SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220216<A|D>

Replace <A|D> with this section's letter

Copy Control

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- The destructor
- The copy constructor
- In-class problem

The destructor

A class with dynamically allocated memory

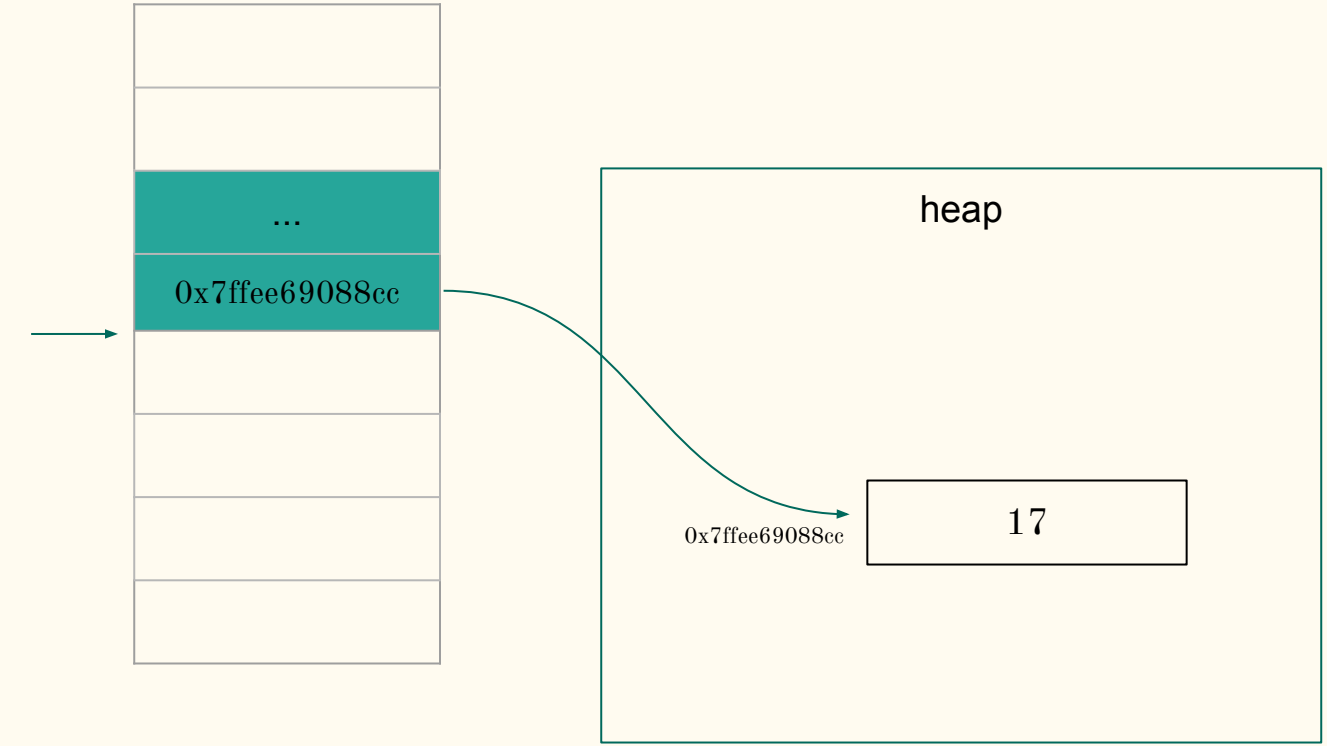
```
class SimpleClass {
public:
    SimpleClass() { ptr = new int(17); }
private:
    int* ptr;
};
```

consider consequences of class design

A class with dynamically allocated memory

```
class SimpleClass {
public:
    SimpleClass() {
        ptr = new int(17);
    }
private:
    int* ptr;
};

void a_function() {
   SimpleClass simp;
   …
}
```

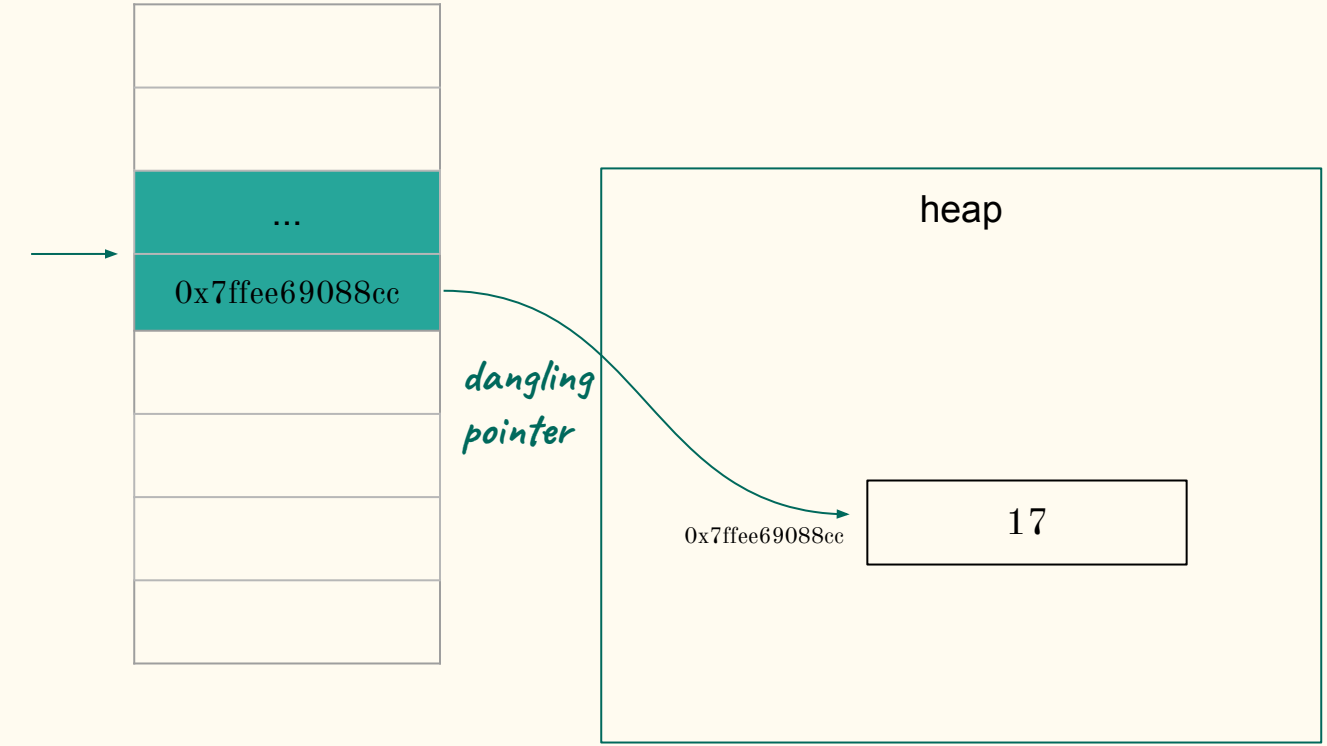


A class with dynamically allocated memory

```
class SimpleClass {
public:
    SimpleClass() {
        ptr = new int(17);
    }
private:
    int* ptr;
};

void a_function() {
   SimpleClass simp;
   …
}

int main() {
   a_function();
   …
}
```



Class destructor

- A destructor frees resources that object is responsible for allocating

```
class ClassName {
public:
    ClassName(p_type1 p_name1, p_type2 p_name2, ...)
        : m_name1(p_name1), m_name2(p_name2),... {
        /* constructor body */
    }

private:
    m_type1 m_name1;
    m_type2 m_name2;
    ...

};
```

Class destructor

- A destructor frees resources that object is responsible for allocating

```
class ClassName {
public:
    ClassName(p_type1 p_name1, p_type2 p_name2, ...)
        : m_name1(p_name1), m_name2(p_name2),... {
        /* constructor body */
    }

    ~ClassName() {
       /* free allocated resources */
    }

private:
    m_type1 m_name1;
    m_type2 m_name2;
    ...

};
```

