

Memory leak

Guoyang Zeng

CST63

1

What is Memory Leak?



▲ What is Memory Leak?

```
#include <stdio.h>
```

```
#include <string.h>
```

```
char* getStr() {
```

```
    char* ret = new char[9];
```

```
    strcpy(ret, "MemLeak!");
```

```
    return ret;
```

```
}
```

```
int main() {
```

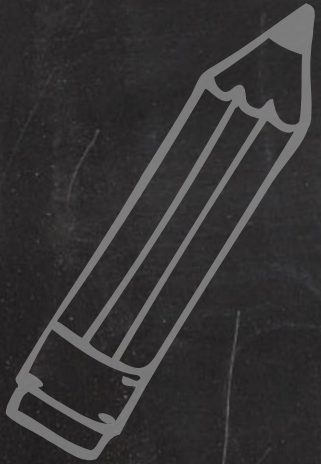
```
    printf("%s\n", getStr());
```

```
    return 0;
```

```
}
```

new a char
array here

forget to delete



What is Memory Leak?



```
#include <stdio>
#include <string>
char* getStr() {
    char* ret = new char[9];
    strcpy(ret, "MemLeak!");
    return ret;
}
int main() {
    char *str = getStr();
    printf("%s\n", str);
    delete[] str;
    return 0;
}
```

new a char
array here



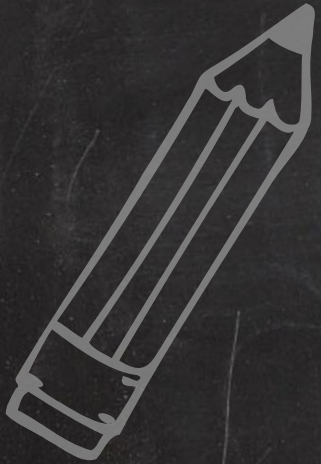
What is Memory Leak?

```
#include <stdio>
#include <string>
char* getStr() {
    char* ret = new char[9];
    strcpy(ret, "MemLeak!");
    return ret;
```

```
}
int main() {
    for (int i = 0; i < 1e9; ++ i)
        getStr();
    return 0;
```

```
}
```

!!! 10^9 times



▲ What is Memory Leak?



名称	6% CPU	38% 内存
> test.exe (32 位)	0%	54.4 MB



名称	8% CPU	43% 内存
> test.exe (32 位)	0%	1,073.3 MB



名称	8% CPU	49% 内存
> test.exe (32 位)	0%	1,939.6 MB



Crashed!

2

Some common errors!





Some common errors!

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
};
```



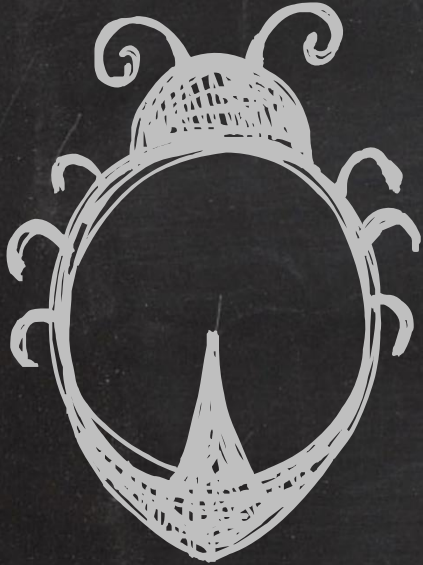


Some common errors!

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
};
```

```
void Bar(Foo a) {  
    // do something  
}
```

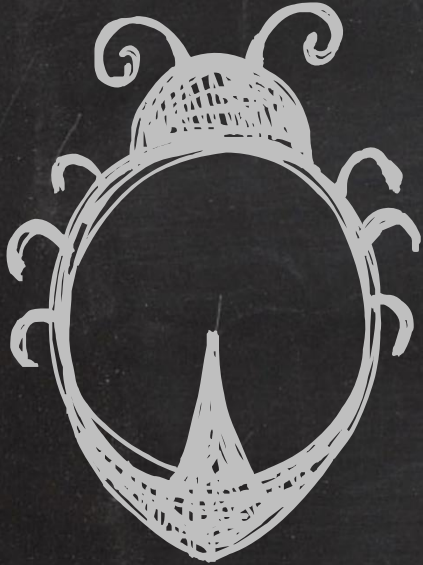
```
Foo foo = Foo(0x3f3f);  
Bar(foo);
```





Some common errors!

