CS 32 Week 2 Discussion 1B

Yiyou Chen / Ian Galvez

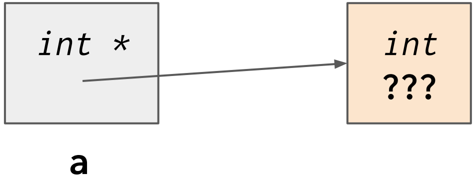
Topics

- Dynamically allocated Array, pointer to an array v.s. Array of pointers.
- Copy Constructor and Assignment operator
- Overload operators

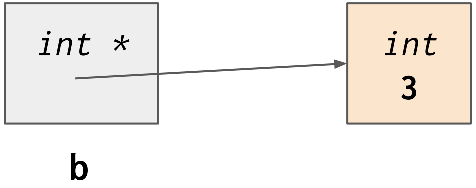
Allocate and deallocate for a pointer

Create a pointer that points to dynamically allocated memory: use **new**

```
int *a = new int;
```

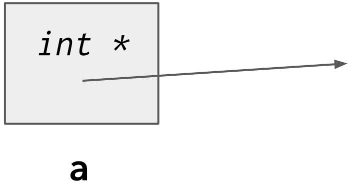


```
int *b = new int(3);
```

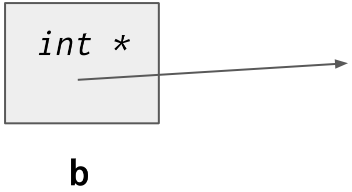


Free a pointer and deallocate the space: use **delete**

```
delete a;
```



```
delete b;
```

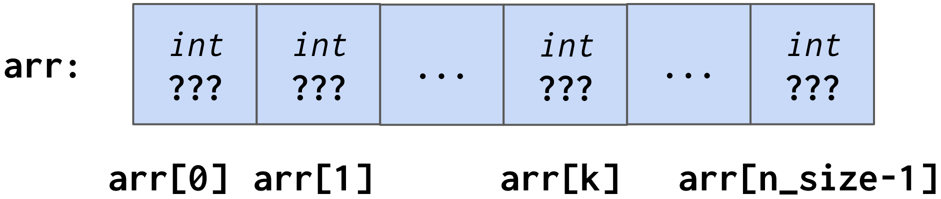


Fixed-size array

Create an array: int arr[n_size];

Access kth index: arr[k];

```
int arr[n_size];
```



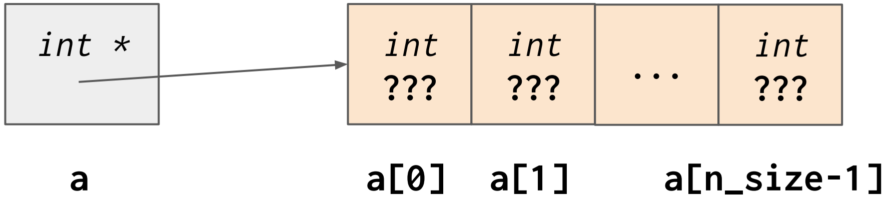
Dynamically allocate and deallocate an array

Dynamically allocate an array: `int *a = new int[n_size];`

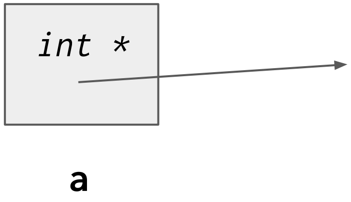
Dynamically deallocate an array: `delete [] a;`