- uses name of class with tilde (~) prepended
- no return type
- declares no parameters
- automatically invoked when object destroyed

Class destructor

```
class SimpleClass {
public:
    SimpleClass() { ptr = new int(17); }

private:
    int* ptr;

};
```

Class destructor

```
class SimpleClass {
public:
    SimpleClass() { ptr = new int(17); }
    ~SimpleClass() { delete ptr; }

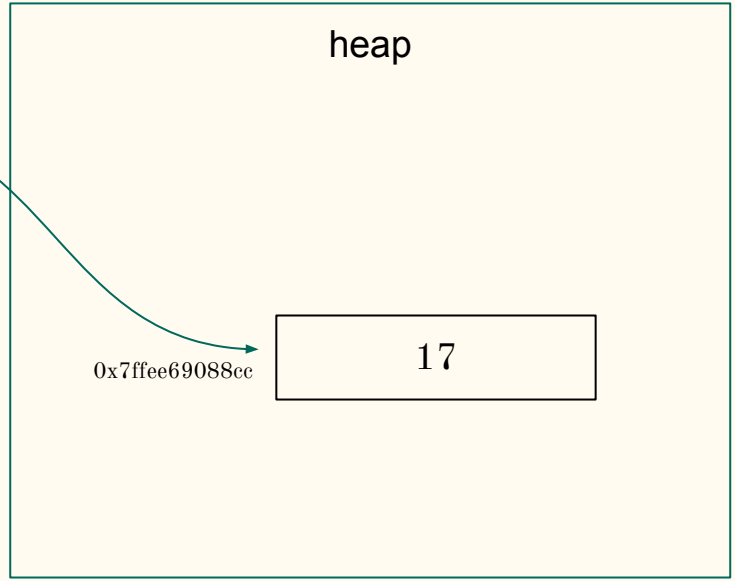
private:
    int* ptr;

};
```

A class with dynamically allocated memory

```
class SimpleClass {
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
void a_function() {
   SimpleClass simp;
   …

}
```



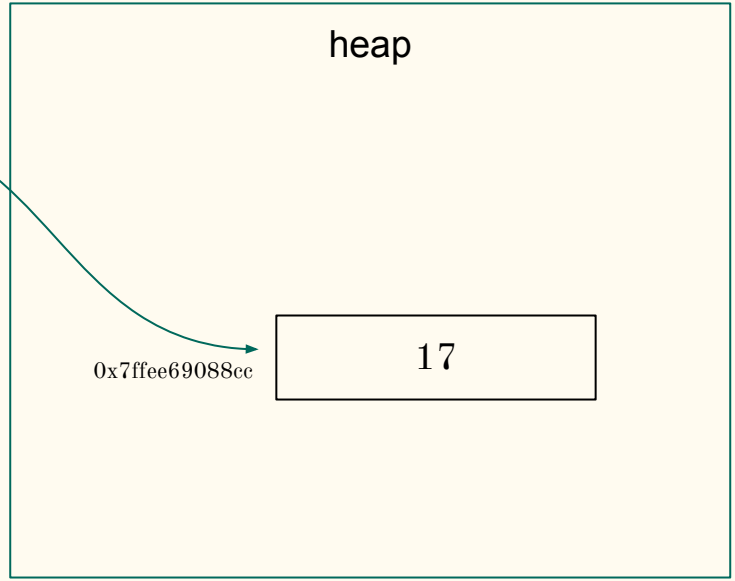
~SimpleClass() called when simp goes "out of scope"

A class with dynamically allocated memory

```
class SimpleClass {
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
void a_function() {
   SimpleClass simp;
   …

}
int main() {
   a_function();
   …
}
```

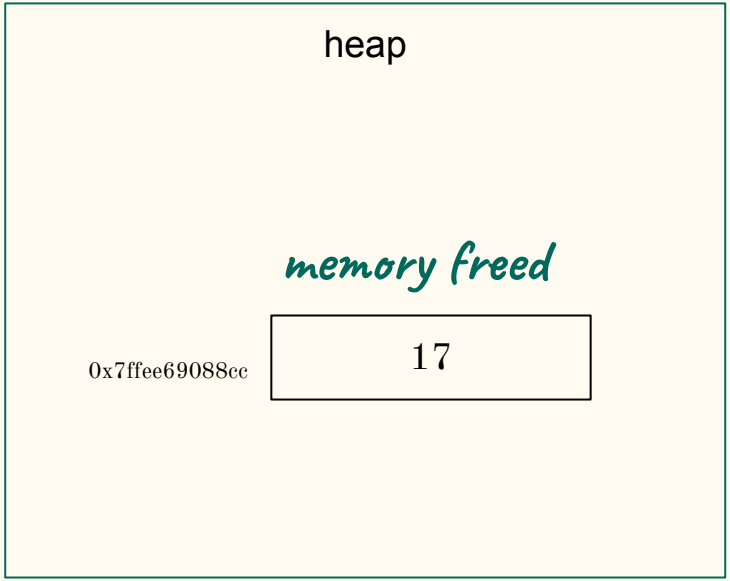
~SimpleClass() called



A class with dynamically allocated memory

```
class SimpleClass {
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
void a_function() {
   SimpleClass simp;
   …

}
int main() {
   a_function();
   …
}
```



The Big 3

- destructor typically implemented with 2 other class components
  - copy constructor
  - assignment operator
- destructor, copy constructor, assignment operator known as "Big 3"
- classes needing 1 implemented typically require other 2

The copy constructor

The copy constructor

parameters provide initial values for object

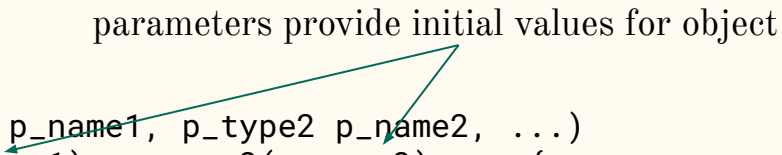
```
class ClassName {
public:
    ClassName(p_type1 p_name1, p_type2 p_name2, ...)
        : m_name1(p_name1), m_name2(p_name2),... {
        /* constructor body */
    }

    ~ClassName() {
       /* free allocated resources */
    }

private:
    m_type1 m_name1;
    m_type2 m_name2;
    ...

};
```

The copy constructor




```
class ClassName {
public:
    ClassName(p_type1 p_name1, p_type2 p_name2, ...)
        : m_name1(p_name1), m_name2(p_name2),... {
        /* constructor body */
    }

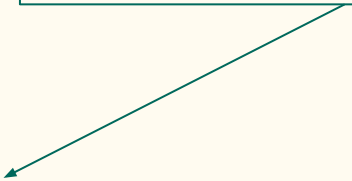
    ~ClassName() {
       /* free allocated resources */
    }

private:
    m_type1 m_name1;
    m_type2 m_name2;
    ...

};
```

The copy constructor

- a copy constructor allows object of same type as parameter



```
class ClassName {
public:
    ...

    ClassName(const ClassName& other) {
        /* initialize object using other */
    }

    ...
private:
    m_type1 m_name1;
    m_type2 m_name2;
    ...

};
```

The copy constructor

- a copy constructor allows object of same type as parameter
- utilized often
  - `SomeClass obj_a;`
    `SomeClass obj_b(obj_a);`
  - `SomeClass obj_a;`
    `SomeClass obj_b = obj_a;`

`obj_b` initialized by object of same type (`obj_a`)
-- both use copy constructor

The copy constructor

- a copy constructor allows object of same type as parameter
- utilized often
  - ```
    SomeClass obj_a;
    SomeClass obj_b(obj_a);
    ```
  - ```
    SomeClass obj_a;
    SomeClass obj_b = obj_a;
    ```
  - ```
    void some_func(SomeClass passed_by_value) {}
    ```

copy constructor used to initialize parameter when object passed-by-value
    ```

The copy constructor

- a copy constructor allows object of same type as parameter
- utilized often
  - `SomeClass obj_a;`
    `SomeClass obj_b(obj_a);`
  - `SomeClass obj_a;`
    `SomeClass obj_b = obj_a;`
  - `void some_func(SomeClass passed_by_value) {}`
  - `SomeClass another_func() {`

