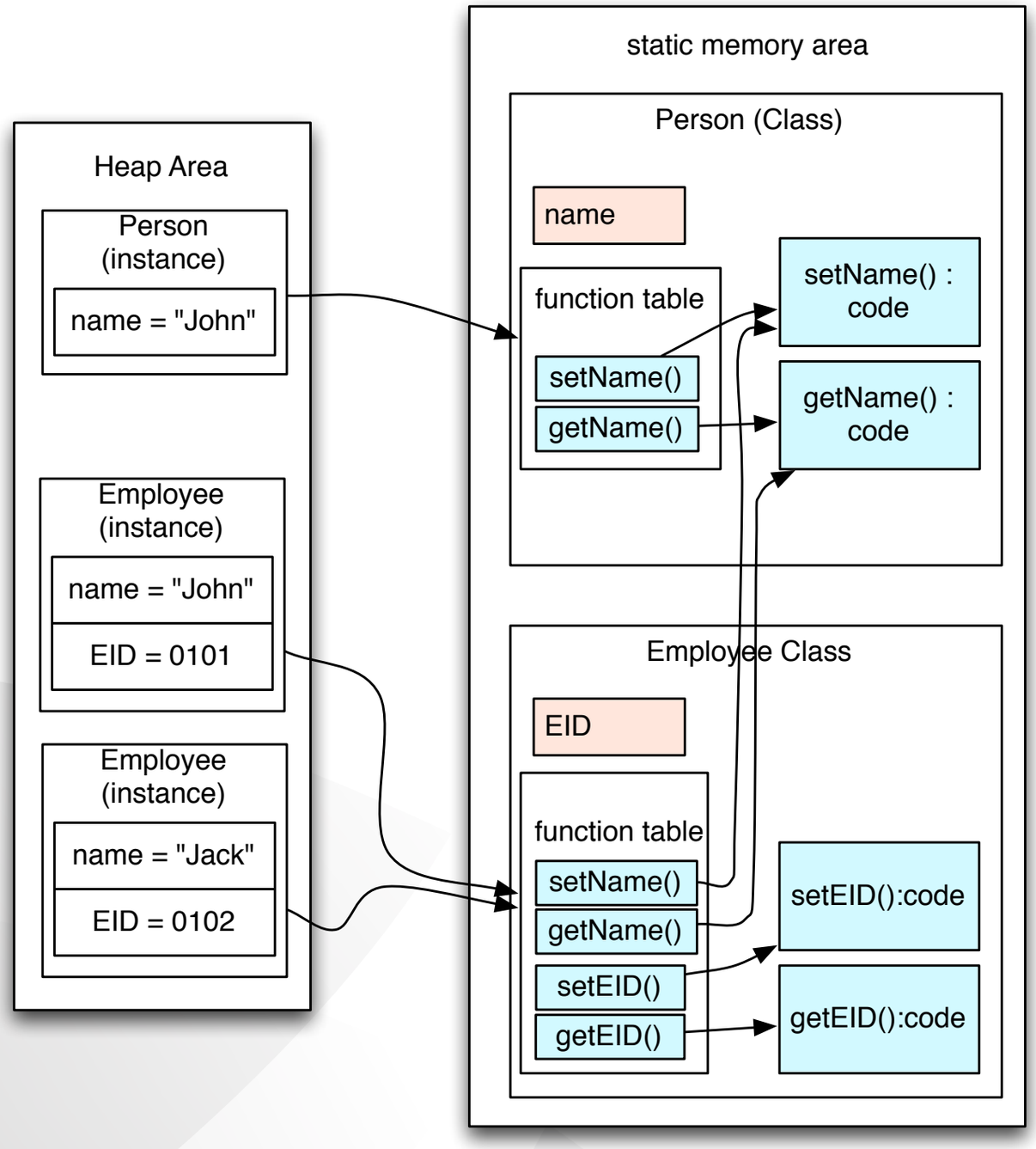
Copyright Warning

Object Oriented Design

Week 13

2nd Review

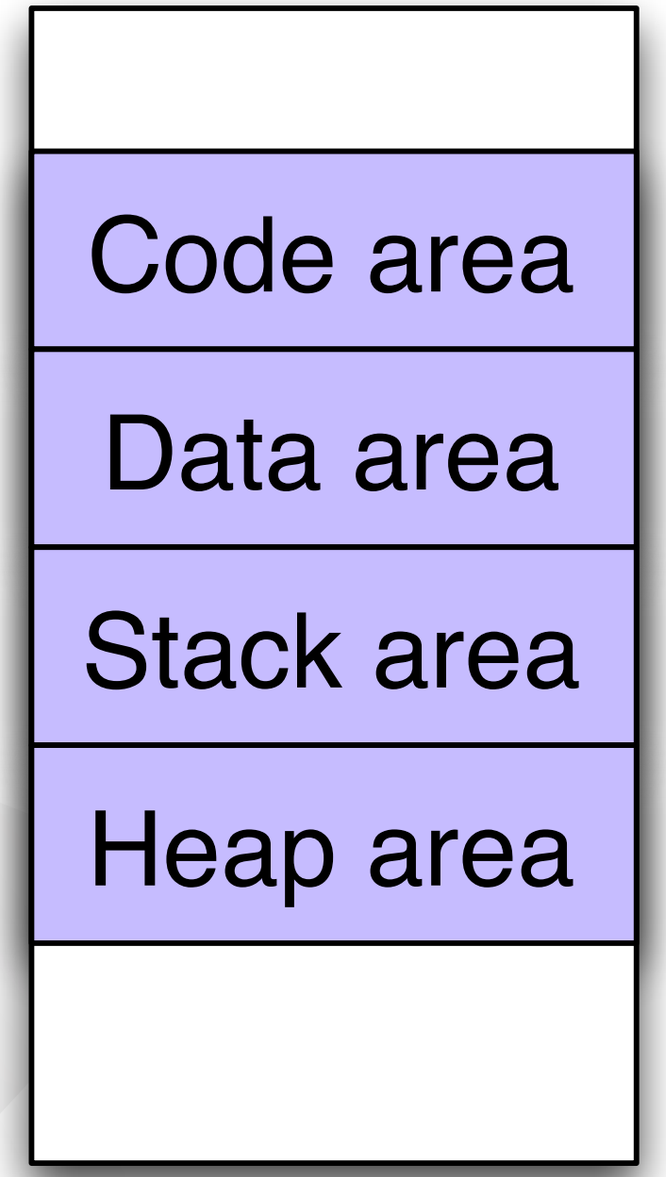
Inheritance and memory




```
class Person {
private:
     string name;
public:
     Person(){};
     Person(string name):name(name){};
     void setName(string name){ this->name = name;};
     string getName() {return name;};
};

class Employee : Person {
private:
     int EID;
public:
     Employee() {}
     Employee(string name, int id) : Person(name), EID(id) {}
     void setEID(int id) {EID = id;}
     int getEID() {return EID;}
};
```

