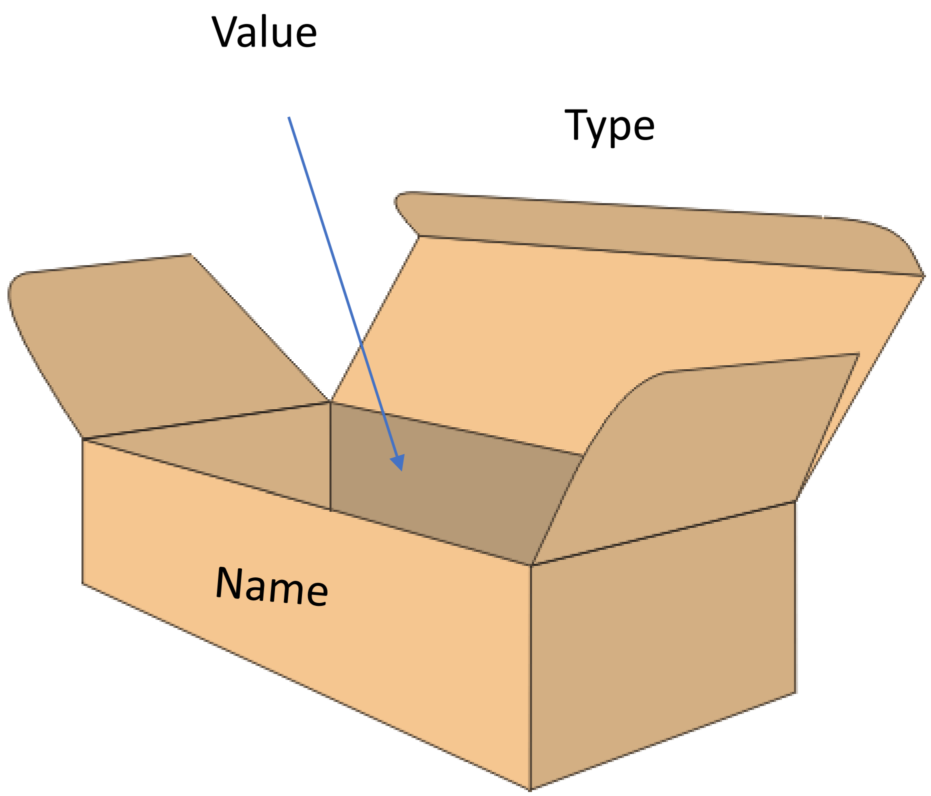
TDDE18 & 726G77

Pointers, Copy, and Move

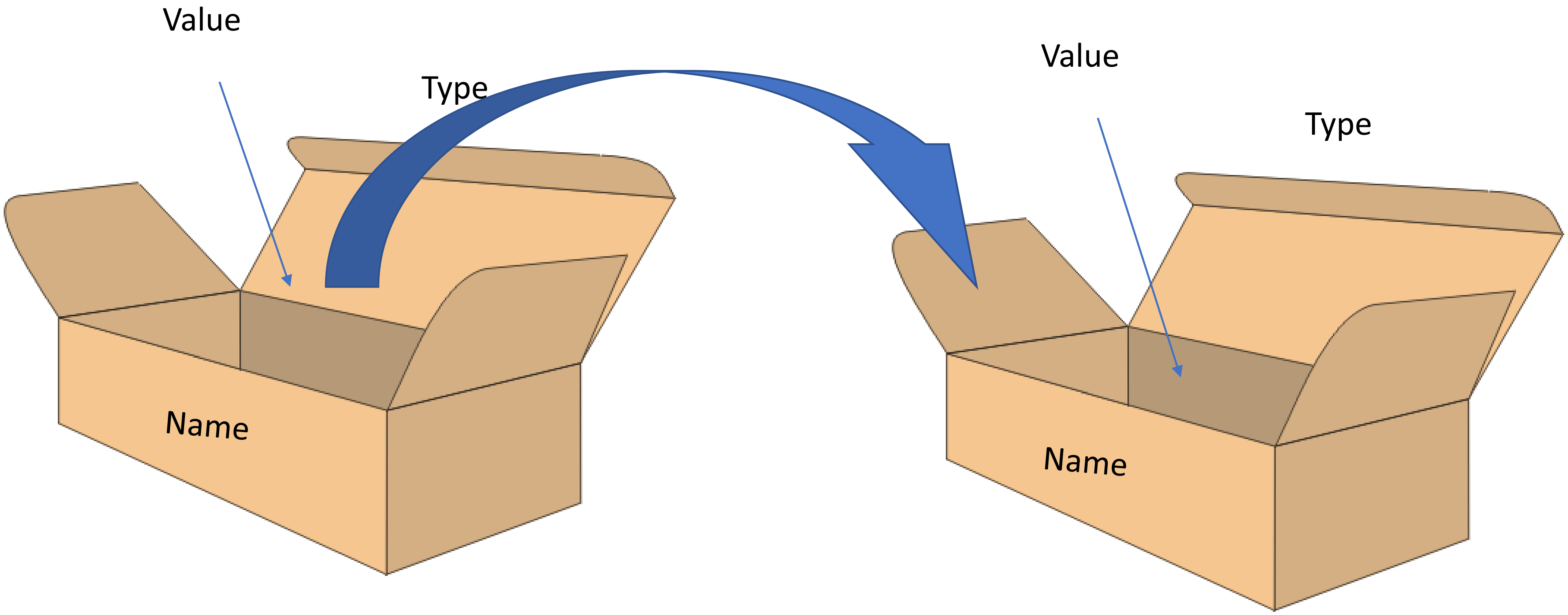
Variable

- 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
  - More about classes later in the course
- Pointer
  - Stores the address of some other variable
  - More about pointers in the course