Access kth index: `a[k] or *(a+k);`

`int *a = new int[n_size];`



`delete [] a;`

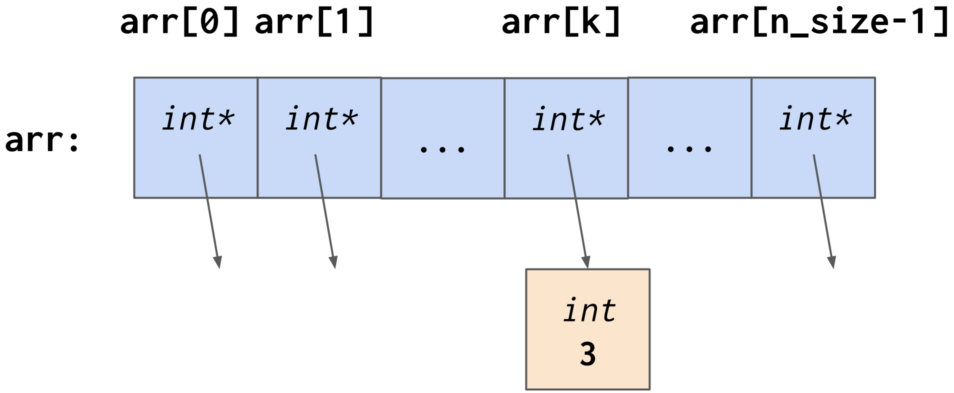


Array of pointers

Create array of pointers: `int *arr[n_size];`

Allocate space for kth index: `arr[k] = new int(3);`

Deallocate the memory: `for (int i = 0; i < n_size; ++i) delete arr[i];`



Default Copy Constructor and Assignment Operator

Copy constructor usage:

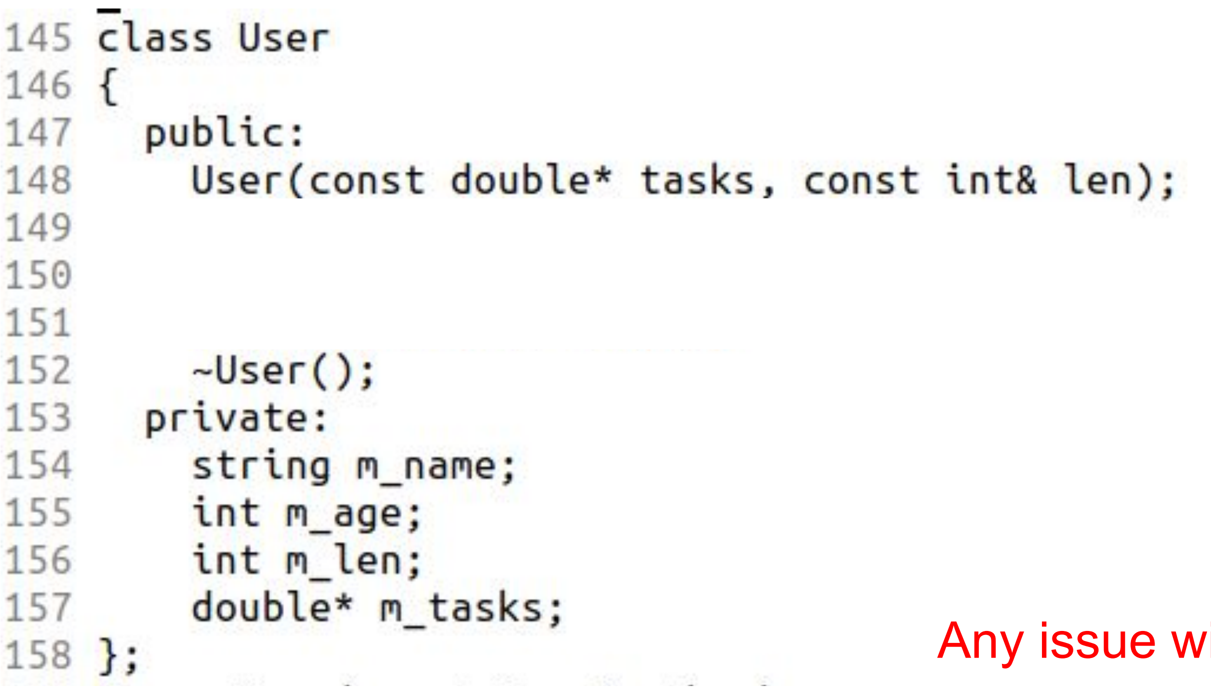
`class_type a = b;` or `class_type a(b);`

Assignment operator usage:

```
a = b; // class_type a, b
```

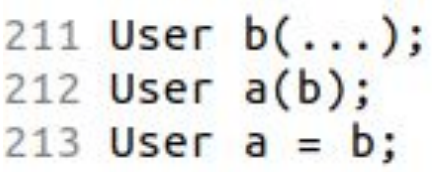
The default ones copy (assign) each class member variables one by one.

