TDDE18 & 726G77

Classes & Pointers

Premise – lab 3

- Start working with `Object Oriented Programming (OOP)`
- Create the class `Sorted_List`
- Learn the difference between `stack and heap`
- Use dynamic memory with `new and delete`

Imperative programming

- Programming paradigm that uses statement that change a program's state.
- Focus on *how* a program operates.
- Revolves around function that operates on data

```
int length_of_string(string s);
string to_string(Time const& t);
```

Object Oriented Programming

- Programming paradigm based on the concept of “objects”
- Objects may contain data and code
  - Data members
  - Member functions
- Revolves around the data

```
str.length();
cin.ignore(5, ’\n’);
```

OOP – Real life definition

- If I'm your coffee getter object
  - *"Can you get me the best coffee, please."* is a question that you asked
  - *"Here is your coffee*" as a result from me.
- You have no idea how I did that.
  - we were able to interact at a very high level of abstraction.

Variable

- Fundamental (also called built-in types)
  - Stores a value of a fundamental type, nothing more
- Object
  - Stores values tied to an derived type (struct, class)
  - Operations associated to the type are provided
- Pointer
  - Stores the address of some other variable

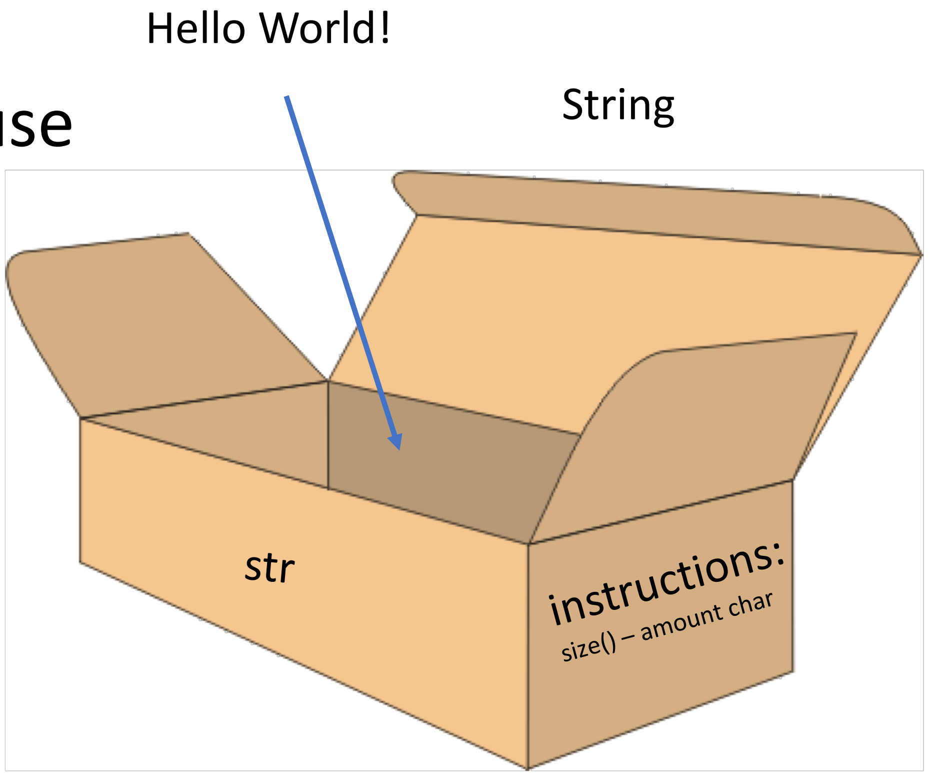
Class

- Data members – store values

```
string str{“Hello World!”};
```

- Member functions – operations available to use

```
str.size();
```



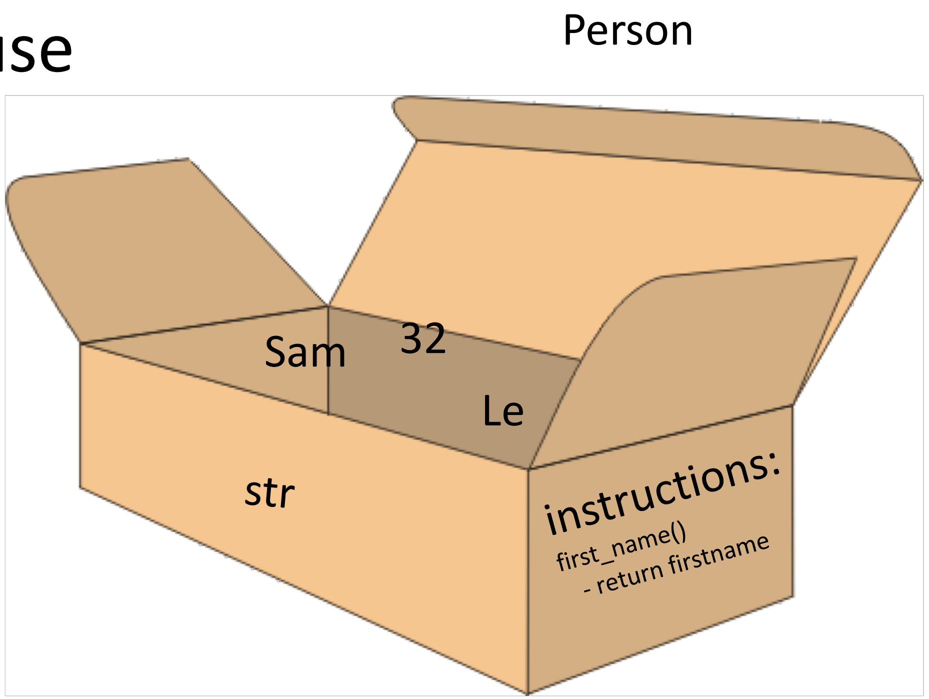
Class – the blueprint of an object

- Data members – store values

```
Person p{"Sam", "Le", 32};
```

- Member functions – operations available to use

```
p.first_name();
```



Class syntax – header file

```
#ifndef _CLASS-NAME_H_
#define _CLASS-NAME_H_
class class-name {
public:
    class-name(); // constructor (Initiator)
    // member functions (methods in Java)
    return-type operation(parameter-list);
private:
    // member variables
    data-type property;
};
#endif
```

Class syntax – implementation file

```
#include “class-name.h”

// Constructor (Initiator)
class-name::class-name() {
    // implementation
}

// Member function
return-type
class-name::operation(parameter-list) {
    // implementation
}
```

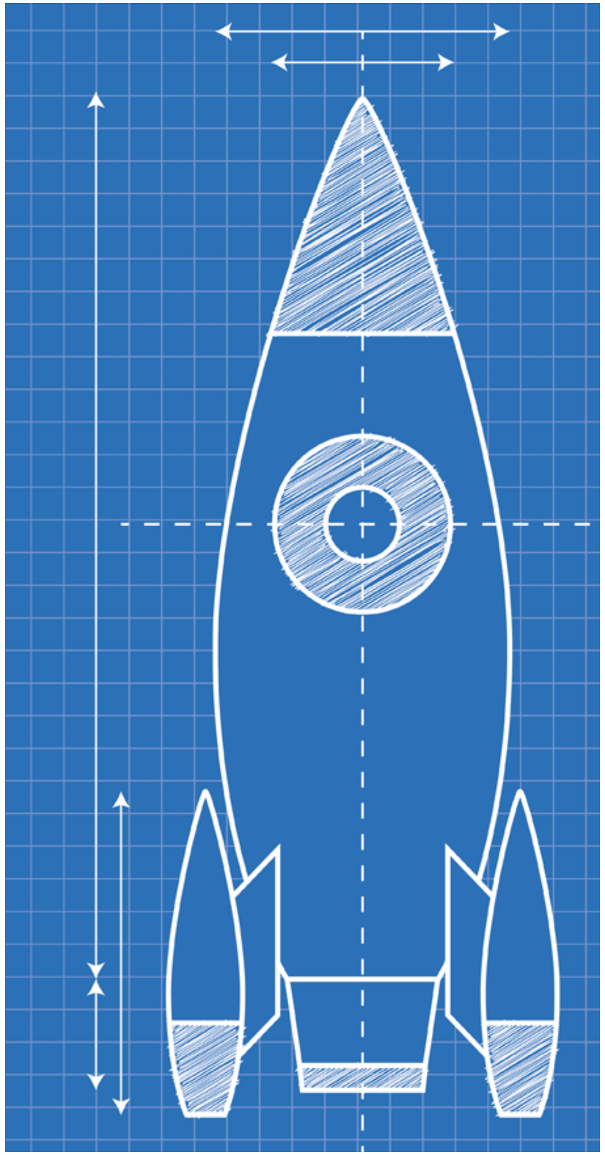
Class

- Provide language support for object orientation
- Having a single purpose, responsibility
- Consist of private member variables and public interface methods
- Can only be manipulated through a well defined interface
- Constructors and interface enables the programmer to depend on always known and correct internal state
- Operators, constructors and destructors allow for easy management

Class vs Instance

- A class only describe the layout. It does not create any data in memory. It’s a description of a data-type with operations ”embedded”.