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
};
```



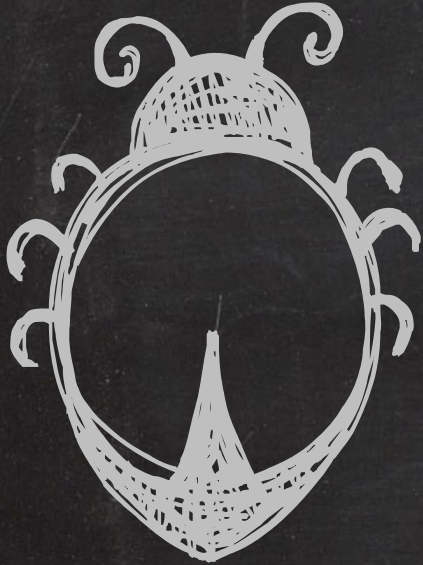
add a Copy constructor

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
    Foo(const Foo &b) { myInt = new int(*b.myInt); }  
};
```



Some common errors!

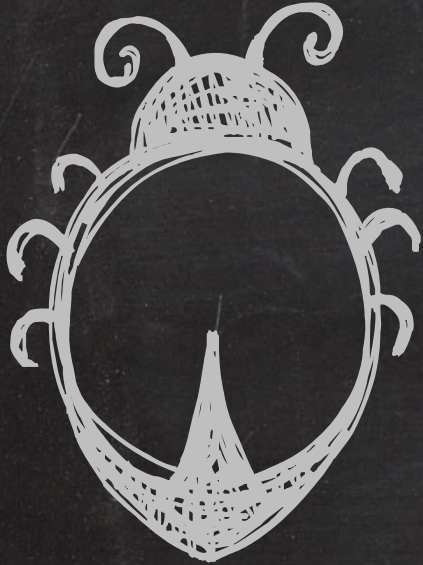
```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
    Foo(const Foo &b) { myInt = new int(*b.myInt); }  
};
```





Some common errors!

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
    Foo(const Foo &b) { myInt = new int(*b.myInt); }  
};
```



```
void Bar() {  
    Foo a = Foo(0x3f3f);  
    Foo b = Foo(0);  
    b = a;  
}
```



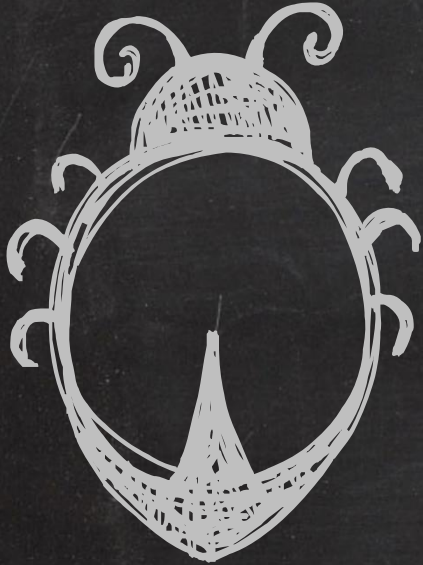
Some common errors!

```
class Foo {  
    int *myInt;  
public:  
    Foo(const int &val = 0): myInt(new int(val)) { }  
    ~Foo() {delete myInt;}  
    Foo(const Foo &b) { myInt = new int(*b.myInt); }  
};
```



overload operator=

```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```





Some common errors!

```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```





Some common errors!

```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```

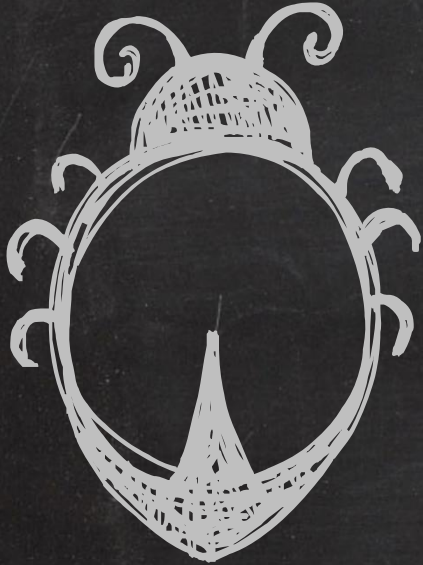


Memory leak



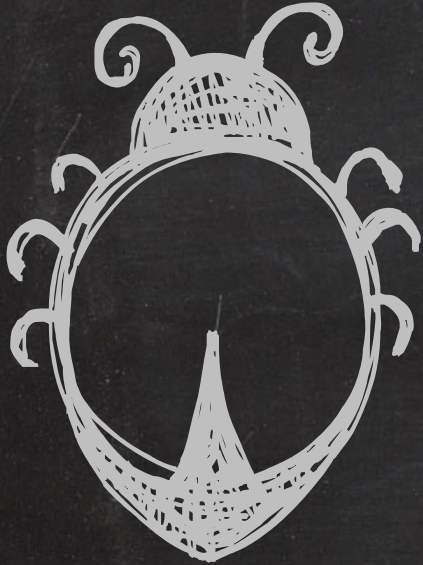
Some common errors!

```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        delete myInt;  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```





Some common errors!



