Dynamic allocation - Is there a problem?

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }

private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();

}
```

```
Thing: 17
```

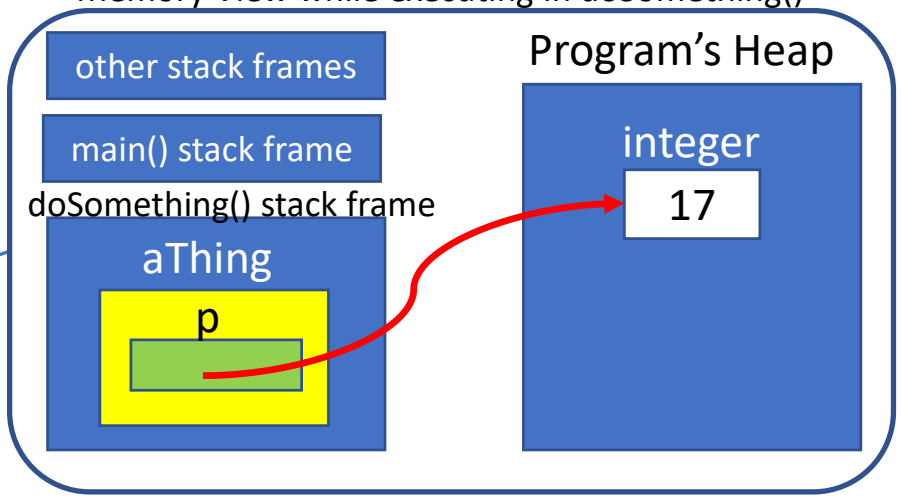
- Is there anything wrong with this code?

Deallocation is needed

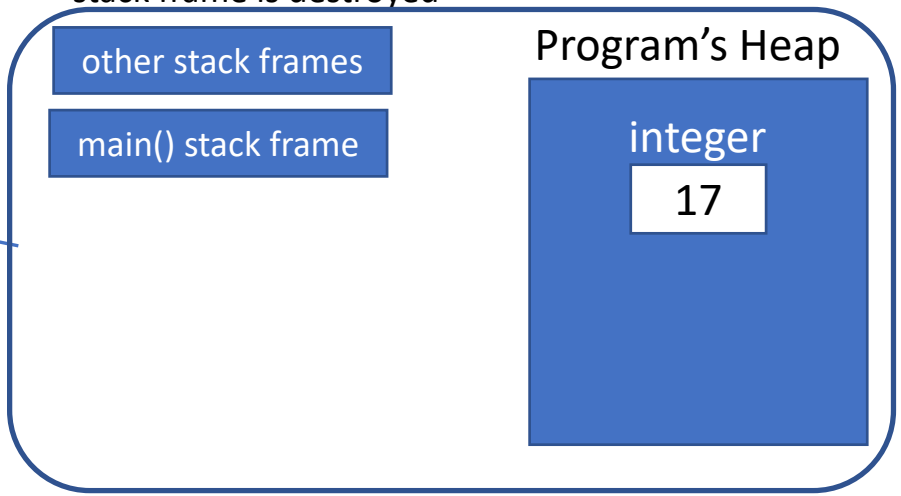
```
#include <iostream>
using namespace std;

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }

private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();

