CP::pair

Create our own data structure

Nattee Niparnan

Intro

- We start writing our own data structure
  - In our own namespace (CP)
- We start with a pair
  - Good introduction on writing a class
  - Some C++ feature we need to use
    - const
    - pass-by-value, pass-by-ref
    - Header file

What should we write in a class

- Class name and namespace
- Member variables which define what data we want to store in the class
- Member functions which define behavior of the class and how member variable is manipulated
  - Include lots of special function such as constructor, destructor and operator overloading

CP::pair version 0.1 (minimal version)

- Templating
  - Parameter of a code
- No member functions

```
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;
  };
}
```

When we declare a pair, T1 and T2 must be declared as a type

```
int main() {
  CP::pair<int,string> p1;
  CP::pair<bool,int> p2;
}
```

In p1, T1 is int, T2 is string

In p2, T1 is bool, T2 is int

Capabilities

- Can hold 2 pieces of data
- Can use template
- Works with operator=
- Has copy constructor

- Cannot check equal
- Cannot check less than

```cpp
#include <iostream>
#include <string>
using namespace std;
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;
  };
}

int main() {
  CP::pair<int,string> p1, p2;  //default ctor
  p1.first = 20; p1.second = "somchai";
  CP::pair<int,string> a(p1);   //copy ctor
  p2 = p1;

  cout << p2.first << "," << p2.second << endl;

  /*
  if (p1 == p2) { //won't compile
    cout << "yes" << endl;
  }

  if (p1 < a) { //won't compile
    cout << "yes" << endl;
  }
  */
}
```

```cpp
// copy constructor
pair(const pair<T1,T2>& a) {
  first = a.first;
  second = a.second;
  cout << "copy" << endl;

}
```

CP::pair version 0.2 (comparator overload)

```cpp
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;
      //----------------- operator -----------------
      bool operator==(const pair<T1,T2> &other) {
        return (first == other.first && second == other.second);
      }

      bool operator<(const pair<T1,T2>& other) const {
        return ((first < other.first) ||
                (first == other.first && second < other.second));

      }
  };
}
```

operator< must be const

```cpp
int main() {
  CP::pair<int,string> p1, p2;  //default ctor
  p1.first = 20; p1.second = "somchai";
  CP::pair<int,string> a(p1);   //copy ctor
  p2 = p1;
  cout << p2.first << "," << p2.second << endl;

  if (p1 == p2) { cout << "yes" << endl; }
  if (p1 < a) { cout << "yes" << endl;  }
  set<CP::pair<int,int>> s;
  s.insert( {1,2} );
  cout << s.begin()->second << endl;

}
```

```
20,somchai
yes
2
```

Now, we can compare and use less<pair<T1,T2>> (and also set, map, priority queue)

const parameter in function

```
void test(int &x, const int& y) {
  x++;                // Okay; doable
  cout << y << endl; // Okay, we only read y
  for (int i = 0;i < y;i++)  { // Okay, too
    cout << i << endl;
  }
  y--; // ERROR: we promise NOT to change Y
}
```

- Declared by putting a keyword const as a prefix
- Const parameter must not be modified inside the function
- Why? So that we know that the function does not modify the data
  - Especially when we pass-by-reference

Recall pass-by-reference && pass-by-value

- Taken from the first slide (c++-intro.pdf)
- Argument = things we give to a function
- Parameter = variables inside a function

```
void pass_by_value(int x) {
  cout << “X is" << x << endl;
  x = 30;
}

void pass_by_reference(int &x) {
  cout << “X is" << x << endl;
  x = 40;
}
```

```
int main() {
 cout << "Pass by Value, direct" << endl;
 pass_by_value(10);
 cout << endl;

 int x = 20;
 cout << "Pass by value, variable" << endl;
 pass_by_value(x);
 cout << "outside PbR function x = " << x << endl;
 cout << endl;

 cout << "Pass by reference" << endl;
 pass_by_reference(x);
 cout << "outside PbR function x = " << x << endl;

 //the following line cannot be compiled
 //because we need reference
 //pass_by_reference(20);
}
```