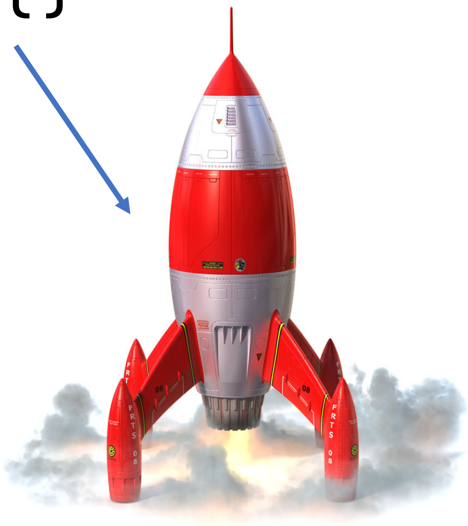
```
class Rocket {
public:
	void fly();
	bool finished;
private:
	int height;
};
```

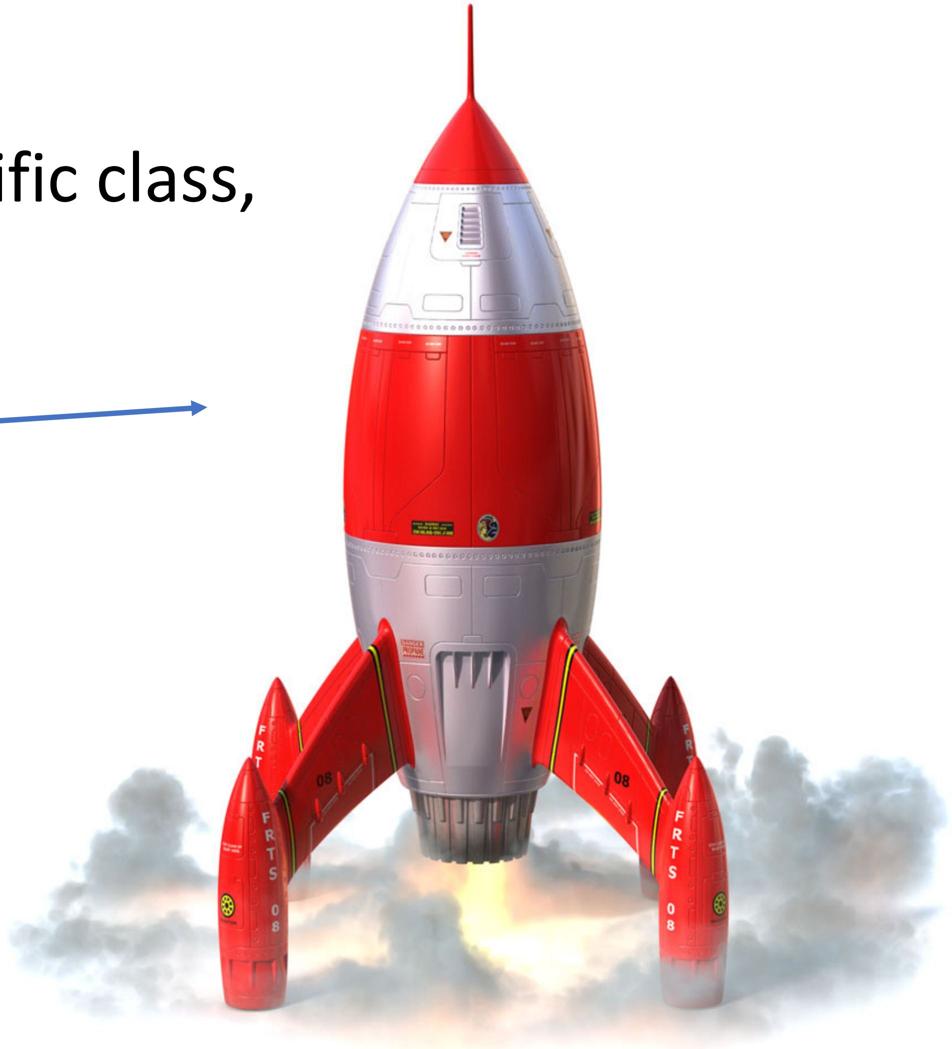


Class vs Instance

- An instance is a variable created of a specific class, an object. You can create many.

```
Rocket r{};
Rocket s{}
```





Class declaration

```
// h-file
class Robot {
public:
  void fly();
  bool finished;
private:
  int height;
};
```

```
// cc-file
void Robot::fly() {
  cout << “I’m flying” << endl;
}
```

Accessing members

- An object variable allow you to access member functions (operations) and member variables of that instance. You use the dot operator

```
// Class definition
class Rocket {
public:
     void fly();
     bool finished;
private:
     int height;
};
```

```
// Access member functions
Rocket r{};
r.finished = true;
r.fly();
```

Accessing members

- Accessing a member inside a class does not require you to tell the compiler which instance you are referring to.

```
// Outside of class
int main() {
   Rocket r{};
   r.finished = true;
}
```

```
// Inside the class
class Rocket {
public:
   void fly() {
      finished = true;
   };
```

The keyword “this”

- Member functions are called “on” an instance and automatically receive that instance to work on, available as the special pointer ***this.***

```
void Robot::fly() {
   finished = true;
   cout << ”I’m finished and I can fly” << endl;
}

void Robot::fly() {
   this -> finished = true;
   cout << ”I’m finished and I can fly” << endl;
}
```

Private members

- Private members are only accessible in functions belonging to the same class

```
class Rocket {
public:
   void fly() {
      model = "M-3"; //OK
   }
};

int main() {
     Rocket r{};
     r.model = "M-3"; //Error
}
```

Friends

- A class can decide to have friends. Friends can access private members!
- Friends should be avoided at all cost, since it makes the two classes highly interdependent.

```
class Rocket {
   ...
   friend bool equals(Rocket r1, Rocket r2);
   ...
};
bool equals(Rocket r1, Rocket r2) {
   return r1.model == r2.model;
}
```

Object lifecycle

- class definition:
  - no object created yet, before birth
- variable definition:
  - object born, memory allocated
  - memory initiated with default values
- variable used...
- variable declaration block ends:
  - memory reclaimed for other variables

Object lifecycle

- class definition:
  - no object created yet, before birth
- variable definition:
  - object born, memory allocated
  - memory initiated with default values
- variable used...
- variable declaration block ends:
  - memory reclaimed for other variables

Constructor

Member functions
Operator functions

Destructor

Lifecycle “hooks”

- Constructor is automatically called when a class variable is defined or allocated
  - have no return value
  - any defined parameters must be specified
- Operators functions are automatically called when variable is used by an operator
  - covered later on
- Destructor is automatically called when a variable goes out of scope or is deleted
  - have neither return value nor parameters

The rocket constructor

```
// h-file
class Rocket {
public:
   Rocket(); // Constructor
private:
   string model;
};
```

```
// cc-file
Rocket::Rocket() {
   model = “Unknown”;
}
```

Using the constructor

- If you define a constructor you must specify all arguments when you create an instance!
- If you do not define a constructor a default constructor that does nothing will be created.
- If you only have private constructors other code can not create instances.

Default constructor

- If you do not define a constructor the compiler will generate a similar default constructor for you.

```
// h-file
class Rocket {
public:
   Rocket(); // Default Constructor
   ...
};
// cc-file
Rocket::Rocket() {
}
```

Constructor Example

```
// h-file
class Rocket {
public:
   Rocket(string m);
   ...
};
// cc-file
Rocket::Rocket(string m) {
   model = m;
}
```

Constructor Example

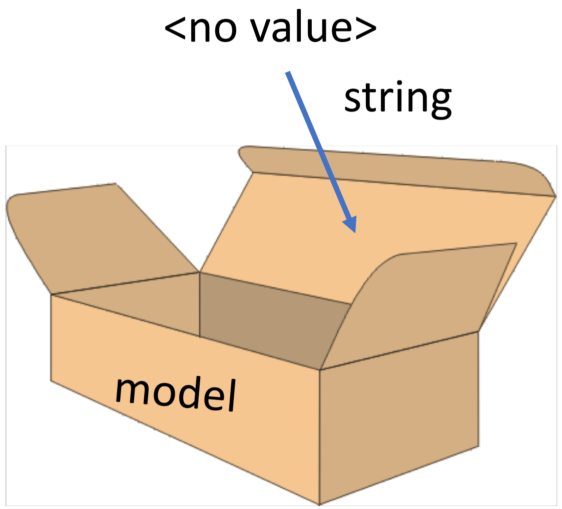
```
// h-file
class Rocket {
public:
   Rocket(string m);
   ...
};
// cc-file
Rocket::Rocket(string m) {
   model = m;
}
```

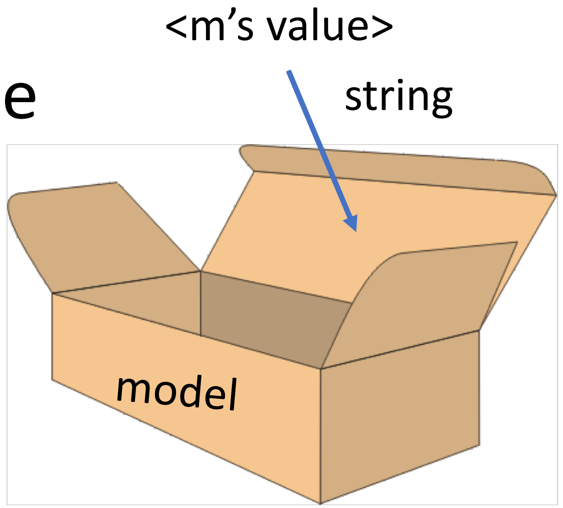
```
// Ok
Rocket r{“M-3”};

// Error no fitting constructor
Rocket s{};
```

Constructor Performance Issue

```
Rocket::Rocket(string m) {
    model = m;
}
```

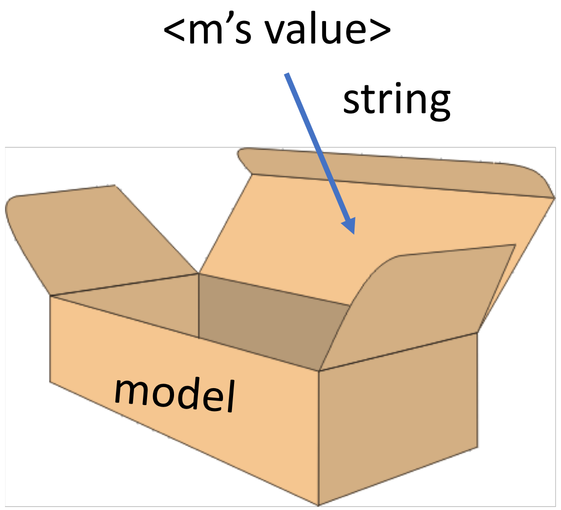
1. Create model variable inside rocket
2. Update model variable with correct value





Constructor Member Initializer List

Robot::Robot(string m) : model{m} {}



Member initializer list specifies the initializers for data members.