}
```

- Is there anything wrong with this code?
  - **Yes,** doSomething() left the dynamically allocated integer behind, with no way of accessing it or de-allocating it later on.
  → leaving garbage behind (**Memory leak**)!
- **How do we fix this?**

Memory View while executing in doSomething()



Memory View after doSomething() exits and its stack frame is destroyed

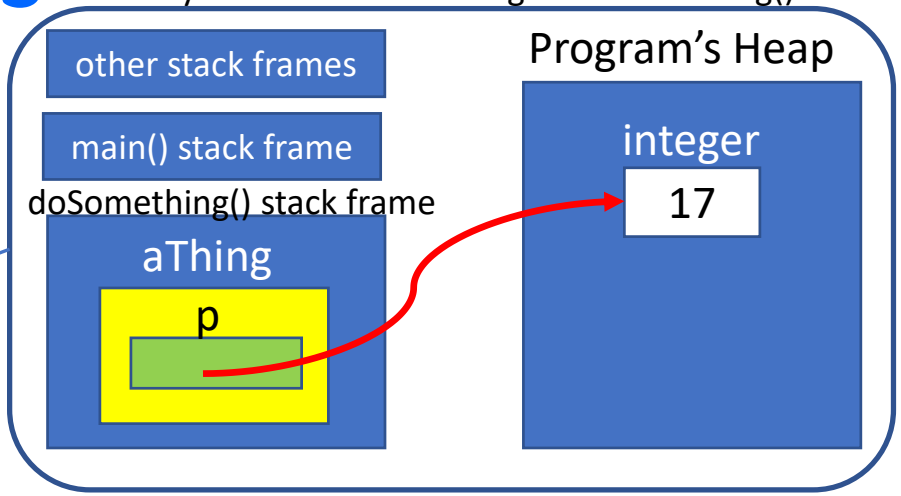


Deallocation using Destructors

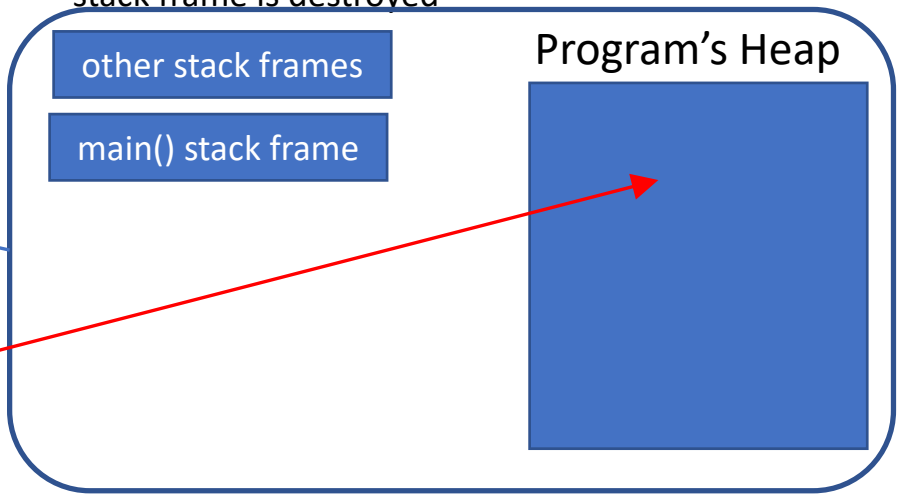
```
#include <iostream>
using namespace std;

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();

}
```

- The destructor is called automatically when the object is being destroyed
- The destructor de-allocated the dynamic memory

Memory View while executing in doSomething()



Memory View after doSomething() exits and its stack frame is destroyed



Destructors – cont.

- The destructor is invoked either
  - When a function returns (i.e. at the closing curly brace)
  - When the `delete` keyword is used.

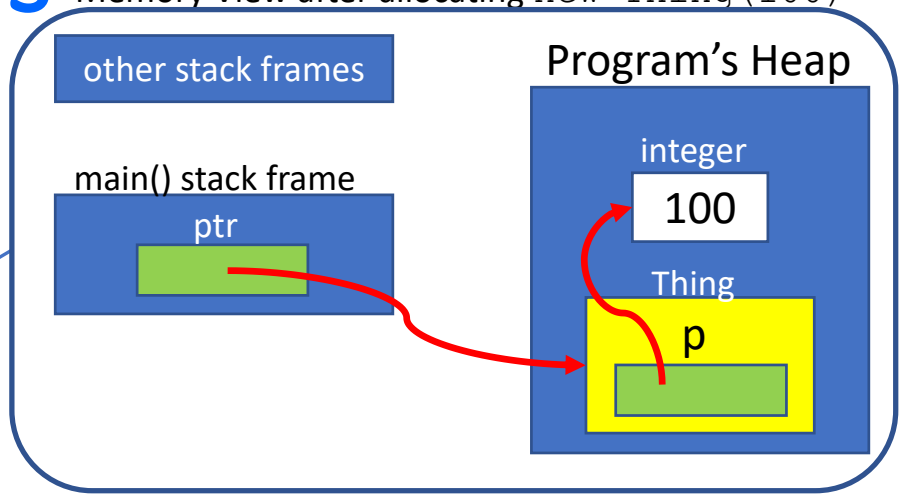
Deallocation using Destructors

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;
}

int main() {
    doSomething();
    Thing* ptr = new Thing(100);
    delete ptr;
}
```