Variable



Pointer



Pointer

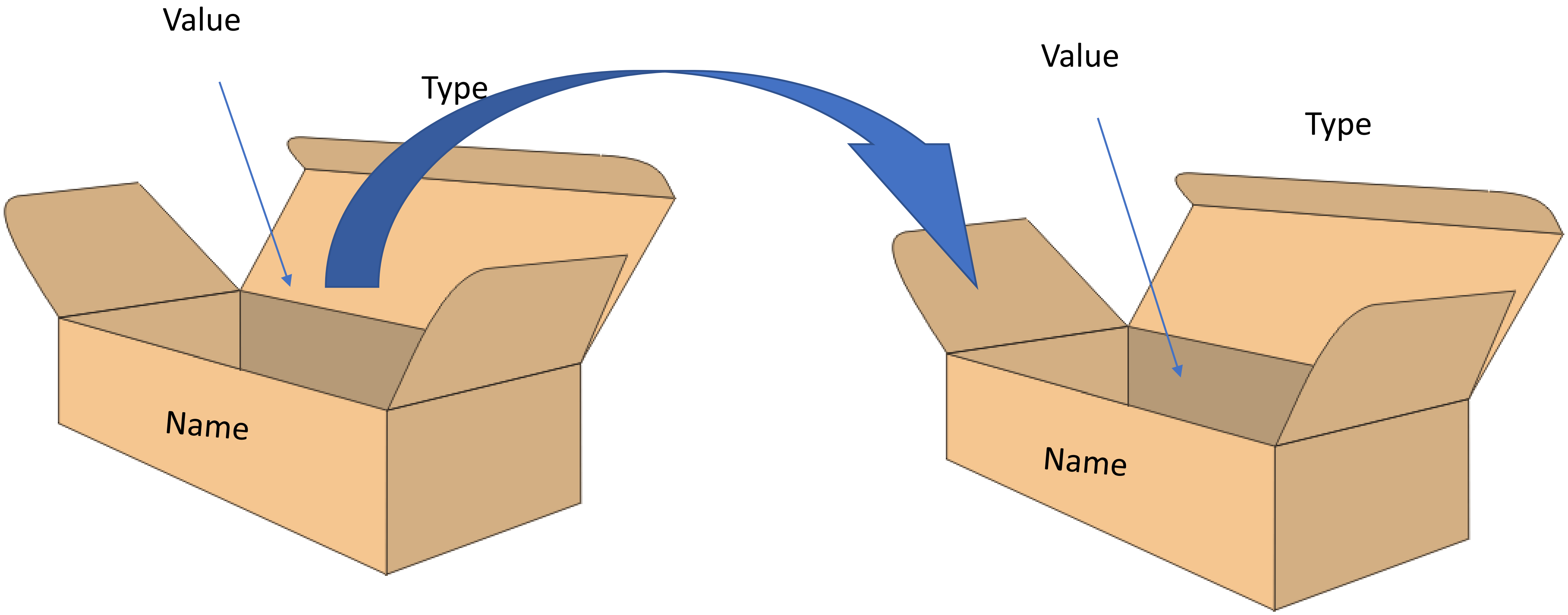
- A variable that stores an address
- Compiler (programmer) keep track of what type each pointer address store in order to index and treat dereference values correct.
- Read declaration backwards

```
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

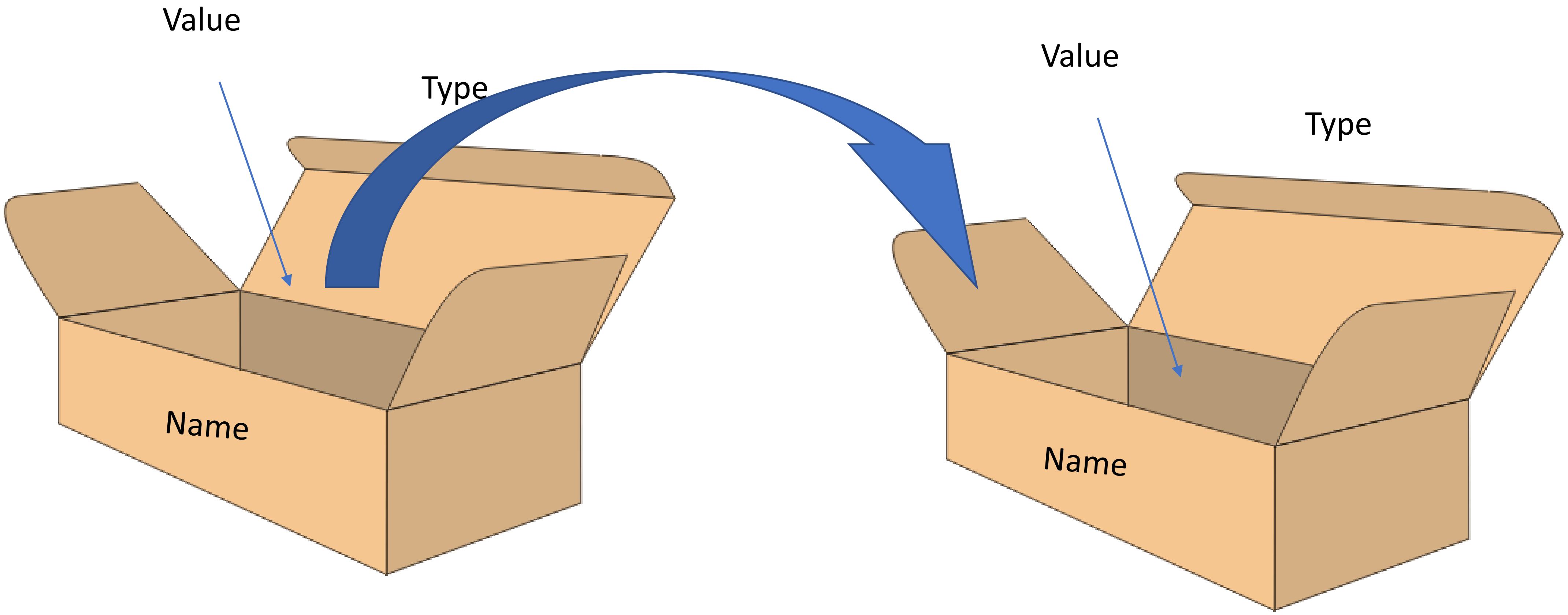
Pointer – Address of



int * int_pointer{&integer_value};

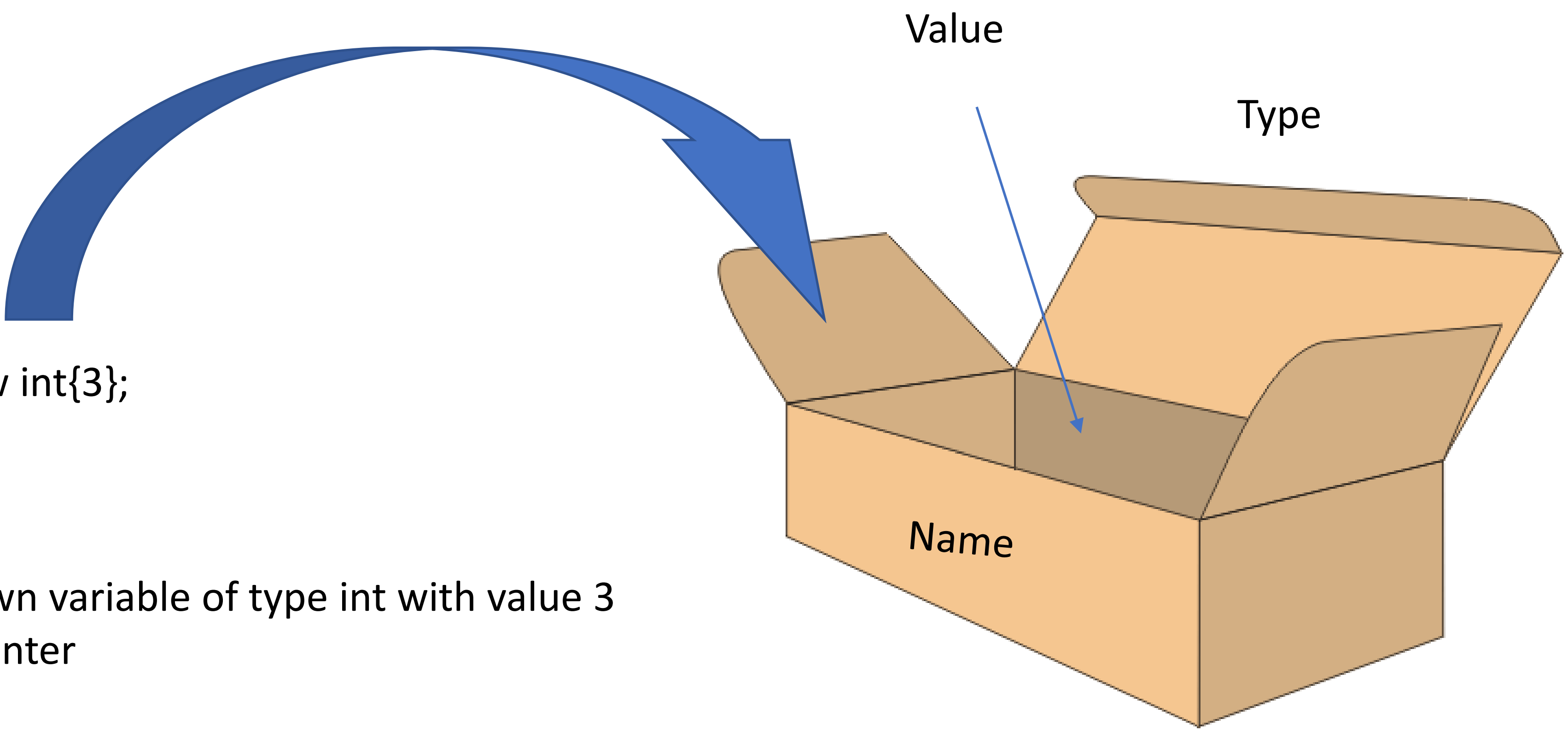
int integer_value{};

Pointer – Dereference



cout << *int_pointer << endl;

Pointer – Allocate



new int{3};

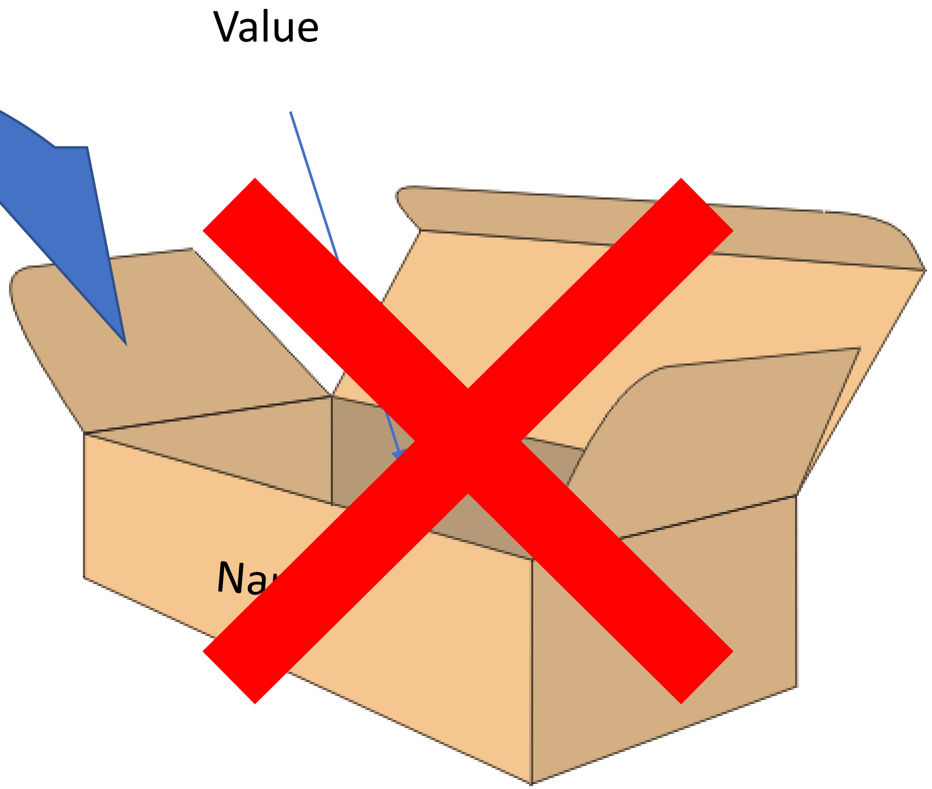
1. Create unknown variable of type int with value 3
2. Return the pointer

Save the pointer by declaring a new variable

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

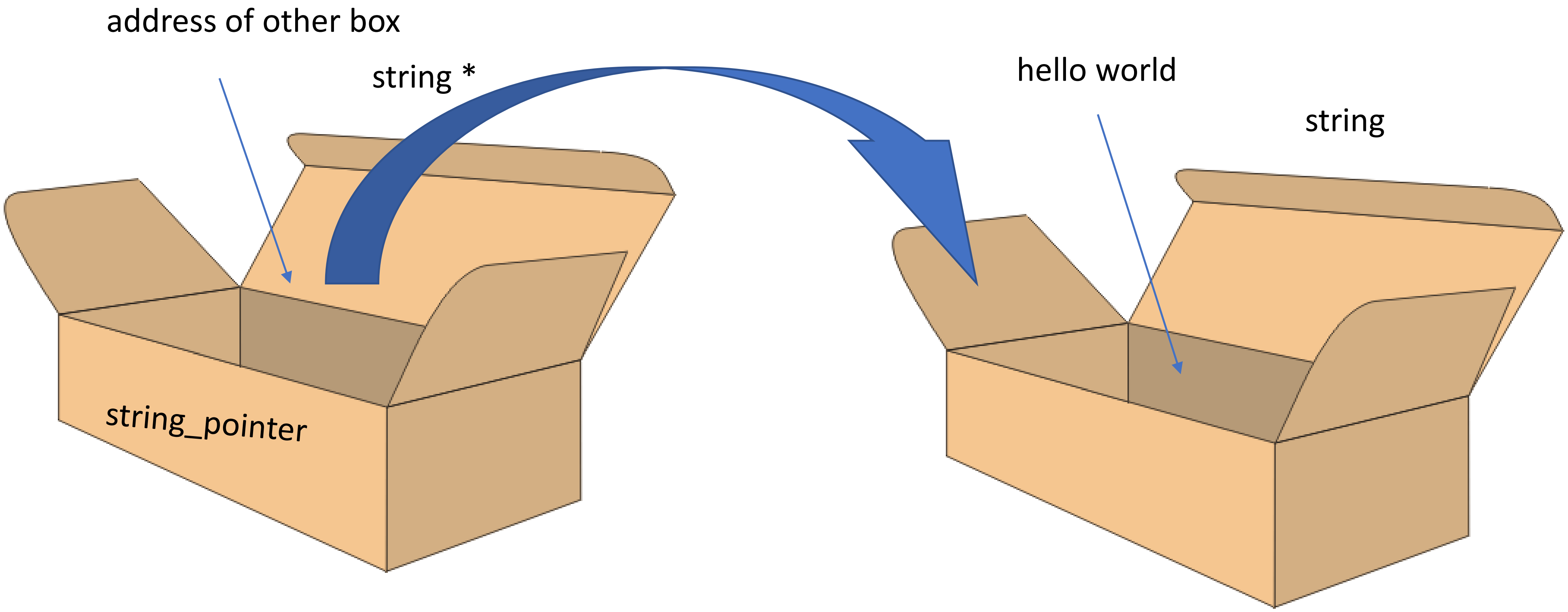
Pointer – Deallocate

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

Class with pointer