Const member variables

- Data members could also be const
- Constant member variable must be initialized in constructor initialization list

```
class Robot {
public:
    ...
    string const model;
};
```

```
Robot::Robot(string m) model{m} {}
```

Reference member variables

- Data members could also be a reference to another variable
- Reference member variables must be initialized in constructor initialization list

```
class Robot {
...
private:
   Person & creator;
};
```

Constructor – Multiple

- Constructor can be overloaded in a similar way as function overloading
- Overloaded constructor have the same name (name of the class) but different number of arguments
- The compiler choose the constructor that fits best with the given input arguments

```
...
    Robot();
    Robot(string m);
    Robot(Person p);
    Robot(Person p, string m);
    etc.
...
```

Constructor delegation

- Many classes have multiple constructors that do similar things
- You could reduce the repetitive code by delegating the work to another constructor

```
Robot::Robot() : Robot{“unknown”} {}
Robot::Robot(string m) : model{m} {}
```

Destructor

- The object calls the destructor when it is about to go out of scope

```
int main() {
   Robot r{};
} // r will call its destructor on this line
```

Destructor

```
// h-file
class Robot {
public:
   ~Robot(); // no return or parameters
   ...
};

// cc-file
Robot::~Robot() { // not useful yet...
   cout << “destructor called” << endl;
}
```

Example class - Money

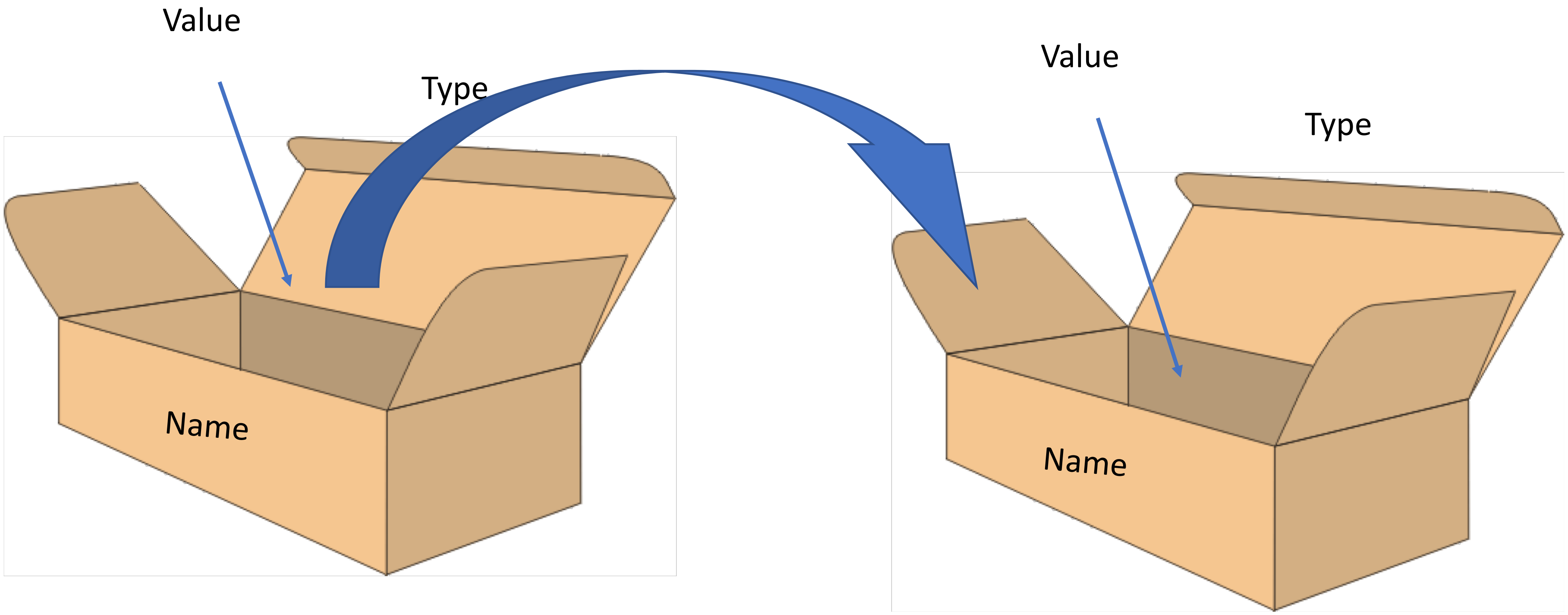
- Class that represent money
- Have the capacity to hold units (Swedish krona)
- Have the capacity to hold hundreds (Swedish öre)
- Can validate that it have valid (non-negative values) in units and hundreds.

Example class

```
class Money {
public:
   Money();
   Money(int unit);
   Money(int unit, int hundred);
   ~Money();
   void validate();
private:
   int unit;
   int hundred;
};
```

```
Money::Money()
   : Money{0} {}
Money::Money(int unit)
   : Money {unit, 0} {}
Money::Money(int unit, int hundred)
   : unit{unit}, hundred{hundred} {
   validate();
}
void Money::validate() {
  if (unit < 0 || hundred < 0)
...
```

Pointer



Pointer

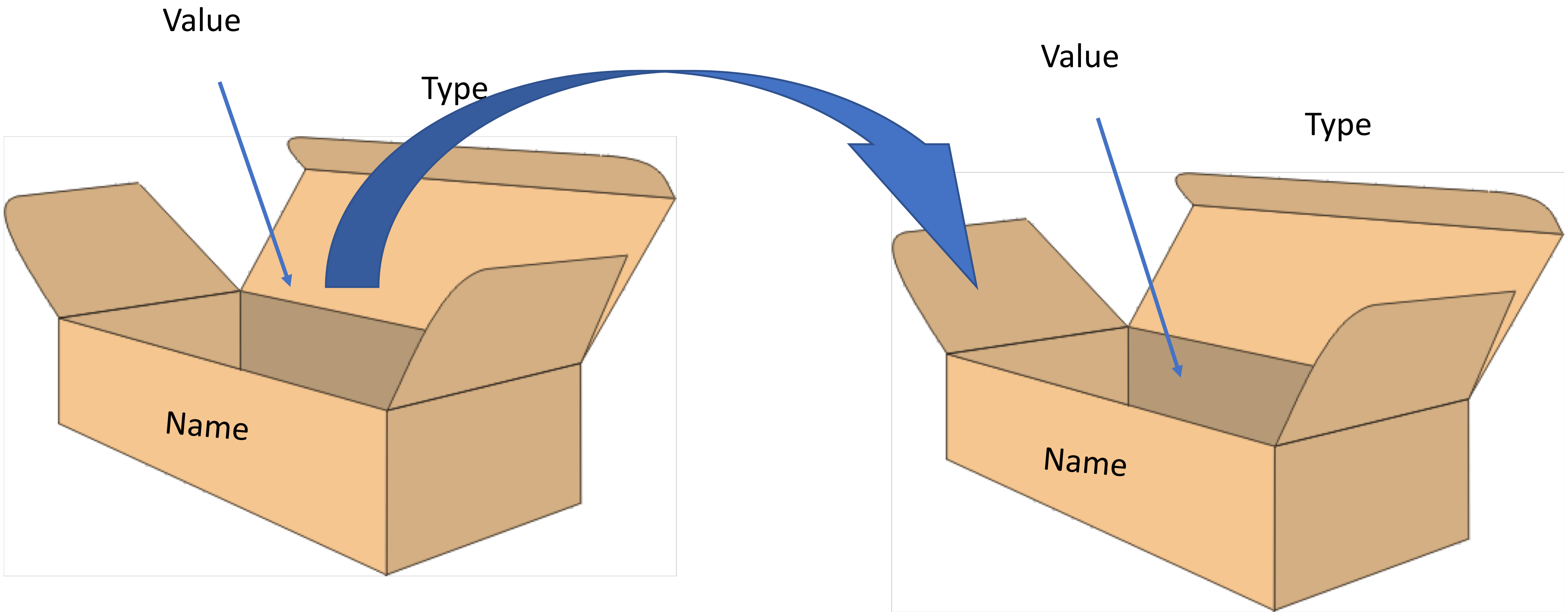
- A variable that stores an address
- Compiler (programmer) keep track of what type each pointer address store in order to index and treat dereference values correct.
- Read declaration backwards

```
int * p;     // A variable p
             // That is a pointer
             // To an int
```