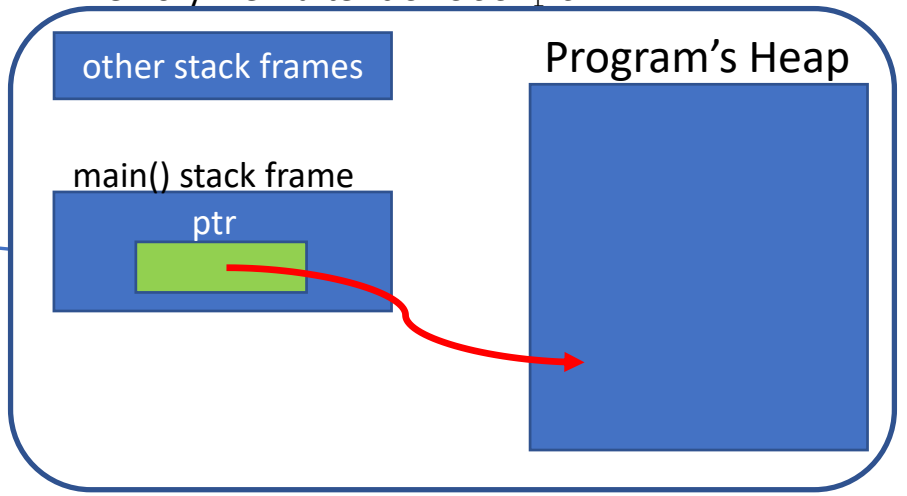
After this statement, ptr is a dangling pointer, but that's okay, we won't use it

- The destructor is called automatically when `delete` is invoked, which
  - Invokes the Thing::~Thing()
  - Deallocates the memory for the Thing object
- The destructor Thing::~Thing() de-allocated the dynamically allocated integer

