

COMP 3522

Object Oriented Programming 2
Week 13 Review

Agenda

Review

1. Pointers vs references
2. Static vs dynamic memory
3. Memory allocation & release

COMP

3522

POINTERS vs REFERENCES

When to use pointers vs references?

Pointers and references **SIMILARITIES:**

1. Different ways to indirectly refer to an **existing object/data**:

```
MyClass mc;
```

```
MyClass *mcPtr = &mc; //mcPtr points at memory address of mc
```

```
MyClass &mcRef = mc; //mcRef alias/nickname of mc
```

2. Avoids copying:

```
MyClass *mcPtr = &mc; //mcPtr points at mc's memory address. Can  
access mc without copying
```

```
MyClass &mcRef = mc; // mcRef alias/nickname of mc. mc is NOT COPIED  
to mcRef. They are the same object and share the same memory space. Changes  
to mcRef affect mc, and vice versa
```

When to use pointers vs references?

Pointers and references SIMILARITIES :

1. Both are used to execute polymorphic code:

```
Bird b;
```

```
Animal *aPtr = &b; // pointer
```

```
Animal &aRef = b; // reference
```

```
aPtr->move(); // polymorphically calls Bird's move function
```

```
aRef.move(); // polymorphically calls Bird's move function
```

When to use pointers vs references?

Pointers and references DIFFERENCES:

1. Syntax when accessing object's functions:

```
MyClass mc;
```

```
MyClass *mcPtr = &mc;
```

```
MyClass &mcRef = mc;
```

```
mcPtr->myFunc(); //calling mc's function with -> notation
```

```
mcRef.myFunc(); //calling mc's functions with . notation
```

2. CAN **dereference** pointers, CAN'T dereference reference to an object:

```
cout << *mcPtr << endl; //dereferences mcPtr to get mc
```

```
cout << *mcRef << endl; //ERROR
```

When to use pointers vs references?

Pointers and references DIFFERENCES:

3. Pointers store memory addresses. References are aliases to variables (behind the scenes they're pointers to other variables)

4. CAN assign nullptr / "nothing" to pointers. Pointers CAN point to different memory addresses later:

```
MyClass mc;  
MyClass *mcPtr = nullptr;  
mcPtr = &mc;
```

5. CAN'T assign nullptr/"nothing" to references. CAN'T assign reference to refer to a different variable later:

```
MyClass &mcRef = nullptr; //ERROR
```

When to use pointers vs references?

6. References MUST be initialized on declaration:

```
MyClass &mcRef2; //ERROR
```

```
MyClass &mcRef3 = mc;
```

Use references when:

1. Need to refer to an **existing object**

```
MyClass &ptr = myClassObject;
```

2. Pass by reference. Avoids copying arguments

```
void myFunction(MyClass &paramRef) {}
```

When to use pointers vs references?

Use pointers when:

1. Need the ability to **assign to null/nothing** temporarily
`MyClass *ptr = nullptr;`
2. Need to **point to memory of existing object** instead of copying it
`MyClass *ptr = &myClassObject;`
`MyClass *ptr2 = ptr;`
3. Avoid copying data by **passing pointers to functions**
`void myFunction(MyClass* paramPtr) {}` //MyClass object not copied, only memory address passed in
4. Need to use **dynamic memory**
`MyClass *ptr = new MyClass;`

STATIC VS DYNAMIC MEMORY

Object *name = new Object vs Object name

```
void createObject() {  
    Object name;  
}
```

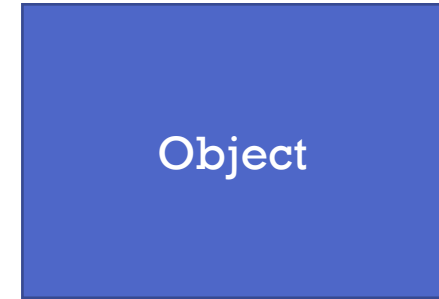
```
//main.cpp  
createObject();  
//some other code
```

- Let's look at statically allocated memory first
- Create object inside of function

Object *name = new Object vs Object name

```
void createObject() {  
    Object name;  
}
```

```
//main.cpp  
createObject();  
//some other code
```