Pointer operators

- Operators relevant to pointers
  - Dereference (content of, “go to”): `*p`
  - Dereference with offset (indexing): `*(p + i)` or `p[i]`
  - Address of: &
  - Dereference and select member: `(*p).m` or `p->m`
  - Allocate (borrow) memory: `p = new t, a = new t[s]`
  - Deallocate (return) memory: `delete p, delete[] a`

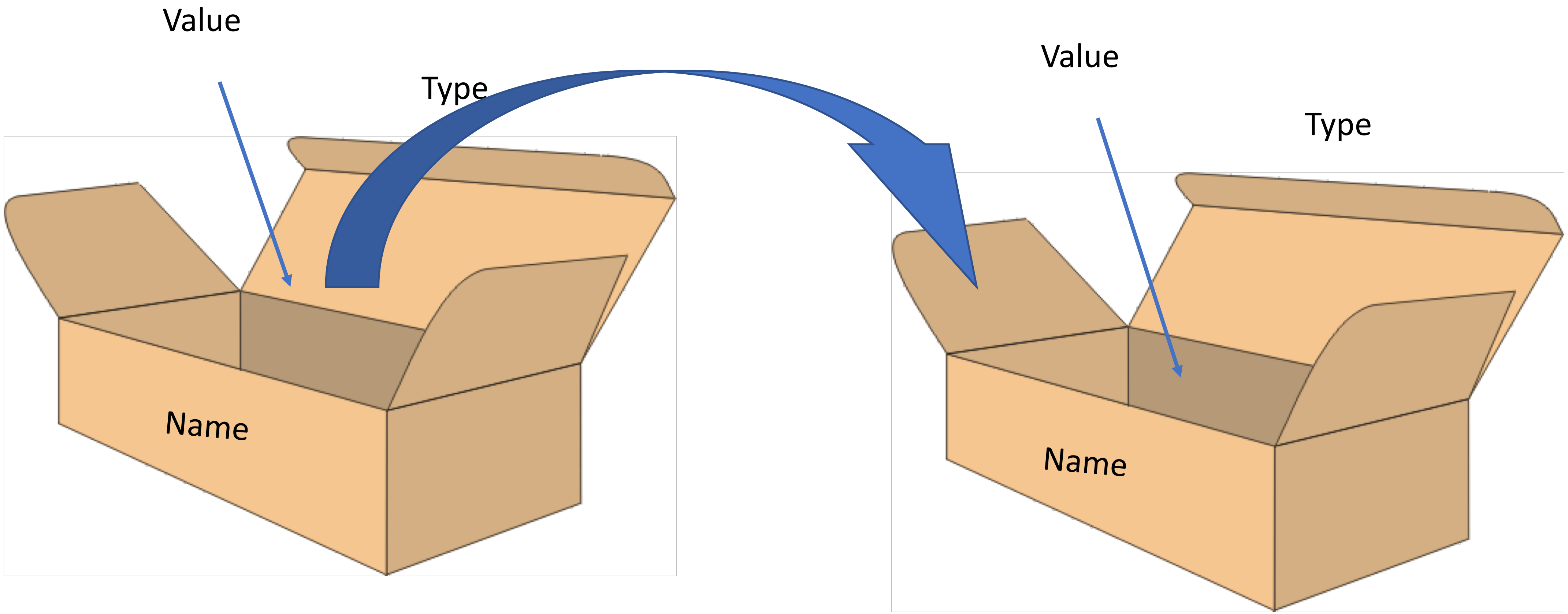
Pointer – Address of




```
int * int_pointer{&integer_value};
```

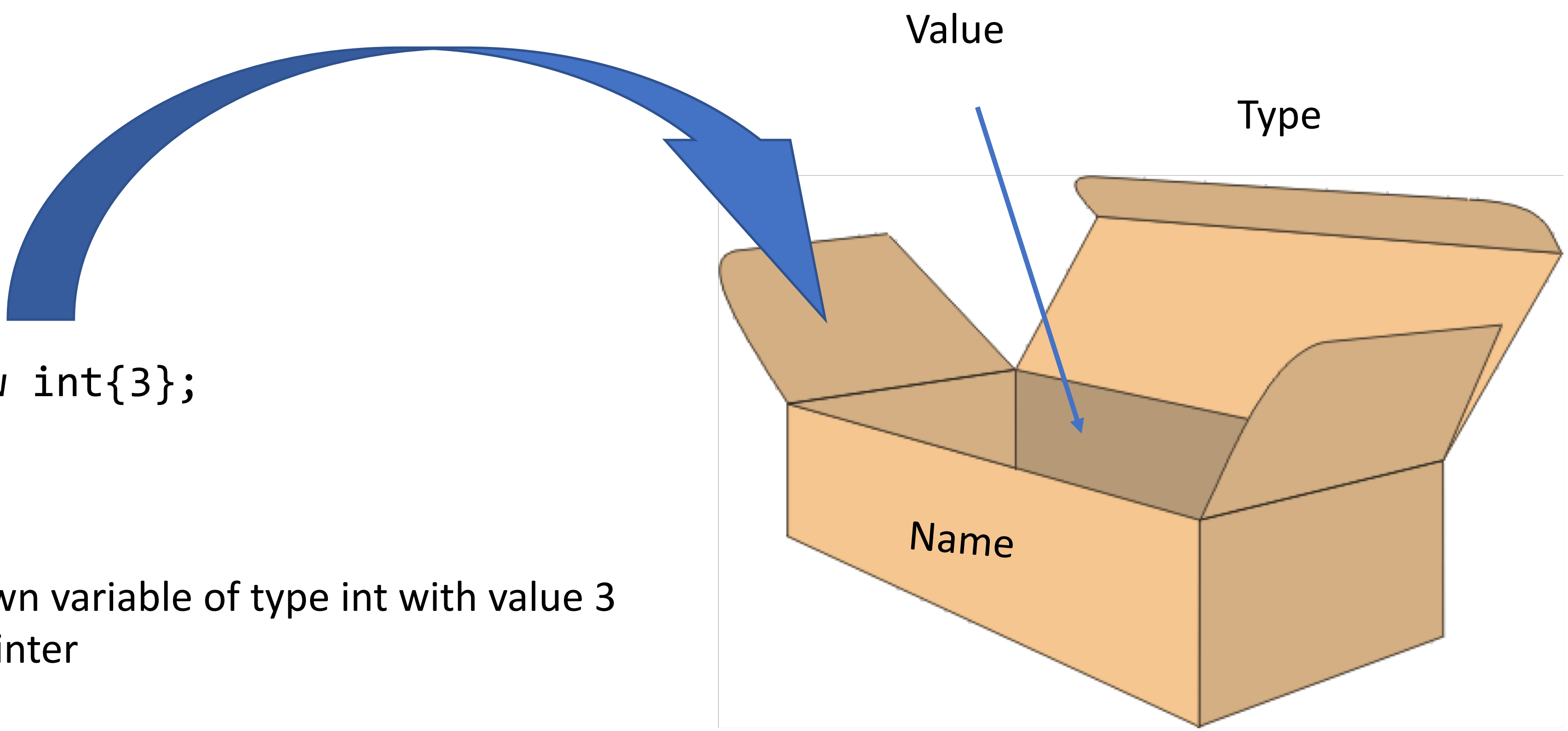
```
int integer_value{};
```

Pointer – Dereference




```
cout << *int_pointer << endl;
```

Pointer – Allocate




```
new int{3};
```

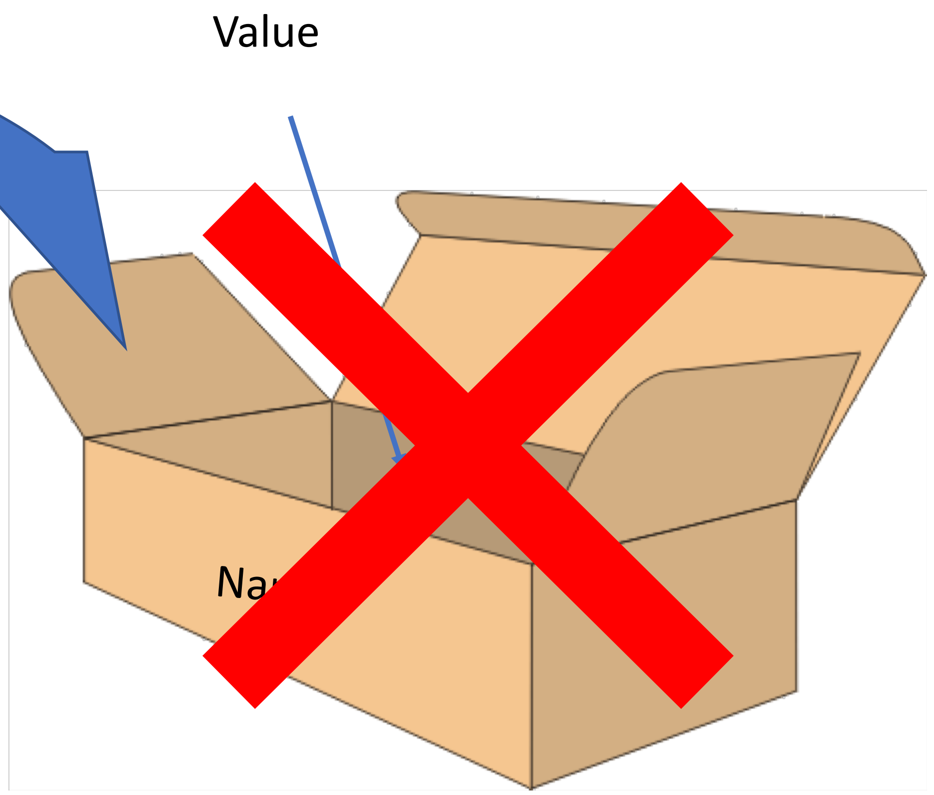
1. Create unknown variable of type int with value 3
2. Return the pointer