Memory View after allocating `new Thing(100)`

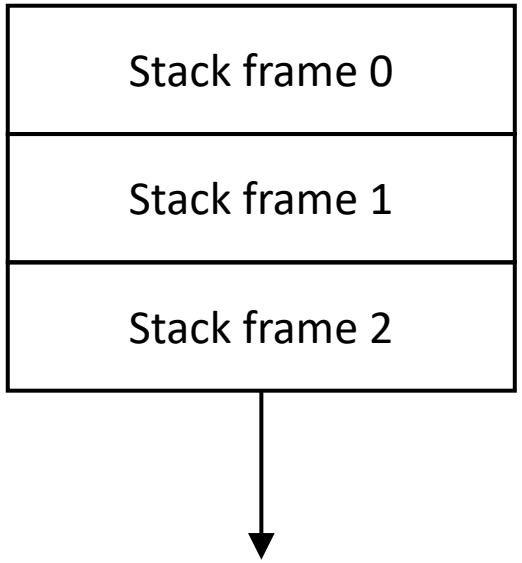


Memory View after `delete ptr`

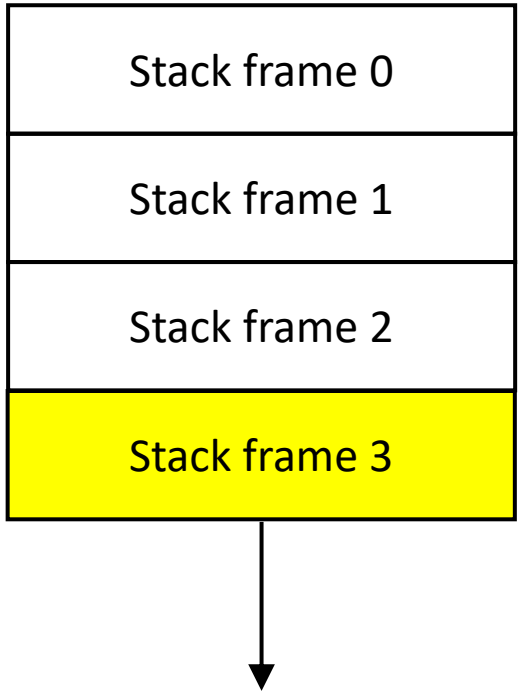


The Stack - reminder

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

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```

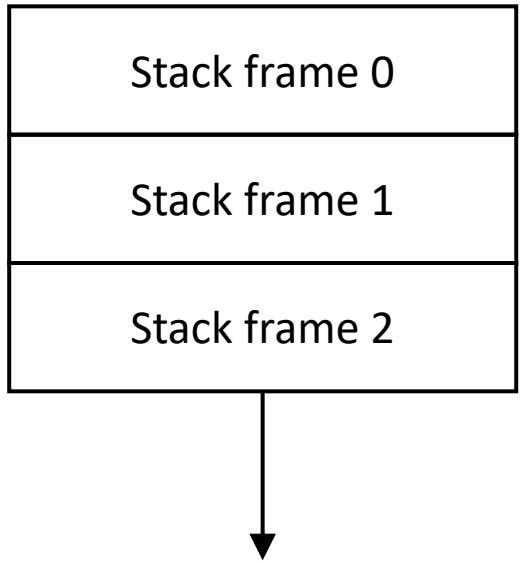


Before calling myfunc()



While myfunc() is executing:

- Parameter x is pushed into stack frame 3
- Local variable y is allocated in frame 3



After myfunc() executed:

- Parameter x is deallocated
- Local variable y deallocated
- Stack frame is no more

Destructors – yet another issue?

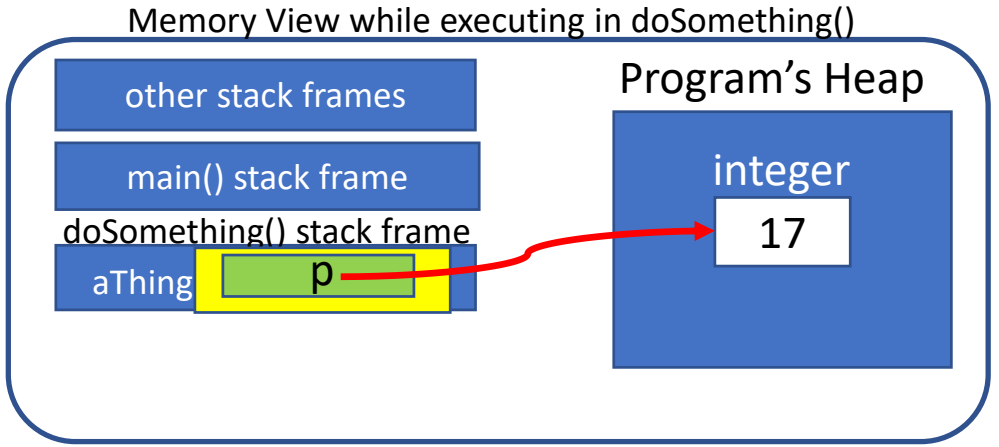
```
#include <iostream>
using namespace std;

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doNothing(Thing x) {}

void doSomething() {
    Thing aThing(17);
    doNothing(aThing);
    cout << aThing << endl;
}

int main() {
    doSomething();
}
```

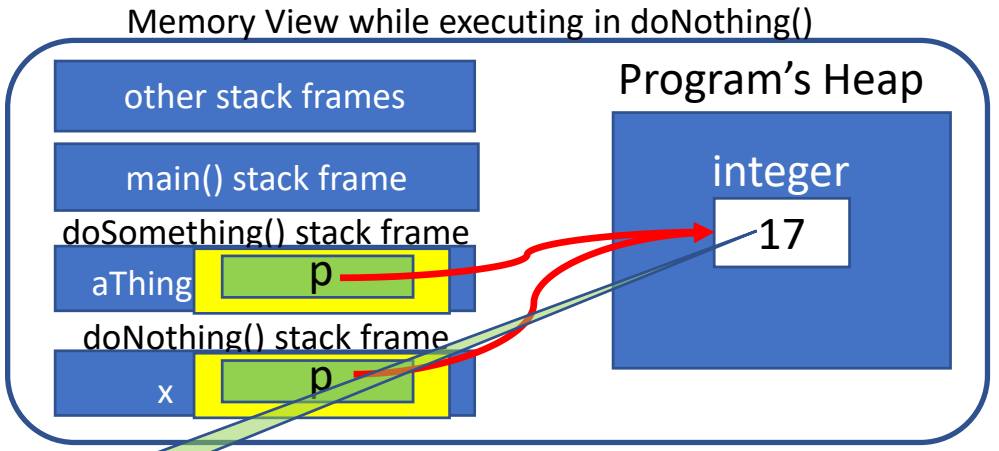
- Is this going to work okay?

