

# Copying versus moving

Bachelor of Science - École polytechnique

[gael.thomas@inria.fr](mailto:gael.thomas@inria.fr)

# Key concepts

- The copy constructor
  - Is used to deeply copy an object
  - `holder_t(const holder_t& t)`
- The move constructor
  - Is used to move an object into another
  - Avoid a deep copy
  - Called when the parameter will be destroyed after the call
  - `holder_t(holder_t&& t)`
  - Don't forget to nullify the elements of `t` that are deleted in the destructor

# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```

the “copy constructor”  
performs a deep copy of t

# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

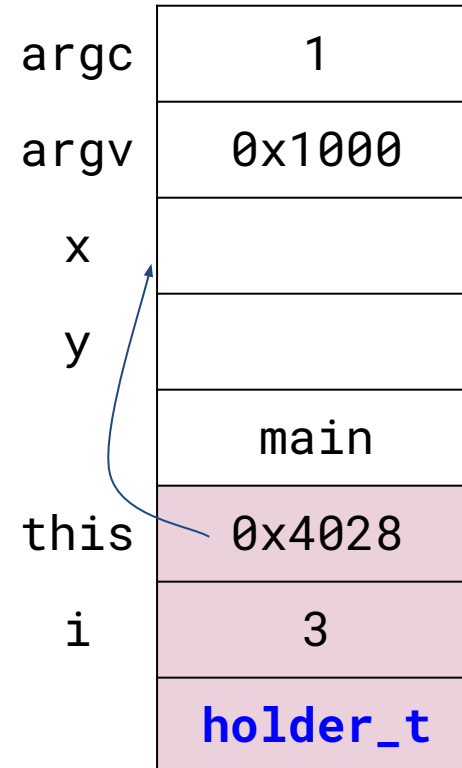
int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```

argc	1
argv	0x1000
x	
y	
	main

# Copy constructor

- Copying an object means performing a deep copy

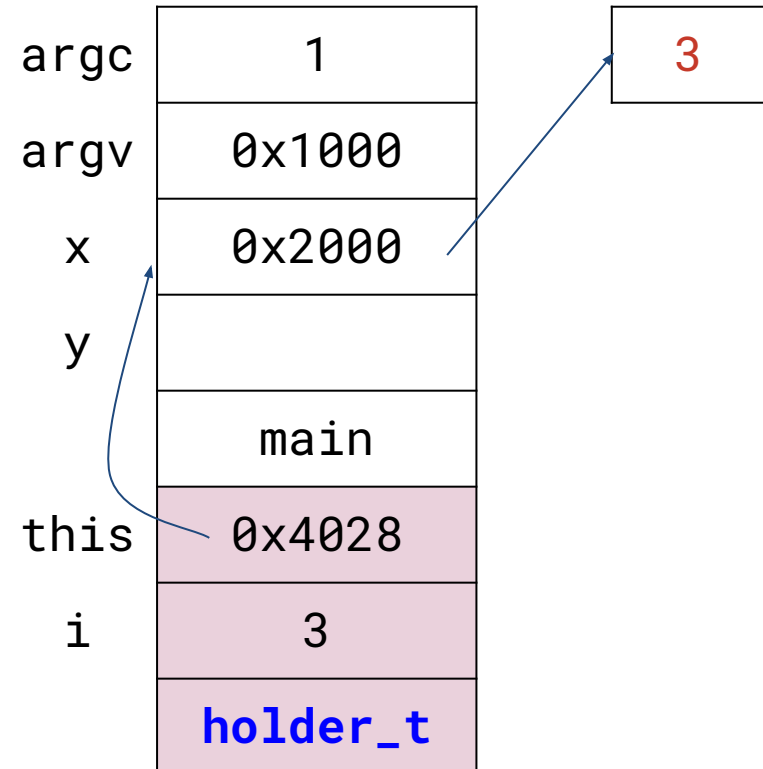
```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```



# Copy constructor

- Copying an object means performing a deep copy

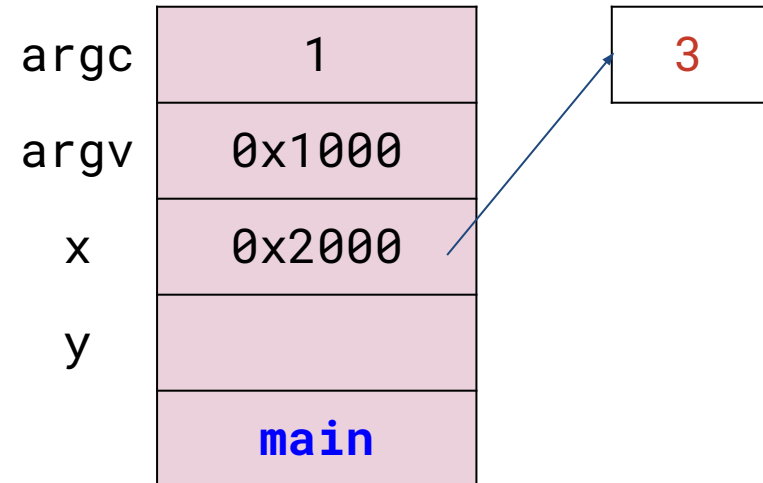
```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```



# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```



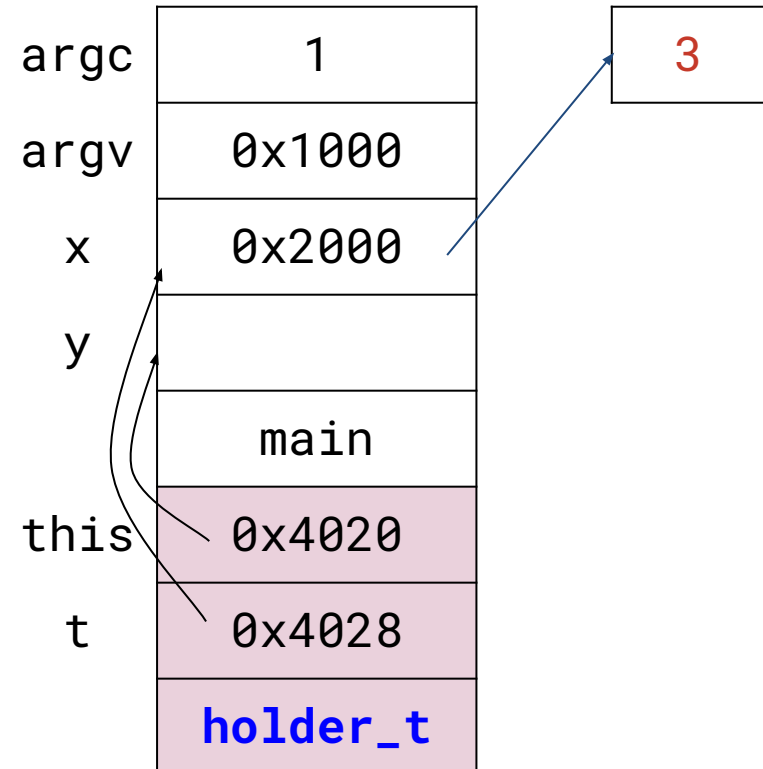
# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```





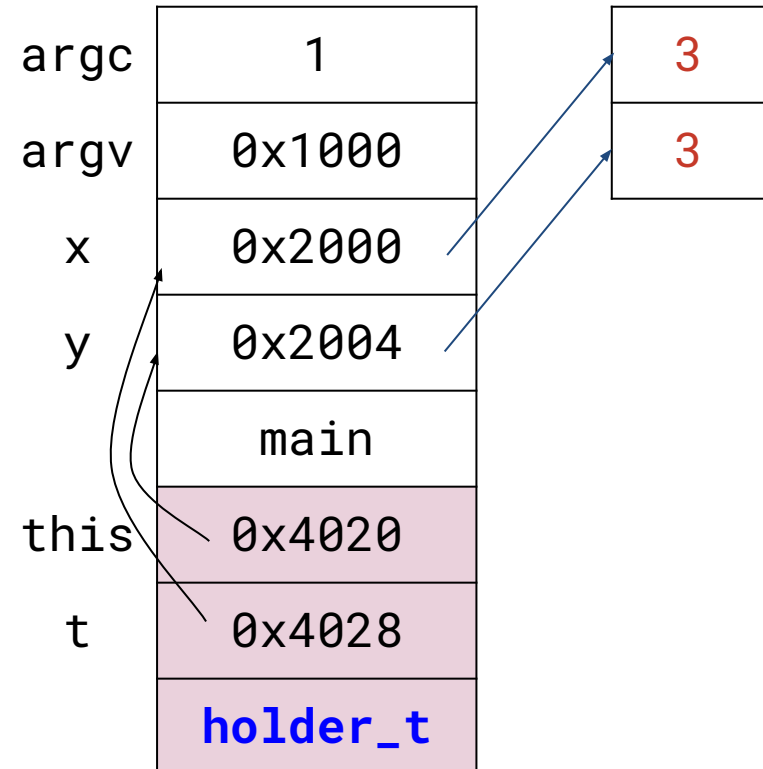
# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```



deep copy of t => allocate  
memory and copy \*t.p  
into \*this->p

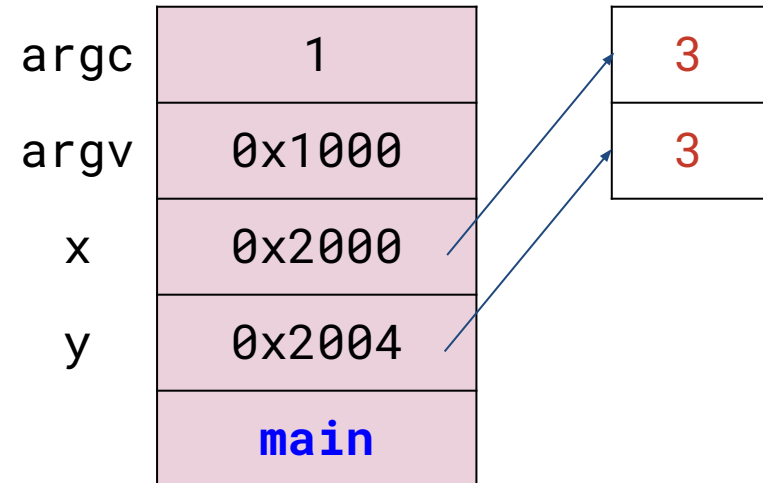
# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```



