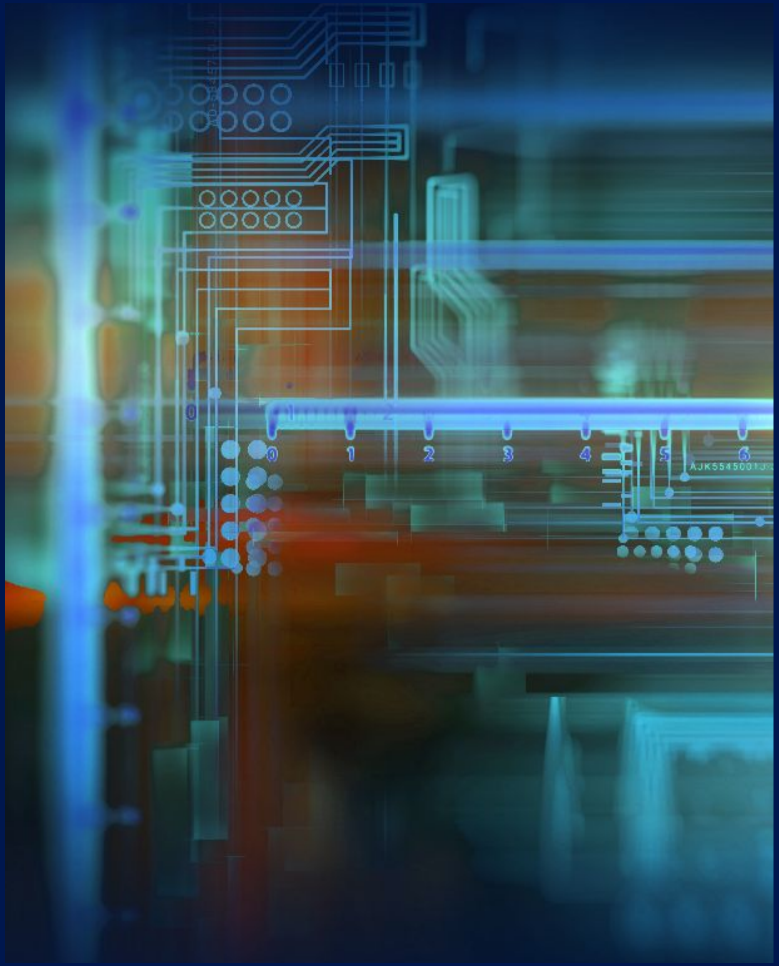
COP3503

Constructors and Destructors

Welcome!



Placeholder for the instructor's welcome message. Video team, please insert the instructor's video here.

Objects Should Be Initialized Before Use

```
class Widget
{
    string name;
public:
    // MUST call this function right away, or
else!
    void Initialize(string widgetName);
    {
        name = widgetName;
    }
    void DoSomething()
    {
        cout << name << endl;
    }
};
```

A way to guarantee that initialization would be helpful!

```
// Proper usage
int main()
{
    Widget widget;
    widget.Initialize();
    widget.DoSomething();
    return 0;
}
```

```
// What if we forget…?
int main()
{
    Widget widget;

    // What happens here?
    widget.DoSomething();
    return 0;
}
```

Constructors

- Class member functions are used to **initialize** an object.
- Constructors are called **only once**, when an object is first instantiated.
- The function name is the same as the class.
- There is no return type (not even **`void`**).

```
class Widget
{
public:
    Widget();    // Constructor
    int Foo();
    void Bar();
};
```

Constructors

Usage

The number of arguments may change, but the intent of a constructor is always the same: initialize an object.

The class will use those parameters for whatever it needs in the implementation of those functions.

```
class Car
{
    double  _price;
    int     _miles;
    string  _make;
    string  _model;
public:
    // Overloading constructors is okay
    Car();
    Car(string make, string model);
    Car(double price, int miles, string make, string model);
};
int main()
{
    Car car; // Default constructor doesn’t need ()
    Car otherCar("Honda", "Civic");
    Car yetAnotherCar(18000, 768, "Toyota", "Prius");

    return 0;
}
```

Constructors

Behind the Scenes

```
class Car
{
    double  _price;
    int     _miles;
    string  _make;
    string  _model;
public:
    // Overloading constructors is okay
    Car();
    Car(string make, string model);
    Car(double price, int miles, string make, string model);
};
```

Constructors

Behind the Scenes

```
Car::Car()
{
    _price = 0;
    _miles = 0;
    _make = "Unknown";
    _model = "Car";
}
```

```
class Car
{
    double  _price;
    int     _miles;
    string  _make;
    string  _model;
public:
    // Overloading constructors is okay
    Car();
    Car(string make, string model);
    Car(double price, int miles, string make, string model);
};
```