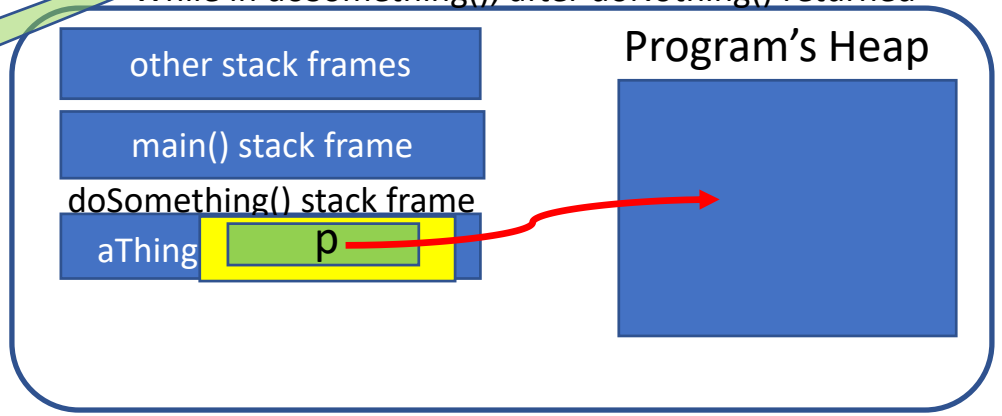
Destructors – anat. issue?

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doNothing(Thing x) {}

void doSomething() {
    Thing aThing(17);
    doNothing(aThing);
    cout << aThing << endl;
}

int main() {
    doSomething();
}
```

Shalllow copying
X ← aThing







Destructors – another issue? – cont.

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    //return os << "Thing: " << *rhs.p;
    os << "Thing: " << *rhs.p;
    return os;
}
void doNothing(Thing x) {}

void doSomething() {
    Thing aThing(17);
    doNothing(aThing);
    cout << aThing << endl;
}

int main() {
    doSomething();
}
```

WE GET AN EXCEPTION !!

- **doNothing, passed a copy of aThing (as parameter x)**, including its integer pointer "p"
- **When doNothing() exits**, the destructor for its local variables and its parameters are invoked.
  - The destructor frees up the int that was allocated during creation of aThing
- **After doNothing() exists**, the pointer residing inside the object "aThing" is not valid → **dangling pointer!**.
- **When doSomething() exits**, the original Thing ends its scope and its destructor tries to free up the int that is "no longer there" (the pointer p is a dangling pointer) → run-time exception
- **Whose fault?**
  - Shall a destructor always nullify the pointer after deallocation AND check for null pointers?
  - Or shall the copying be **deep**, rather than **shallow**?

Destructors – another issue? – cont.

- Built-in copy constructor only copies the pointer but does not allocate additional data (i.e. **shallow copying**)

- A **copy constructor** implementation is needed. It should perform:
  - Allocating a new space for the date members (int in our example)
  - Populating the values in these data members