Difference

Pass-by-value

- The arguments can be either constants or variables
- Modifying parameters (variable inside the function) does not change the argument's value
- The argument's value is copied to the parameter
  - SLOWER!!! Because we have to copy

Pass-by-reference

- The arguments must be variables
  - Except when we also use const
- The parameters are the argument's variables
  - Modify the parameter also modify the argument's variable
- Faster

const member function

```
class ccc {
public:
  int a,b;

  void inspect() const { // This function promises NOT to change anything
    if (a < b) cout << "yes" << endl; // Okay

    // b += 20;   // <--- NOT OKAY
  }

  void mutate() { // This function might change something
    if (a < b) a += 10; // Okay
  }
};

void test2(ccc& changeable, const ccc& unchangeable)
{
  changeable.inspect();   // Okay: doesn't change a changeable object
  changeable.mutate();    // Okay: changes a changeable object
  unchangeable.inspect(); // Okay: doesn't change an unchangeable object
  unchangeable.mutate();  // ERROR: attempt to change unchangeable object
}
```

- Declared by putting a keyword const after the function declaration
- Const member function cannot modify any member data
  - Also cannot call any other member function that is not const
- Why? So that we know that the function does not modify the data

Custom Constructor

- In STL spec of std::pair, it has custom constructor called initialization constructor

```
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;

      // custom constructor
      pair(const T1 &a,const T2 &b) {
        first = a;
        second = b;
      }
      //---------------- operator ----------------
      bool operator==(const pair<T1,T2> &other) {…}
      bool operator<(const pair<T1,T2>& other) const {…}
  };
}
```

```
int main() {
  CP::pair<int, bool> p(10,false);
  CP::pair<string, int> q("abc",42), r("",0);

  cout << (q < r) << endl;
  priority_queue<CP::pair<string,int>> pq;
  pq.push(r);
  pq.push(q);
  cout << pq.top().first << endl;

  CP::pair<string, int> x(q);
  CP::pair<string, int> y = x;

  //-- all below cannot be compiled --
  //CP::pair<string, int> w;
  //vector<CP::pair<int,int>> v(10);
}
```

```
0
abc
```

When we have a constructor, a default constructor is not auto-generated

Initialization List

- Instead of writing a code to assign a value to each member, we can use initialization list
  - A little bit shorter code
  - Also little bit faster
  - Only way to init const member

```
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;

      // custom constructor, using initializer list
      pair(const T1 &a, const T2 &b) : first(a), second (b) { }

}
```

Default Constructor

- A constructor is used when we simply declare an object
- Auto-generated as initialization of all members with its default constructor
- Won't be auto-gen if we have any other constructor

```
namespace CP {
  template <typename T1,typename T2>
  class pair{
    public:
      T1 first;
      T2 second;

      // --------- constructor ------------
      pair() : first(), second() { }
      pair(const T1 &a, const T2 &b) : first(a), second (b) { }
      //----------------- operator -----------------
      //...
  };
}
```

```
int main() {
  CP::pair<int, bool> p(10,false);
  CP::pair<string, int> q("abc",42), r("",0);

  cout << (q < r) << endl;
  priority_queue<CP::pair<string,int>> pq;
  pq.push(r);
  pq.push(q);
  cout << pq.top().first << endl;

  CP::pair<string, int> x(q);
  CP::pair<string, int> y = x;

  //-- all below Okay now  --
  CP::pair<string, int> w;
  vector<CP::pair<int,int>> v(10);
  for (auto &x: v) { cout << x.first << endl;}
}
```

