The final exam will be online supervised exam (via Proctorio).

Outline

OOP in C++

1. What is it?
2. Why OOP?

Class and Object

3. Create a C++ Class
4. Generate a C++ Object
5. Attributes, Method

Encapsulation

6. Struct and Class
7. Public, Private, and Protected

Object Management

8. Constructors
9. *Destructors ('this' pointer)

Copy operations

10. Copy constructor
11. Copy assignment
12. Deep and shallow copy

Function overloading

1. C++ OOP

What is OOP?

Concept: Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define the data type of a data structure, and also the types of operations (functions) that can be applied to the data structure.

Source: https://www.webopedia.com/TERM/O/object_oriented_programming_OOP.ht

Why OOP?

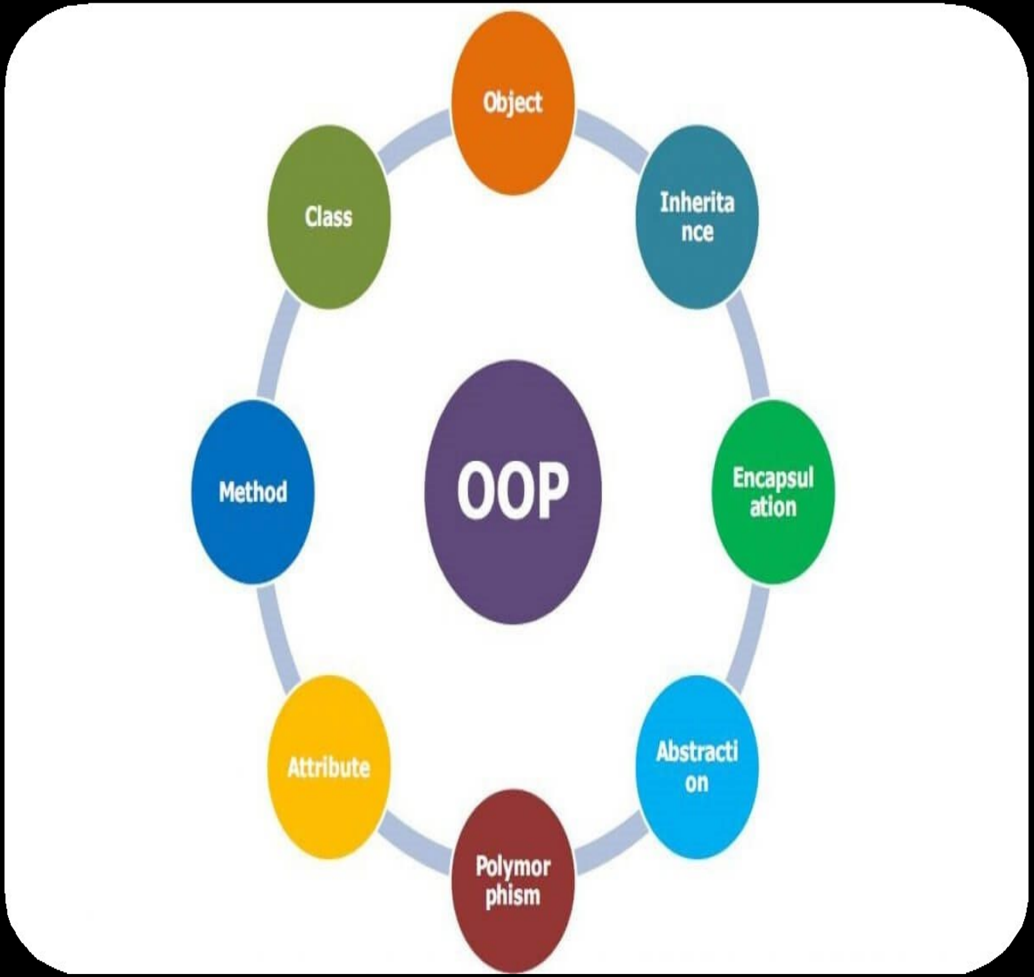
- To create new type of data structure
- Clear structure for the programs
- To reflect things in the real world.
- Faster and easier to execute
- Keep code DRY “Don’t Repeat Yourself”

Source: https://www.w3schools.com/cpp/cpp_oop.asp

OOP Components?

Components relating to OOP

- Object
- Class
- Attributes
- Method
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



2. Class and Object

What is C++ Class?

Class is most important in OOP in C++.

Class is the new type of data structure that includes attributes and methods to reflect things in the real world.

What is C++ Class?

A class is a user defined data type:

- provide a description for building objects.
- provide prototypes or blueprints for objects.
- provide a way to group related data and the functions which are used to process the data
- you create an object from the class, you automatically create all the related fields

Abstract data type (ADT): a type you define.

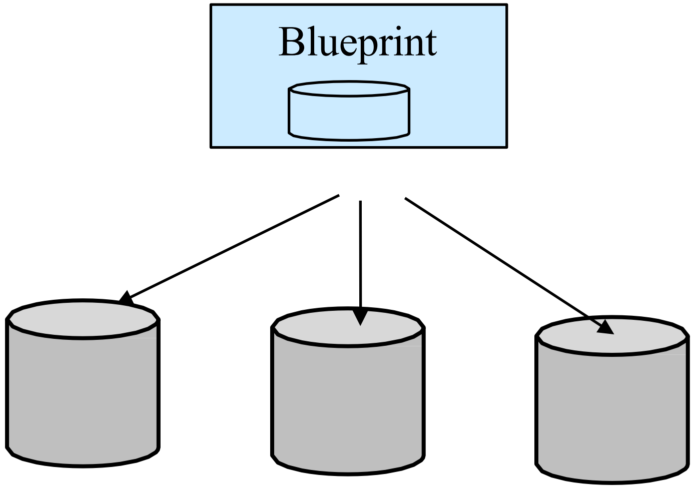
Why we use Class?

- To create new type of data structure
- To reflect things in the real world.
- To model real world problem.

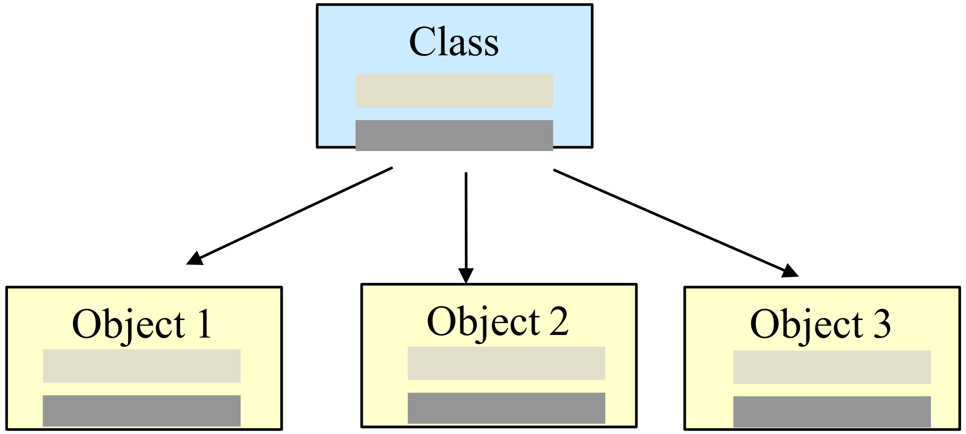
Source: https://www.w3schools.com/cpp/cpp_oop.asp

Class and Objects

Parts of a technical system are produced based upon their description: blueprints.