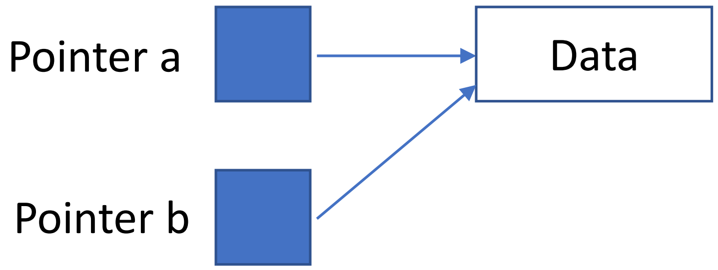
```
class Array {
public:
   Array(int size);
   ...
private:
   int size_;
   int * data;
};
```

What if …

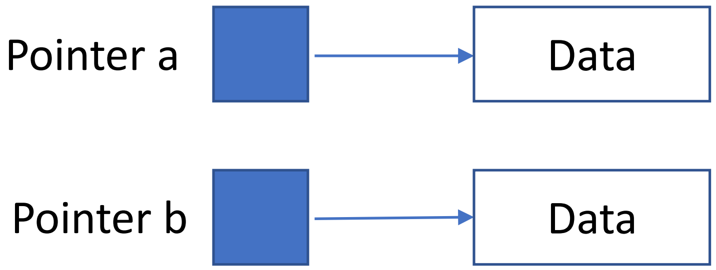
- We pass Array variables as parameter?
- We assign (copy) Array variables?
- We want to initialize an array from another?
- Destroy an Array variable?
- Move an Array variable about to be destroyed to another array?

Shallow copy vs deep copy

Shallow copy

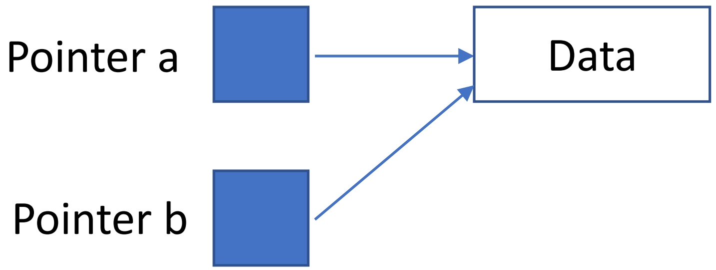


Deep copy



Shallow copy vs deep copy

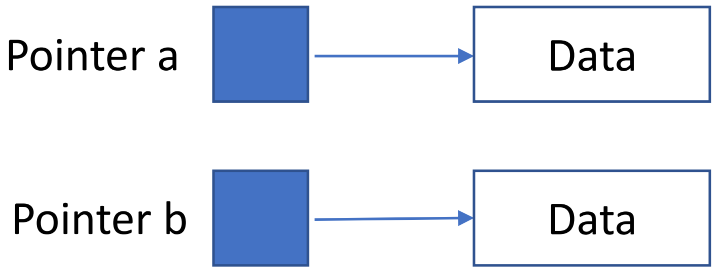
Shallow copy



Example code:

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

Deep copy



Example code:

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

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

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
  - covered later on

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();
  ```