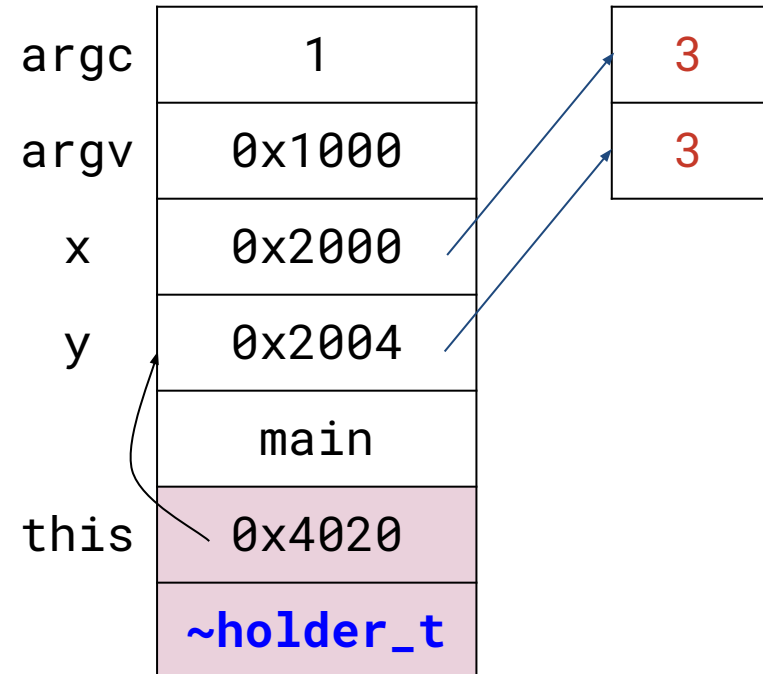
# Copy constructor

- Copying an object means performing a deep copy

```
class holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

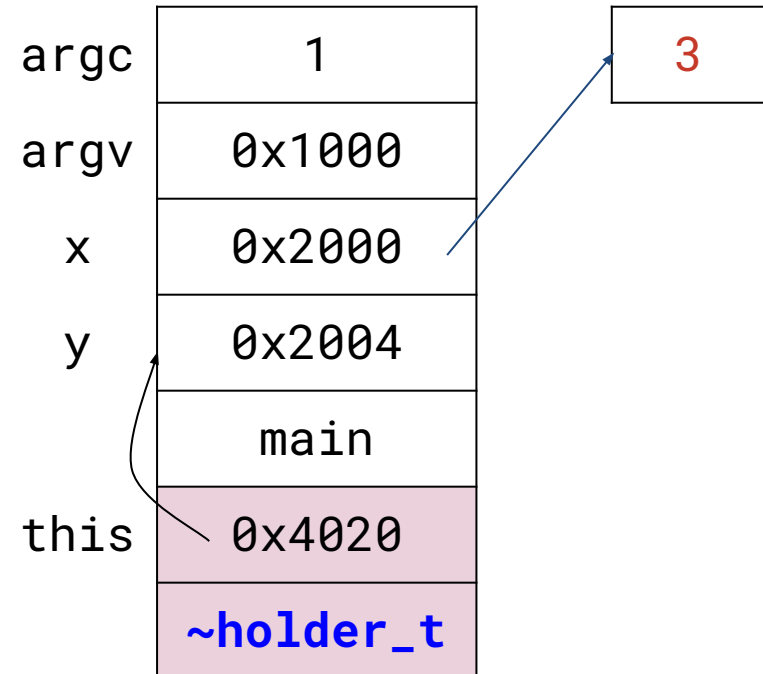
int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```



# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```



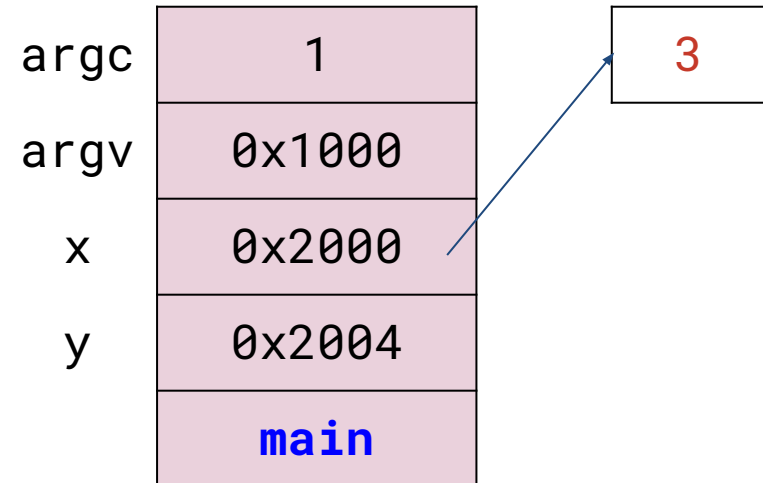
# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```



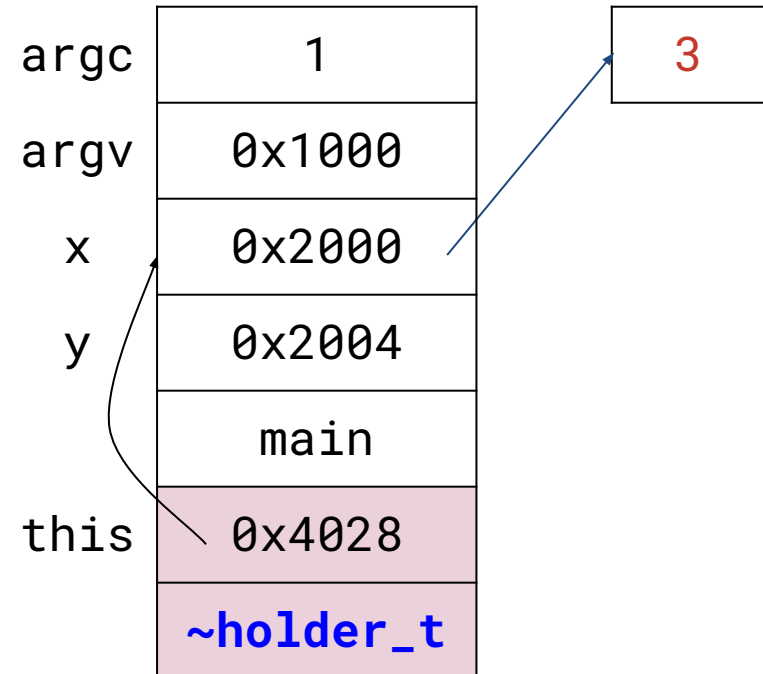
# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

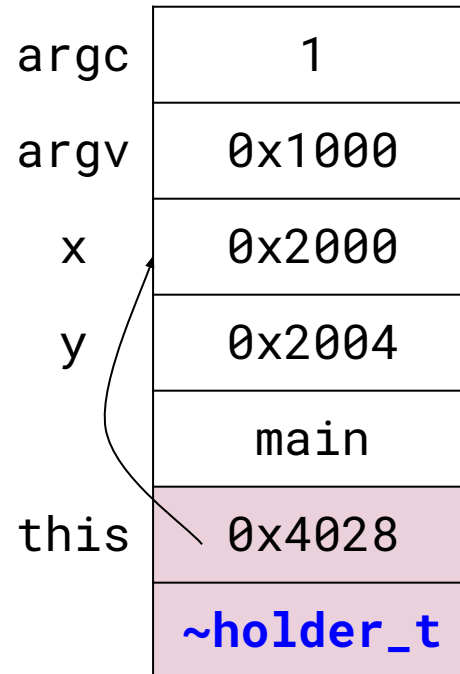
int main(int argc, char* argv[]) {
    holder_t x { 3 };
    holder_t y { x };
    return 0;
}
```