Constructors

Behind the Scenes

```
class Car
{
    double  _price;
    int     _miles;
    string  _make;
    string  _model;
public:
    // Overloading constructors is okay
    Car();
    Car(string make, string model);
    Car(double price, int miles, string make, string model);
};
```

```
Car::Car()
{
    _price = 0;
    _miles = 0;
    _make = "Unknown";
    _model = "Car";
}
```

```
Car::Car(string make, string model)
{
    _price = 0;
    _miles = 0;
    _make = make;
    _model = model;
}
```

Constructors

Behind the Scenes

```
class Car
{
    double  _price;
    int     _miles;
    string  _make;
    string  _model;
public:
    // Overloading constructors is okay
    Car();
    Car(string make, string model);
    Car(double price, int miles, string make, string model);
};
```

```
Car::Car()
{
    _price = 0;
    _miles = 0;
    _make = "Unknown";
    _model = "Car";
}
```

```
Car::Car(string make, string model)
{
    _price = 0;
    _miles = 0;
    _make = make;
    _model = model;
}
```

```
Car::Car(double price, int miles, string make, string
model)
{
    _price = price;
    _miles = miles;
    _make = make;
    _model = model;
}
```

Default Constructors

- A specific form of constructor that either:
  - Takes **no arguments**, **or**
  - Takes in all **default arguments**.
- A class can have many constructors, but no more than **one** default.

Default arguments can be a helpful shortcut, and they provide you with convenient options.

```
class Car
{
    double _price;
    int _miles;
public:
    Car(double price = 1000, int miles = 0);
};
```

Default values for these arguments go in the function prototype.

```
Car one(0, 500);
Car two(2750); // Same as: two(2750, 0)
Car default;   // Same as: default(1000, 0);
```

Default Constructors

Limit 1 Per Class

```cpp
class Car
{
    double _price;
    int _miles;
public:
    Car() // Default
    {
        _price = 0;
        _miles = 0;
    }
    Car(double price = 1000, int miles = 0) // Also a default
    {
        _price = price;
        _miles = miles;
    };
};
```

Both of these constructors qualify as a default, but you can only have one.

Or is it 0?

Is it 1000?

What's the price of this car?

```cpp
Car instance;
```

Quick Detour to Default Arguments

- Once you assign one argument a default value, all arguments **after** that must also have a default value.

After == next argument to the right

```
void Foo(int x = 5, int y = 2, int z = 0); // Okay
Foo(17, 6); // z == 0, as a default

void Foo(int x, int y = 2, int z = 0); // Okay
Foo(199); // y == 2 and z == 0 as defaults

void Foo(int x, int y, int z = 0); // Okay
void Foo(51, 40, 12); // Okay, overwriting the default value

void Foo(int x = 100, int y, int z = 25); // Not okay
Foo(5);         // What is assigned the 5? x? y?
Foo(__, 12);    // How to use a default for x, but not y?
```

What if You Don’t Write a Constructor?

- If you don’t write one, an **implicit constructor** is created.
- It takes **no arguments** and **does nothing** but qualifies as a default.

```
class Example
{
    Example() // Implicitly declared, default constructor
    {
        // No parameters, does nothing
    }
};
```

If you write **any** constructors, default or not, the compiler will not create this empty constructor for you.

Why Should You Have a Default Constructor?

```cpp
// Given this class…
class Point
{
public:
    int _x, _y;
    Point(int x, int y);
};

Point::Point(int x, int y)
{
    _x = x;
    _y = y;
}
```

```cpp
int main()
{
    Point pt(2, 4); // Constructor
    Point pt2; // No default constructor, error

    // This MUST use the default constructor
    // Create 10 DEFAULT objects
    Point points[10];

    return 0;
}
```

This class has **no** default constructor (that's not inherently a bad thing).

It just means that the object **must** be constructed with values.

Without a default constructor, you can't create an array of objects of this type (each element in the array gets the default constructor called)

Do Constructors Have Size Limits?

```cpp
class BigObject
{
    int x, y, z;
    float a, b, c;
    string strings[5];
public:
    BigObject(int val1, int val2, int val3, float val4, float val5, float val6,
                   string s1, string s2, string s3, string s4, string s5);
};
```

```cpp
BigObject instance(2, 4, 6, 1.2, 2.4, 2.5, "This", "is", "a", "lot", "of", "arguments");
```

```cpp
// Break initialization into smaller pieces
BigObject instance();
instance.SetIntegers(2, 4, 6);
instance.SetFloats(1.2, 2.4, 2.5);
instance.SetStrings("This", "is", "a", "lot", "of", "arguments");
```

Two options:
1. A really big constructor
2. A small constructor with multiple initialization steps afterward

Destructors

- Class member function that is automatically invoked when an object **falls out of scope**
- The last function an object invokes
- Used to "clean up" the object

```
class Example
{
public:
    Example(); // Constructor
    ~Example();    // Destructor
};
```