Default Copy Constructor



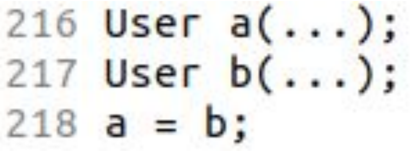
```
class User
{
  public:
    User(const double* tasks, const int& len);



    ~User();
  private:
    string m_name;
    int m_age;
    int m_len;
    double* m_tasks;
};
```

Copy constructor:



```
User b(...);
User a(b);
User a = b;
```

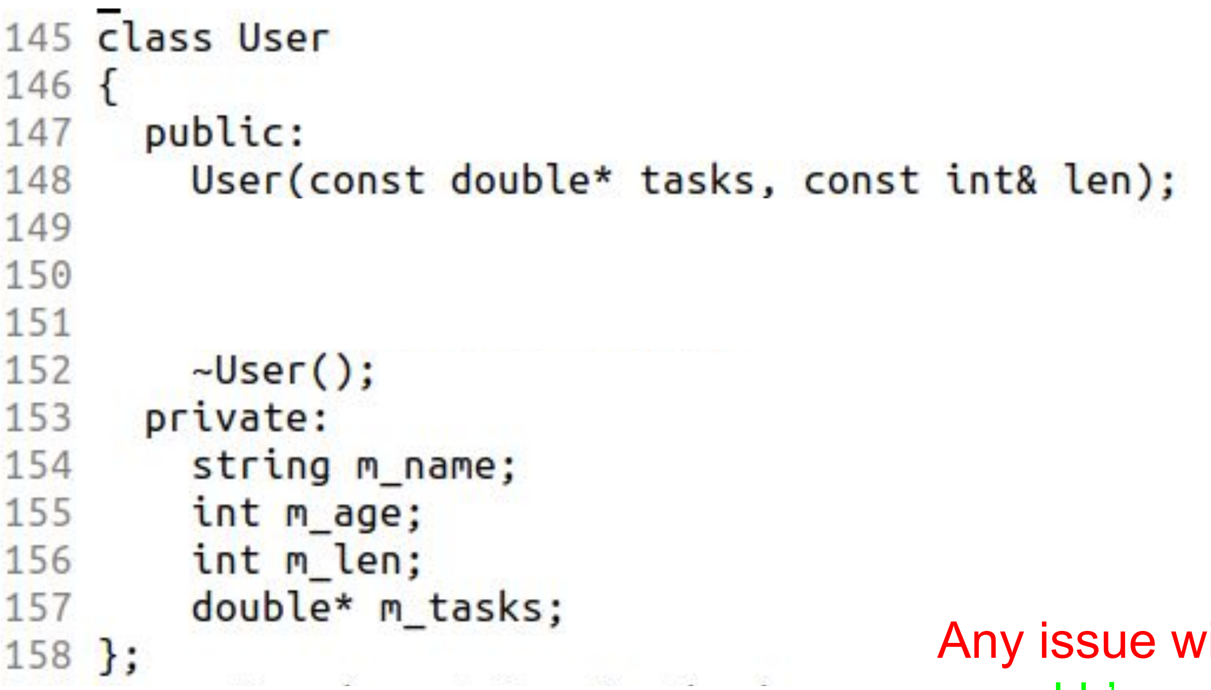
Assignment operator:



```
User a(...);
User b(...);
a = b;
```

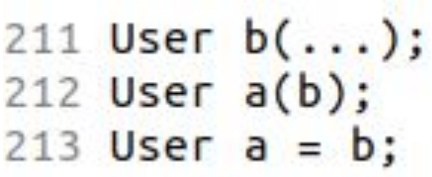
Any issue with the default ones?

Default Copy Constructor



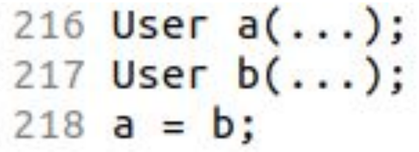
```
class User
{
  public:
    User(const double* tasks, const int& len);



    ~User();
  private:
    string m_name;
    int m_age;
    int m_len;
    double* m_tasks;
};
```

Copy constructor:



```
User b(...);
User a(b);
User a = b;
```

Assignment operator:

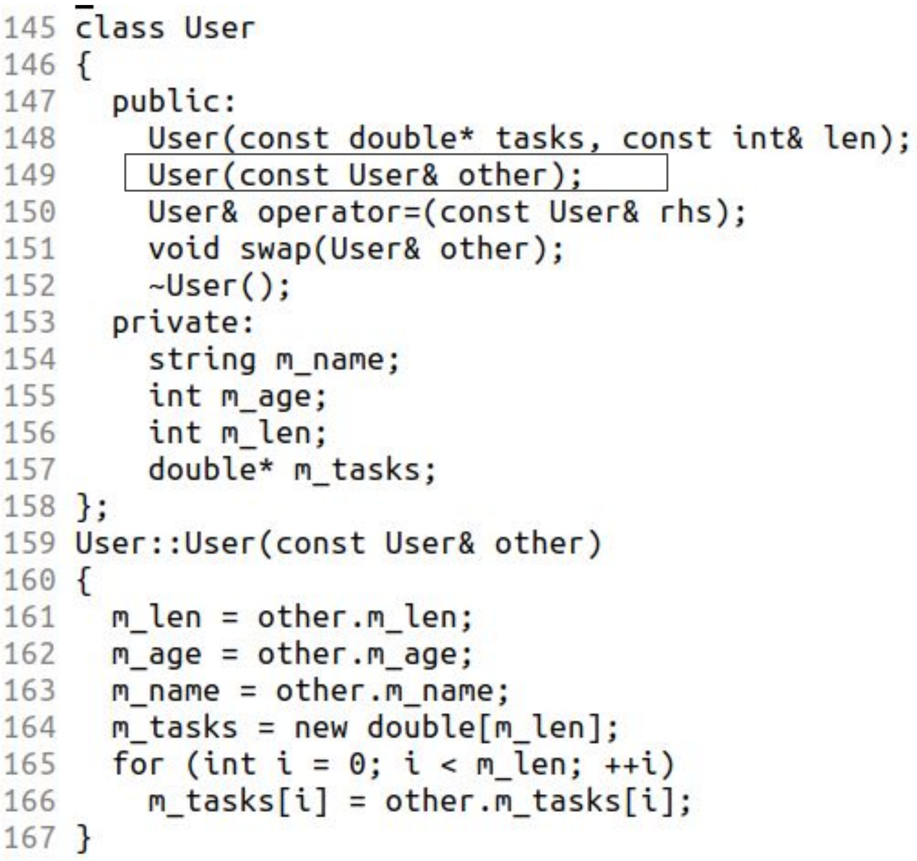


```
User a(...);
User b(...);
a = b;
```

Any issue with the default ones?

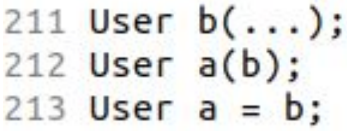
a and b’s m_tasks will point to the same location in memory. Share of memory.

Copy Constructor



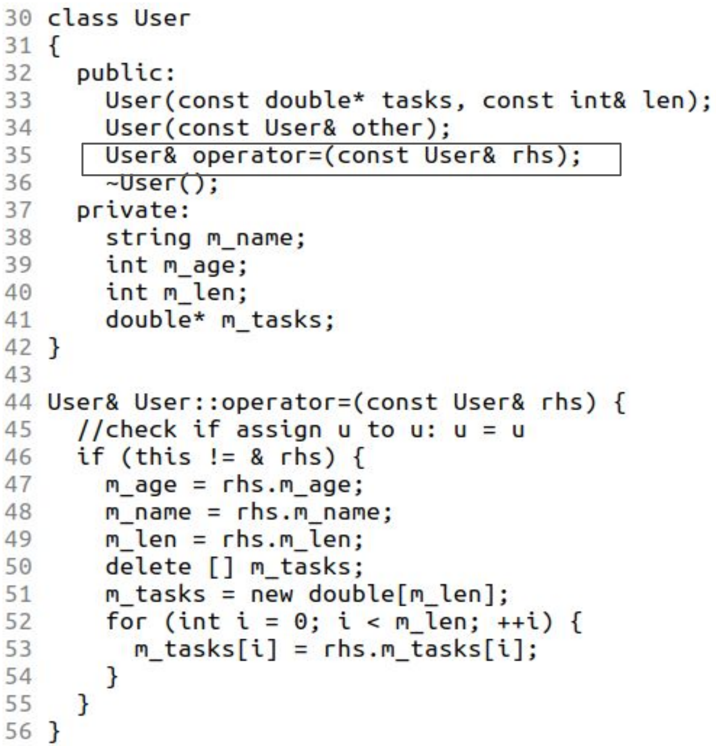
```
class User
{
  public:
    User(const double* tasks, const int& len);
    User(const User& other);
    User& operator=(const User& rhs);
    void swap(User& other);
    ~User();
  private:
    string m_name;
    int m_age;
    int m_len;
    double* m_tasks;
};
User::User(const User& other)
{
  m_len = other.m_len;
  m_age = other.m_age;
  m_name = other.m_name;
  m_tasks = new double[m_len];
  for (int i = 0; i < m_len; ++i)
    m_tasks[i] = other.m_tasks[i];
}
```