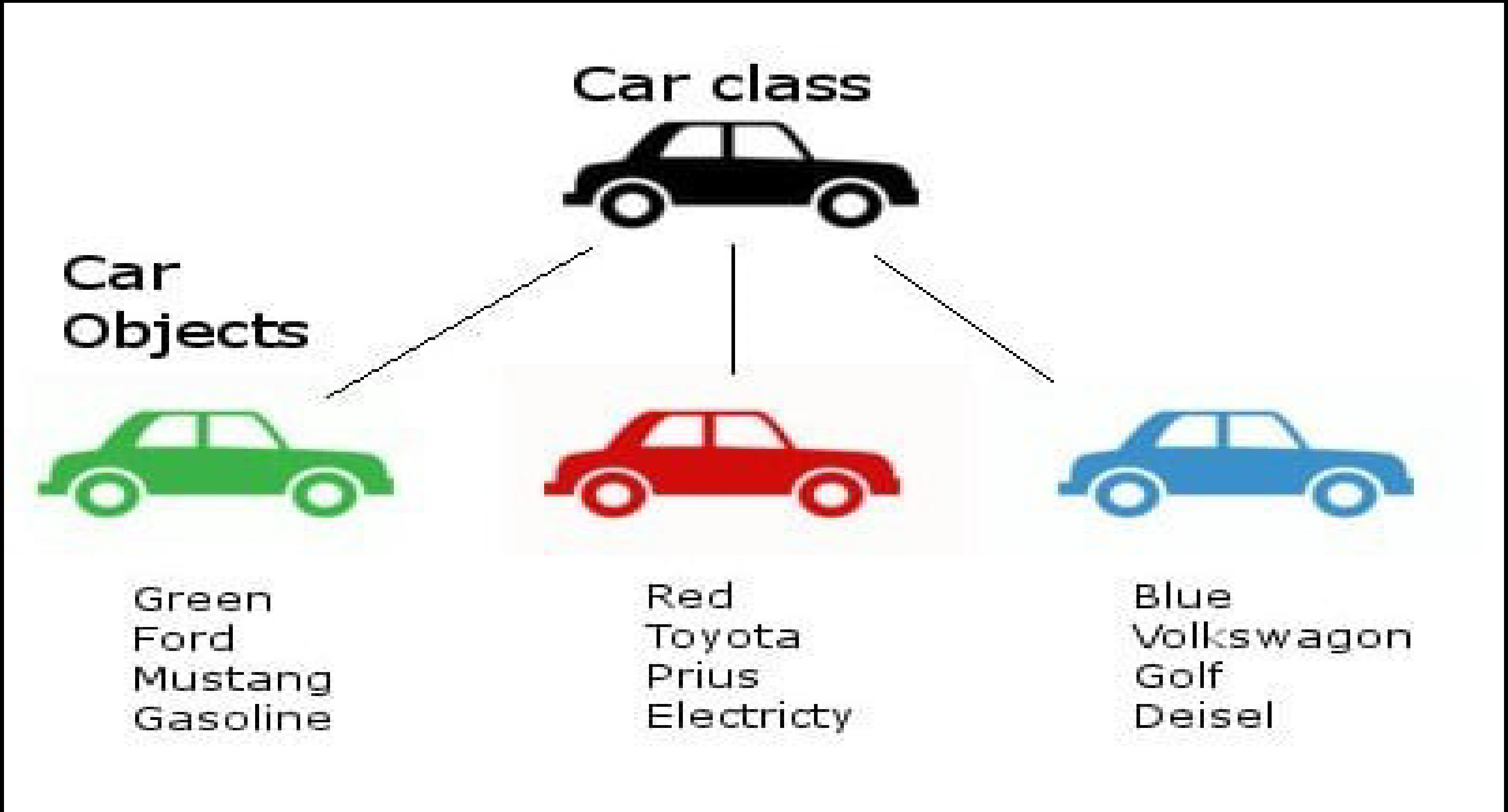


Parts of a software system are generated based upon their description: classes.



- A class is an abstraction, it is an abstract data type.
- An object is an instance generated and placed in computer memory.