A program



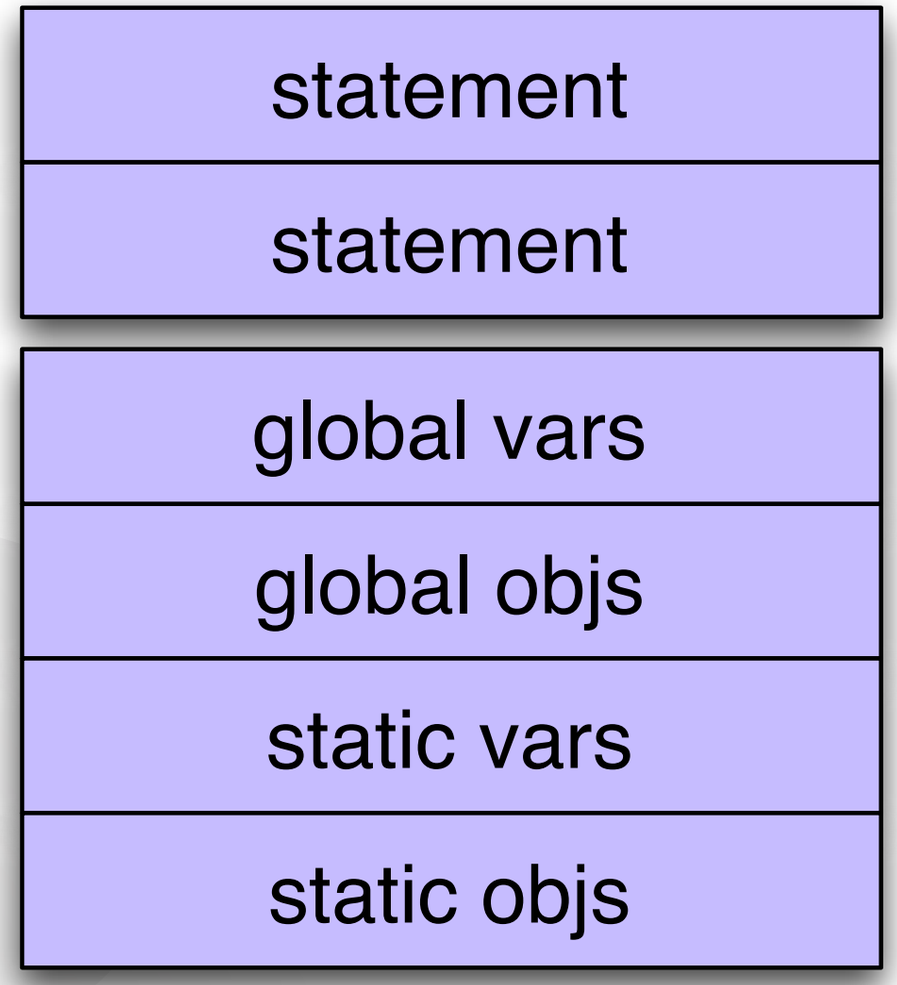
commands/statements

global variables/objects/static

args, local vars/objects

new-ed objects/vars

code/data area



commands/statements

global variables/objects/static

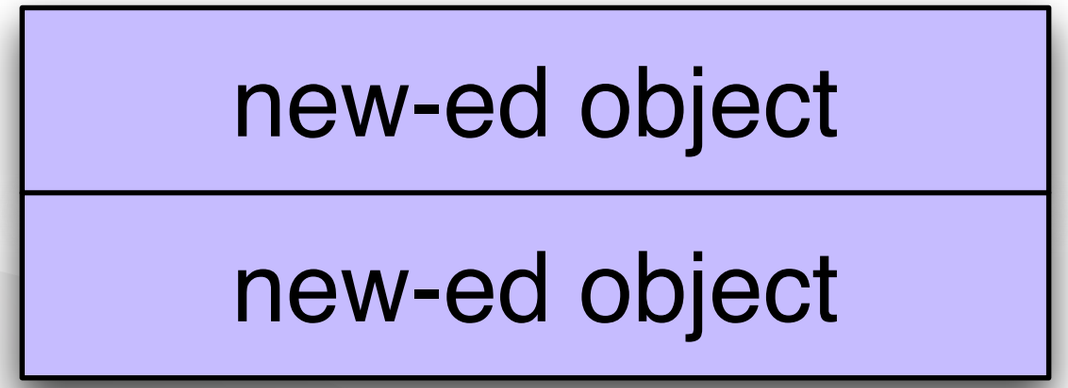
stack

- contents will dynamically change.

args, local vars/objects

heap

- contents will dynamically change.



new-ed objects/vars

Design Patterns

- Generation:
  - Singleton
  - Factory Method
  - Abstract Factory
  - Prototype
  - Builder

Design Patterns

- Structural:
  - Adapter
  - Composite
  - Facade
  - Proxy
  - Bridge
  - Flyweight
  - Decorator

Design Patterns

- Behaviour
- Memento
- Template Method
- State (structurally same as Strategy)
- Iterator (covered in Tute) (Visitor)

Design Patterns

- Given a simple description of a design pattern
- being able to apply it to derive a design
- Scenario :
  - Given class definitions
  - and new requirements
  - apply some design pattern(s)
  - derive UML diagrams and codes (at least class defs)