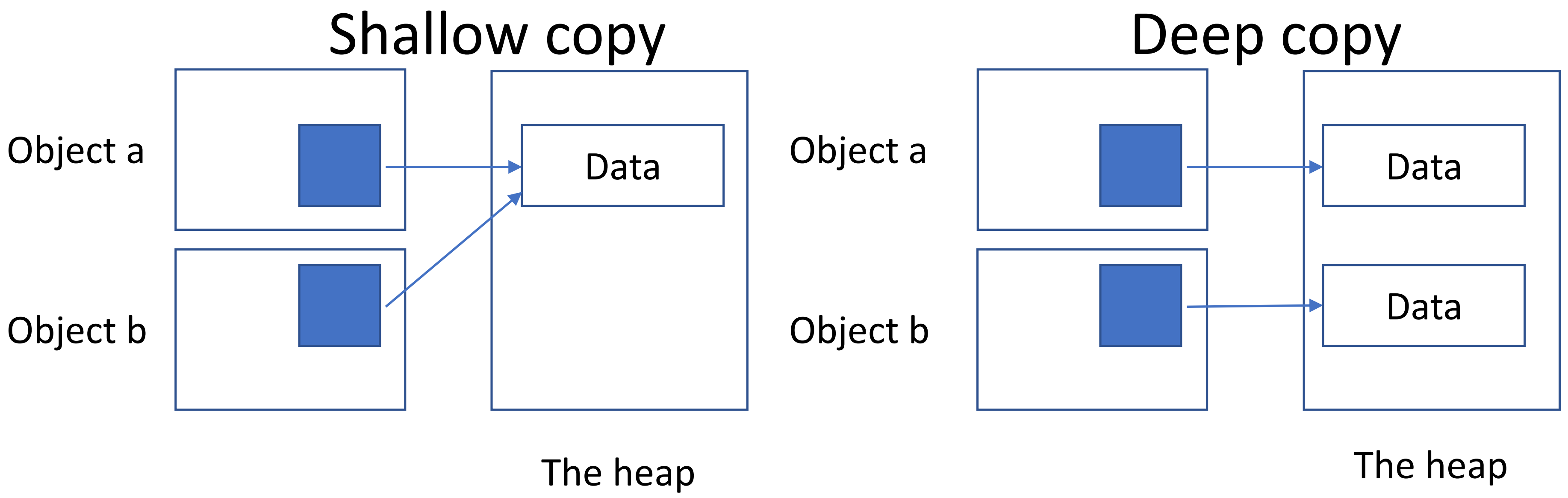
When?