Class Example



Car class

Car Objects

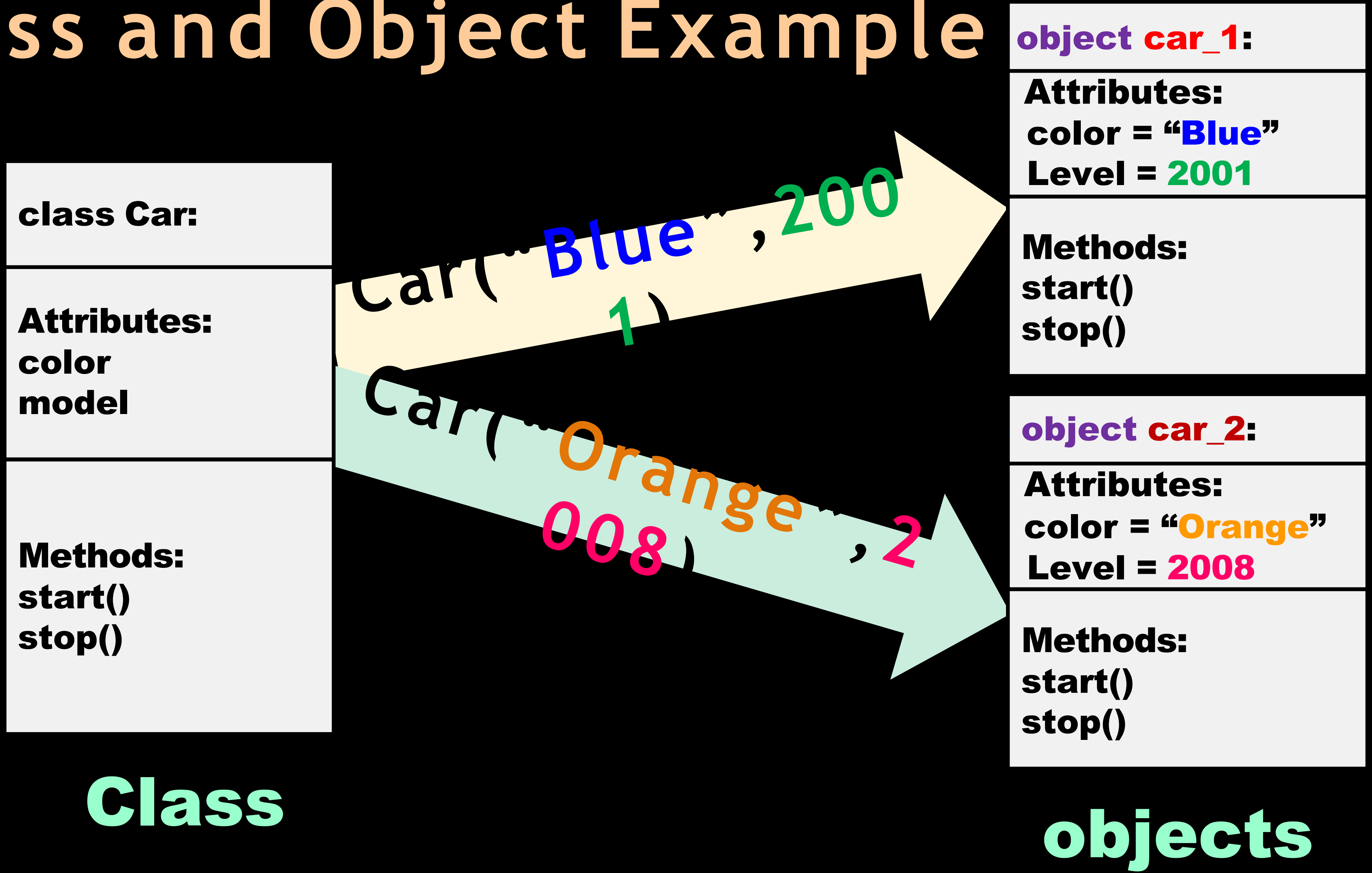
Green
Ford
Mustang
Gasoline

Red
Toyota
Prius
Electricty

Blue
Volkswagon
Golf
Deisel

https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Class and Object Example



class Car:

Attributes:
color
model

Methods:
start()
stop()

Car("Blue",2001)

Car("Orange",2008)

object car_1:

Attributes:
color = “Blue”
Level = 2001

Methods:
start()
stop()

object car_2:

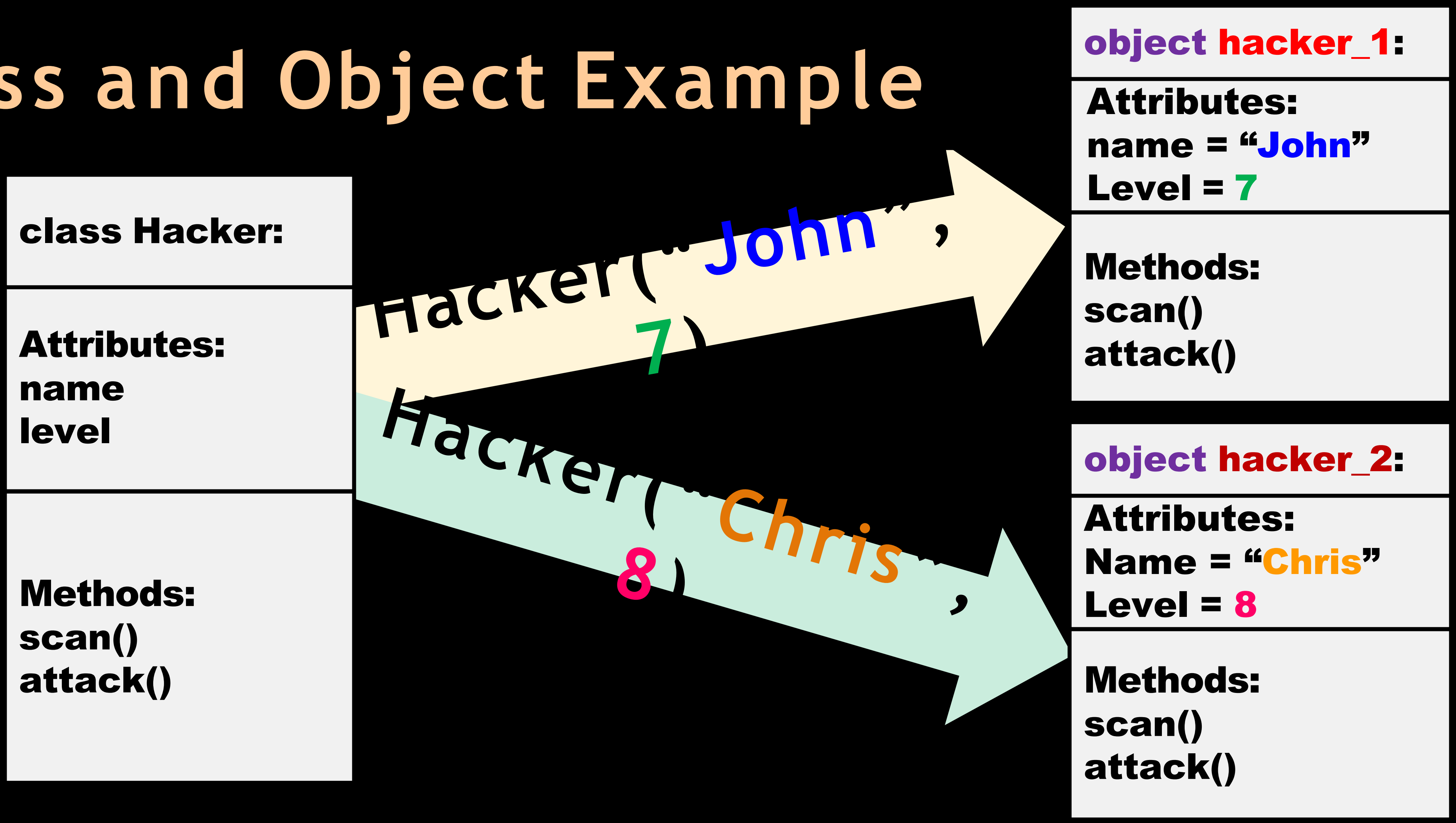
Attributes:
color = “Orange”
Level = 2008

Methods:
start()
stop()

Class

objects

Class and Object Example



class Hacker:

Attributes:
name
level

Methods:
scan()
attack()

Hacker(“John”, 7)

Hacker(“Chris”, 8)

object hacker_1:

Attributes:
name = “John”
Level = 7

Methods:
scan()
attack()

object hacker_2:

Attributes:
Name = “Chris”
Level = 8

Methods:
scan()
attack()

Class

objects

What is object?

An object is an instance of a class.

Source: https://www.w3schools.com/cpp/cpp_oop.asp

How to create C++ Class?