  → **deep copying**

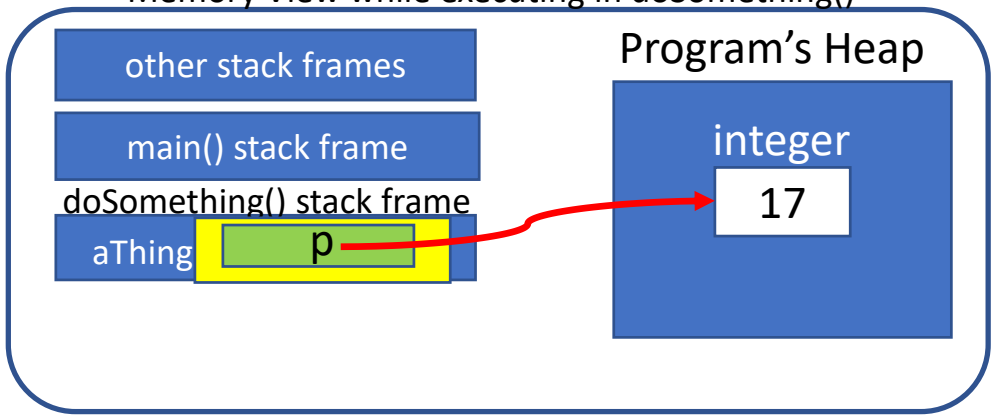
Copy constructor

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor
private:
    int* p;
};
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doNothing(Thing x) {}
void doSomething() {
    Thing aThing(17);
    doNothing(aThing);
    cout << aThing << endl;
}

int main() {
    doSomething();
}
```

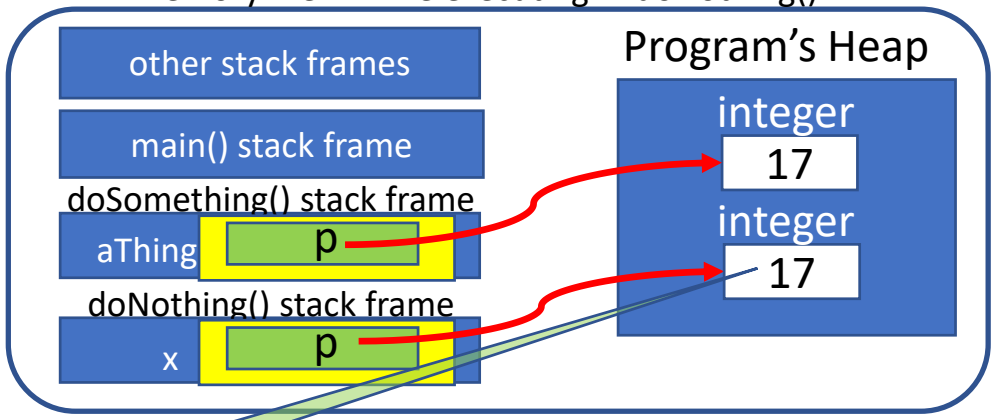
Thing: 17

Deep copying
X ← aThing

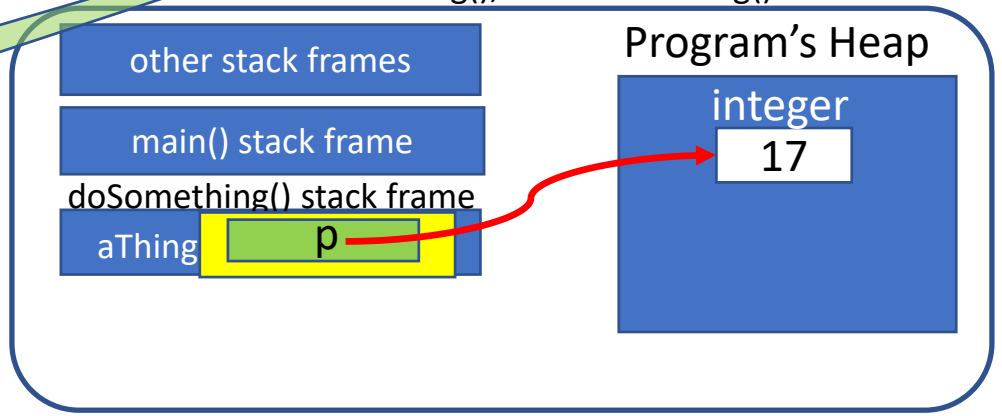
Memory View while executing in doSomething()



Memory View while executing in doNothing()



While in doSomething(), after doNothing() returned



Overloading operator=

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }

private:
    int* p;
};
```