- A variable can fall out of scope when:
  - A function ends
  - A program ends
  - A block of code ends (such as ifs/loops)
  - The delete keyword is called on a dynamically allocated pointer (this is a later topic)

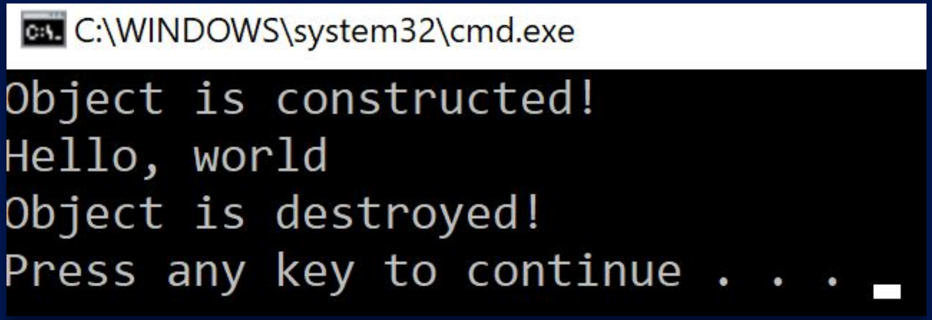
Destructors

What Do They Do?

- Whatever a class needs to “finish” its job
- For many classes, that might be nothing at all.
- Critical when **dynamic memory** is involved (more on this later!)

```cpp
class Example
{
public:
    Example()    { cout << "Object is constructed!" << endl; }
    ~Example() { cout << "Object is destroyed!" << endl; }
};

int main()
{
    Example instance;
    cout << "Hello, world" << endl;
    return 0;
}
```



C:\WINDOWS\system32\cmd.exe

```
Object is constructed!
Hello, world
Object is destroyed!
Press any key to continue . . .
```

Do You Have to Write a Destructor?

- C++ requires a destructor for all classes.
- If you don’t write one, an **implicit destructor** is created for you.

If you don’t use dynamic memory, then you probably don’t need a destructor.

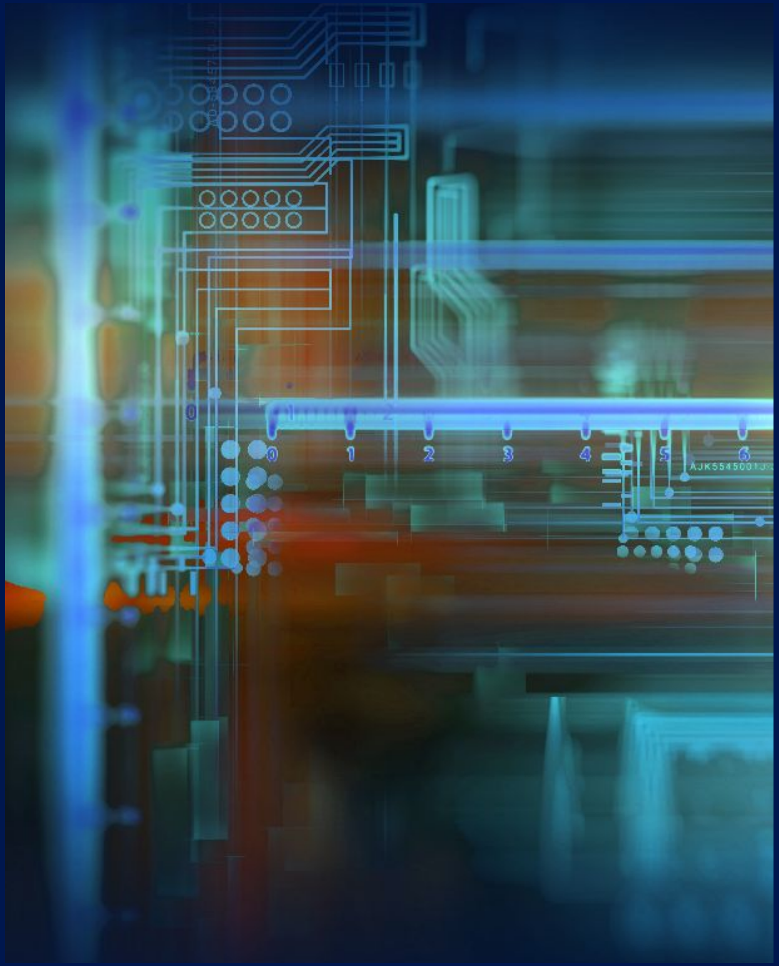
Some languages may not even have destructors!

```
class Example
{
public:
    ~Example()
    {
        // Does nothing by default
    }
};
```

Recap

- All objects must be created and destroyed.
- **Constructors** are invoked on object creation.
- **Destructors** are invoked when objects fall out of scope.
  - Or when they are deleted, with dynamic memory
- The details of each of those steps depends on the class.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.