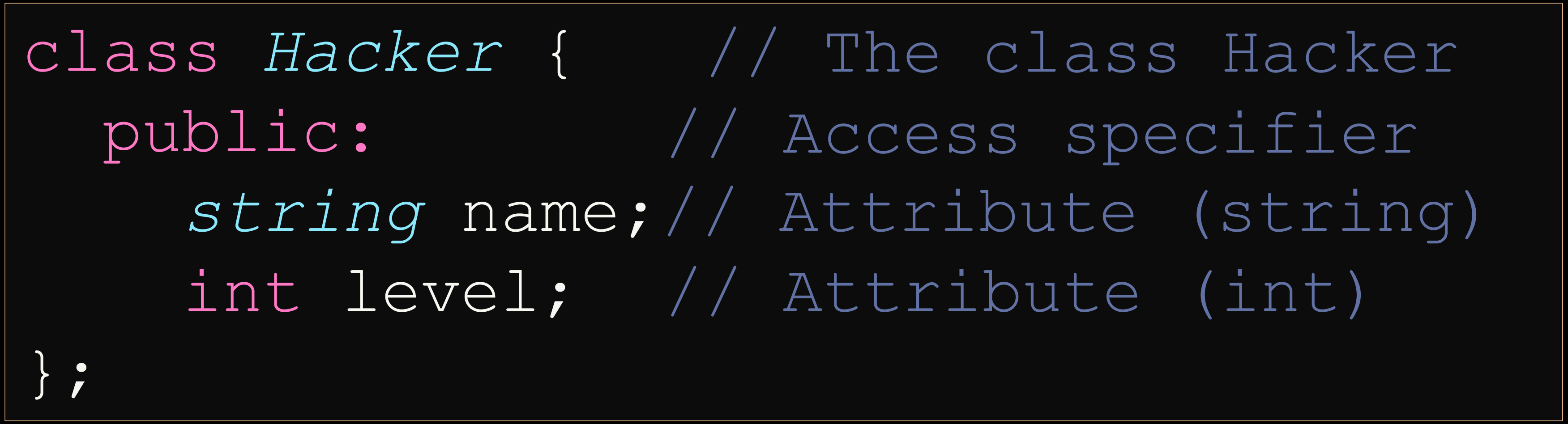
To create a C++ Class

- Syntax:

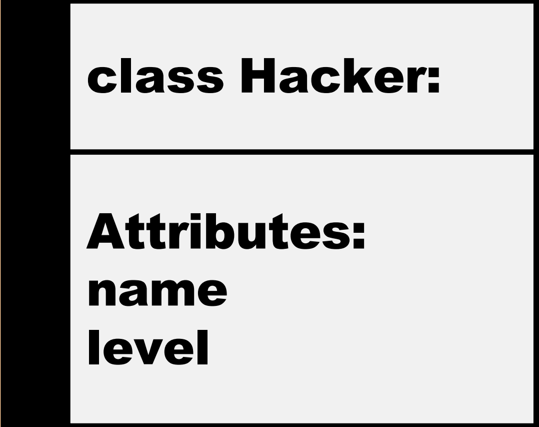
```
class Class_name{   // The class name
  public:          // Access specifier
    int variable; // Attributes
};                   // Semicolon
```

How to create C++ Class?

To create a Hacker Class with attributes



```
class Hacker {     // The class Hacker
  public:          // Access specifier
    string name;// Attribute (string)
    int level;     // Attribute (int)
};
```



class Hacker:

Attributes:
name
level

Instantiate C++ object

To create C++ object

- Syntax: Class_name object;

```
Hacker hacker_1; // Create an Object of Hacker
```

Instantiate C++ object

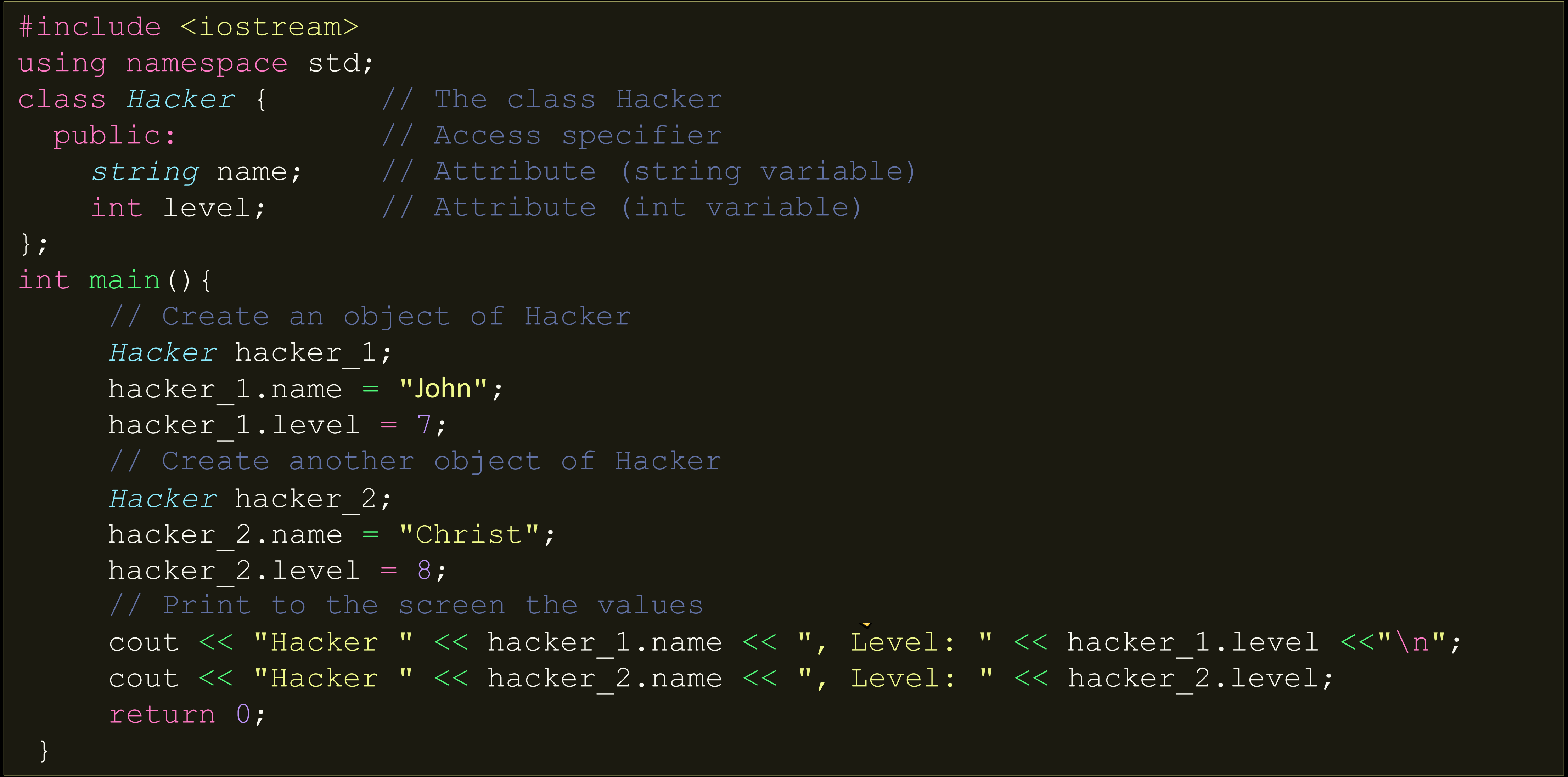
Set the values and process data

object hacker_1:

Attributes:
name = “John”
level = 7

```
#include <iostream>
using namespace std;

class Hacker {          // The class Hacker
  public:               // Access specifier
    string name;        // Attribute (string variab
le)
    int level;          // Attribute (int variable)
};

int main(){
     Hacker hacker_1; // Create an Object of Ha
cker
     // Access the attributes and set values
    hacker_1.name = "John";
     hacker_1.level = 7;
     // Print to the screen the values
     cout << hacker_1.name << "\n";
     cout << hacker_1.level;
     return 0;
 }
```