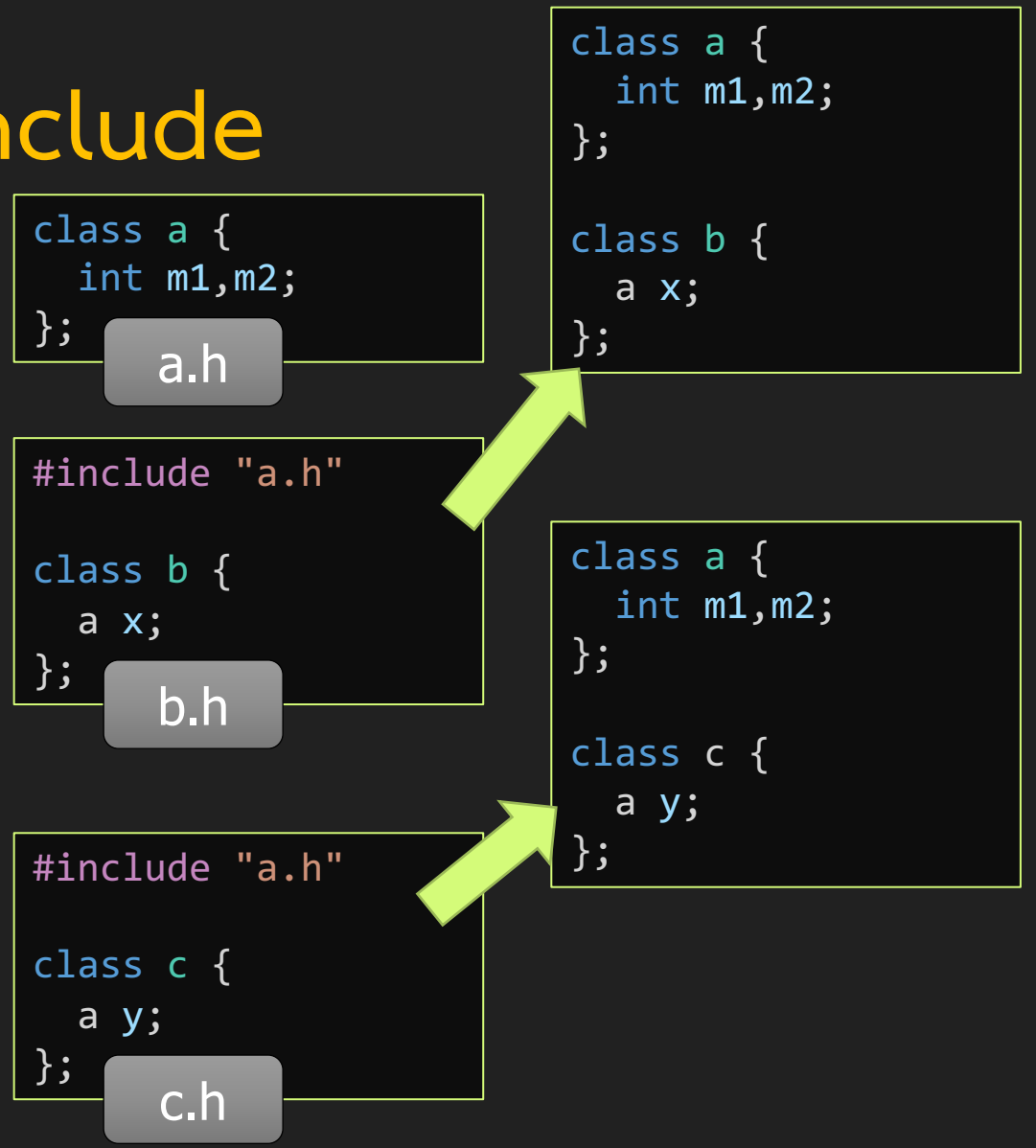
```
// default constructor
pair() {
  first = T1();
  second = T2();
}
...
```

Include File

- Usually, our data structure (which is written as a class) will be used by several programs in several files.
  - It is better NOT TO copy our data structure code into each each file
- Rather, put our code in a file and include it where it is needed
  - Better if we want to change it
  - Better compilation
- Introducing “.h” files

C++ header file (.h) and #include

- To put a content of one file into another file, we use `#include “filename”` keyword
- C++ will simply put the content of filename into where we #include it
- #includes has more benefit
  - separation of declaration (what it is) and definition (how it works)
  - Not really explored in this class
- We usually use .h for a file that we will include but this is not a rule, we can use other extension