Save the pointer by declaring a new variable

```
int * integer_pointer{new int{3}};
```

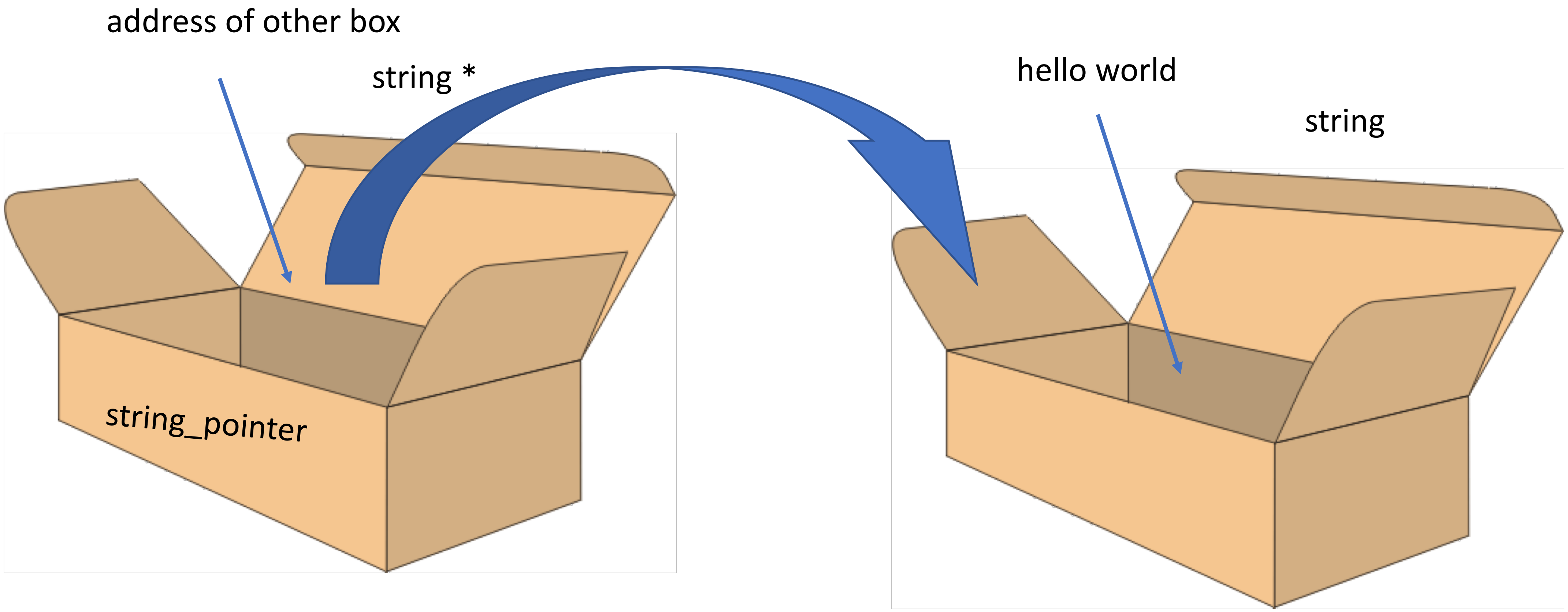
Pointer – Deallocate

```
delete integer_pointer;
```



1. Delete the variable that the pointer points to
2. Does not remove the pointer!

Pointer – Dereference and select member




```
string * string_pointer{new string{"hello world"}};
string_pointer->length();
```

Dynamic memory

- Memory for variables can be dynamically allocated and deallocated
  - Dynamic: During program execution
  - Normal/Static: During compile time
  - Allocate: Borrow from operating system
  - Deallocate: return to operating system
- Each allocation must be deallocated exactly once, as soon as possible

What if …

- We assign (copy) pointer variables?

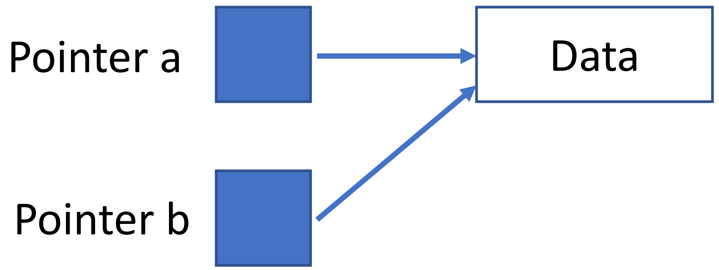
```
int * a_ptr { new int { 4 } };
int * b_ptr { a_ptr };
```

- We pass pointer variables as parameter?

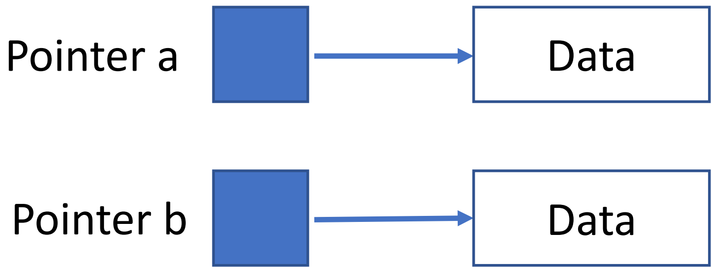
```
void foo(int * p);
```

Shallow copy vs deep copy

Shallow copy

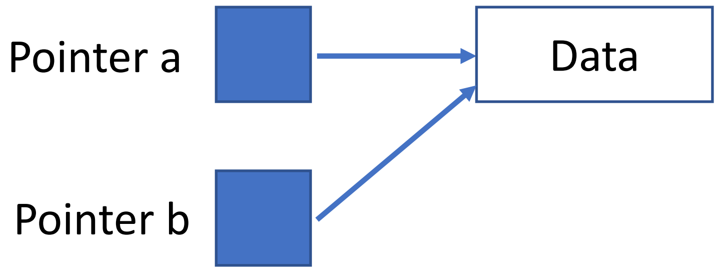


Deep copy



Shallow copy vs deep copy

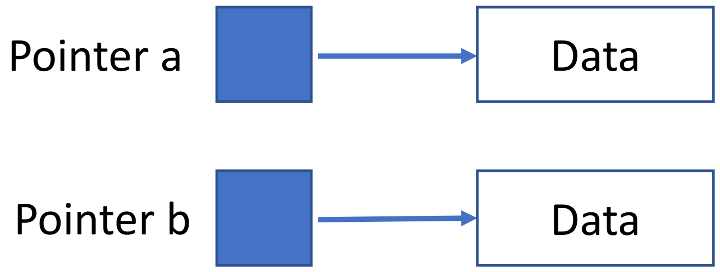
Shallow copy



Example code:

```
int * a{new Integer{3}};
int * b{a};
```

Deep copy



Example code:

```
int * a{new Integer{3}};
int * b{new Integer{*a}};
```

Class with pointer

```
class Array {
public:
   Array(int size);
   ...
private:
   int size_;
   int * data;
};
```

What if …

- We pass Array variables as parameter?
- We assign (copy) Array variables?
- We want to initialize an array from another?
- Destroy an Array variable?
- Move an Array variable about to be destroyed to another array?

Lifecycle “hooks”

- Constructor is automatically called when a class variable is defined or allocated
  - have no return value
  - any defined parameters must be specified
- Operators functions are automatically called when variable is used by an operator
  - Set an object equals to another object
- Destructor is automatically called when a variable goes out of scope or is deleted
  - have neither return value nor parameters

Lifecycle "hooks"

- Constructor is automatically called when a class variable is defined or allocated
  - have no return value
  - any defined parameters must be specified

Eg. Default constructor

- Operators functions are automatically called when variable is used by an operator
  - Set an object equals to another object

Eg. Assignment operator

- Destructor is automatically called when a variable goes out of scope or is deleted
  - have neither return value nor parameters

Destructor

Three essential "hooks"

- Copy constructor
  - Called automatically when a fresh object is created as a copy of an existing object

```
Array(Array const&);
```

- Assignment operator
  - Called automatically when an existing object is overwritten by another object (or itself)

```
Array & operator=(Array const&);
```

- Destructor
  - Called automatically when an object is destroyed

```
~Array();
```

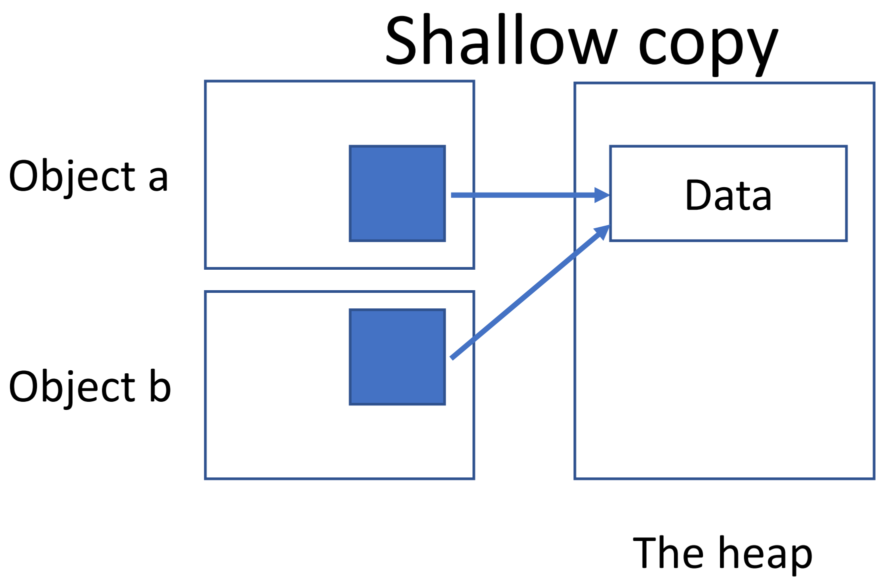
When?