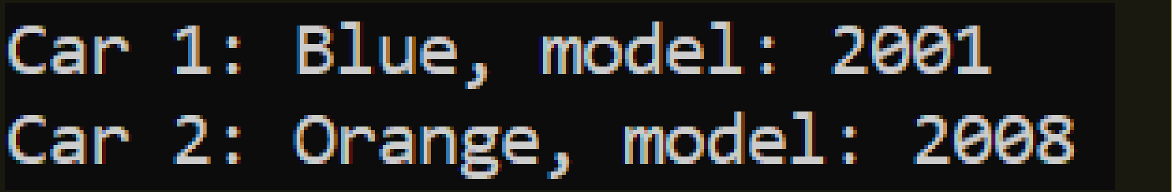
Instantiate multiple C++ objects



```
#include <iostream>
using namespace std;
class Hacker {       // The class Hacker
  public:            // Access specifier
    string name;     // Attribute (string variable)
    int level;       // Attribute (int variable)
};
int main(){
     // Create an object of Hacker
     Hacker hacker_1;
     hacker_1.name = "John";
     hacker_1.level = 7;
     // Create another object of Hacker
     Hacker hacker_2;
     hacker_2.name = "Christ";
     hacker_2.level = 8;
     // Print to the screen the values
     cout << "Hacker " << hacker_1.name << ", Level: " << hacker_1.level <<"\n";
     cout << "Hacker " << hacker_2.name << ", Level: " << hacker_2.level;
     return 0;
 }
```

Instantiate multiple C++ objects

```cpp
#include <iostream>
using namespace std;
class Car {           // The class Car
  public:             // Access specifier
    string color;     // Attribute color
    int model;        // Attribute model
};
 int main(){
     // Create an object of Car
     Car car_1;
     car_1.color = "Blue";
     car_1.model = 2001;
     // Create another object of Car
     Car car_2;
     car_2.color = "Orange";
     car_2.model = 2008;
     // Print to the screen the values
     cout << "Car 1: " << car_1.color << ", model: " << car_1.model <<"\n";
     cout << "Car 2: " << car_2.color << ", model: " << car_2.model;
    return 0;
 }
```



```
Car 1: Blue, model: 2001
Car 2: Orange, model: 2008
```

How to create C++ Class?

To create a Hacker Class with **methods**

There are two ways to define functions (methods):

- Inside class (create function inside class)
- Outside class (declare inside and create outside)

Method - inside

```
#include <iostream>
using namespace std;

class Hacker {      // The class Hacker
  public:           // Access specifier
    void scan(){    // Create function/
method inside class
        cout << "I am scanning the target";
    }
};

 int main(){
    Hacker hacker_1; // Create an Object of Hacker
    hacker_1.scan(); // Call the method by "." operato
r
    return 0;
 }
```

Method - outside

```
#include <iostream>
using namespace std;

class Hacker {          // The class Hacker
  public:               // Access specifier
    void scan();        // Declare the function/method
};

// Create function/method outside class
void Hacker::scan(){
        cout << "I am scanning the target";
    }

int main(){
    Hacker hacker_1; // Create an Object of Hacker
    hacker_1.scan(); // Call the method by "." operator
    return 0;
}
```

Remember double colons

Why we define - outside?

3. Encapsulation

Struct and Class

Syntactic difference between struct and class

```
struct sStudent {
    string name;
    int id;
};
```

&

```
class cStudent {
    string name;
    int id;
};
```

Struct and Class

Create object for each type

```
sStudent studentS;
```

&

```
cStudent studentC;
```

Set the data

```
studentS.id = 777;
```

&

```
studentC.id = 888;
```

What happen?

Access Specifier

For modifying the feature of attributes and methods, we have 3 different type of access specifiers:

- Public can be directly accessed from outside the object.
- Private can only be directly accessed by internal methods.
- Protected can be directly accessed by objects of subclasses (inherited class), but not by arbitrary external objects.

We will look at protected later, but private and public allow us to capture the idea of encapsulation.

Access Specifier

What happen?

```
#include <iostream>
using namespace std;
class Hacker {          // The class Hacker
  public:               // Access specifier
    string name;        // Attribute (string variable)
  private:
    int level;          // Attribute (int variable)
};
int main(){
     Hacker hacker_1; // Create an Object of Hacker
     // Access the attributes and set values
    hacker_1.name = "John";
     hacker_1.level = 7;
     // Print the values to screen
     return 0;
}
```

Encapsulation - What is it?

The meaning of Encapsulation, is to make sure that “sensitive” data is hidden from users. To achieve this, we must declare class variables/attributes as private (cannot be accessed from outside the class).

Source:
https://www.w3schools.com/cpp/cpp_encapsulation.asp

Why Encapsulation

- It is considered good practice to declare your class attributes as private (as often as you can).
- Ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

How to do Setter and Getter?

If we want others to read or modify the value of a private member, we can provide public `get` and `set` methods.

Encapsulation - Setter and Getter

```
#include <iostream>
using namespace std;
class Hacker {
    private:
        int level;
    public:
        void setLevel(int l){
            level = l;
        }
        int getLevel() const{
            return level;
        }
};
int main(){
    Hacker hacker_1;
    hacker_1.setLevel(7);
    cout << hacker_1.getLevel(
);
    return 0;
}
```

Explain the code:

- Level attribute is private, we cannot access directly
- Public setLevel() take variable l and assign to level attribute.
- Public getLevel() return level value.

So, in the main(), we create object hacker_1, using setLevel to pass 7 as a value. Level attribute will be 7. through function getLevel to return the level attribute is 7.