```
    SomeClass returned_by_value;

    return returned_by_value;

}
```

copy constructor used to create a copy of an object when returning by value

The copy constructor

- a copy constructor allows object of same type as parameter
- utilized often
- a copy constructor provided automatically by compiler

```
class SimpleClass {
    friend
    ostream& operator<<(ostream& os, const SimpleClass& rhs) {
        os << rhs.ptr << ' ' << *rhs.ptr;
        return os;
    }
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
```

```
int main() {
    SimpleClass simp1;

    SimpleClass simp2(simp1);

    cout << "1. " << simp1 << endl;
    cout << "2. " << simp2 << endl;

}
```

```
% g++ -std=c++11 simple.cpp -o simple.o
% ./simple.o
1. 0x7f8c37c05990 17
2. 0x7f8c37c05990 17
```

The copy constructor

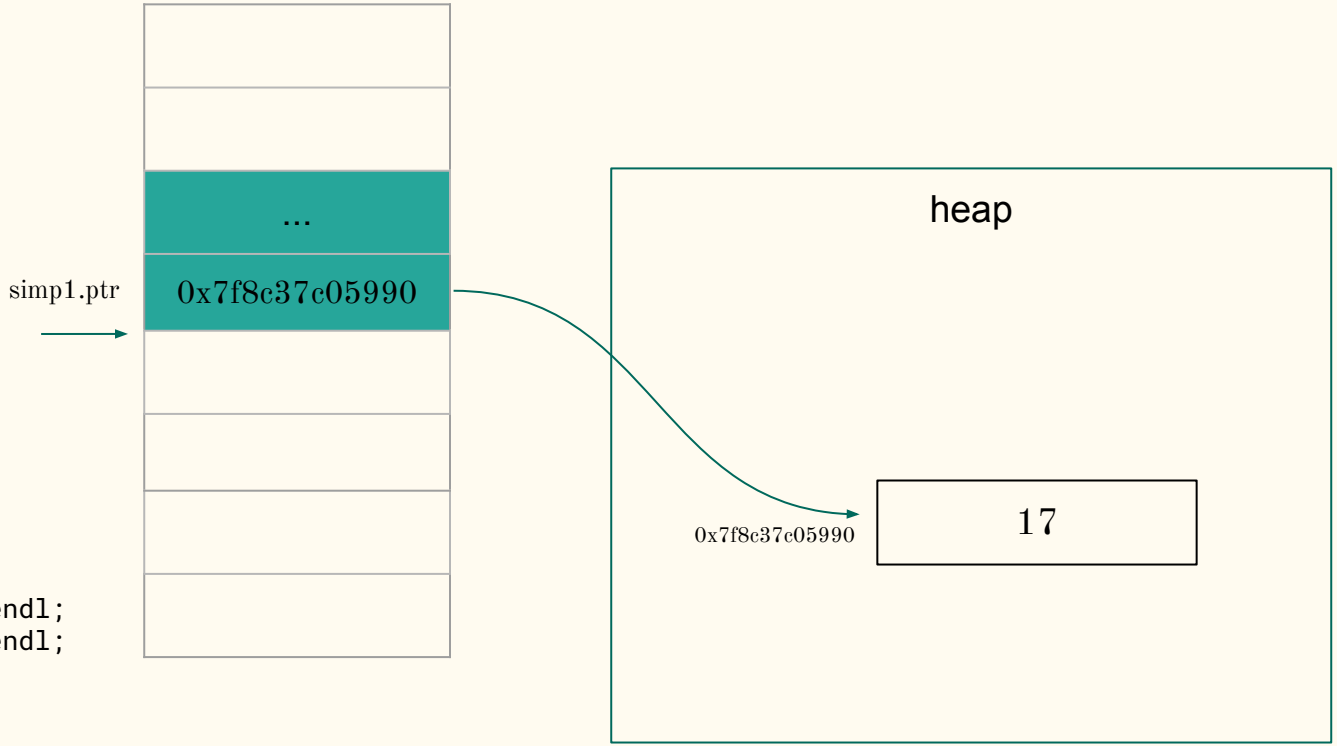
```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};


int main() {
    SimpleClass simp1;

    SimpleClass simp2(simp1);

    cout << "1. " << simp1 << endl;
    cout << "2. " << simp2 << endl;

}
```



The copy constructor

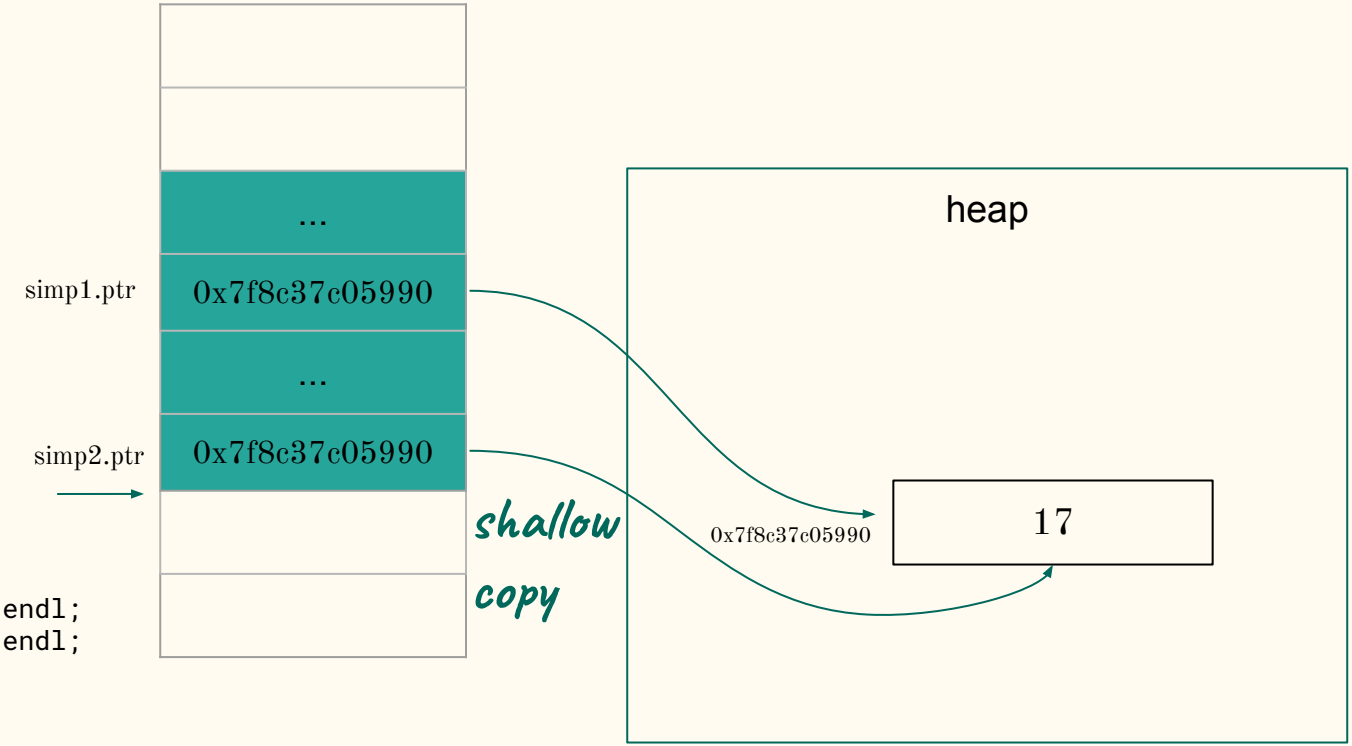
```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};


int main() {
    SimpleClass simp1;

    SimpleClass simp2(simp1);

    cout << "1. " << simp1 << endl;
    cout << "2. " << simp2 << endl;