- 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

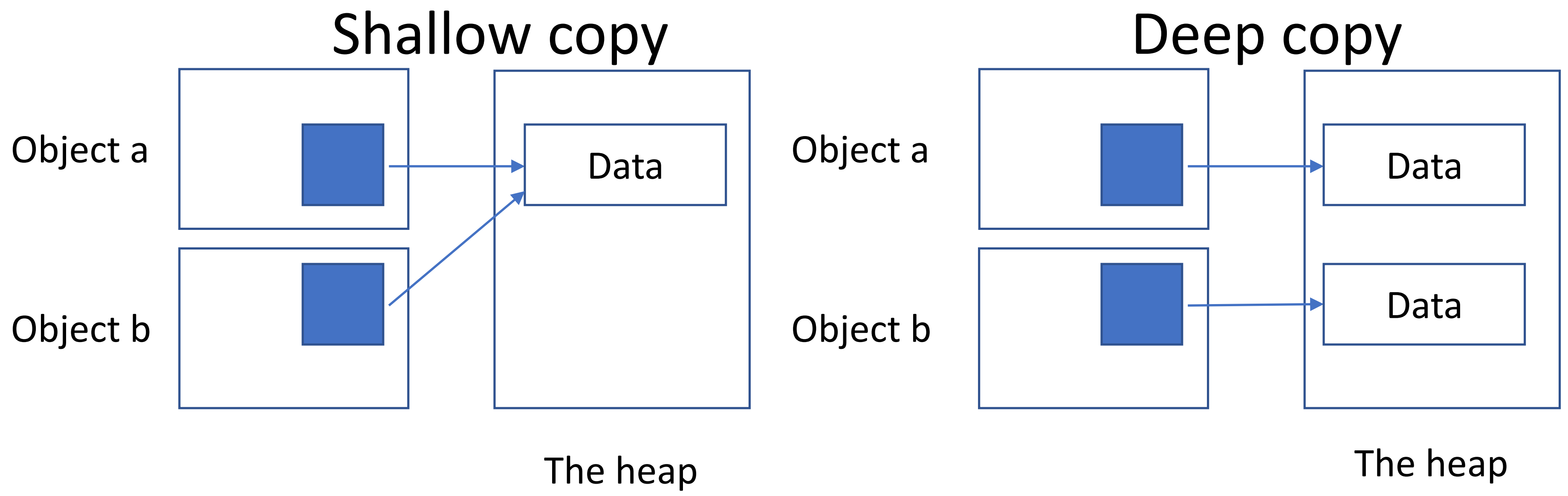
Array class

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

Shallow copy vs deep copy



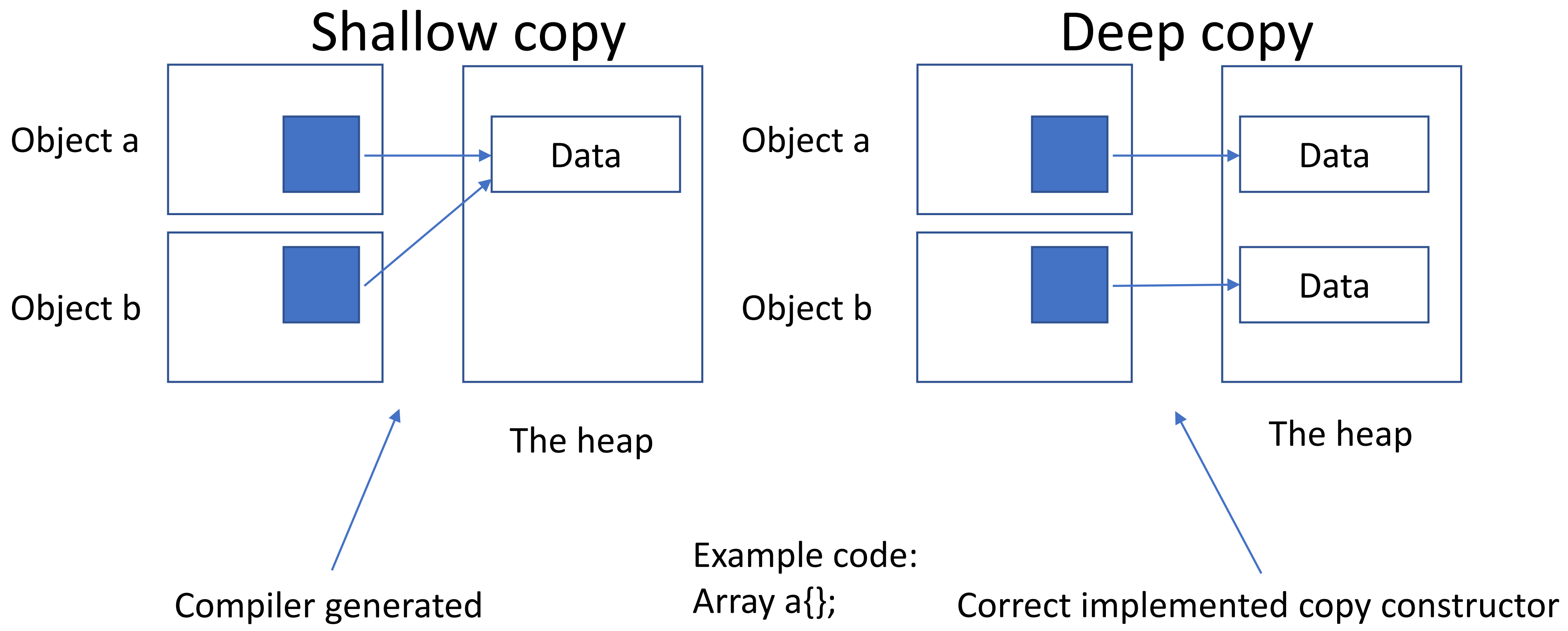
Shallow copy vs deep copy



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);
   ...
};
```

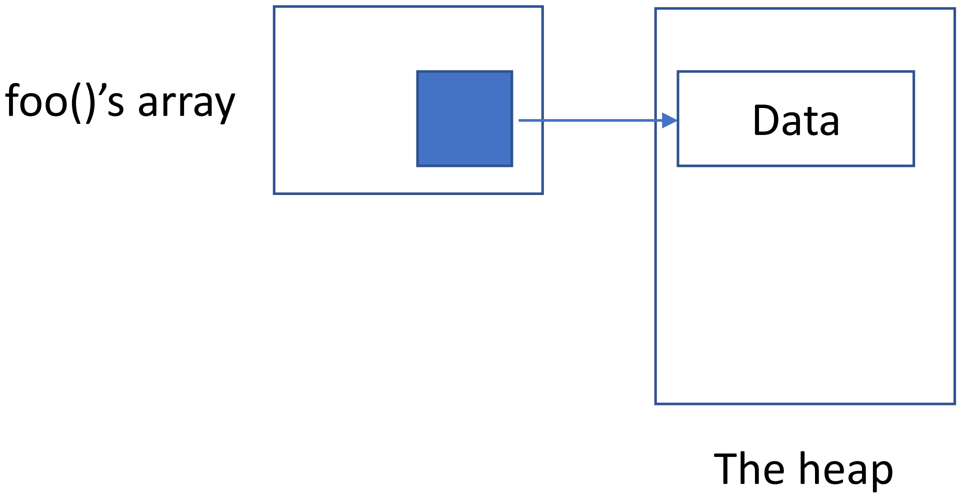
```
// cc-file
Array::Array(Array const& other) {
   // allocate new memory
   // etc
}
```

Temporary variable

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

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

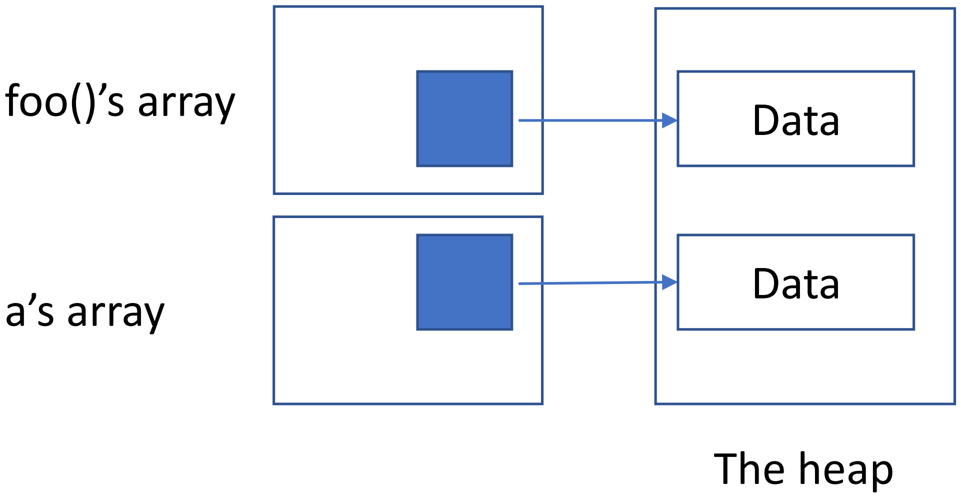
Temporary variable

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

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



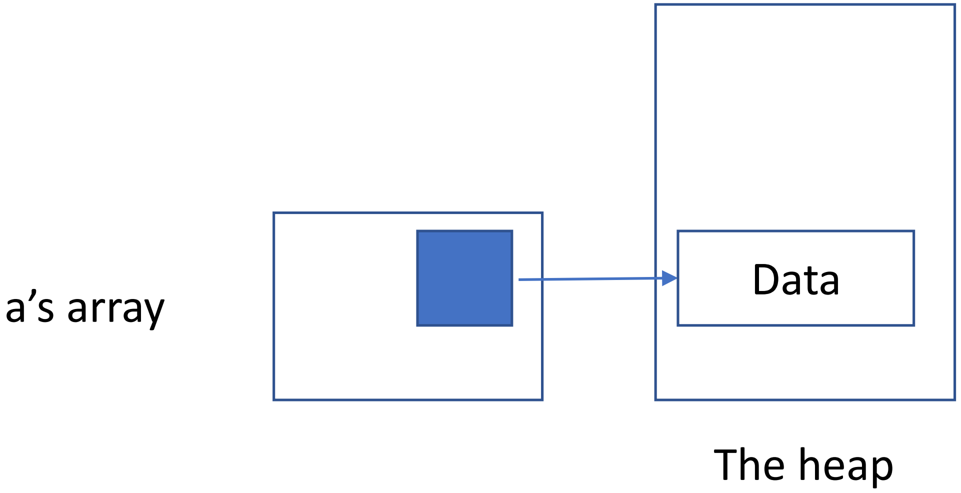
Temporary variable