# Copy constructor

- Copying an object means performing a deep copy

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(const holder_t& t)  
        : p { new int { *t.p } } {};  
    ~holder_t() { delete p; }  
};  
  
int main(int argc, char* argv[]) {  
    holder_t x { 3 };  
    holder_t y { x };  
    return 0;  
}
```



# A deep copy is sometimes inefficient

- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# A deep copy is sometimes inefficient

- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```

argc	1
argv	0x1000
x	
	main

# A deep copy is sometimes inefficient

- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```

argc	1
argv	0x1000
x	
	main
h	
	self

Smartly the compiler avoids a copy by directly constructing the parameter of `self` from `3`

# A deep copy is sometimes inefficient

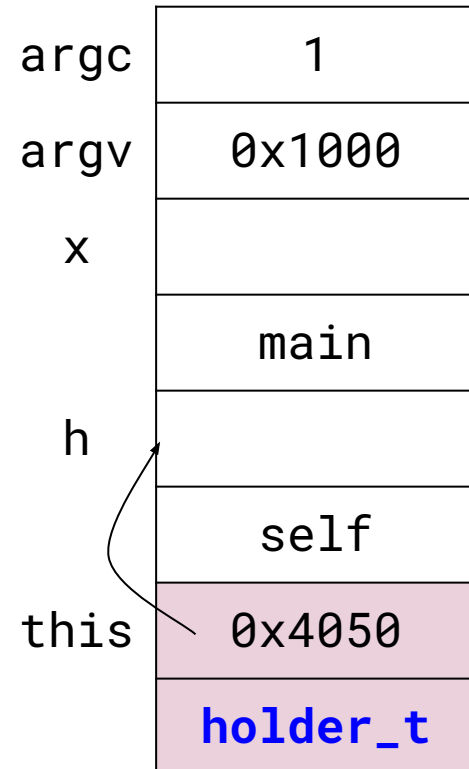
- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# A deep copy is sometimes inefficient

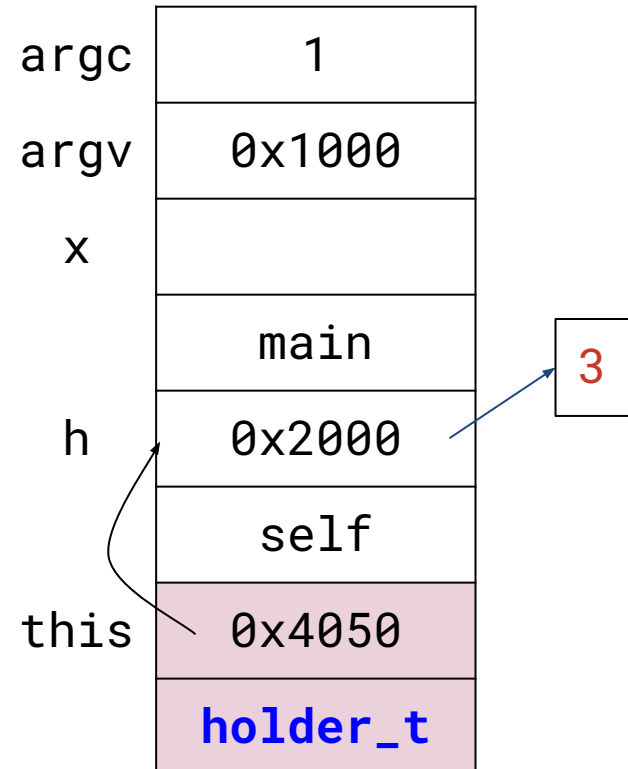
- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# A deep copy is sometimes inefficient