}
```



The copy constructor

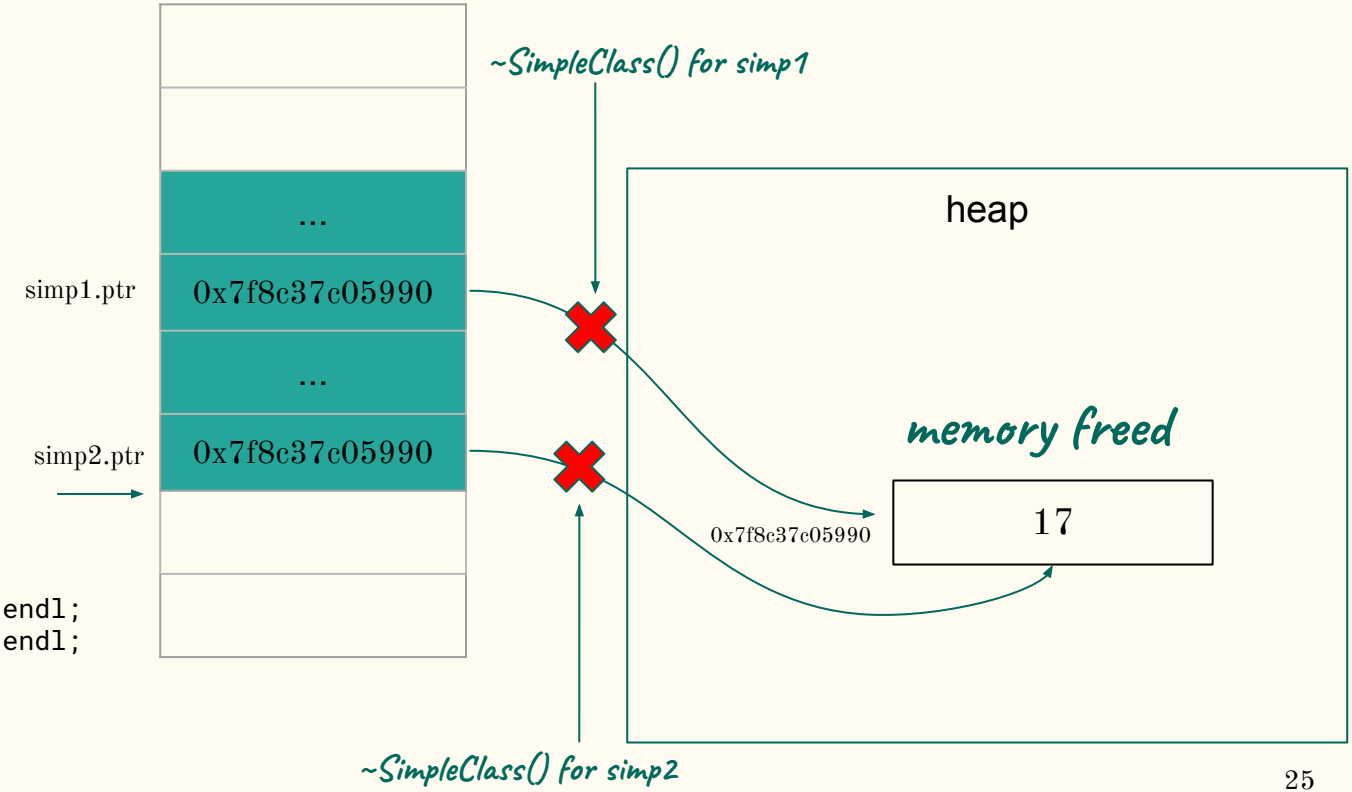
```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};


int main() {
    SimpleClass simp1;

    SimpleClass simp2(simp1);

    cout << "1. " << simp1 << endl;
    cout << "2. " << simp2 << endl;

}
```



The copy constructor

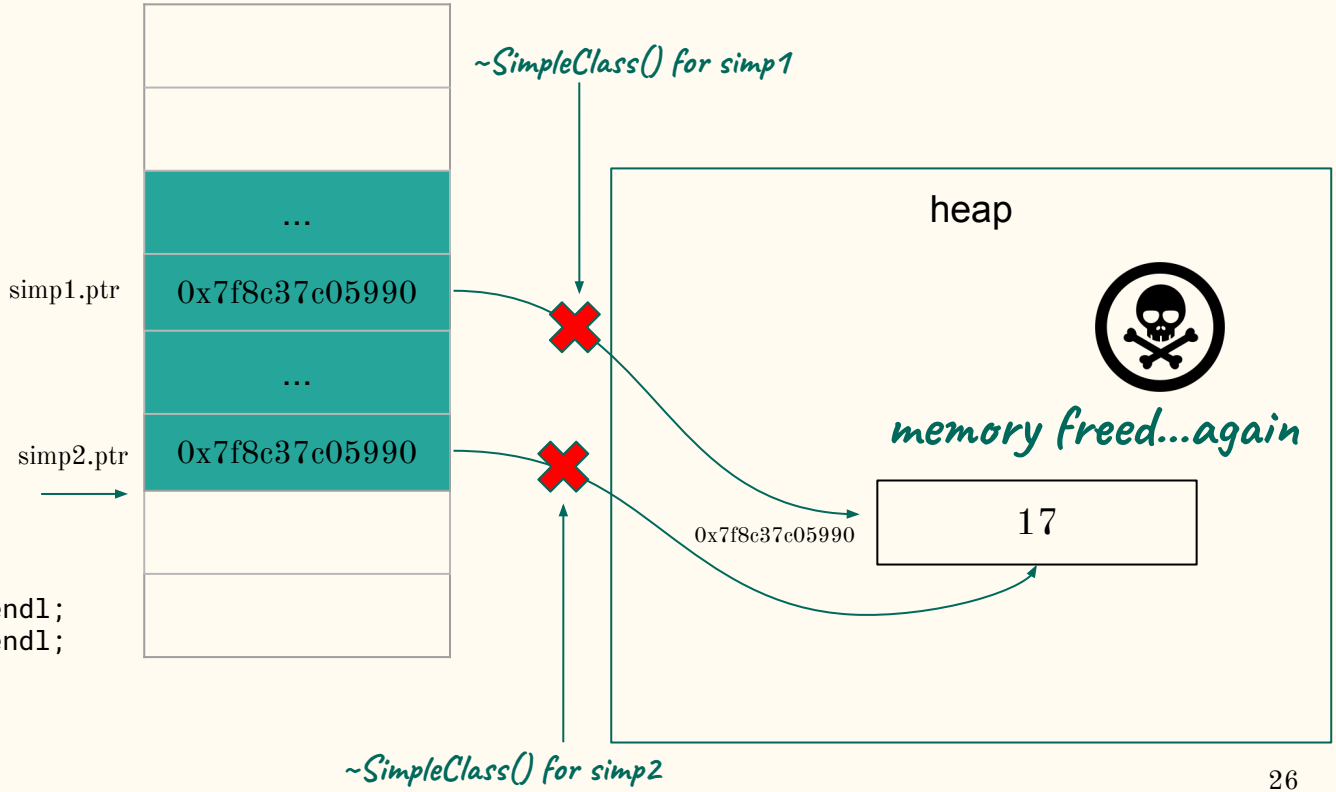
```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};


int main() {
    SimpleClass simp1;