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

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



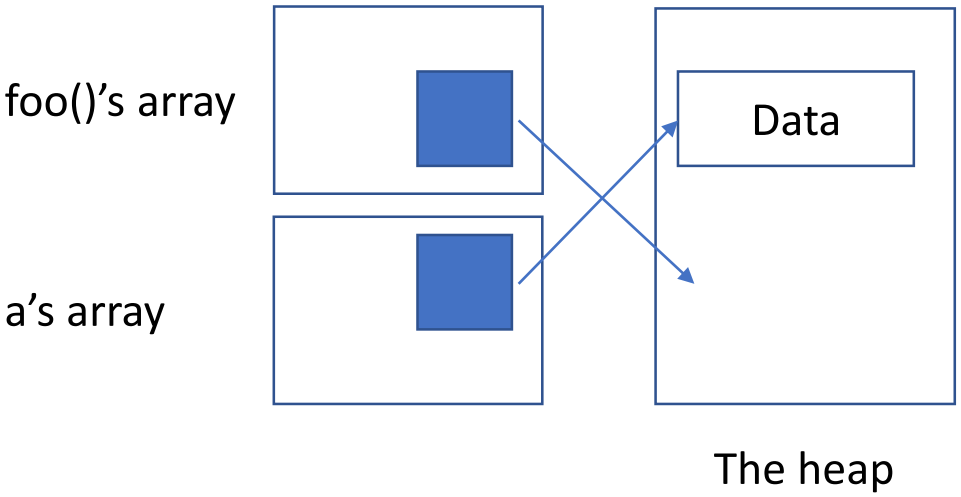
Temporary variable

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

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



Temporary variable

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

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



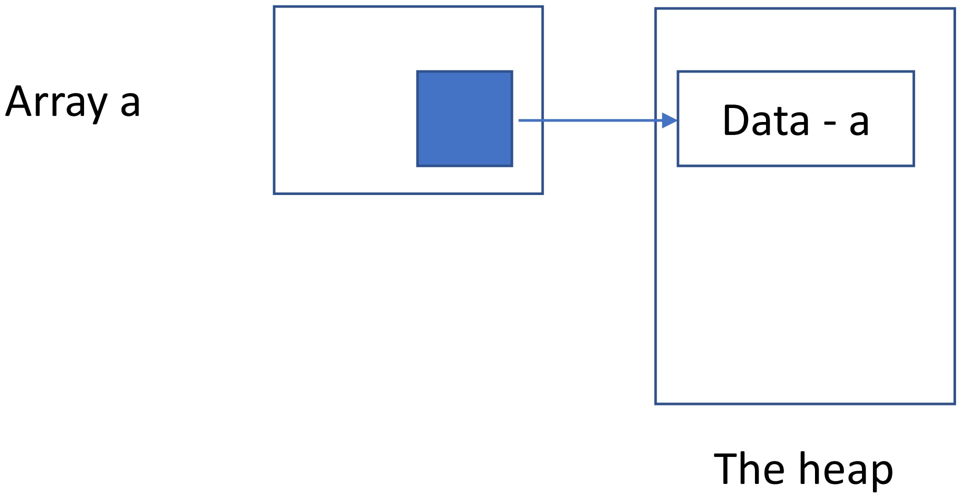
Move constructor – syntax

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

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

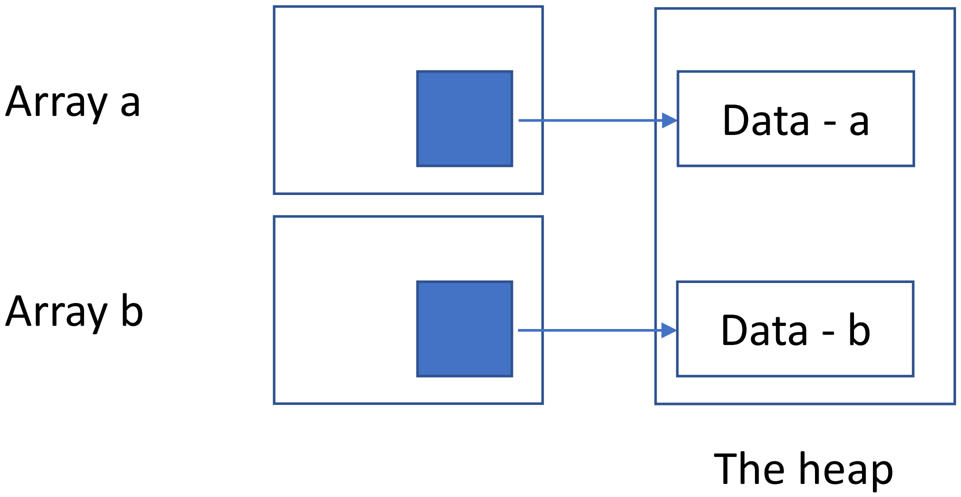
Problems that might occur with copy assignment

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



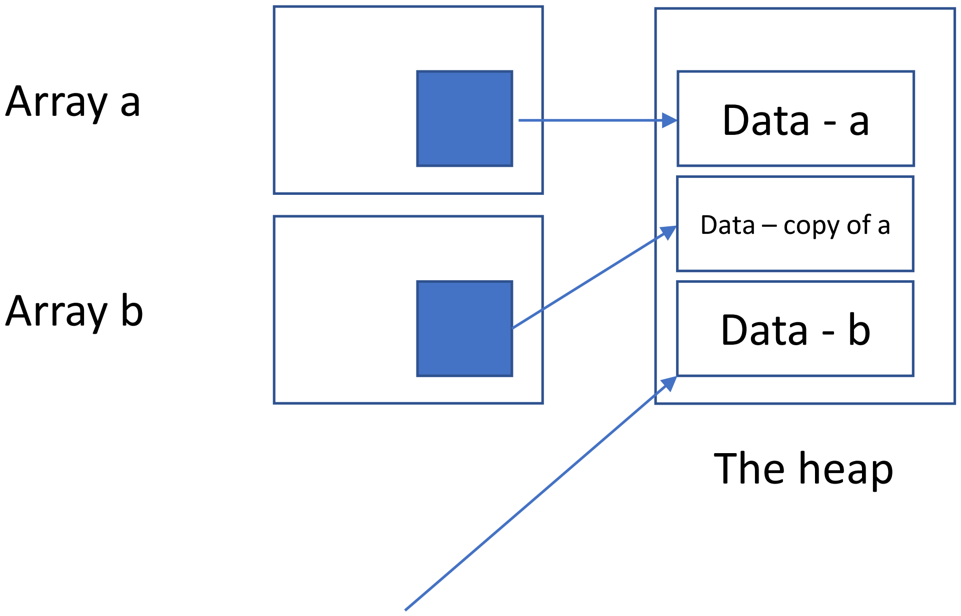
Problems that might occur with copy assignment