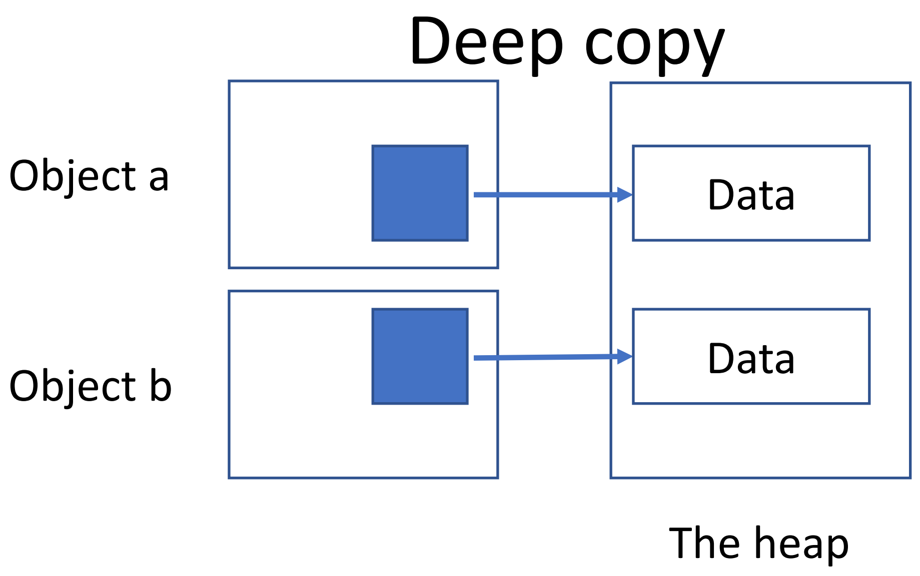
- If you have a class with pointers you need the three essential hooks to prevent memory leaks
- The compiler generate default versions if they do not exist, but the compiler version **WILL NOT** be adequate or enough
- If your class have **no pointers**, you do not have to care, the compiler version will be enough

Array class

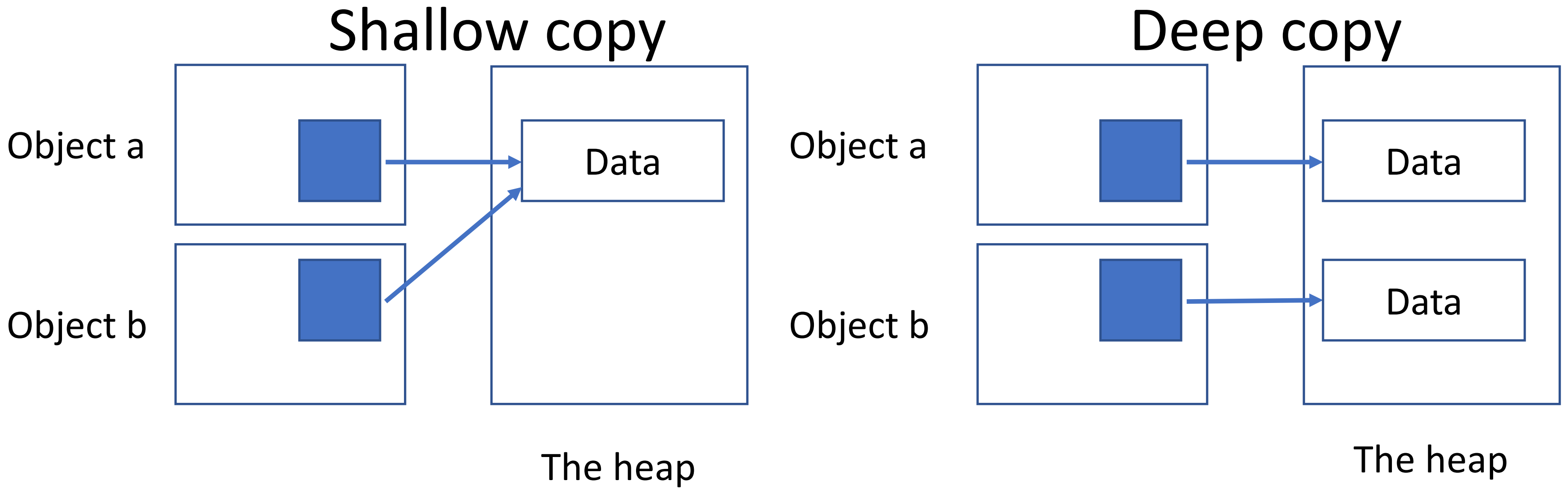
```
class Array {
public:
   Array(int size);
   ...
private:
   int size_;
   int * data;
};
```

Shallow copy vs deep copy



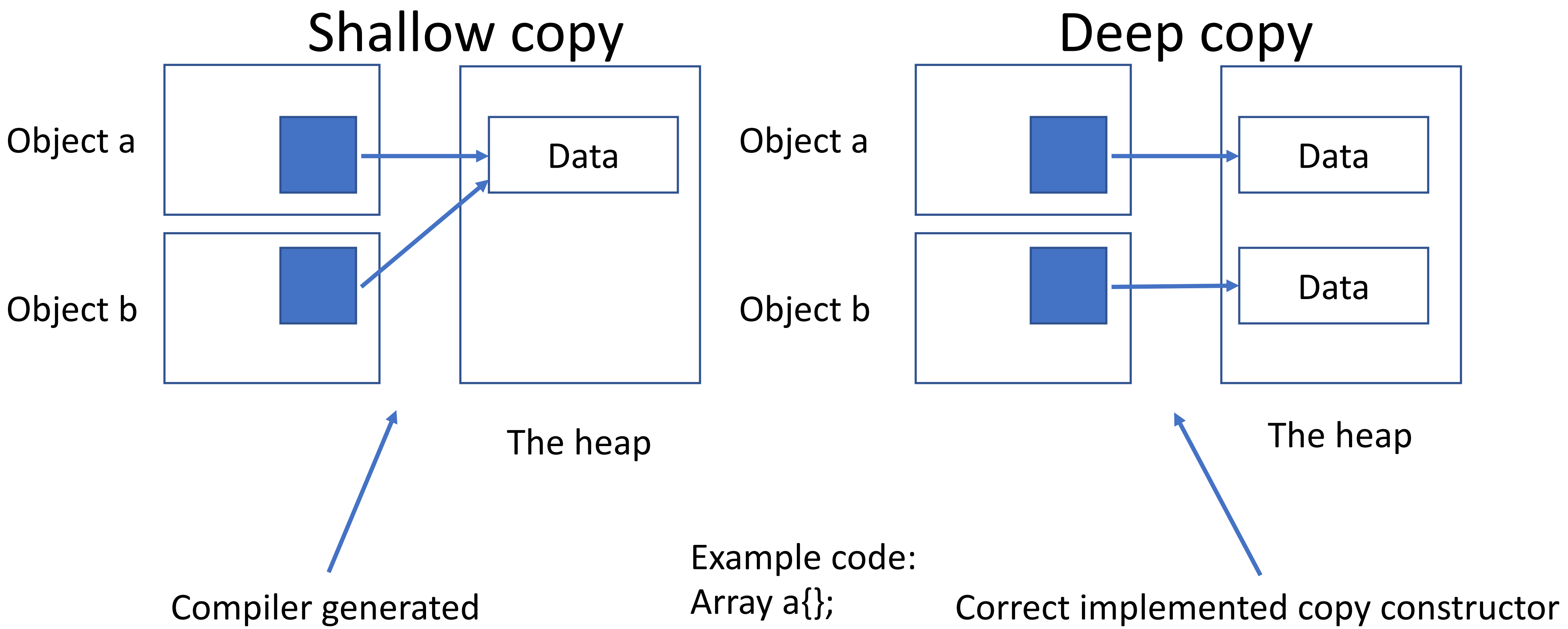


Shallow copy vs deep copy



Example code:
Array a{};
Array b{a};

Shallow copy vs deep copy



Example code:

```
Array a{};
Array b{a};
```

Copy constructor – syntax

```
class Array {
   ...
   Array(Array const& a);
   ...
};
```