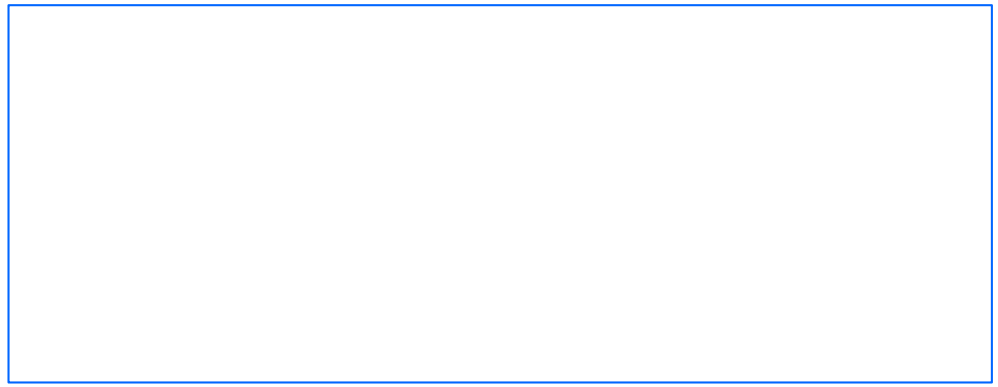
To support proper copy control, you should implement:

- copy constructor
- overloaded operator=
- destructor

Overloading operator=

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doNothing(Thing x) {}

void doSomething() {
    Thing aThing(17);
    doNothing(aThing);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something; //same as aThing.operator=(something);
    cout << aThing << endl;
}

int main() {
    doSomething();

}
```

```
Thing: 17
Thing: 6
```



Assigning to int

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        cout << "Copy constructor invoked: " << endl;
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }
    Thing& operator=(int x) {
        *p = x;
        return *this;
    }

private:
    int* p;
};
```

```
Thing: 17
Thing: 6
Thing: 5
```

- Let's add operator=(int x)

Assigning to int - cont.

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something;
    //aThing.operator=(something); // same as “aThing = something;”
    cout << aThing << endl;
}

int main() {
    doSomething();

    Thing mything = 5;
    cout << mything << endl;

}
```

```
Thing: 17
Thing: 6
Thing: 5
```

Assigning to int - which one is called?

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        cout << "Copy constructor invoked: " << endl;
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }
    Thing& operator=(int x) {
        *p = x;
        return *this;
    }

private:
    int* p;
};
```

- But hold on…
- Which one of these two was invoked?

Assigning to int - which one is called? - cont.

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); cout << "constructor invoked: " << endl; }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        cout << "Copy constructor invoked: " << endl;
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        cout << "operator= invoked: " << endl;
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }
    Thing& operator=(int x) {
        cout << "operator= invoked: " << endl;
        *p = x;
        return *this;
    }

private:
    int* p;
};
```

- **Generally speaking, you do not need to be printing to stdout within your constructor, this is only for demonstration.**

Assigning to int - which one is called? - cont.

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something;
    //aThing.operator=(something); // same as “aThing = something;”
    cout << aThing << endl;
}

int main() {
    doSomething();

    Thing mything = 5;
    cout << mything << endl;

}
```

```
constructor invoked:
Thing: 17
constructor invoked:
operator= invoked:
Thing: 6
constructor invoked:
Thing: 5
```

- Okay, so it’s the constructor that was invoked here… just as we stated before
  - When we use the assignment operator on the same statement as the object instantiation, this is an initialization → constructor takes care of it.

Assigning to int - which one is called (2)?

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); cout << "constructor invoked: " << endl; }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        cout << "Copy constructor invoked: " << endl;
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        cout << "operator= invoked: " << endl;
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }
    Thing& operator=(int x) {
        cout << "operator= invoked: " << endl;
        *p = x;
        return *this;
    }

private:
    int* p;
};
```

Assigning to int - which one is called (2)? - cont.

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something; //same as aThing.operator=(something);
    cout << aThing << endl;
}

int main() {
    doSomething();

    Thing mything(200);
    cout << mything << endl;
    mything = 5;
    cout << mything << endl;

}
```

```
constructor invoked:
Thing: 17
constructor invoked:
operator= invoked:
Thing: 6
constructor invoked:
Thing: 200
operator= invoked:
Thing: 5
```

- All makes sense so far:
  - Constructor invoked when needed
  - Assignment invoked when needed
- Can I remove the assignment operator from my class? **Lets' try it!**

Removing operator=(int)