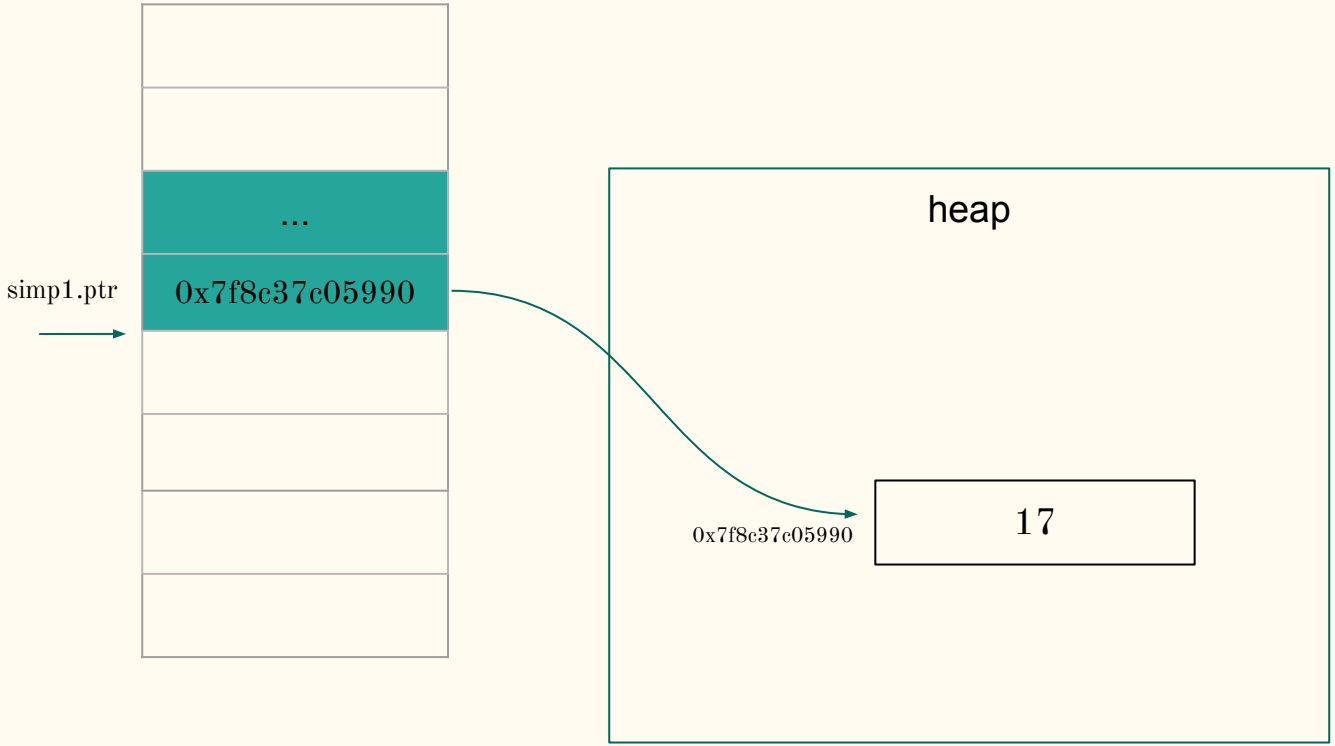
    SimpleClass simp2(simp1);

    cout << "1. " << simp1 << endl;
    cout << "2. " << simp2 << endl;

}
```

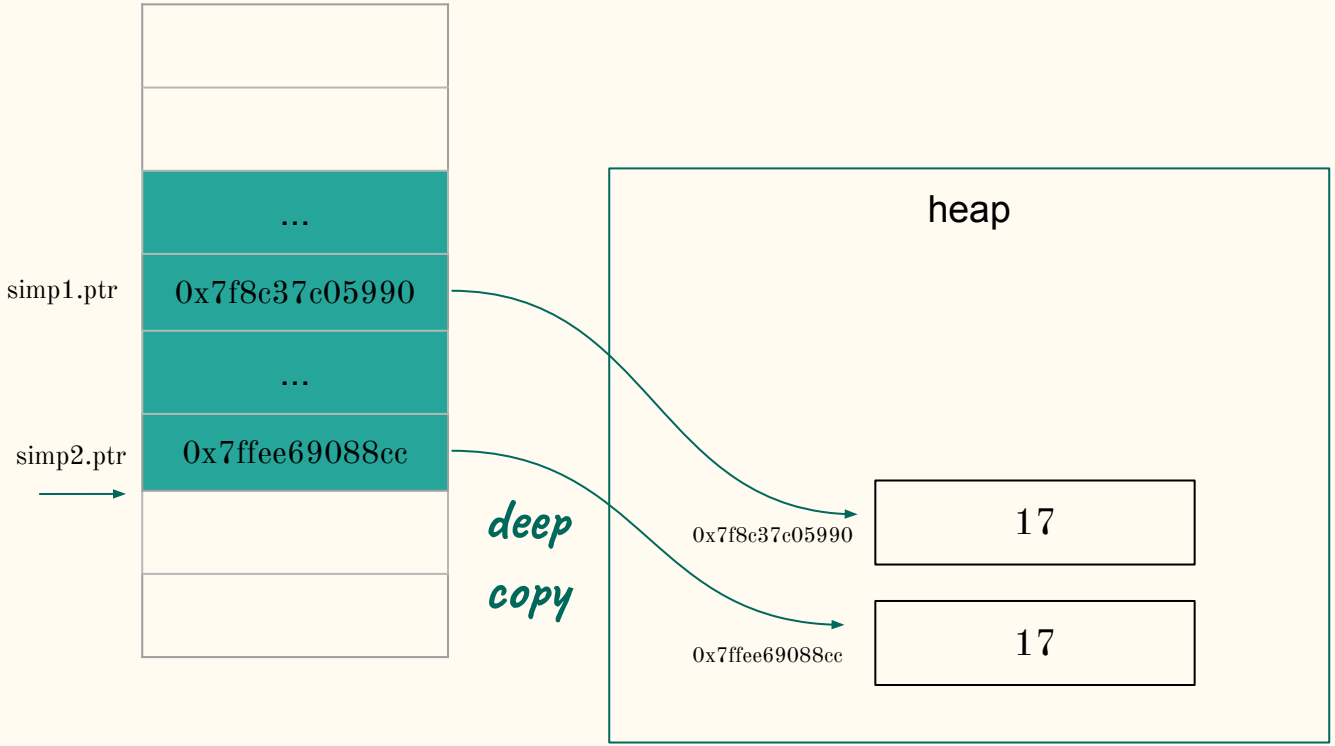


The copy constructor



The copy constructor

```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }
    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
```

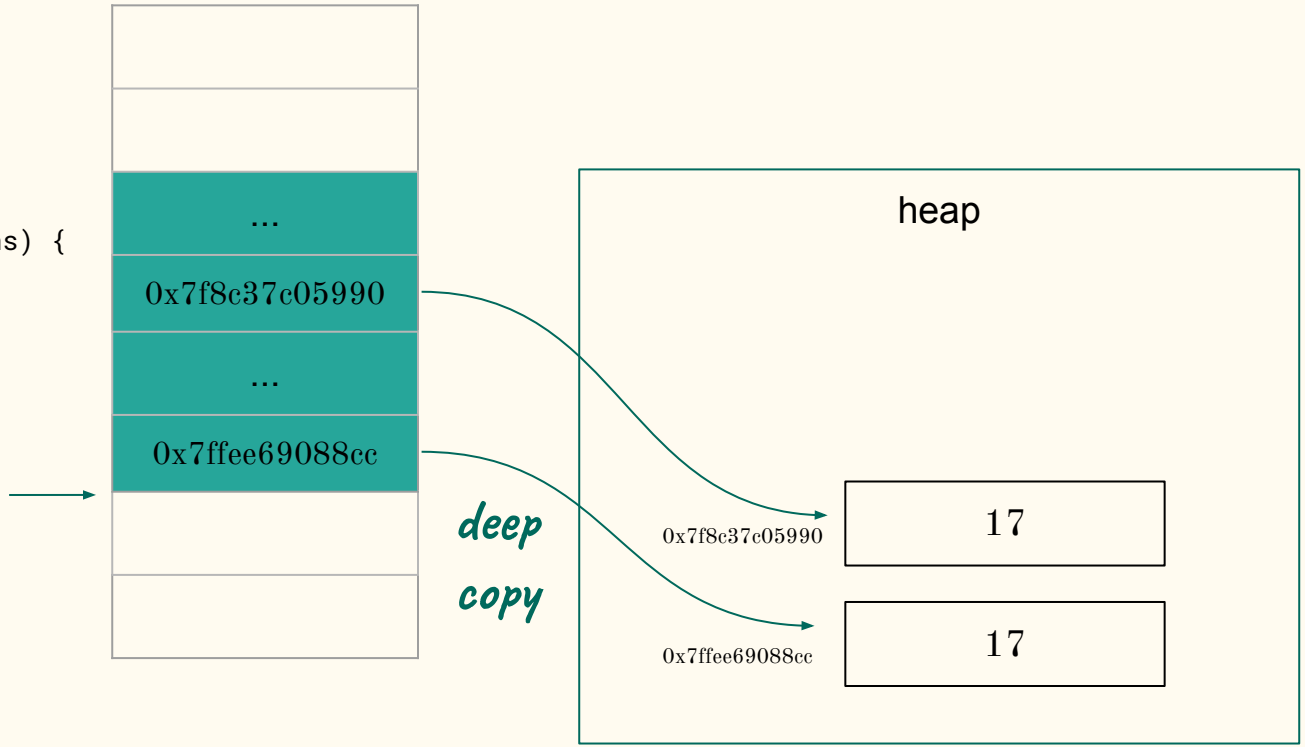


The copy constructor