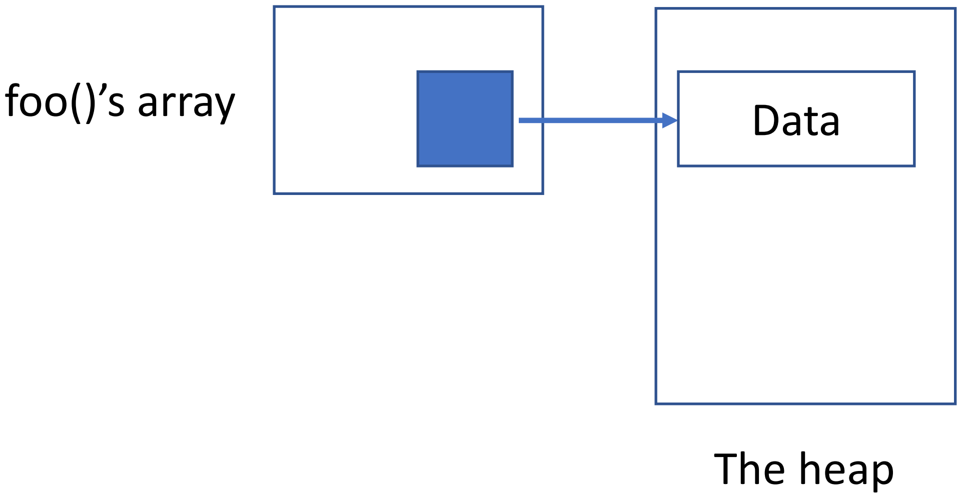
```
// cc-file
Array::Array(Array const& other) {
   // allocate new memory
   // etc
}
```

Temporary variable

```
Array foo() {
   return Array{};
}

int main() {
   Array a{foo()};
}
```

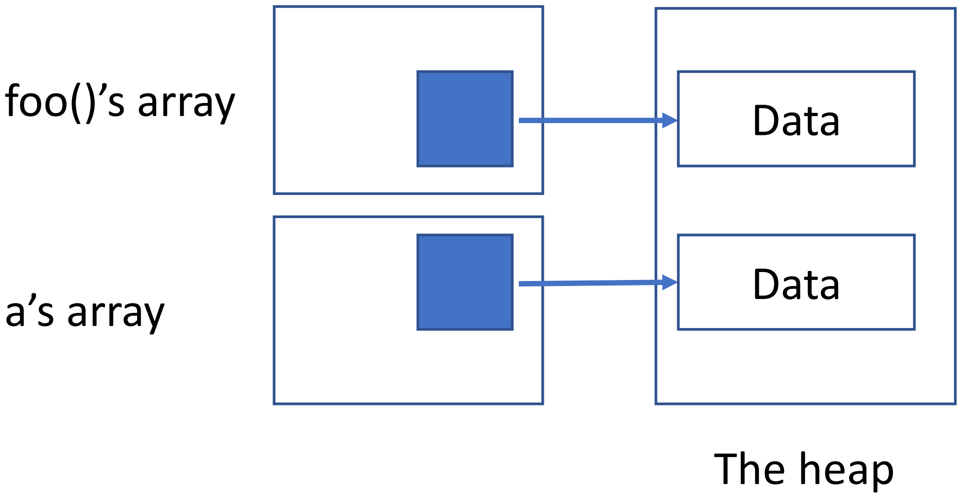
Temporary variable

```
Array foo() {
    return Array{};
}

int main() {
    Array a{foo()};
}
```



Temporary variable

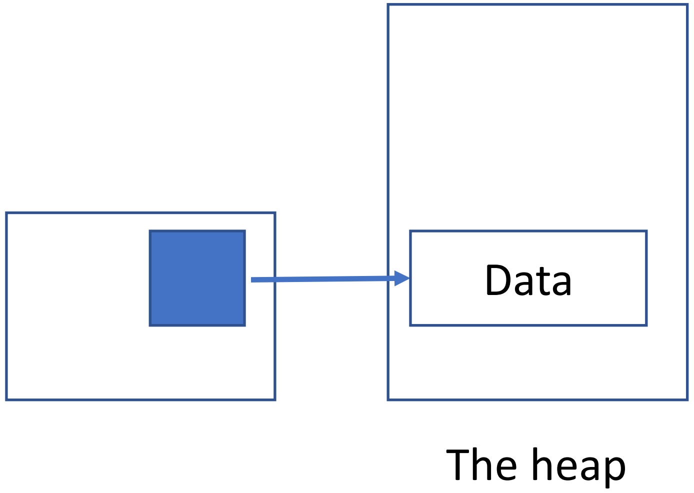
```
Array foo() {
    return Array{};
}

int main() {
    Array a{foo()};
}
```



Temporary variable

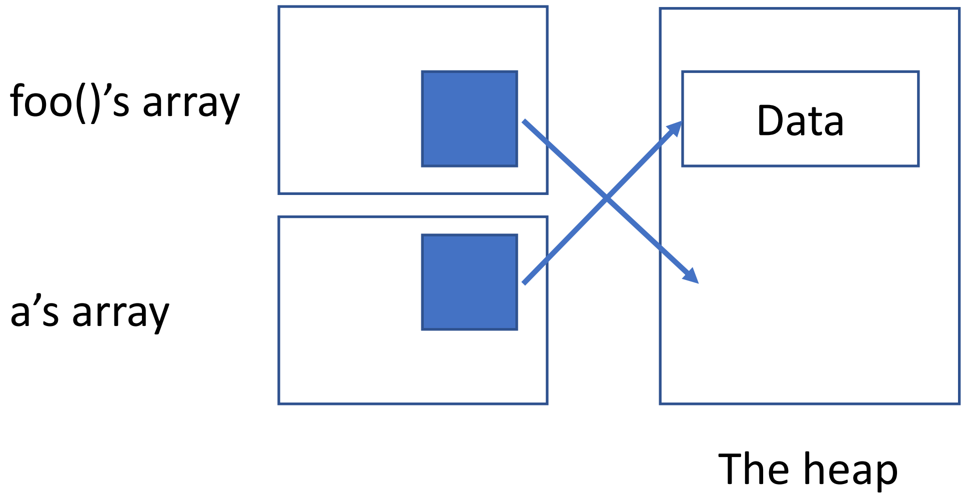
```
Array foo() {
   return Array{};
}

int main() {
   Array a{foo()};
}
```

a’s array



Temporary variable

```
Array foo() {
    return Array{};
}

int main() {
    Array a{foo()};
}
```



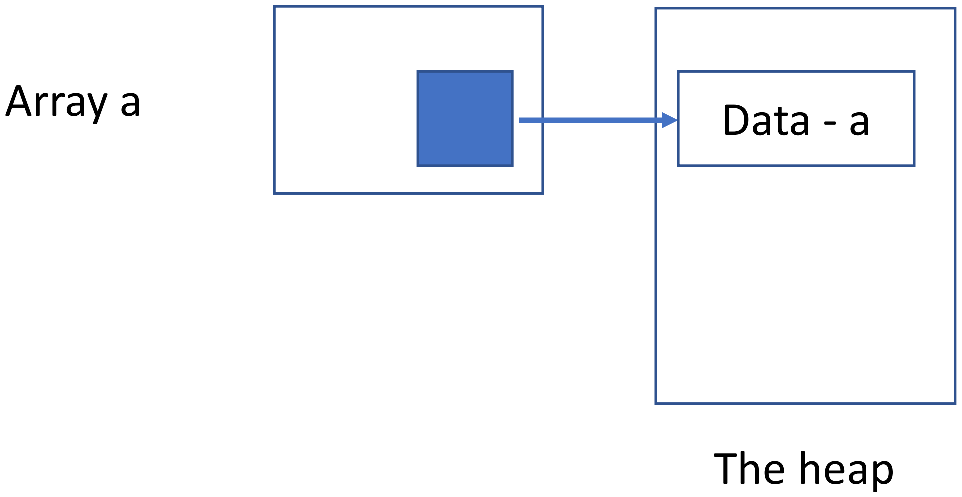
Move constructor – syntax

```
class Array {
   ...
   Array(Array && a);
   ...
};
```

```
// cc-file
Array::Array(Array && other) {
   // swap the pointers
   // etc
}
```

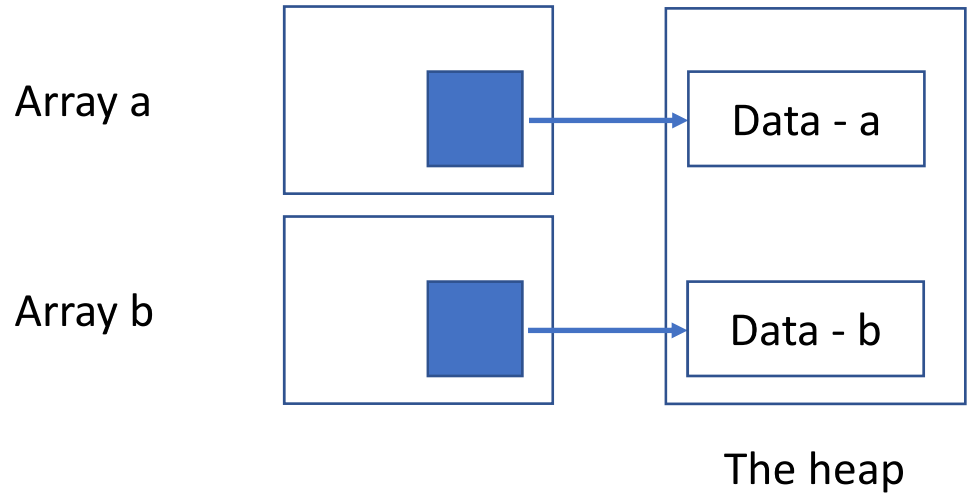
Problems that might occur with copy assignment

```
int main() {
    Array a{};
    Array b{};
    b = a;
}
```