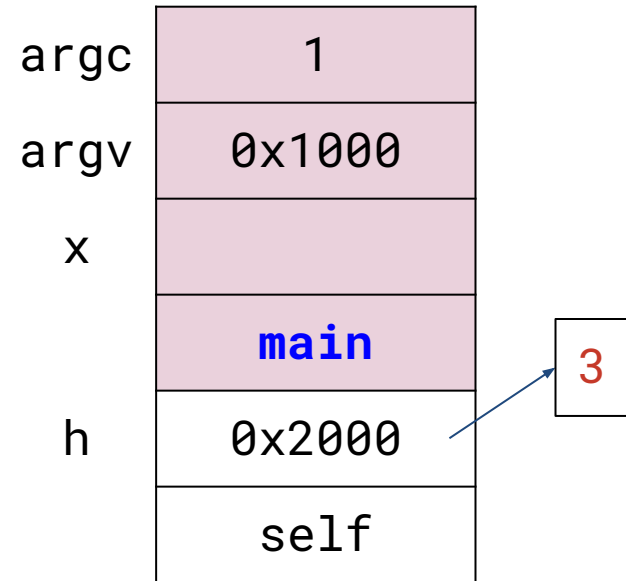
- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# A deep copy is sometimes inefficient

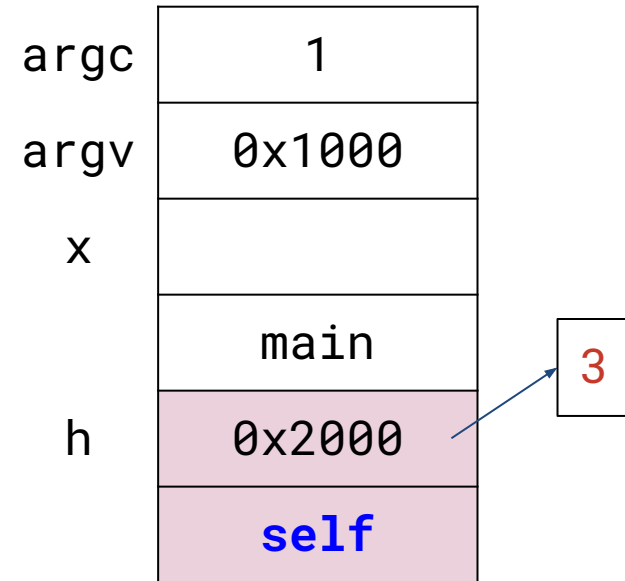
- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



Smartly, the compiler avoids a copy by directly constructing the result in the caller

# A deep copy is sometimes inefficient

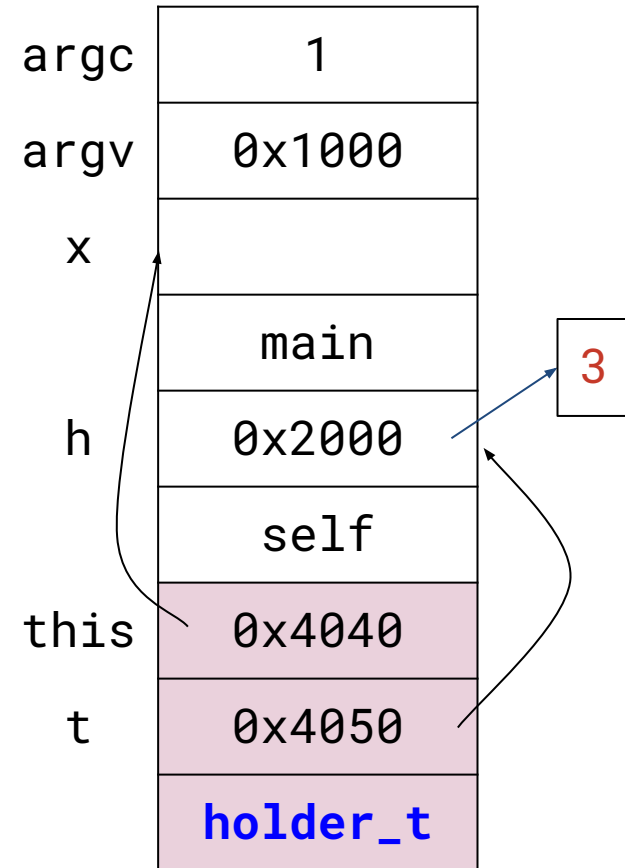
- Useless copies since at each time a single instance exists

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# A deep copy is sometimes inefficient

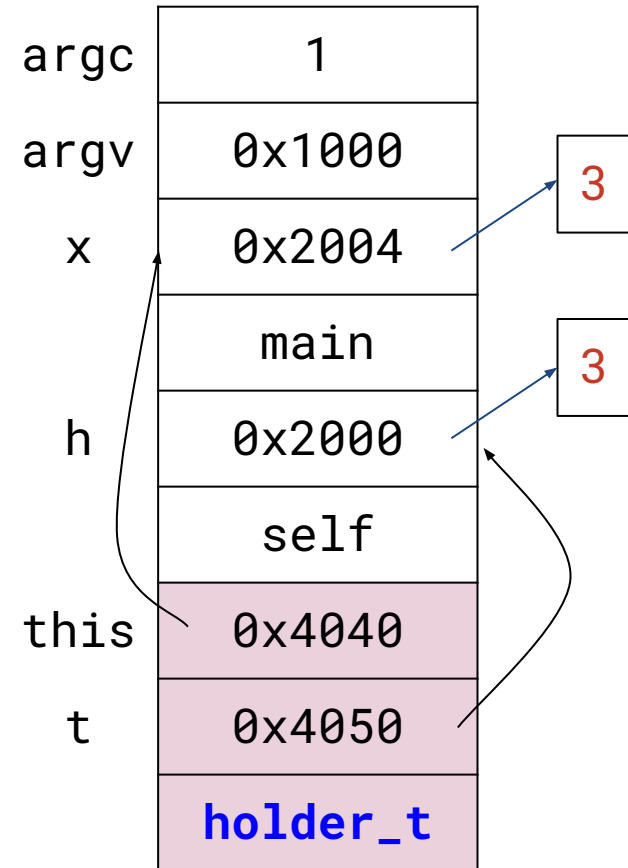
- The deep copy here is useless since h will be destroyed after

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(const holder_t& t)
        : p { new int { *t.p } } {};
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```





# The move constructor

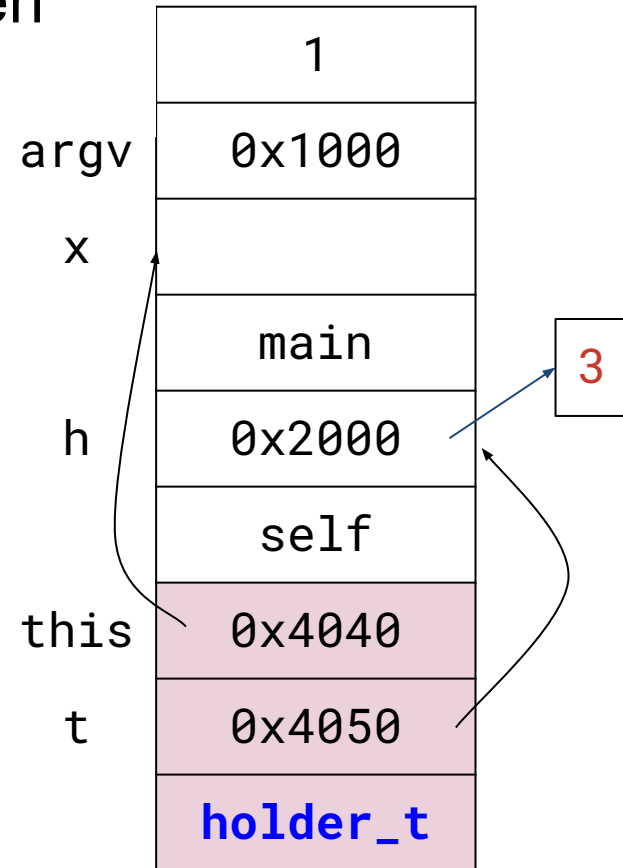
- A move constructor receives a `holder_t&& t` used instead of the copy constructor when the parameter is destroyed just after

S