Most of the time, a copy constructor passes by constant reference. (Guarantees you can’t modify what’s being passed in)



```
User b(...);
User a(b);
User a = b;
```

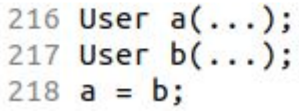
Assignment Operator



```
class User
{
  public:
    User(const double* tasks, const int& len);
    User(const User& other);
    User& operator=(const User& rhs);
    ~User();
  private:
    string m_name;
    int m_age;
    int m_len;
    double* m_tasks;
}

User& User::operator=(const User& rhs) {
  //check if assign u to u: u = u
  if (this != & rhs) {
    m_age = rhs.m_age;
    m_name = rhs.m_name;
    m_len = rhs.m_len;
    delete [] m_tasks;
    m_tasks = new double[m_len];
    for (int i = 0; i < m_len; ++i) {
      m_tasks[i] = rhs.m_tasks[i];
    }
  }
}
```

Aliasing: two different variables have the same reference (e.g. `u = u`). Always be cautious of aliasing!

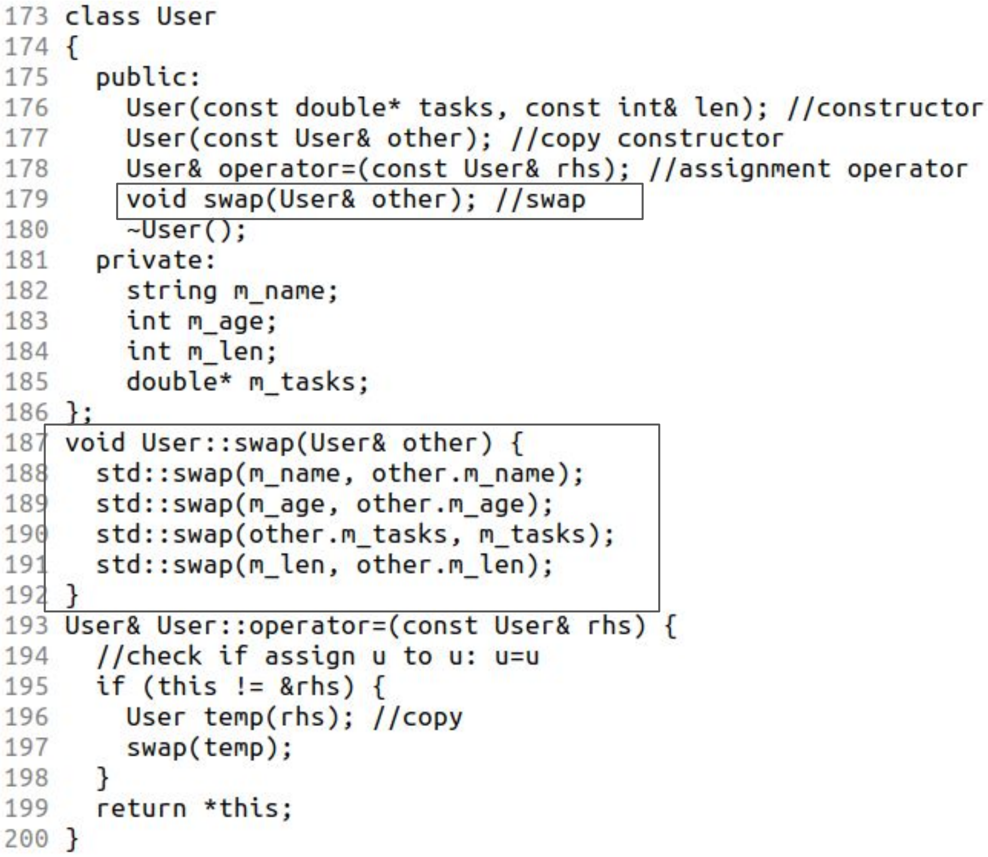
This is the traditional way. Not widely used. ***Why?***



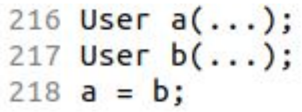
```
User a(...);
User b(...);
a = b;
```

Assignment Operator

This is the modern way to assign.
It makes sure there's enough resource for assignment first, by trying to create a copy of `rhs` (which we call `temp`).
If it succeeds, it swaps `temp` with `this`, i.e. `this` is now `temp`.