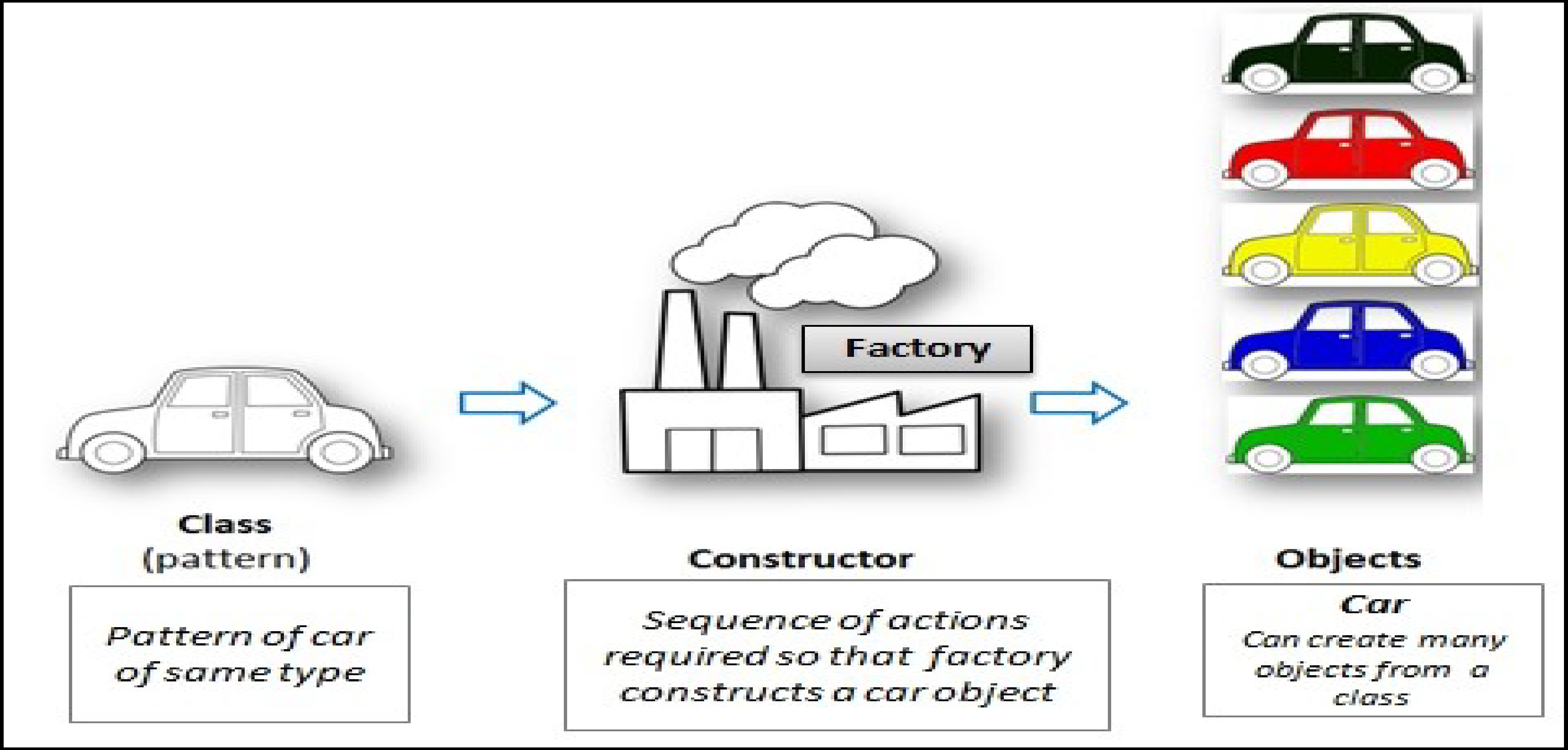
4. Object Management

Constructor and Destructor

Constructors - What is it?

A constructor in C++ is a special method/function/procedure that is automatically called when an object of a class is created.

Constructor - Idea



Class (pattern)

Pattern of car of same type

Constructor

Sequence of actions required so that factory constructs a car object

Objects

Car

Can create many objects from a class

https://www.listendata.com/2019/08/python-object-oriented-programming.html

Constructors - How to use it

To create a constructor, use the same name as the class, followed by parentheses ():

In this case, it is default constructor without arguments.

```
#include <iostream>
using namespace std;
class Hacker {
    public:
        Hacker ()
{  // Constructor
     cout << "Hi CSCI291";
   }
};
int main(){
    Hacker hacker_1;
    return 0;
}
```

Constructors - Parameters

Constructors can also take parameters like regular functions. We use this technique to set initial values for attributes.

It is called non-default constructor.

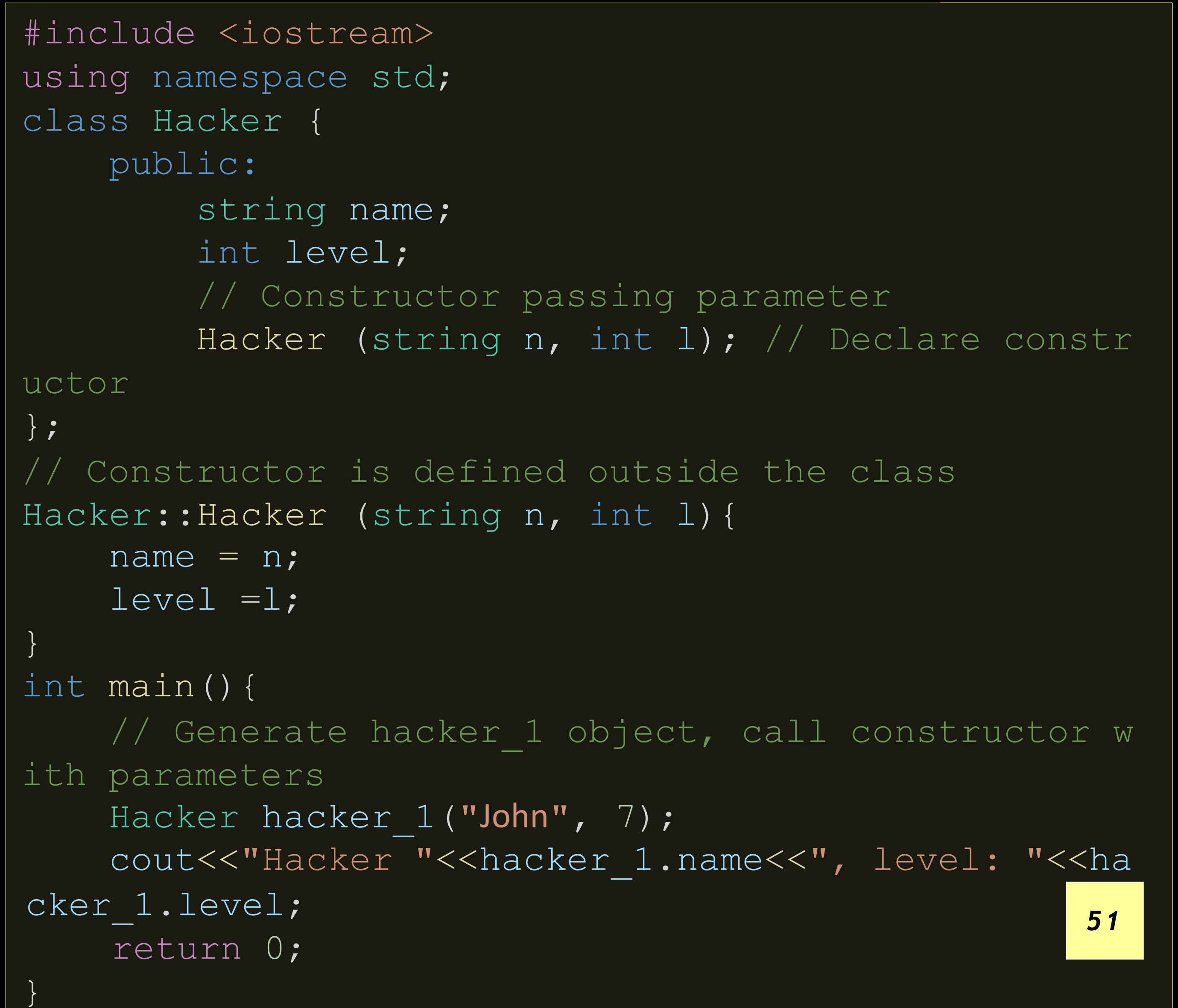
Constructors - Parameters

We pass 2 parameter name and level from creating object

```
#include <iostream>
using namespace std;
class Hacker {
    public:
        string name;
        int level;
        // Constructor passing parameter
        Hacker (string n, int l){
            name = n;
            level =l;
        }
};
int main(){
    // Generate hacker_1 object, call construct
or with parameters
    Hacker hacker_1("John", 7);
    cout<<"Hacker "<<hacker_1.name<<", level: "
<<hacker_1.level;
    return 0;
}
```