```
holder_t(int i)
    : p { new int { i } } {}
holder_t(holder_t&& t)
    : p { t.p } { t.p = nullptr; };
~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

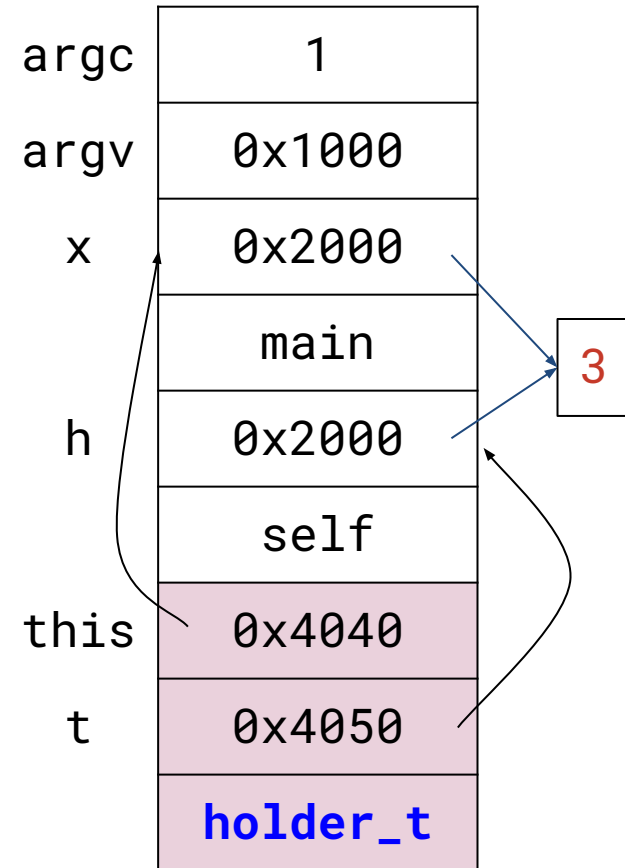
int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# The move constructor