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

class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x) { p = new int(x); cout << "constructor invoked: " << endl; }
    // Copy Constructor
    Thing(const Thing& anotherThing) {
        cout << "Copy constructor invoked: " << endl;
        p = new int(*anotherThing.p); // deep copy
    }
    ~Thing() { delete p; } // destructor

    Thing& operator=(const Thing& rhs) {
        cout << "operator= invoked: " << endl;
        if (this != &rhs) { // 0. self check
            // 1. Free up whatever heap space the lhs is holding
            delete p;
            // Deep copy
            // 2. allocate
            // 3. copying
            p = new int(*rhs.p);
        }
        // 4. return yourself
        return *this;
    }
//     Thing& operator=(int x) {
//         cout << "operator= invoked: " << endl;
//         *p = x;
//         return *this;
//     }

private:
    int* p;
};
```

Removing operator=(int) - converting int to thing

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something; //same as aThing.operator=(something);
    cout << aThing << endl;
}

int main() {
    doSomething();

    Thing mything(200);
    cout << mything << endl;
    mything = 5; // same as mything.operator=(5);
    cout << mything << endl;

}
```

```
constructor invoked:
Thing: 17
constructor invoked:
operator= invoked:
Thing: 6
constructor invoked:
Thing: 200
constructor invoked:
operator= invoked:
Thing: 5
```

- What happened here?
  - There is no assignment operator that takes rhs of type int, but there is one that takes rhs of type “Thing”
  - First: compiler converts int into Thing
  - Second: assigns the converted object to your “mything” object.
- Thus, our constructor has acted as a **converting constructor,** aka a casting constructor

Removing operator=(int)-converting float to thing

```
ostream& operator<<(ostream& os, const Thing& rhs) {
    os << "Thing: " << *rhs.p;
    return os;
}

void doSomething() {
    Thing aThing(17);
    cout << aThing << endl;

    Thing something(6);
    // assignment operator
    aThing = something; //same as aThing.operator=(something);
    cout << aThing << endl;
}

int main() {
    doSomething();

    Thing mything(200);
    cout << mything << endl;
    mything = 5.4;
    cout << mything << endl;

}
```

```
constructor invoked:
Thing: 17
constructor invoked:
operator= invoked:
Thing: 6
constructor invoked:
Thing: 200
constructor invoked:
operator= invoked:
Thing: 5
```

- It'll easily cast the double into an integer, but it'll give you a **warning at compile-time**:

```
Warning   C4244      'argument': conversion from
'double' to 'int', possible loss of data
            C:\Dropbox\CS2124_OOP_2023_Spring\l
ect_code\04.Copy Control\1.copy-control-
B_om.cpp 53
```

Copy control – adding a non-primitive attribute

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

// Thing class now has TWO fields, a pointer and a string
class Thing {
    friend ostream& operator<<(ostream& os, const Thing& rhs);
public:
    Thing(int x, const string& name) : name(name) {
        p = new int(x);
    }
    void setValue(int val) { *p = val; }
    int getValue() const { return *p; }
    // Destructor
    ~Thing() { delete p; }
    // copy constructor
    Thing(const Thing& anotherThing) : name(anotherThing.name) {
        p = new int( *anotherThing.p );
    }
    // Assignment operator
    Thing& operator=(const Thing& rhs) {
        if ( &rhs != this ) {// 0. Self check
            // 1. Free up the memory (avoid leaks)
            delete p;

            // Deep copy
            p = new int( *rhs.p );
            name = rhs.name;
        }
        return *this;
    }

private:
    int* p;
    string name;
};
```

```
17
17
```

Copy control – adding a non-primitive – cont.

```
Thing doNothing(Thing something) {
    something.setValue(42);
    return something;
}

void doSomething() {
    Thing aThing(17, "moe");
    cout << aThing << endl;

    doNothing(aThing);
    cout << aThing << endl;


    Thing anotherThing(64, "larry");
    aThing = anotherThing;

    anotherThing = anotherThing;

    Thing thingA(aThing);
    Thing thingB = aThing;
}

int main() {
    doSomething();

    Thing* thingPtr = new Thing(6, "curly");
    delete thingPtr;

}

ostream& operator<<(ostream& os, const Thing& rhs) {
    os << *rhs.p;
    return os;
}
```

```
17
17
```