Constructors - from Outside

It is similar to the regular function. It can be defined outside the Class



```cpp
#include <iostream>
using namespace std;
class Hacker {
    public:
        string name;
        int level;
        // Constructor passing parameter
        Hacker (string n, int l); // Declare constructor
};
// Constructor is defined outside the class
Hacker::Hacker (string n, int l){
    name = n;
    level =l;
}
int main(){
    // Generate hacker_1 object, call constructor with parameters
    Hacker hacker_1("John", 7);
    cout<<"Hacker "<<hacker_1.name<<", level: "<<hacker_1.level;
    return 0;
}
```

Multiple Constructors

We can have multiple constructors. Default and non. So, it gives us the options to use.

In this case, we have hacker_1 with default constructor, hacker_2 with parameters

```
class Hacker {
public:
    string name;
    int level;
    Hacker(){
        cout<<"Hi CSCI251\n";
    }
    Hacker(string n, int l){
        name=n;
        level=l;
    }
};

int main(){
    Hacker hacker_1;
    Hacker hacker_2("John", 7);
    cout<<"Hacker "<<hacker_2.name<<", level: "<<hacker_2.l
evel;
    return 0;
}
```

Destructors - What is it?

Destructors are called for an object whenever the object goes out of scope.

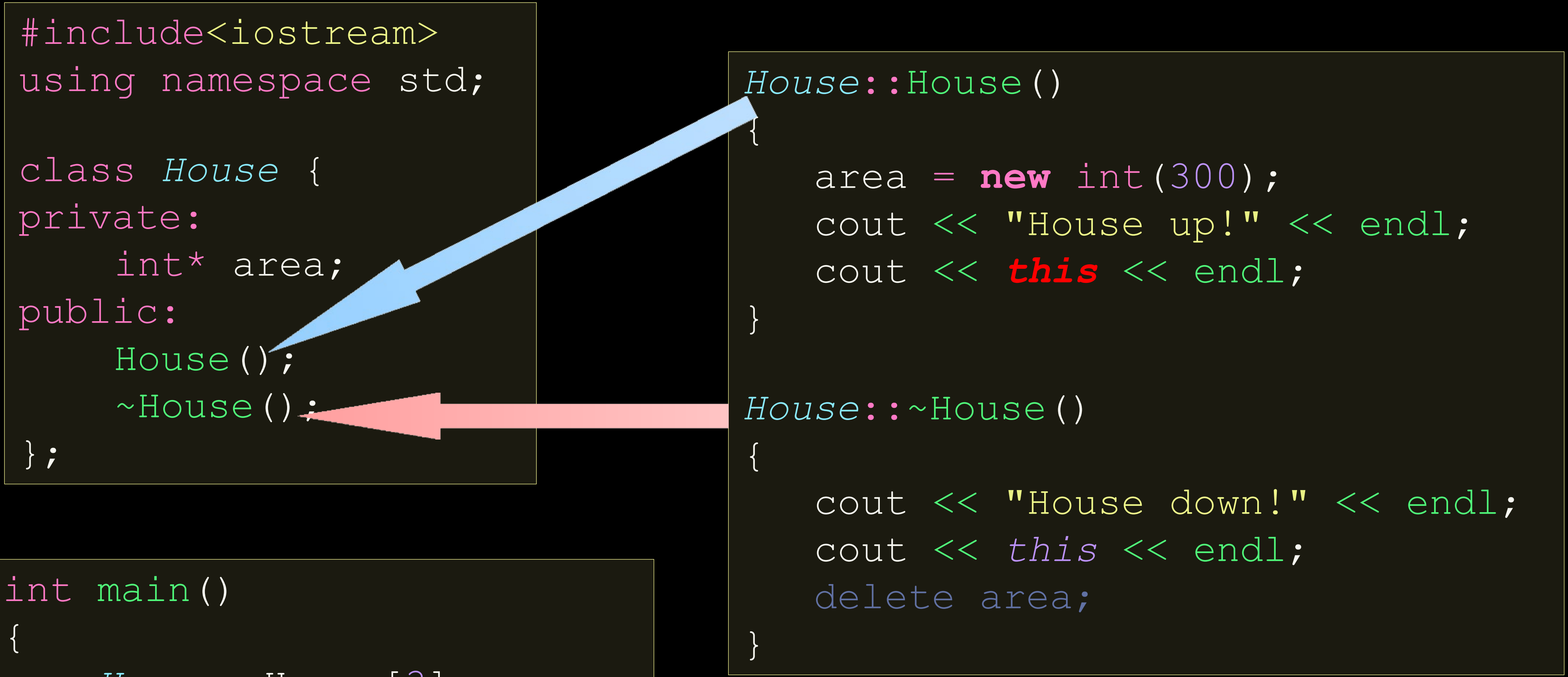
Why we use Destructors?

- Think of using the constructor to set up objects and the destructor to remove them nicely.
- Destructors should allow memory to be released, avoiding memory leaks.
- They could be used to write some information about the object to a file or to standard out.

How to use it?

- They have the name as the class but with a leading tilde (~).
- One destructor per class. Default set up by the compiler if none is specified.
- No parameters, no return type.

order of destruction

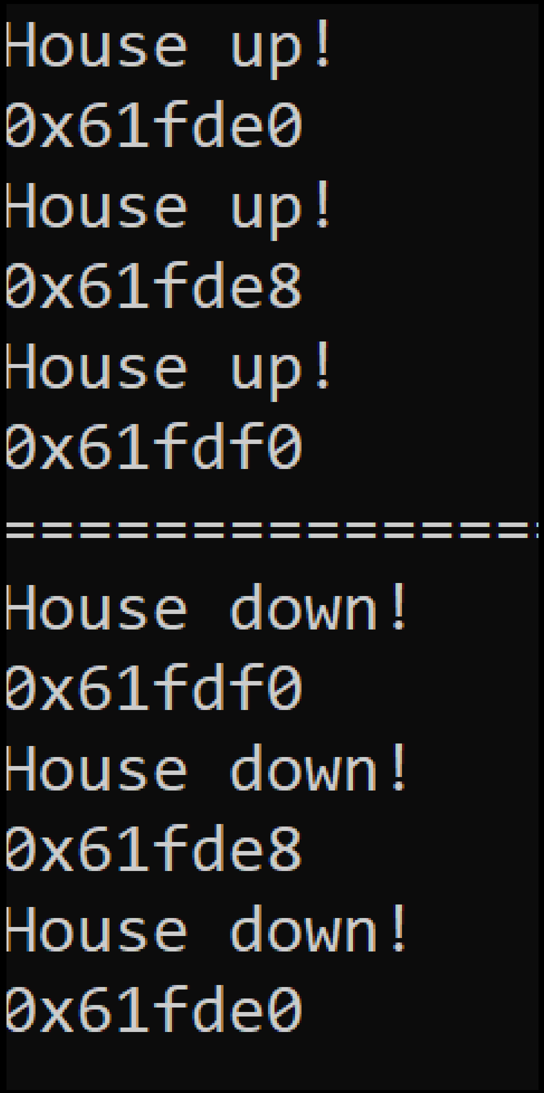


```
#include<iostream>
using namespace std;

class House {
private:
    int* area;
public:
    House();
    ~House();
};
```

```
House::House()
{
    area = new int(300);
    cout << "House up!" << endl;
    cout << this << endl;
}

House::~House()
{
    cout << "House down!" << endl;
    cout << this << endl;
    delete area;
}
```

```
int main()
{
    House aHouse[3];
    cout << “======" << en
dl;
}
```

