << "Size of V is " << v.size() << ":";
    for (int i = 0;i < v.size();i++) cout << v[i] << ", ";
    cout << endl;
}

int main() {
    vector<int> v(3,10);
    print(v);
    v.resize(6);
    print(v);
    v.resize(1);
    print(v);
    v.resize(7);
    print(v);
}
```

```
Size of V is 3:10,10,10,
Size of V is 6:10,10,10,0,0,0,
Size of V is 1:10,
Size of V is 7:10,0,0,0,0,0,0,
```

Modify

- pop_back
  - Erase last element

- insert(position, value)

- erase(position)

- Careful!
  - Both insert and erase position must be valid
  - If not valid, it can crash

```cpp
#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> v) {
    cout << "Size of V is " << v.size() << ": ";
    for (int i = 0;i < v.size();i++) cout << v[i] << ", ";
    cout << endl;
}


int main() {
    vector<int> v(3,8);
    v.insert( v.begin(), 1);
    v.insert( v.begin()+3, 2);
    v.insert( v.end(), 3);
    print(v);
    v.erase(v.begin());
    v.erase(v.begin()+2);
    //print(v);
    //v.pop_back();
    //print(v);
}
```

```
Size of V is 6: 1, 8, 8, 2, 8, 3,
```

Functions that work with vector

- sort

- find

- min_element

- max_element

- lower_bound

- upper_bound

Need to use **iterator**

Find

- find(a, b, c)
  - a and b are position (iterator)
  - find c from a to before b
  - If not found, return b

- Must #include <algorithm>

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    vector<int> v = {9,-1,3,7,5,2,1,4};

    int x = 5;
    if ( find(v.begin(), v.end(), x) != v.end() ) {
        cout << "found" << endl;
    } else {
        cout << "not found" << endl;
    }

    if (find(v.begin(), v.begin()+3, 3) != v.begin()+3) cout << "FOUND" << endl;

    //how many "YES" will be printed? (CHEAT QUESTION)
    //if (find(v.begin()  , v.begin()+2, x) != v.end()) cout << "YES" << endl;
    //if (find(v.begin()  , v.begin()+4, x) != v.end()) cout << "YES" << endl;
    //if (find(v.begin()+4, v.begin()+2, x) != v.end()) cout << "YES" << endl;
    //if (find(v.begin()+4, v.begin()+8, x) != v.end()) cout << "YES" << endl;
}
```

Sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(vector<int> v) {
    cout << "Size of V is " << v.size() << ": ";
    for (int i = 0;i < v.size();i++) cout << v[i] << ", ";
    cout << endl;
}

int main() {
    vector<int> v = {9,-1,3,7,5,2,1,4};

    print(v);
    sort(v.begin()+2, v.begin()+6);
    print(v);
}
```

- sort(a,b)
  - sort everything from a to before b

- Must #include <algorithm>

Vector iterator

Iterator

- Iterator is a pointer to position

- First element is begin()

- The one after the last element is end()

- For insert, valid position is from begin() to end(), inclusive

- For erase, valid position is from begin() up to before end()

- Different vector, different iterator
  - v1.end() is not the same as v2.end()

Iterator arithmetic

- We can add by integer to an iterator

- We can subtract two iterators of the same type

- We can use increment (**++**) and decrement (**--**)

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int>   v1 = {0, 10, 20 ,30, 40, 50, 60, 70, 80};
    vector<float> v2 = {0.2, -4, 0.13, 3.14, 2.73};

    vector<int>::iterator it1 = v1.begin() + 3;
    vector<float>::iterator it2 = v2.end();

    //getting value at iterator by using "*" operator
    cout << *it1 << endl;
    cout << *(it2-1) << endl;
    cout << *it1+2 << endl;

    //iterator arithmetics
    vector<int>::iterator it3 = it1 + 2;
    cout << "data at it3: " << *it3 << endl;
    cout << "different of it3,it1: " << (it3 - it1) << endl;

    vector<float>::iterator it4 = v2.end();
    it4--;
    cout << "data at it4: " << *it4 << endl;

    //this cannot be done
    //cout << (it2 - it1) << endl;
}
```

Iterate all elements by iterator

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int>   v1 = {0, 10, 20 ,30, 40, 50, 60, 70, 80};
  vector<float> v2 = {0.2, -4, 0.13, 3.14, 2.73};

  cout << "----v1----" << endl;
  auto it = v1.begin();
  while (it < v1.end() ) {
    cout << it - v1.begin() << ": " << *it << endl;
    it++;
  }

  cout << "----v2----" << endl;
  for (auto it = v2.begin(); it < v2.end(); it++) {
    cout << it - v2.begin() << ": " << *it << endl;
  }
}
```

- We can compare iterator
  - Beware!! Iterator of some data structure does not support > or < comparator
  - But for vector, its' ok
- We can use auto keyword to automatically define a type of a variable

Some functions that use iterators

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int>   v1 = {3,0,1,2,4,-3,9,8};
  vector<float> v2 = {10.2, -4, 0.13, 3.14, 2.73};

  auto it1 = min_element(v1.begin(),v1.end());
  auto it2 = max_element(v2.begin()+2,v2.end());

  cout << *it1 << endl;
  cout << *it2 << endl;
}
```

- min_element and max_element get the iterator of the minimum (and maximum) element.

New idiom that iterates all elements

- Shorter syntax for-loop

- Called range-based for loop

- Can use reference operator (&)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

int main() {
  vector<string> v1 = {"somchai", "somying", "somsak"};

  //range-based for loop
  for (string x : v1) {
    // x is a copy of each element in v1
    cout << x << ", ";
  }
  cout << endl;

  //using reference
  // x is THE element of v1, meaning we can modify it
  for (auto &x : v1) { x.replace(0,4,"--"); }
  for (auto &x : v1) { cout << x << " ";}
  cout << endl;
}
```

Iterator Invalidation

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v1 = {10,20};

  auto it = v1.end()-1;  // this point to 20
  // we resize (enlarge) v1, now [it] is invalidated
  v1.resize(10);

  //this might not crash
  //but it actually points to somewhere not in v1
  cout << *it << endl;

  //this will crash the program
  v1.insert(it,99);
  for (auto &x: v1) {cout << x << " ";}

}
```

- Some operation on the data structure invalidate existing iterators

- When an iterator is invalidated, it must not be used

- For vector, this include any addition and deletion of element (including resize)

- Another example: when we call v.erase(it), the iterator it is invalidated, because we have already deleted it.