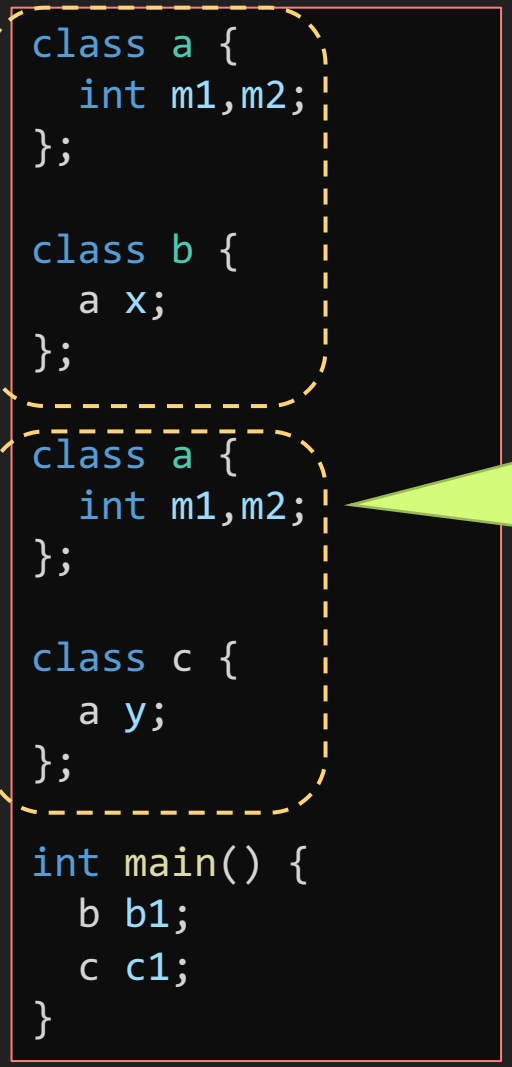
Problem with include

```
class a {
  int m1,m2;
};
```

```
#include "a.h"

class b {
  a x;
};
```

```
#include "a.h"

class c {
  a y;
};
```

```
#include "b.h"
#include "c.h"

int main() {
  b b1;
  c c1;
}
```



There is multiple copy of **class a**

Problem with include

```
#ifndef A_H
#define A_H
class a {
  int m1,m2;
};
#endif
```

```
#include "a.h"

class b {
  a x;
};
```

```
#include "a.h"

class c {
  a y;
};
```

```
#include "b.h"
#include "c.h"

int main() {
  b b1;
  c c1;
}
```

```
#ifndef A_H
#define A_H
class a {
  int m1,m2;
};
#endif

class b {
  a x;
};
```

```
#ifndef A_H
#define A_H
class a {
  int m1,m2;
};
#endif

class c {
  a y;
};
```

```
int main() {
  b b1;
  c c1;
}
```

class a here is taken out of compilation, because A_H is defined

Final Version

```cpp
#ifndef _CP_PAIR_INCLUDED_
#define _CP_PAIR_INCLUDED_

#include <iostream>

namespace CP {
  template <typename T1,typename T2>
  class pair  {
    public:
      T1  first;
      T2  second;

      // default constructor, using list initialize
      pair() : first(), second() {}

      // custom constructor, using list initialize
      pair(const T1 &a,const T2 &b) : first(a), second(b) { }

      // equality operator
      bool operator==(const pair<T1,T2> &other) {
        return (first == other.first && second == other.second);
      }
      // comparison operator
      bool operator<(const pair<T1,T2>& other) const {
        return ((first < other.first) ||
                (first == other.first && second < other.second));
      }
  };
}
#endif
```

Summary

Major

- Templating: allow use to write a code that works with different data type
- Constructor: a function called when we create a variable
  - Default constructor
- Operator overloading for less-than and equality
- pass-by-ref vs pass-by-value

Minor

- Header file
- const
- List initialization

Exercise (no grader)

1. Modify operator< so that it compare second before first
2. Modify operator< so that when we call sort(v.begin(), v.end()) where w is a vector of our pair, it is sorted from Max to Min
3. Write operator!= and operator>=