```
class SimpleClass {
    ...
public:
    SimpleClass() {
        ptr = new int(17);
    }

    SimpleClass(const SimpleClass& rhs) {
        ptr = new int;
        *ptr = *rhs.ptr;
    }

    ~SimpleClass() {
        delete ptr;
    }
private:
    int* ptr;
};
```



In-class problem

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    /* private member variables */
};
```

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    // pointer to integer
};
```

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    // pointer to integer
    ---
};
```

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    // pointer to integer
    _1_
};
```

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220216<A|D>

Replace <A|D> with this section's letter

Which variable declaration replaces blank #1 to create an integer pointer named `i_ptr`;

```
class Thing {

public:
    /* public interface implementation */

private:
    // pointer to integer
    _1_
};
```

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    // pointer to integer
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    /* public interface implementation */

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    ___()

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    _2_()

private:
    int* i_ptr;
};
```

What name do we give to this constructor (replacing blank #2)?

```
class Thing {

public:
    // define a simple constructor to assign int's value
    _2_()

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing() {}

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(___ val) {}

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(_3_ val) {}

private:
    int* i_ptr;
};
```

Which type (replacing blank #3) do we declare for `val` to use this parameter for initializing the value *pointed to* by `i_ptr`?

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(_3_ val) {}

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) {}

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = ___; }

private:
    int* i_ptr;
};
```

`i_ptr` is a pointer type, what kind of value will it hold?

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = ___; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = ___ ___; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = _5_ ___; }

private:
    int* i_ptr;
};
```

To assign memory from the heap for the integer, which operator must replace blank #5?

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = _5_ ___; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = new ___; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = new _6_; }

private:
    int* i_ptr;
};
```

Which expression replaces blank #6 to create an integer initialized to `val`?

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = new _6_; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    // define a simple constructor to assign int's value
    Thing(int val) { i_ptr = new int(val); }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ___()

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    _7_()

private:
    int* i_ptr;
};
```

Which name is used for the destructor of the `Thing` class?

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    _7_()

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing()

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing() {}

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing() { ___ }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing() { _8_ }

private:
    int* i_ptr;
};
```

Which statement replaces blank #8 to free the memory allocated by the constructor?

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing() { _8_ }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    // define class destructor
    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {
    // implement operator<<

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {
    // implement operator<<
    ---

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {
    // implement operator<<
    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << ___;
    }

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {
    // implement operator<<
    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << _9_;
    }

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Which expression replaces blank #9 to insert the value pointed to by `i_ptr` to the `ostream& os`?

```
class Thing {
    // implement operator<<
    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << _9_;
    }

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {
    // implement operator<<
    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }

public:
    Thing(int val) { i_ptr = new int(val); }

    ~Thing() { delete i_ptr; }

private:
    int* i_ptr;
};
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   // double int associated with Thing
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

compilation error

Which type of method do we need to add to the `Thing` class's public interface to access the value pointed to by `i_ptr`?

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

compilation error

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

compilation error

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    ___

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

compilation error

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() { return ___; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

    // double int associated with Thing
    int result = *a_thing.i_ptr * 2;
}
```

compilation error

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() { return _10_; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

    // double int associated with Thing
    int result = *a_thing.i_ptr * 2;
}
```

compilation error

Which expression replaces blank #10 to return the integer pointed to by `i_ptr`?

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() { return _10_; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

compilation error

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() _11_ { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

Which keyword replaces blank #11 to indicate that the `get_value()` method does not modify the `Thing` object?

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() _11_ { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // accessor method for int pointed to by i_ptr
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   // double int associated with Thing
   int result = *a_thing.i_ptr * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

    // double int associated with Thing
    int result = a_thing.get_value() * 2;
}
```

~~compilation error~~

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int main() {
   Thing a_thing(100);

   int result = a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};



int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // define mutator method to change value of integer pointed to by i_ptr
    ___
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // define mutator method to change value of integer pointed to by i_ptr
    void set_value(int val) { ___ }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // define mutator method to change value of integer pointed to by i_ptr
    void set_value(int val) { _12_ }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Which statement changes the value of the integer pointed to by `i_ptr` to `val`?

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // define mutator method to change value of integer pointed to by i_ptr
    void set_value(int val) { _12_ }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    // define mutator method to change value of integer pointed to by i_ptr
    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {

    friend ostream& operator<<(ostream& os, const Thing& rhs) {
        return os << "Thing: " << *rhs.i_ptr;
    }
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing() { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
}
```

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

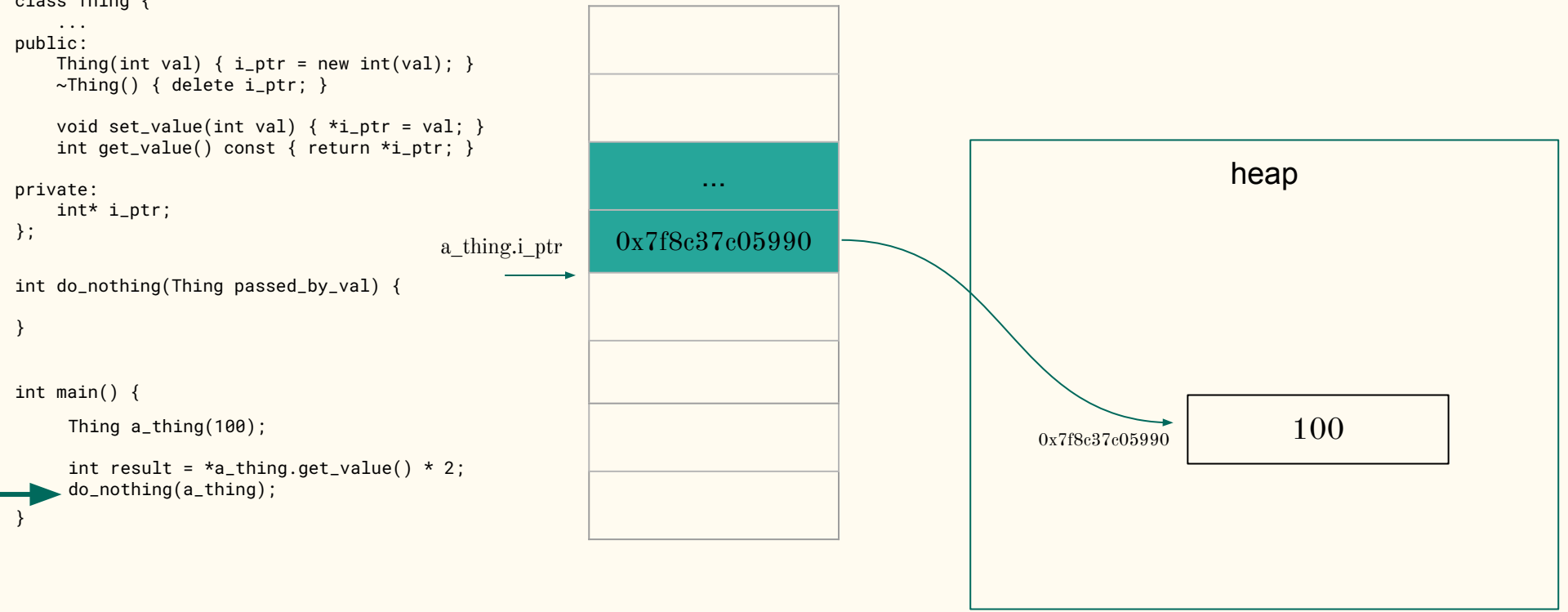
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) {