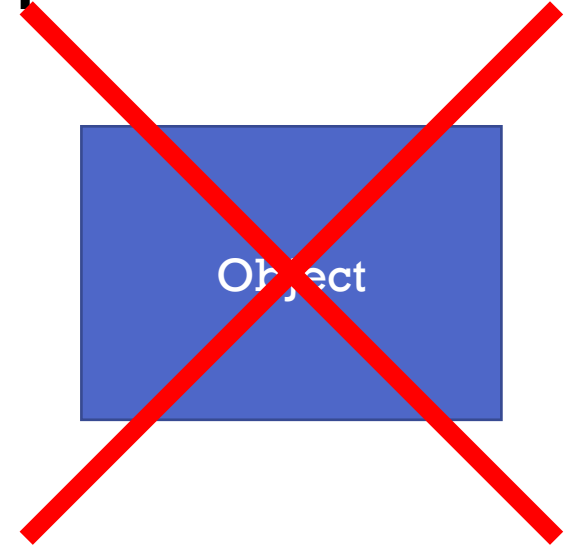


Create object in memory
(stack)

Object *name = new Object vs Object name

```
void createObject() {  
    Object name;  
}
```

```
//main.cpp  
createObject();  
//some other code
```



After leaving createObject function, object is released from memory (stack)

Object *name = new Object vs Object name

```
void createObject() {  
    Object *name = new Object;  
}
```

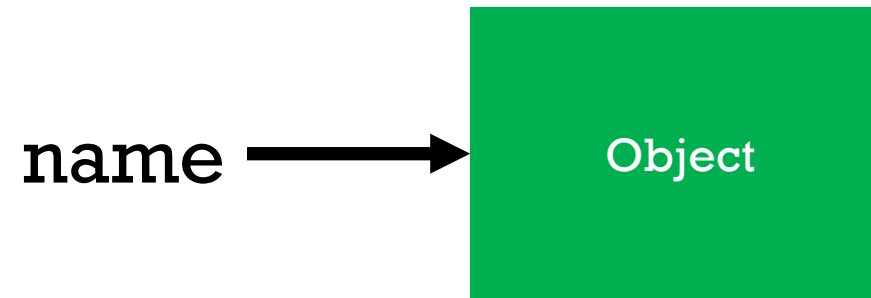
```
//main.cpp  
createObject();  
//some other code
```

- Now look at dynamically allocated memory

Object *name = new Object vs Object name

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```



- Create **object** in memory (heap)
- name pointer points to **object**

Object *name = new Object vs Object name

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```

~~name~~



After leaving function, pointer memory is released, but **object** still exists in memory (heap)

Object *name = new Object vs Object name

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```



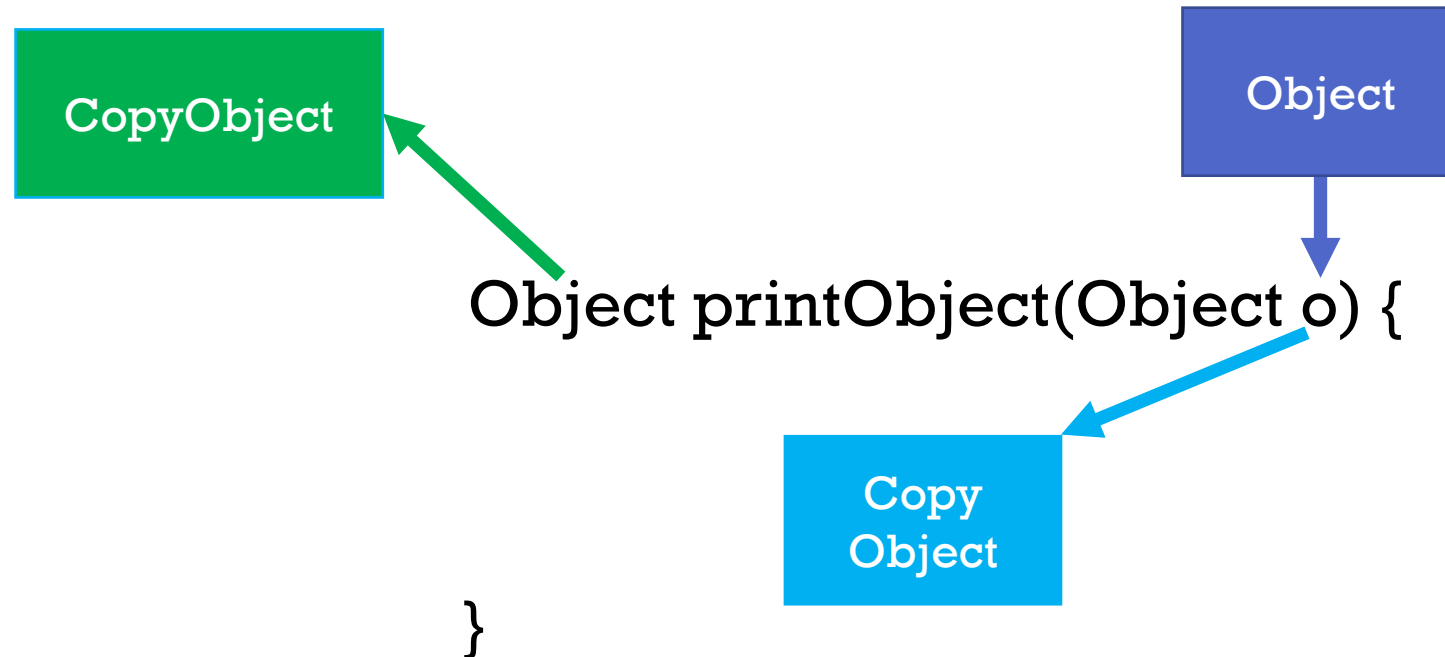
- This is a memory leak because there's no pointer pointing to **object**
- No way to call delete on the **object**