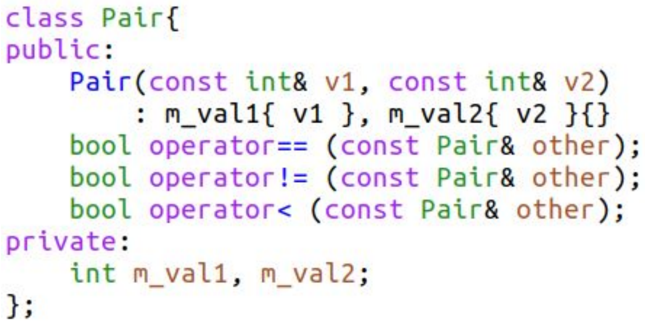
```cpp
class User
{
  public:
    User(const double* tasks, const int& len); //constructor
    User(const User& other); //copy constructor
    User& operator=(const User& rhs); //assignment operator
    void swap(User& other); //swap
    ~User();
  private:
    string m_name;
    int m_age;
    int m_len;
    double* m_tasks;
};
void User::swap(User& other) {
  std::swap(m_name, other.m_name);
  std::swap(m_age, other.m_age);
  std::swap(other.m_tasks, m_tasks);
  std::swap(m_len, other.m_len);
}
User& User::operator=(const User& rhs) {
  //check if assign u to u: u=u
  if (this != &rhs) {
    User temp(rhs); //copy
    swap(temp);
  }
  return *this;
}
```



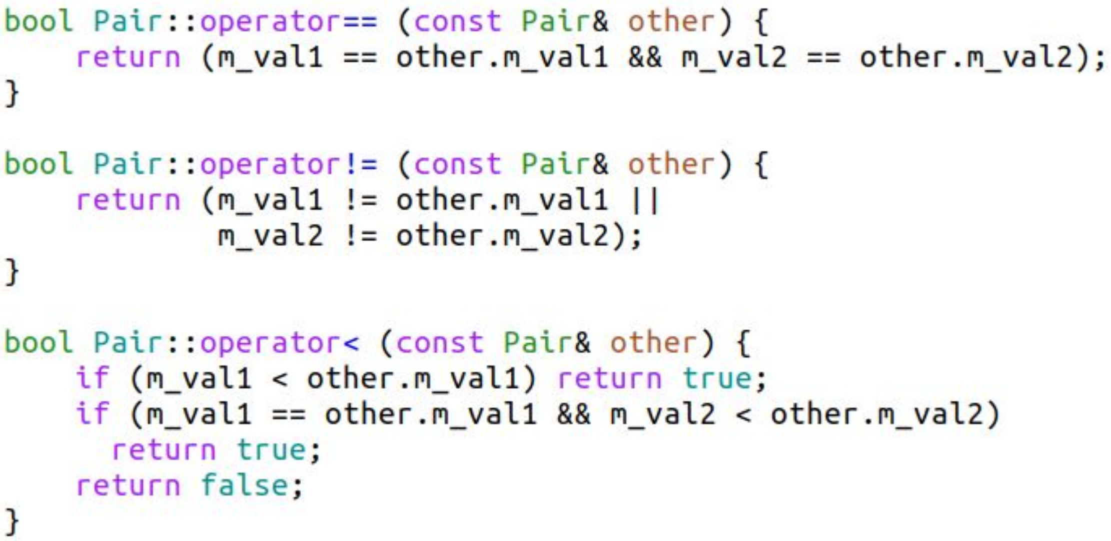
```cpp
User a(...);
User b(...);
a = b;
```

Overload Operators

We have shown an example to define (overload) assignment operators. Indeed, we can overload other operators as well. For example, we can overload the comparison operators (`==`, `!=`, `<`, etc.)



```
class Pair{
public:
    Pair(const int& v1, const int& v2)
        : m_val1{ v1 }, m_val2{ v2 }{}
    bool operator== (const Pair& other);
    bool operator!= (const Pair& other);
    bool operator< (const Pair& other);
private:
    int m_val1, m_val2;
};
```



```
bool Pair::operator== (const Pair& other) {
    return (m_val1 == other.m_val1 && m_val2 == other.m_val2);
}

bool Pair::operator!= (const Pair& other) {
    return (m_val1 != other.m_val1 ||
            m_val2 != other.m_val2);
}

bool Pair::operator< (const Pair& other) {
    if (m_val1 < other.m_val1) return true;
    if (m_val1 == other.m_val1 && m_val2 < other.m_val2)
      return true;
    return false;
}
```