```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        delete myInt;  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```

```
Foo a;  
a = a; // a.myInt has been deleted
```



Some common errors!

```
class Foo {  
    int* myInt;  
public:  
    Foo(const int &val = 0) : myInt(new int(val)) { }  
    ~Foo() { delete myInt; }  
    Foo(const Foo &b) {myInt = new int(*b.myInt);}  
    Foo& operator = (const Foo &b) {  
        if (&b == this) return *this;  
        delete myInt;  
        myInt = new int(*b.myInt);  
        return *this;  
    }  
};
```



Important

3 Ways to find memory leaks





Ways to find memory leaks

1

visual c++ (on windows)

```
#define CRTDBG_MAP_ALLOC
#include <crtdbg.h>
int main() {
    int *p = new int[15];
    _CrtDumpMemoryLeaks();
    return 0;
}
```

← crtdbg.h



output:

Detected memory leaks!

Dumping objects ->

{76} normal block at 0x001D9130, 60 bytes long.

Data: < > CD CD CD CD CD CD CD CD CD
CD CD CD CD CD CD CD

Object dump complete.



Ways to find memory leaks

1

visual c++ (on windows)

2

on linux

```
valgrind --tool=memcheck --leak-check=full ./a.out
```

output:

.....

==14609== LEAK SUMMARY:

==14609== definitely lost: 60 bytes in 1 blocks

==14609== indirectly lost: 0 bytes in 0 blocks

==14609== possibly lost: 0 bytes in 0 blocks

.....





Ways to find memory leaks

1

visual c++ (on windows)

2

on linux

3

XCode (on mac os)

leaks [PID]

output:

.....

Process [PID]: 1 leaks for 60 total leaked bytes

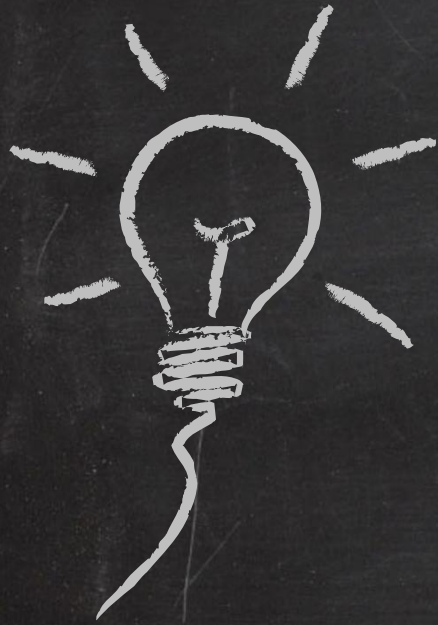


4 Ways to avoid memory leaks



▲ Ways to avoid memory leaks

c++11's Smart Pointer:



1 `unique_ptr`

2 `shared_ptr`

▲ Ways to avoid memory leaks

C++11's Smart Pointer:

1 *unique_ptr*



```
#include <memory>
unique_ptr<int> u_i;
u_i.reset(new int(3));
unique_ptr<int> u_i2(new int(4));
int *p_i = u_i2.release();
unique_ptr<int> u_i2 = std::move(u_i);
u_i2=nullptr;
```

▲ Ways to avoid memory leaks

C++11's Smart Pointer:

1 `unique_ptr`

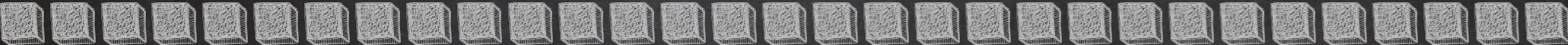
2 `shared_ptr`



```
#include <memory>
unique_ptr<int> u_i;
u_i.reset(new int(3));
unique_ptr<int> u_i2(new int(4));
int *p_i = u_i2.release();
unique_ptr<int> u_i2 = u_i;
u_s2=nullptr;
```

difference





Thanks for listening