Why use dynamic memory?

- There are situations where you want memory to **persist beyond the scope of where the memory is created**
- This is where you use **dynamic memory**
- It's more efficient than statically allocating memory and copying that memory everywhere it's needed
- Cases where you don't know how much memory you'll need at run-time
 - Think of linked lists
 - Don't know ahead of time how long the linked list is
 - Need the linked list to dynamically add new nodes

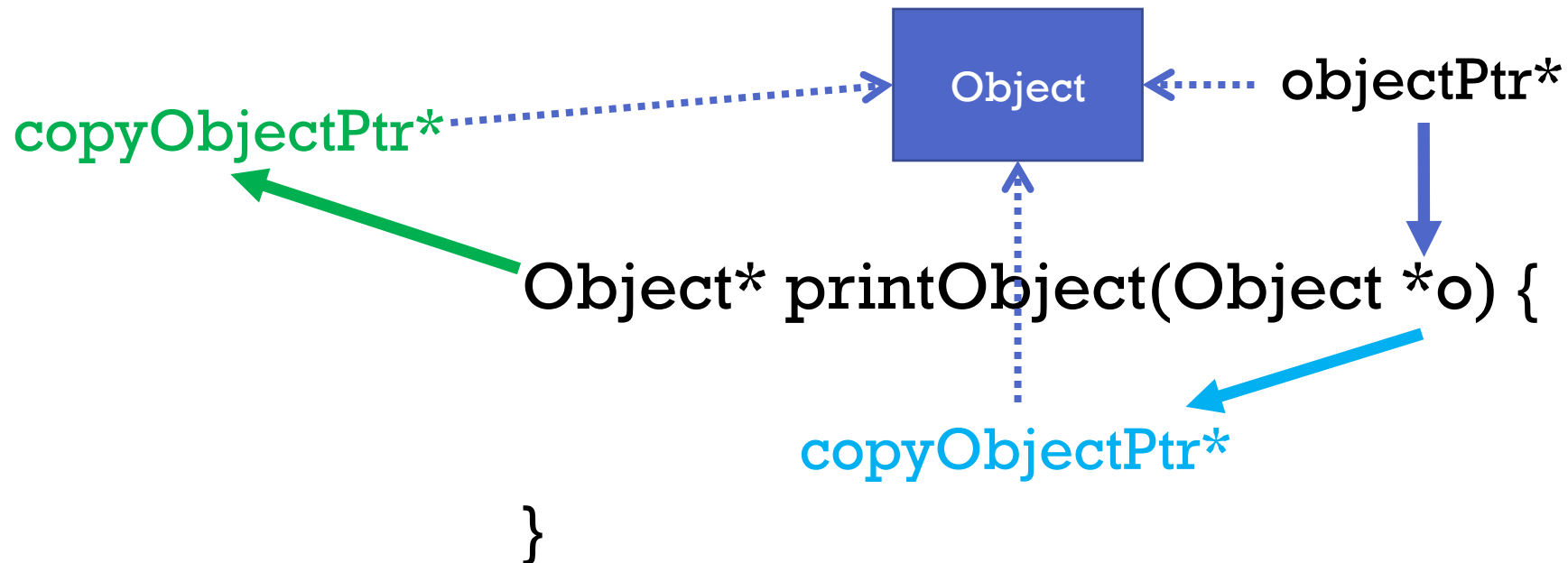
Pass by value

- This shows calling a function by passing an object
- Look at all the **copied objects**
 - Expensive operation and memory allocation



Pass by value

- This shows calling a function by an object pointer
- Only **one object** is pointed at, no object copies made
- Only the pointers are copied, this is fast compared to copying object



MEMORY: ALLOCATION & RELEASE

`dynamicMemory.cpp`