C++ – Static object allocation

Construction

```
Class object = Class();
```

`Animal a = Animal();`	An Animal object is **constructed** (created) …
`Animal a = Animal();`	… and **stored** as the value of the variable **a**
	The object is the **value** of the variable **a**: it is inside **a**

Assignment

```
Class object = otherObject;
```

`Animal b = a;`	The object in variable **a** (the value of **a**) is **copied** …
`Animal b = a;`	… and the new object copy is **stored** as the value of variable **b**
	If the variable **b** is being declared, then this assignment of **a** to **b** implicitly calls the copy constructor of the **`Animal`** class in order to construct the object **b**, passing **a** as the object to copy
	If the variable **b** was already declared, then this assignment of **a** to **b** calls the appropriate operator overload for assignment
	There are now 2 independent objects
	After this, changes to one object do not affect the other

Member access

`object.member`	Use the **dot operator `.`** to access members of the object through the object variable
`a.field = 10;`	Assign a new value to the object's field
`cout << a.field;`	Access the value of the object's field
`a.method();`	Call the object's method

C++ – Dynamic object allocation

Construction

```
Class* pointer = new Class();
```

`Animal* aPtr = new Animal();`	An Animal object is allocated and **constructed** on the heap …
`Animal* aPtr = new Animal();`	… and its **address** is **stored** in the pointer variable `aPtr`

Assignment

```
Class* pointer = otherPointer;
```

`Animal* bPtr = aPtr;`	The value of the pointer variable `aPtr` is **copied** …
`Animal* bPtr = aPtr;`	… and **stored** in another pointer variable, `bPtr`
	The **object is not copied**
	Only the pointer value is copied (the address of the object)
	Only 1 object exists, but now 2 pointers point to it
	After this, changes made to the object through one pointer are visible through the other pointer, and vice versa, because they're both accessing the same object

Member access

`pointer->objectMember`	Use the **arrow operator `->`** to access members of the object through the pointer variable
`aPtr->field = 10;`	Assign a new value to the object's field
`cout << aPtr->field;`	Access the value of the object's field
`aPtr->method();`	Call the object's method

Java – Dynamic object allocation

Construction

```
Class reference = new Class();
```

`Animal aRef = new Animal();`	An Animal object is allocated and **constructed** on the heap …
`Animal aRef = new Animal();`	… and a **reference** to it is **stored** in the reference variable `aRef`

Assignment

```
Class reference = otherReference;
```

`Animal bRef = aRef;`	The value of the reference variable `aRef` is **copied** …
`Animal bRef = aRef;`	… and **stored** in another reference variable, `bRef`
	The **object is not copied**
	Only the reference value is copied (a reference to the object)
	Only 1 object exists, but now 2 references point to it
	After this, changes made to the object through one reference are visible through the other reference, and vice versa, because they're both accessing the same object

Member access

`reference.objectMember`	Use the **dot operator** `.` to access members of the object through the reference variable
`aRef.field = 10;`	Assign a new value to the object's field
`System.out.println(aRef.field);`	Access the value of the object's field
`aRef.method();`	Call the object's method