```
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

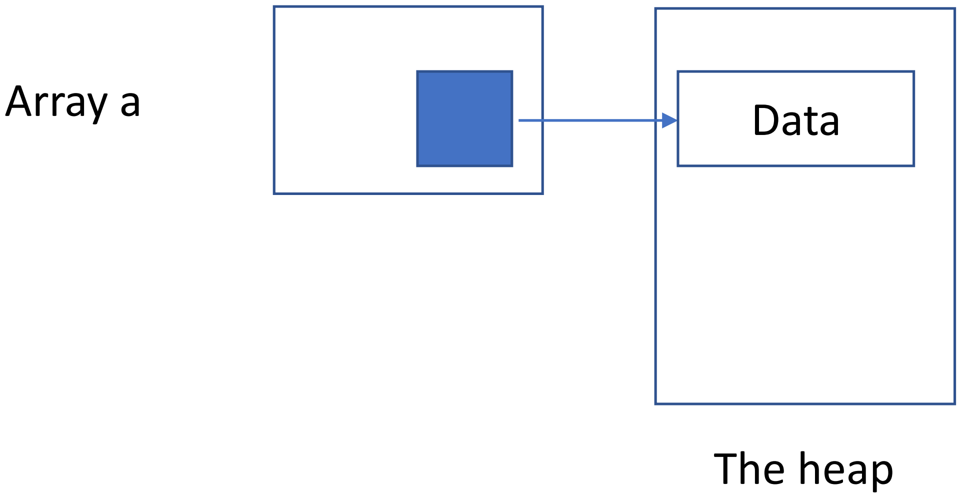
```
// h-file
class Array {
   ...
   Array & operator=(Array  const& other);
   ...
};

// 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
};
```

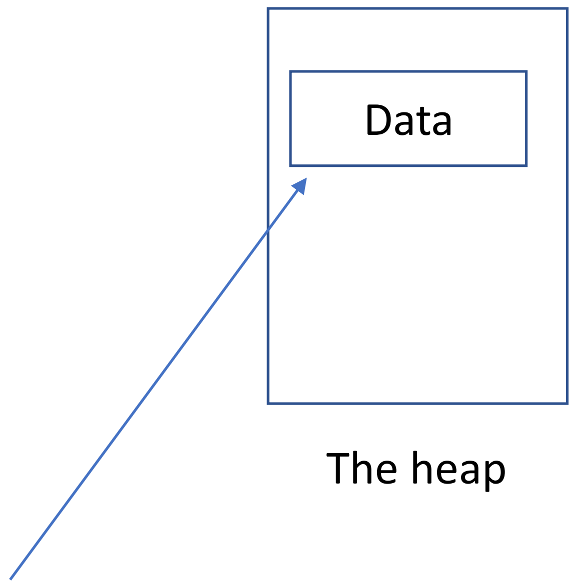
Object that is going to be removed

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



Object that is going to be removed

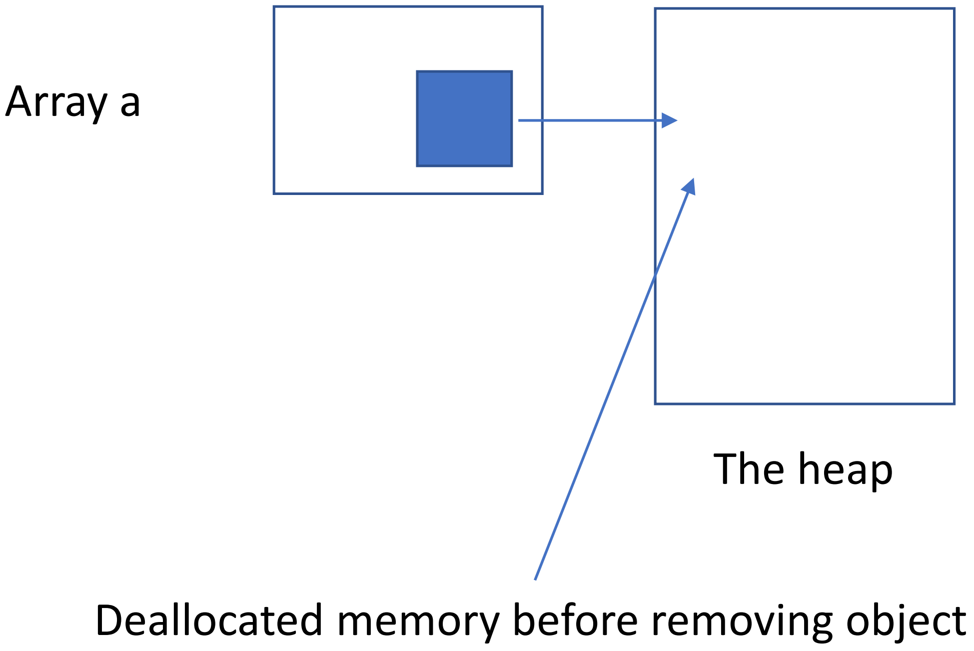
```
int main() {
  Array a{};
}  // a will be removed here
```