}


int main() {
     Thing a_thing(100);

     int result = *a_thing.get_value() * 2;
     do_nothing(a_thing);
}
```



Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

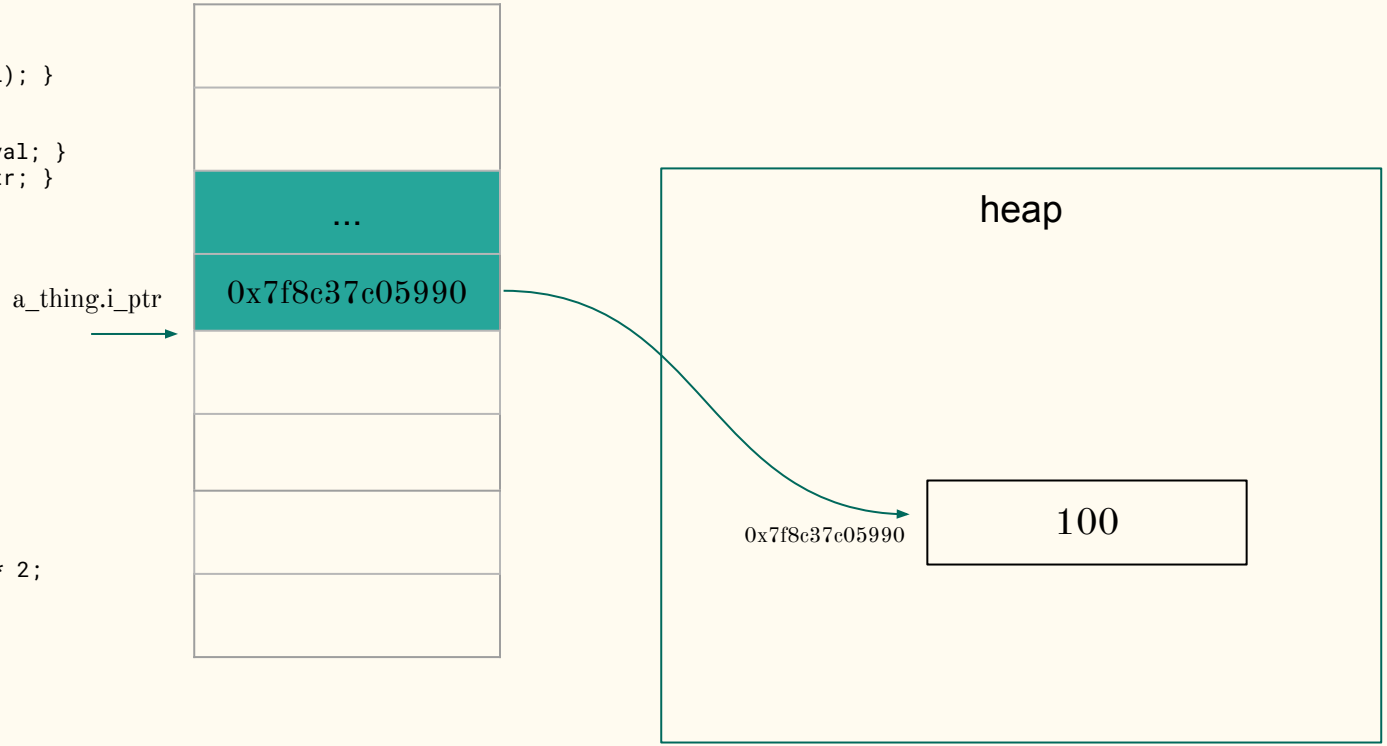
int do_nothing(Thing passed_by_val) {

}


int main() {

     Thing a_thing(100);

     int result = *a_thing.get_value() * 2;
     do_nothing(a_thing);
}
```



Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

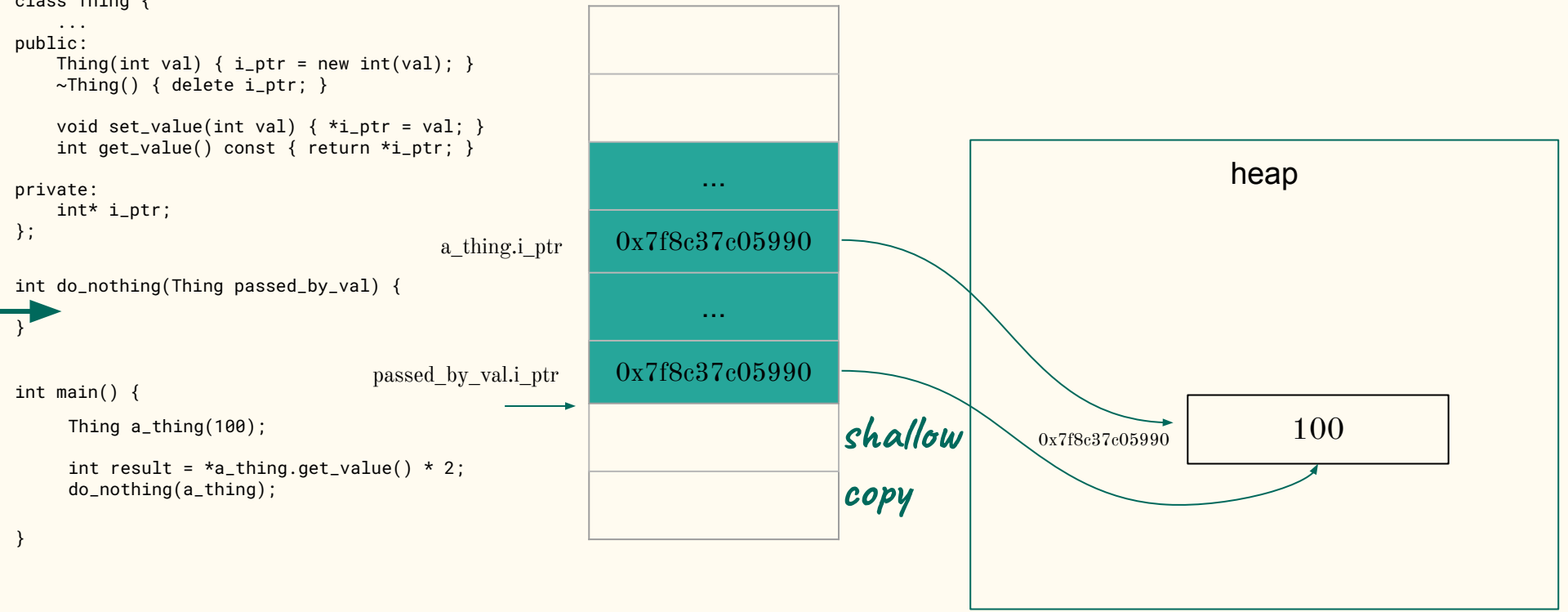
int do_nothing(Thing passed_by_val) {

}

int main() {
     Thing a_thing(100);

     int result = *a_thing.get_value() * 2;
     do_nothing(a_thing);