Template and STL

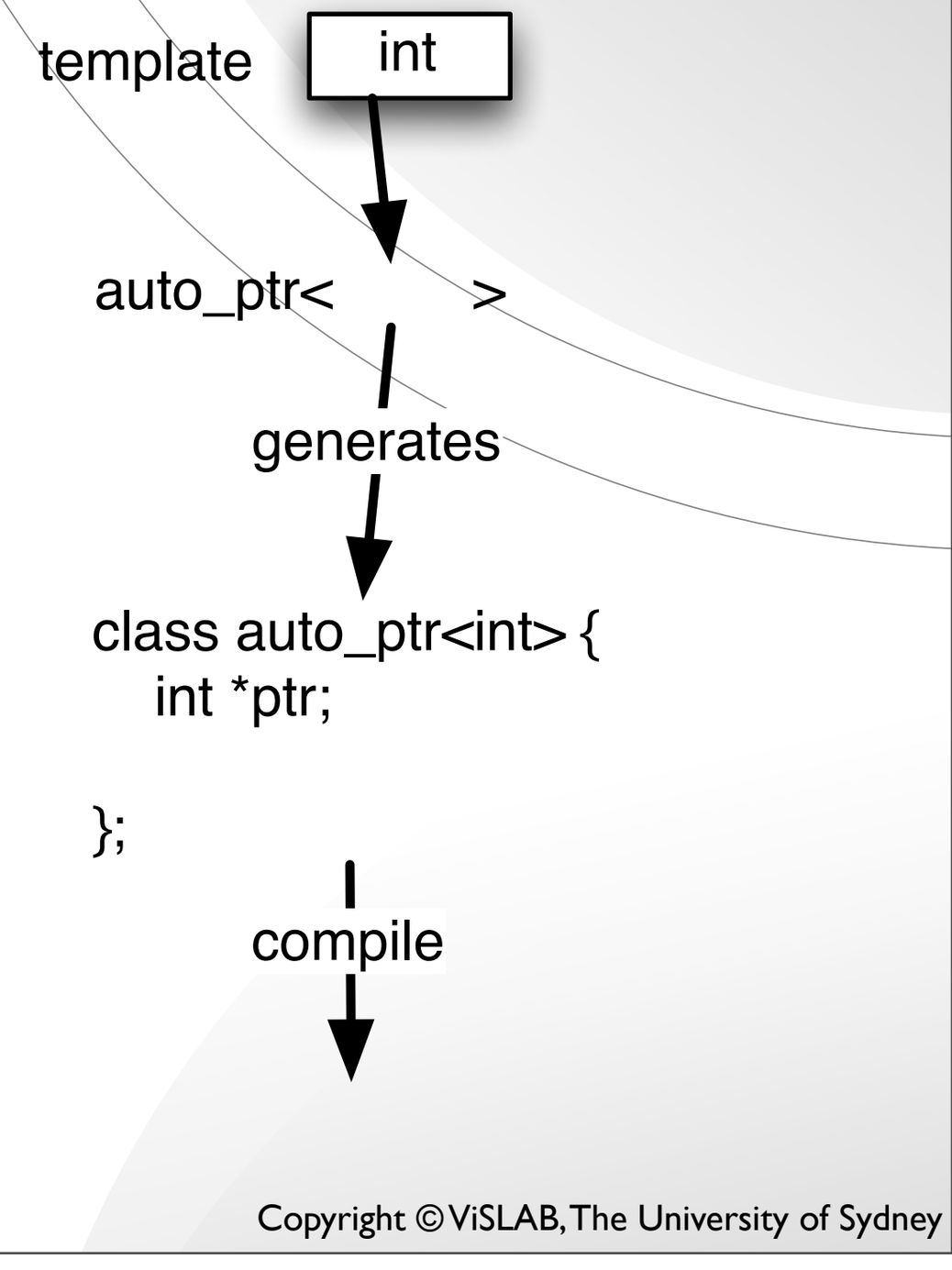
- In Week 4 tutorial
- AutoFreePtr class:

```
class AutoFreePtr {
    char *ptr;   // a pointer to an allocated char array
public:
    // you need implement here
}
```

```
#include <memory>
using namespace std;

void foo() {
   auto_ptr<int> p(new int);
   *p = 10;
}

int main(void) {
   foo();
   return 0;
}
```



Template Parameter (T/type)

```
template <typename type> void display(type arg) {
   cout << arg << endl;
}
```

or

```
template <typename T> void display(T arg) {
   cout << arg << endl;
}
```

or

```
template <class T> void display(T arg) {
   cout << arg << endl;
}
```

```
#include <iostream>
#include <string>
using namespace std;

template <typename T> void display(T arg) {
   cout << arg << endl;
}

int main() {
   display(30.0);
   string str = "hello";
   display(str);
   return 0;
}
```