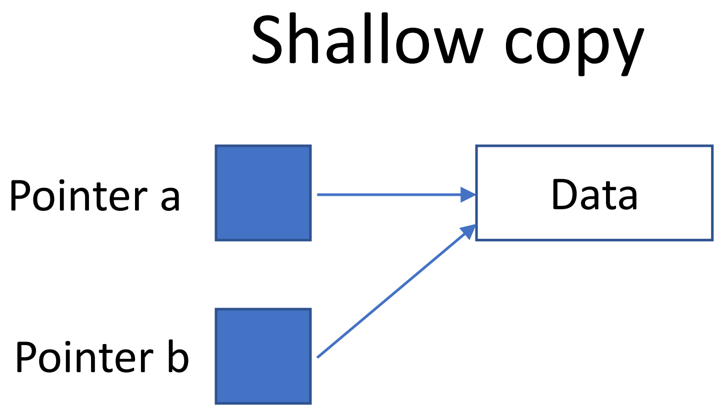
Compiler generated destructor
Data still on the heap

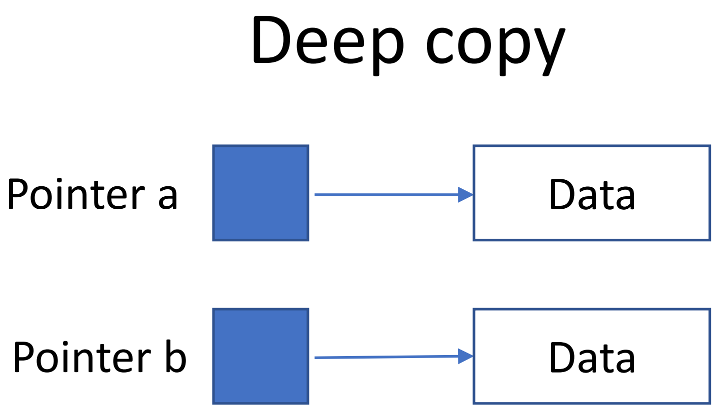
Destructor – syntax

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



Shallow copy vs deep copy





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]
```

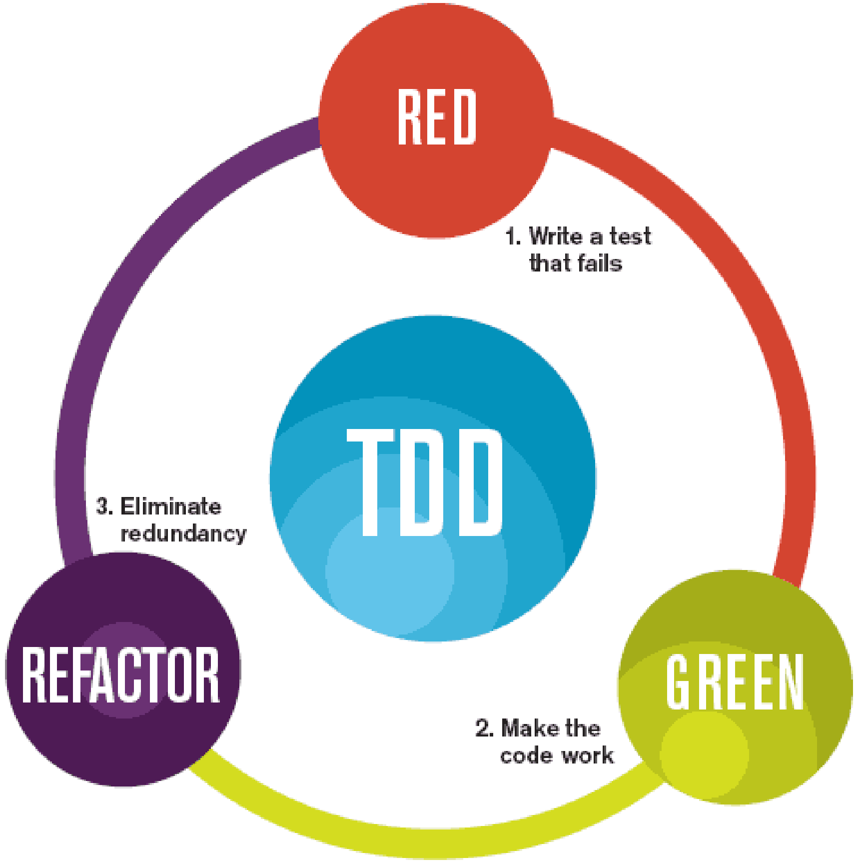
Further reference:

en.cppreference.com

Test first approach

- In lab 4 we want you to write the test before implementation
- We are going to ask you during the lab which test case you are working on
- Catch testing library https://github.com/philsquared/Catch

Test Driven Development



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Using the debugger – command line

- Make sure your compilation command contain the ‘-g’ flag:

```
g++ -g some_buggy_program.cc
```

- Load your program in the debugger:

```
gdb a.out
```

- Start your program, add command line arguments if needed

```
run arg1 arg2 arg3
```

- Do whatever causes your program to crash, and then retrieve a backtrace

```
backtrace
```

- The backtrace will show where the program was executing, and how it got there

A backtrace example

g++11 –g debug_example.cc

gdb a.out

(gdb) **run 1234-56-89**

Starting program: /home/klaar/Cplusplus/a.out 1234-56-89

[Thread debugging using libthread_db enabled]

[New Thread 1 (LWP 1)]

/home/klaar/Cplusplus/a.out is not a date

1234-56-89 is a date

Program received signal SIGSEGV, Segmentation fault.

[Switching to Thread 1 (LWP 1)]

0xff132d50 in strlen () from /lib/libc.so.1

(gdb) **backtrace**

#0 0xff132d50 in strlen () from /lib/libc.so.1

#1 0x00043554 in is_date (str=0x0) at debug_example.cc:10

#2 0x000436b0 in main (argc=2, argv=0xffbfe104) at debug_example.cc:29

Using the debugger – Visual studio code

- https://code.visualstudio.com/docs/editor/debugging