}
```



Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) {

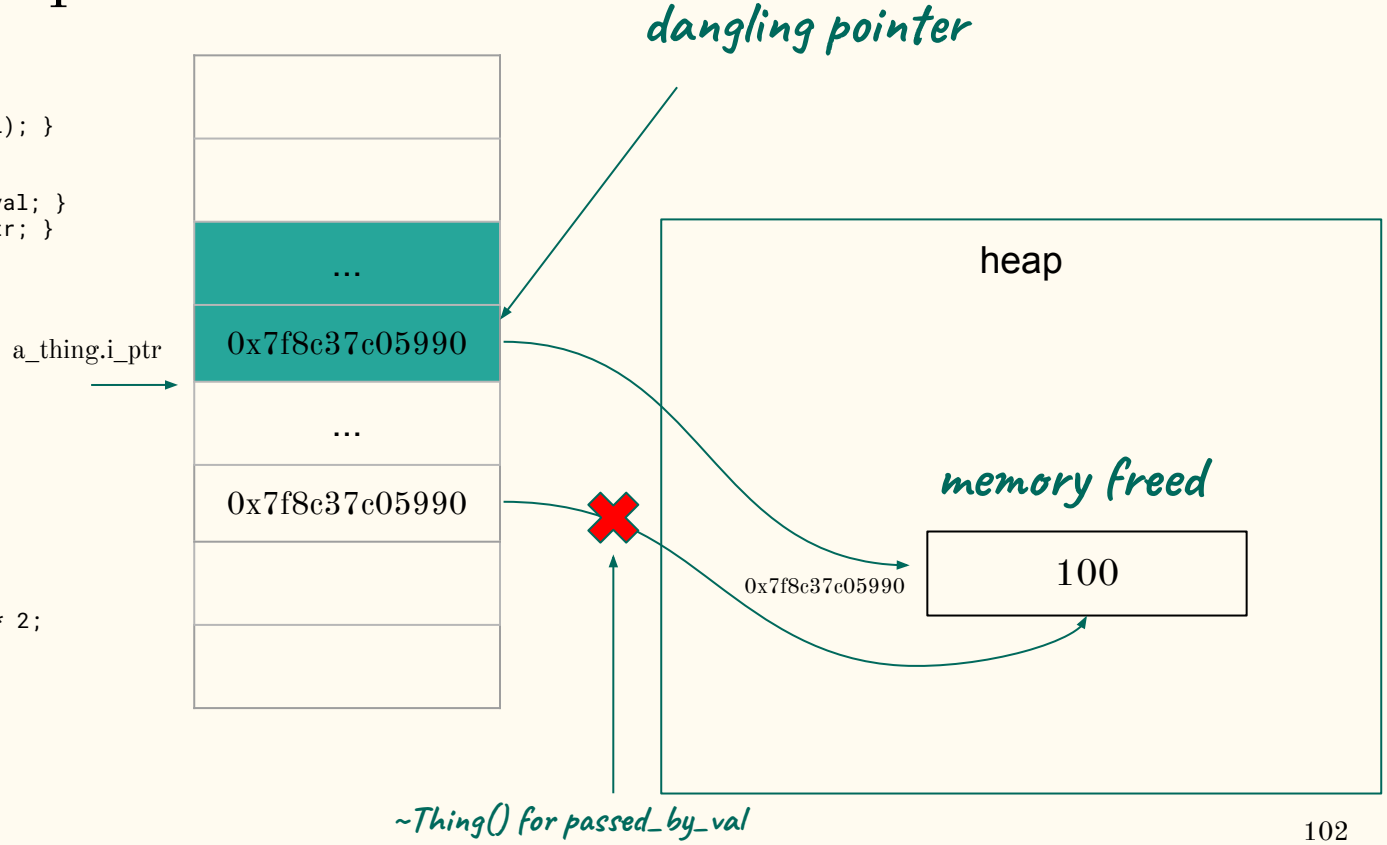
}


int main() {

     Thing a_thing(100);

     int result = *a_thing.get_value() * 2;
     do_nothing(a_thing);

}
```



Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};


int do_nothing(Thing passed_by_val) {

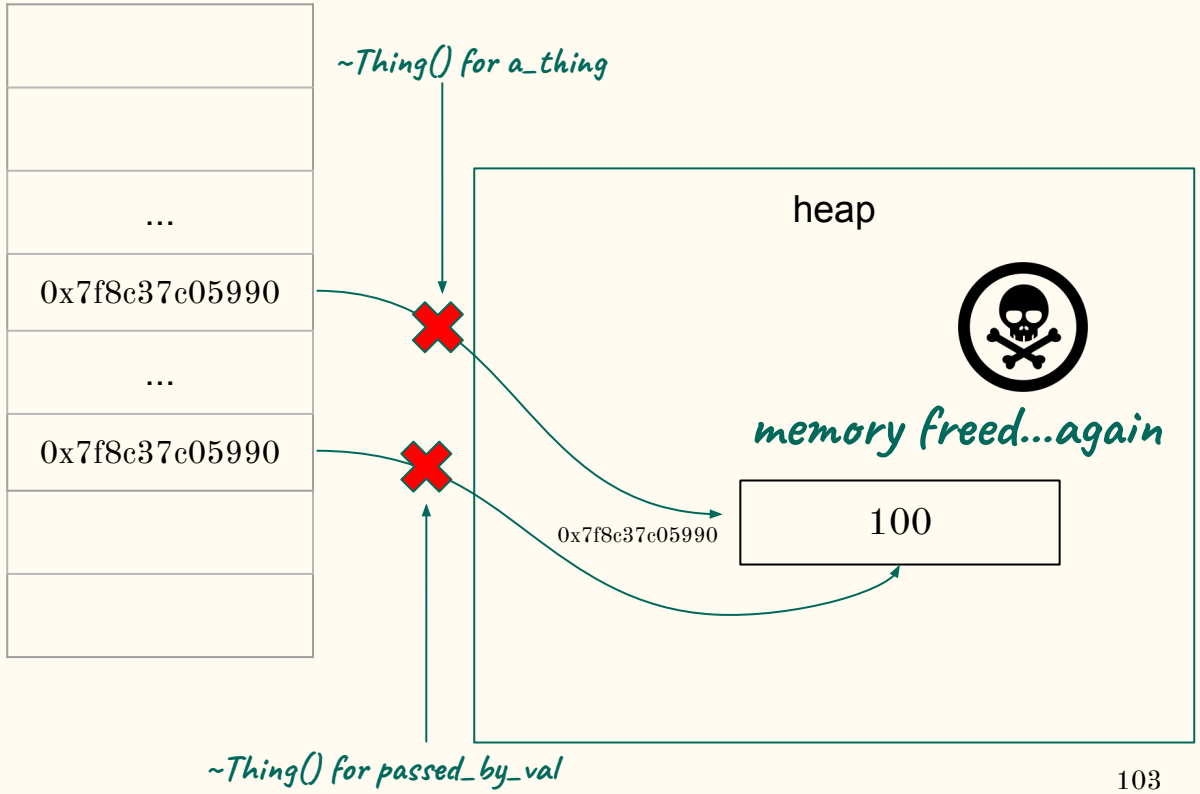
}


int main() {

     Thing a_thing(100);

     int result = *a_thing.get_value() * 2;
     do_nothing(a_thing);

}
```



Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

What needs to be added to the Thing class to avoid the memory issues described?

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }

    void set_value(int val) { *i_ptr = val; }
    int get_value() const { return *i_ptr; }

private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor

private:
    int* i_ptr;
};


int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    ___
private:
    int* i_ptr;
};


int do_nothing(Thing passed_by_val) { }

int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(___ another_thing) {

    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(___ another_thing) {

    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(_12_ another_thing) {

    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

What is the type of the `another_thing` parameter of the copy constructor (replacing blank #12)?

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(_12_ another_thing) {

    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {

    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {
        i_ptr = new int(___); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {
        i_ptr = new int(_13_); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

What expression is passed to the `int` constructor to make a deep copy of the integer associated with `another_thing` (replacing blank #13)?

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {
        i_ptr = new int(_13_); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {
        i_ptr = new int(_13_); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...
    // define copy constructor
    Thing(const Thing& another_thing) {
        i_ptr = new int(*another_thing.i_ptr); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

memory freed twice

Copy control implementation

```
class Thing {
    ...
public:
    Thing(int val) { i_ptr = new int(val); }
    ~Thing() { delete i_ptr; }
    ...

    Thing(const Thing& another_thing) {
        i_ptr = new int(*another_thing.i_ptr); // deep copy
    }
private:
    int* i_ptr;
};

int do_nothing(Thing passed_by_val) { }
int main() {
   Thing a_thing(100);

   int result = *a_thing.get_value() * 2;
   do_nothing(a_thing);
}
```

~~memory freed twice~~