- `void display(int arg)`
- `void display(string arg)`

```
template <typename T> T abs(T arg) {
   T temp;
   temp = (arg < 0) ? -arg : arg;
   return temp;
}

template <> char *abs(char *arg) {
    // some implemenation to abs a string
    ...
}

int main() {
   char *p = "hello";
   p = abs(p);

   return 0;
}
```

explicit use

```
#include <iostream>
#include <string>
using namespace std;

template <typename T> void display(T arg) {
   cout << arg << endl;
}

int main() {
   display<double>(30.0);
   string str = "hello";
   display<string>(str);
   return 0;
}
```

STL : Container

- vector
- deque
- list
- set/multiset
- map/multimap
- queue/stack

vector : iterator and delete

```
#include <vector>
using namespace std;
int main() {
   vector<int> v;
   vector<int>::iterator it;
   for (it = v.begin(); it != v.end(); ++it) {
      ...
   }
   ...
   return 0;
}
```

- v.begin() ... first iterator
- v.end() ... last iterator

- it != v.end() rather than
  - it < v.end()

vector : iterator and delete

```
#include <vector>
using namespace std;
int main() {
   vector<int> v;
   v.push_back(5);
   v.push_back(4);
   v.push_back(3);

   vector<int>::iterator it;
   for (it = v.begin(); it != v.end(); ++it) {
      cout << *it << endl;
   }
   return 0;
}
```

- use * to access the element
- not a pointer!
- operator* is defined
- -> (member access) can be used for
  - class or struct

STL : Algorithm

- STL containers provided convenient storages.
  - dynamically change sizes, etc.
  - generic programming
  - easy to change the containers

- Searching, sorting elements in the container
- other frequently used processes

Custom Class

```
#include <iostream>

class MyData {
    int data1;
    char* data2;
    friend std::ostream& operator<<(std::ostream& os, MyData& a) {
        return os << a.data1 << ", " << a.data2;
    }
    void init_data2(const char* d2) {
        if (d2) {
            this->data2 = new char[strlen(d2)+1];
            strcpy(this->data2, d2);
        }
    }
public:
    MyData(int d1, char* d2) : data1(d1), data2(0) {
        init_data2(d2);
    }
    MyData(const MyData& src) : data1(src.data1), data2(0) {
        init_data2(src.data2);
    }
};
```

Custom Class

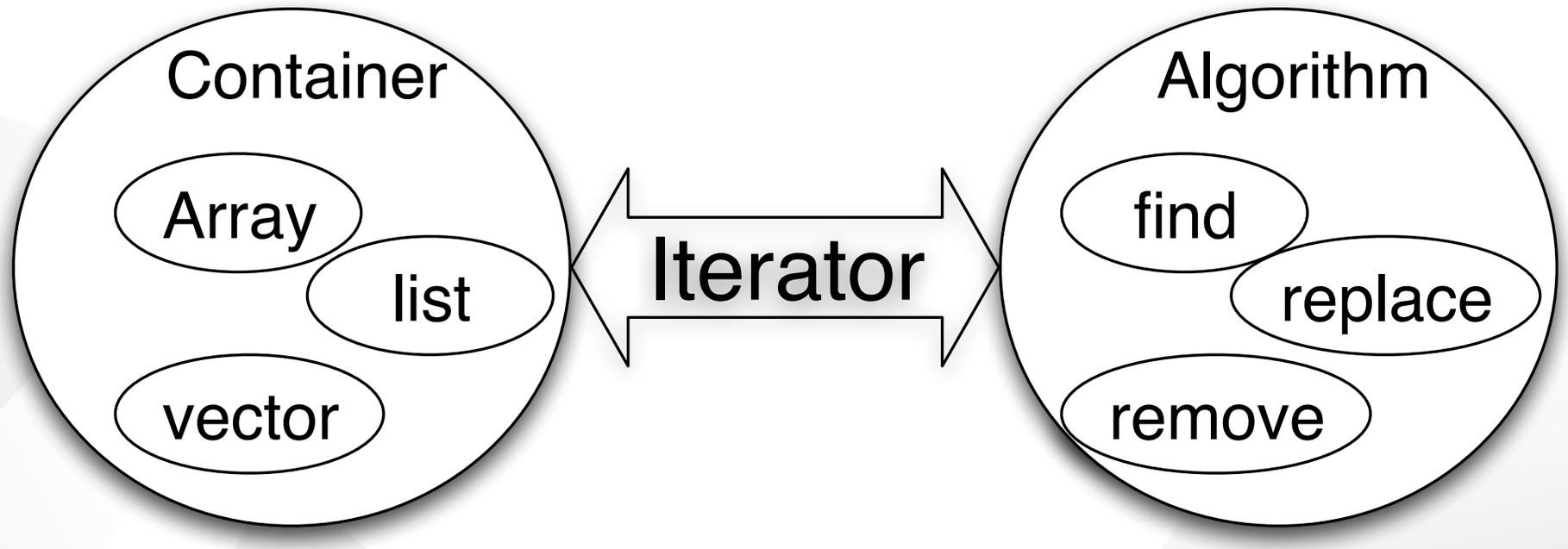
```
class MyData {
int data1;
char* data2;
friend std::ostream& operator<<(std::ostream& os, MyData& a) {
   return os << a.data1 << ", " << a.data2;
}
friend bool operator==(const MyData &left, const MyData &right) {
   return left.data1 == right.data1;
}
void init_data2(const char* d2) {
   if (d2) {
      this->data2 = new char[strlen(d2)+1];
      strcpy(this->data2, d2);
   }
}
public:
   MyData(int d1, char* d2) : data1(d1), data2(0) {
      init_data2(d2);
   }
   MyData(const MyData& src) : data1(src.data1), data2(0) {
      init_data2(src.data2);
   }
};
```