- A move constructor receives a `holder_t&& t`

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(holder_t&& t)  
        : p { t.p } { t.p = nullptr; };  
    ~holder_t() { delete p; }  
};  
  
holder_t self(holder_t h) { return h; }  
  
int main(int argc, char* argv[]) {  
    holder_t x { self(3) };  
    return 0;  
}
```



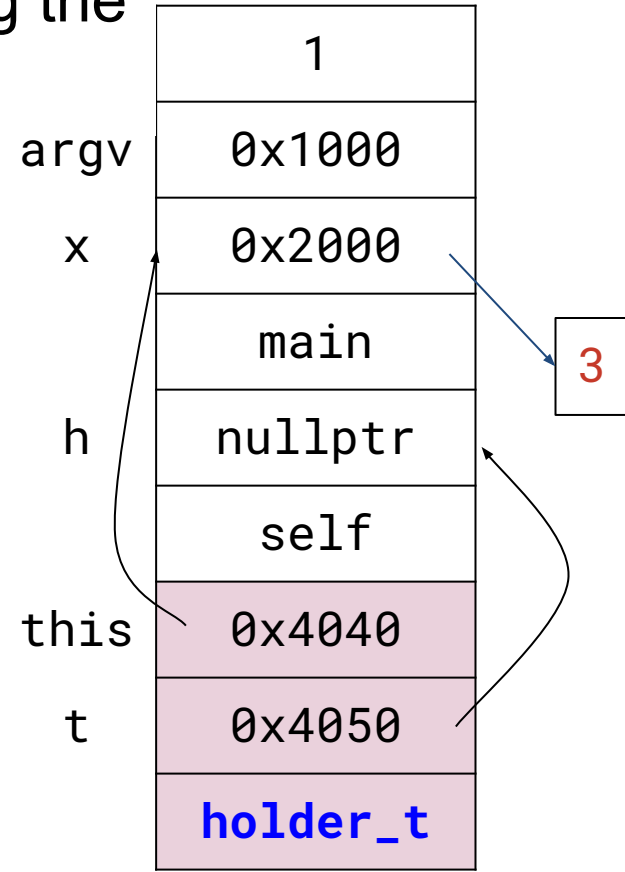
# The move constructor

- A move constructor receives a `holder_t&& t`  
`t.p = nullptr` in order to avoid deleting the integer at `0x2000` in the destructor

```
holder_t(int i)
    : p { new int { i } } {}
holder_t(holder_t&& t)
    : p { t.p } { t.p = nullptr; };
~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

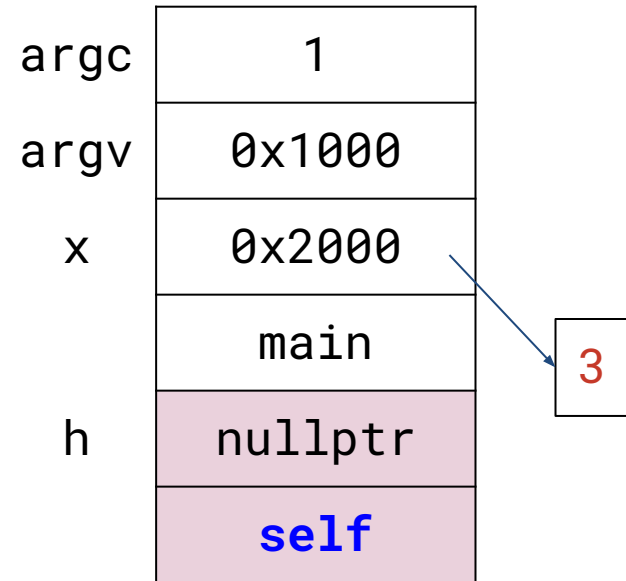
int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# The move constructor

- A move constructor receives a `holder_t&& t`

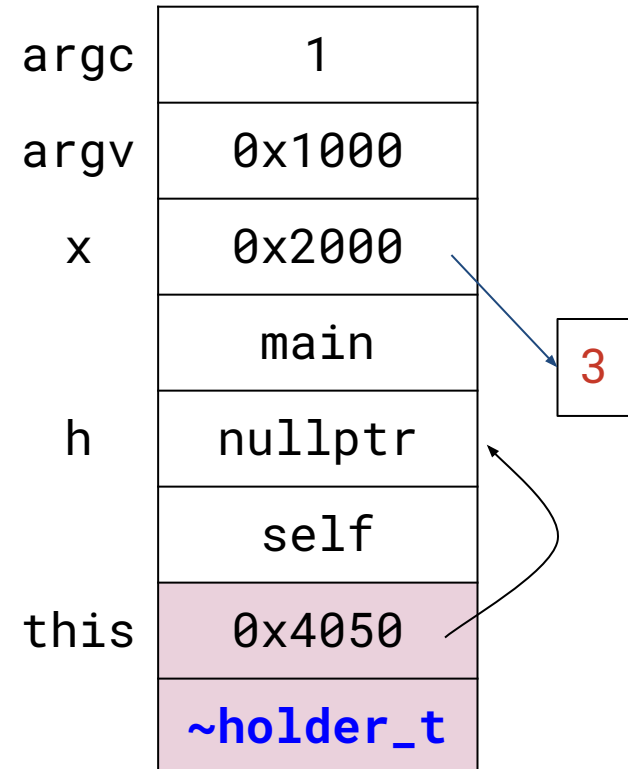
```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(holder_t&& t)  
        : p { t.p } { t.p = nullptr; };  
    ~holder_t() { delete p; }  
};  
  
holder_t self(holder_t h) { return h; }  
  
int main(int argc, char* argv[]) {  
    holder_t x { self(3) };  
    return 0;  
}
```