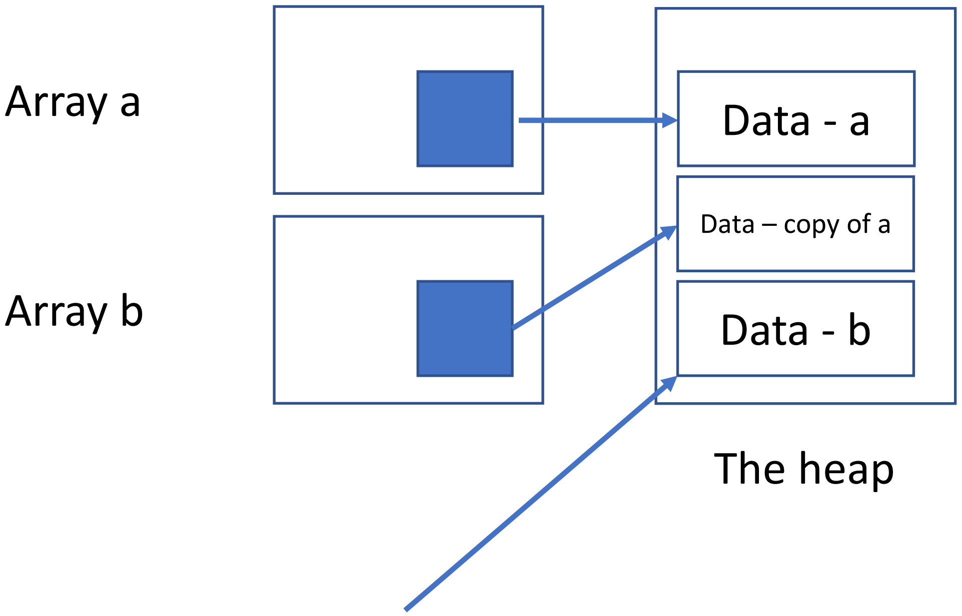
Problems that might occur with copy assignment

```
int main() {
    Array a{};
    Array b{};
    b = a;
}
```



Problems that might occur with copy assignment

```
int main() {
    Array a{};
    Array b{};
    b = a;
}
```



Still in memory – Memory leak
You must remove this manually in your
- copy assignment
- move assignment

Copy assignment - syntax

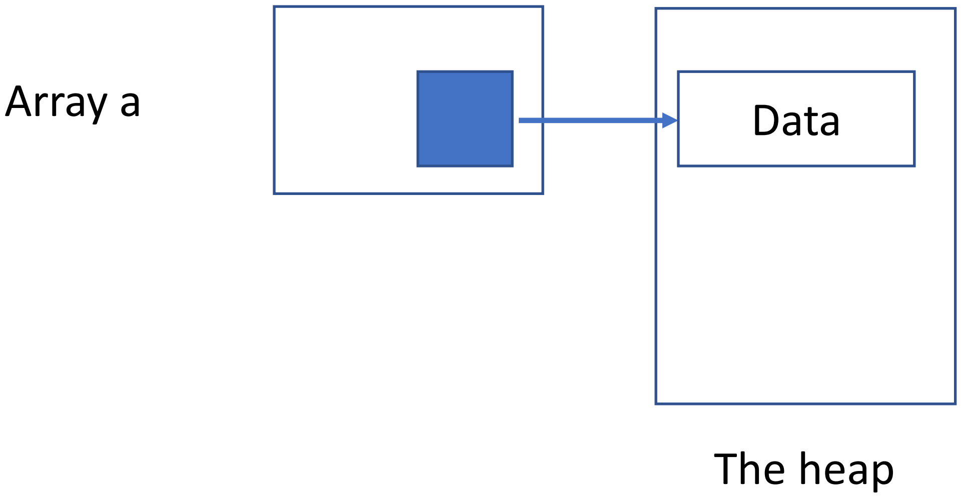
```
// h-file
class Array {
   ...
   Array & operator=(Array  const& other);
   ...
};

// cc-file
Array & Array::operator=(Array const& other) {
   // implementation
};
```

Move assignment - syntax

```
// h-file
class Array {
   ...
   Array & operator=(Array  && other);
   ...
};

// cc-file
Array & Array::operator=(Array && other) {
   // implementation
};
```

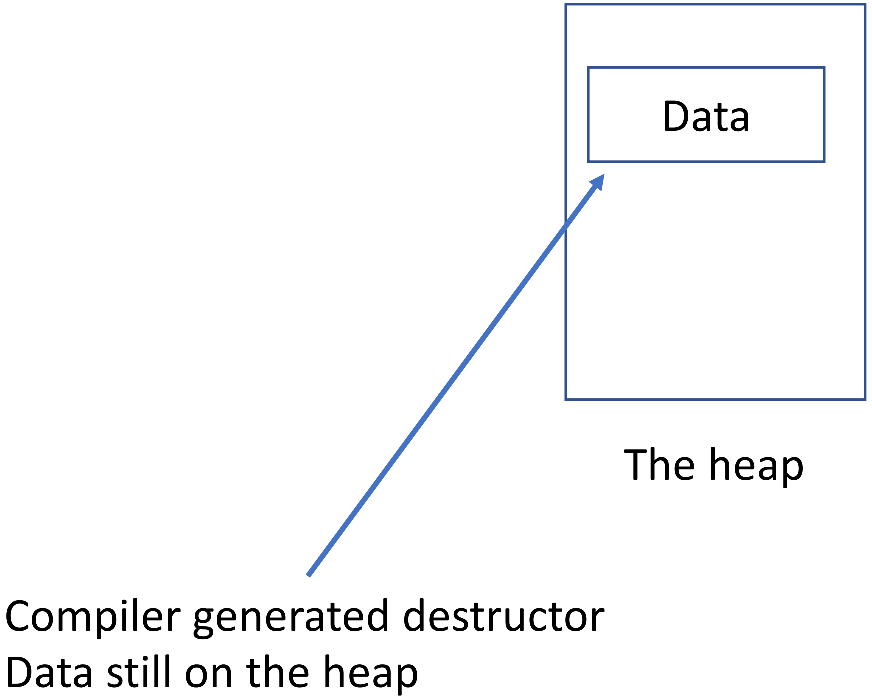
Object that is going to be removed

```
int main() {
  Array a{};
}  // a will be removed here
```



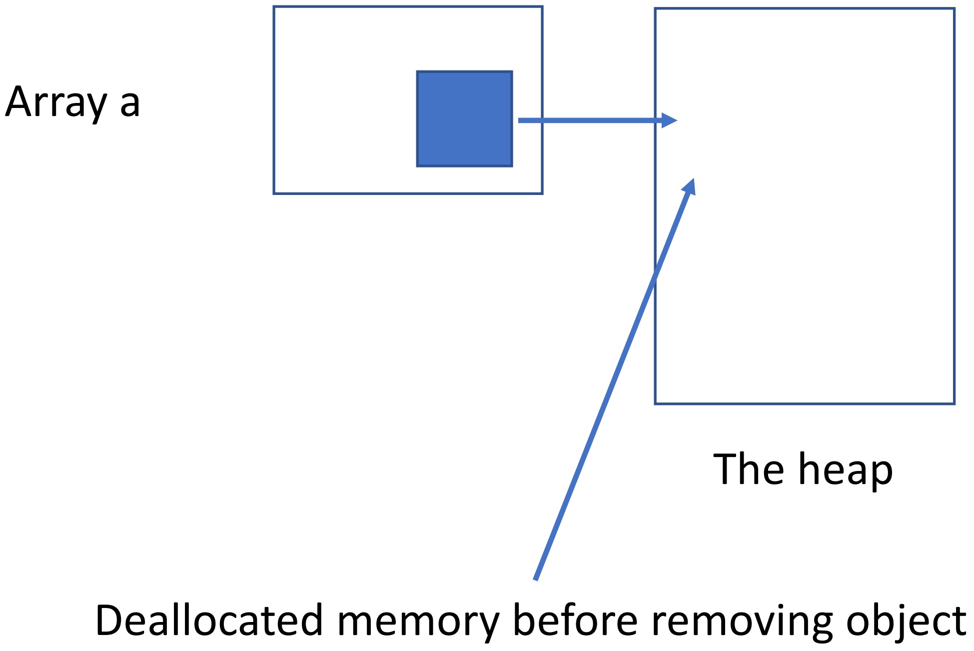
Object that is going to be removed

```
int main() {
  Array a{};
}  // a will be removed here
```



Destructor – syntax

```
// h-file
class Array {
   ...
   ~Array();
   ...
}
// cc-file
Array::~Array() {
   // deallocate memory
}
```



Constructors

- Constructor – Called when creating a new object
- Copy constructor – Called when creating a new object from an old object
- Move constructor – Called when creating a new object from an object that is about to be removed
- Copy assignment – Assign an existing object the same values as another object
- Move assignment – Assign an existing object the same values as an object that is about to be removed
- Destructor – Called when an existing object is about to be removed

Random number generator

```
#include <random>
random_device rand{};
uniform_int_distribution<int> die(1, 6);
int n = die(rand); // random in [1 .. 6]
```

Further reference:

en.cppreference.com