overloading operator

- find ....."=="
- sort ....."<",">"

- Cannot find "operatorXX"
  - then overload it.
  - use "const" for the arg

- Array + pointer works with std::copy
- why iterator?



Function Object

```
class Func {
public:
   int operator() (int a, int b) const {
      return a + b;
   }
}
```

- works like a function, but it's not a function
- has a function "operator()"

```
Func f;
...
cout << f(5, 53) << endl;
```

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
    list<int> l;

    l.push_back(5);
    l.push_back(3);
    l.push_back(2);
    l.push_back(1);
    l.push_back(4);

    list<int>::iterator it;
    for (it = l.begin(); it != l.end();) {
        it = find_if(it, l.end(), bind2nd(greater<int>(), 3));
        if (it != l.end()) {
            cout << "found! " << *it << endl;
            ++it;
        }
    }
```

- 1st and 2nd args : specify range of search
- `bind2nd(greater<int>(), 3)`
  - `std::greater<int>()` : function object (>)
  - `<int>` : template
  - `()` : calling a constructor
  - `bind2nd` : combined F.O(>) and 3 ....3>

Function Object

```
f(5, 53)

f.operator()(5, 53)
```

`operator()` : is the name of the function.

you can override `operator()` with different args.

```
    class StrSizeComp {
public:
    bool operator() (const string& a, const string& b) const {
        return a.size() < b.size();
    }
}

int main() {
    string s;
    vector<string> data;
    while (getline(cin, s))
        data.push_back(s);
    StrSizeComp f;
    sort(data.begin(), data.end(), f);
    vector<string>::iterator it;
    for ( it = data.begin(); it != data.end(); ++it)
        std::cout << *it << std::end;
    return 0;
}

    StrSizeComp f;
    sort(data.begin(), data.end(), f);
```

or

```
    sort(data.begin(), data.end(), StrSizeComp());
```

stream iterator

```
int main() {
int a[] = {3, 5, 2, 1};
std::list<int> l;

l.push_back(0);
l.push_back(0);
l.push_back(0);
l.push_back(0);

std::copy(a, a + 5, l.begin());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, " "));
return 0;
}
```

- More in tute.

Custom Class

```
iterator find(iterator begin, iterator end, T value) {
iterator it;
for (it = begin; it != end; ++it)
   if (*it == value)
      return it;
return end;
}
```

- inside “find”
- == to compare values (int, double are ok, but..)
- you need to overload “operator==”

Final Exam

- 18/June 6pm
- 10 min (reading) + 2 hours
- 4 sections
  - each section will have 2 or 3 sub-questions.

Final Exam

- OO literacy
- OO and memory
- C++ and OO
- C++ : template
- Design Patterns

OO

- OOD, OOP : need lots of practice (do practice from the tutorial sessions)
- Assignment 1 - 4 : good bases to practice