# The move constructor

- A move constructor receives a `holder_t&& t`

```
struct holder_t {  
    int* p;  
  
    holder_t(int i)  
        : p { new int { i } } {}  
    holder_t(holder_t&& t)  
        : p { t.p } { t.p = nullptr; };  
    ~holder_t() { delete p; }  
};  
  
holder_t self(holder_t h) { return h; }  
  
int main(int argc, char* argv[]) {  
    holder_t x { self(3) };  
    return 0;  
}
```



# The move constructor

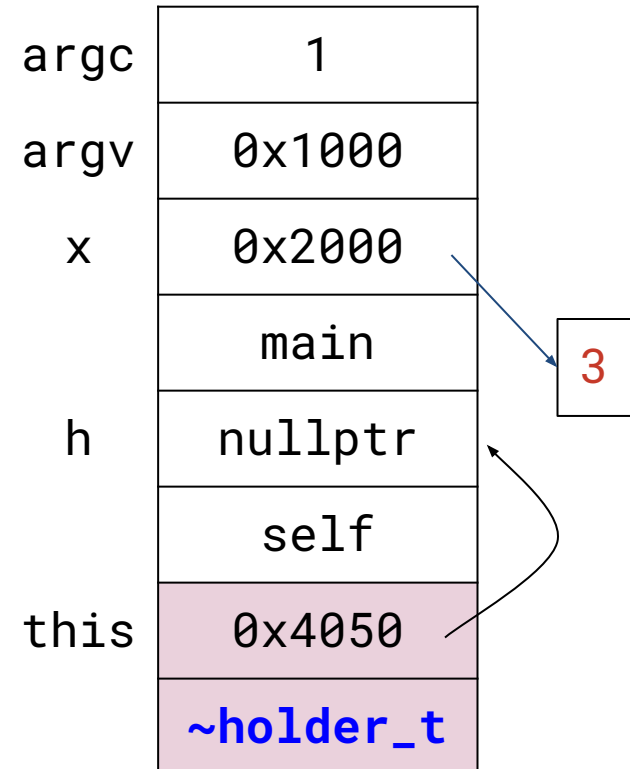
- A move constructor receives a `holder_t&& t`

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(holder_t&& t)
        : p { t.p } { t.p = nullptr; };
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



`delete` has no effect

# The move constructor

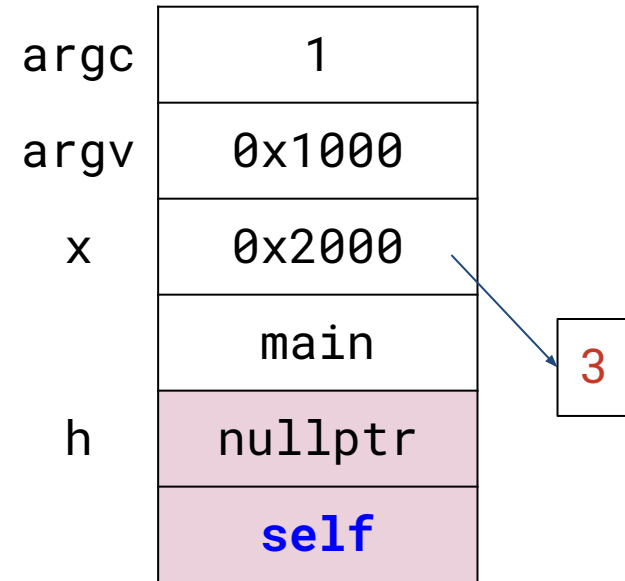
- A move constructor receives a `holder_t&& t`

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(holder_t&& t)
        : p { t.p } { t.p = nullptr; };
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



# The move constructor

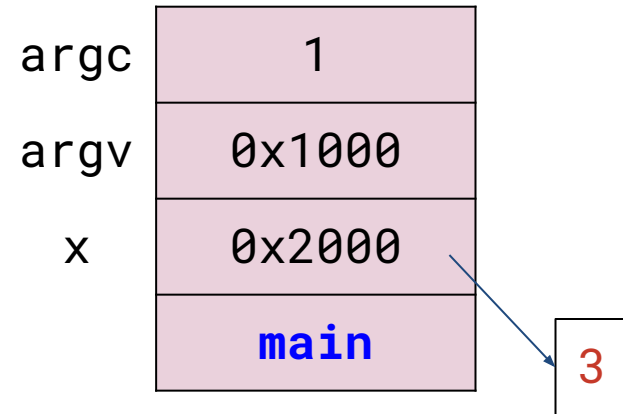
- A move constructor receives a `holder_t&& t`

```
struct holder_t {
    int* p;

    holder_t(int i)
        : p { new int { i } } {}
    holder_t(holder_t&& t)
        : p { t.p } { t.p = nullptr; };
    ~holder_t() { delete p; }
};

holder_t self(holder_t h) { return h; }

int main(int argc, char* argv[]) {
    holder_t x { self(3) };
    return 0;
}
```



Thanks to the move constructor: 0 deep copy in this code



# The rvalue reference (&&)

- A rvalue reference is a reference identified with &&
  - The compiler call a method with a rvalue reference when the parameter will be destroyed after the call
  - Useful to move instead of copy an object
- A rvalue reference is used for the move constructor
  - But can be useful for the operator =
  - And other cases

# Key concepts

- The copy constructor
  - Is used to deeply copy an object
  - `holder_t(const holder_t& t)`
- The move constructor
  - Is used to move an object into another
  - Avoid a deep copy
  - Called when the parameter will be destroyed after the call
  - `holder_t(holder_t&& t)`
  - Don't forget to nullify the elements of `t` that are deleted in the destructor