order of destruction



```
House up!
0x61fde0
House up!
0x61fde8
House up!
0x61fdf0
================
House down!
0x61fdf0
House down!
0x61fde8
House down!
0x61fde0
```

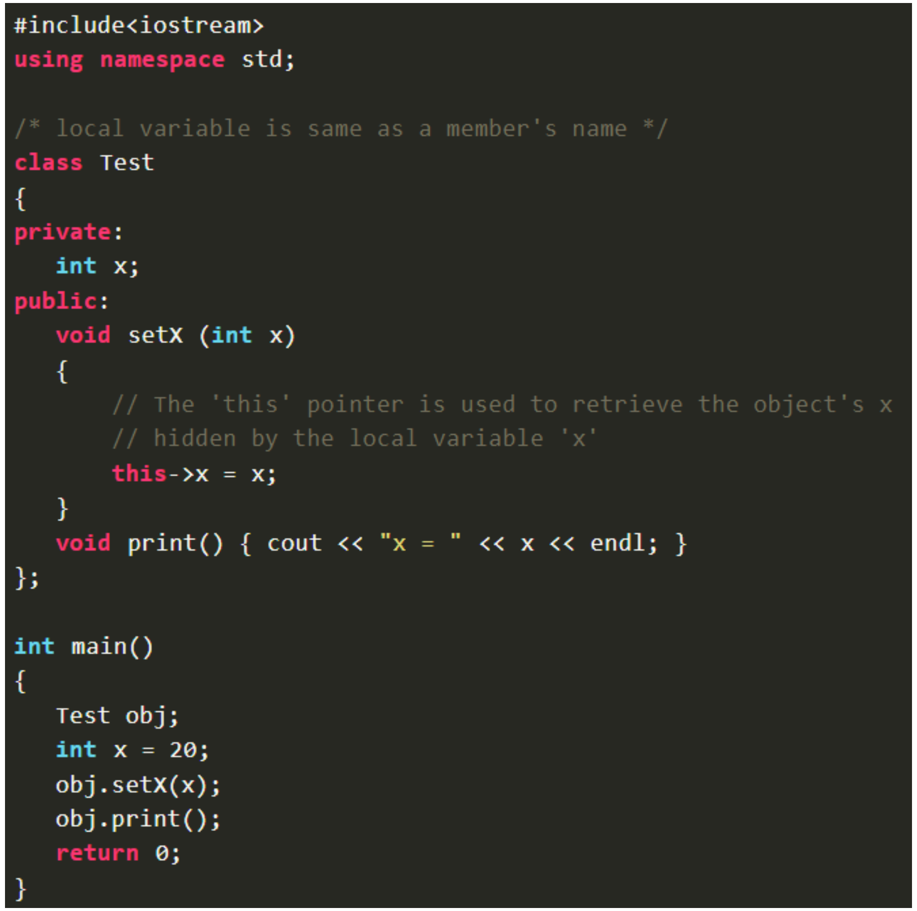
When constructors are created with 3 objects. We can have each constructor creation of 8 byte data (pointer).

Then when calling destructors, each will delete the memory in the stack following LIFO (Last In First Out)

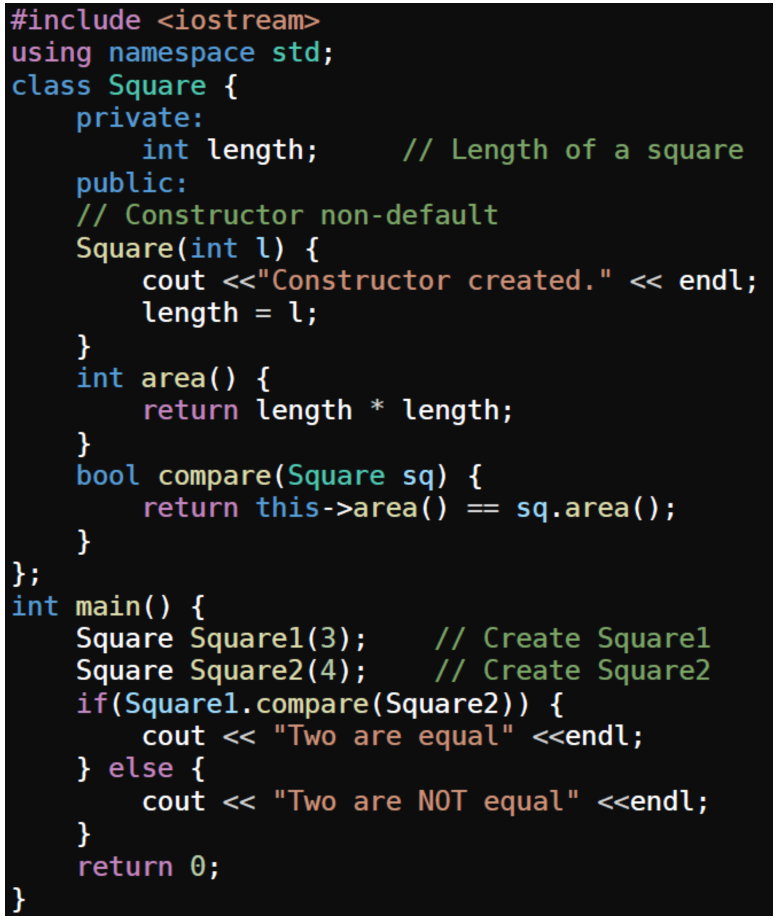
It means the last created is the first destroyed

‘this’ pointer

It represents the address of the current object instance on which the member function is being called. The 'this' pointer allows objects to refer to themselves and access their own member variables and member functions.



```
#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```



```
#include <iostream>
using namespace std;
class Square {
    private:
        int length;     // Length of a square
    public:
    // Constructor non-default
    Square(int l) {
        cout <<"Constructor created." << endl;
        length = l;
    }
    int area() {
        return length * length;
    }
    bool compare(Square sq) {
        return this->area() == sq.area();
    }
};
int main() {
    Square Square1(3);    // Create Square1
    Square Square2(4);    // Create Square2
    if(Square1.compare(Square2)) {
        cout << "Two are equal" <<endl;
    } else {
        cout << "Two are NOT equal" <<endl;
    }
    return 0;
}
```

Copy operations

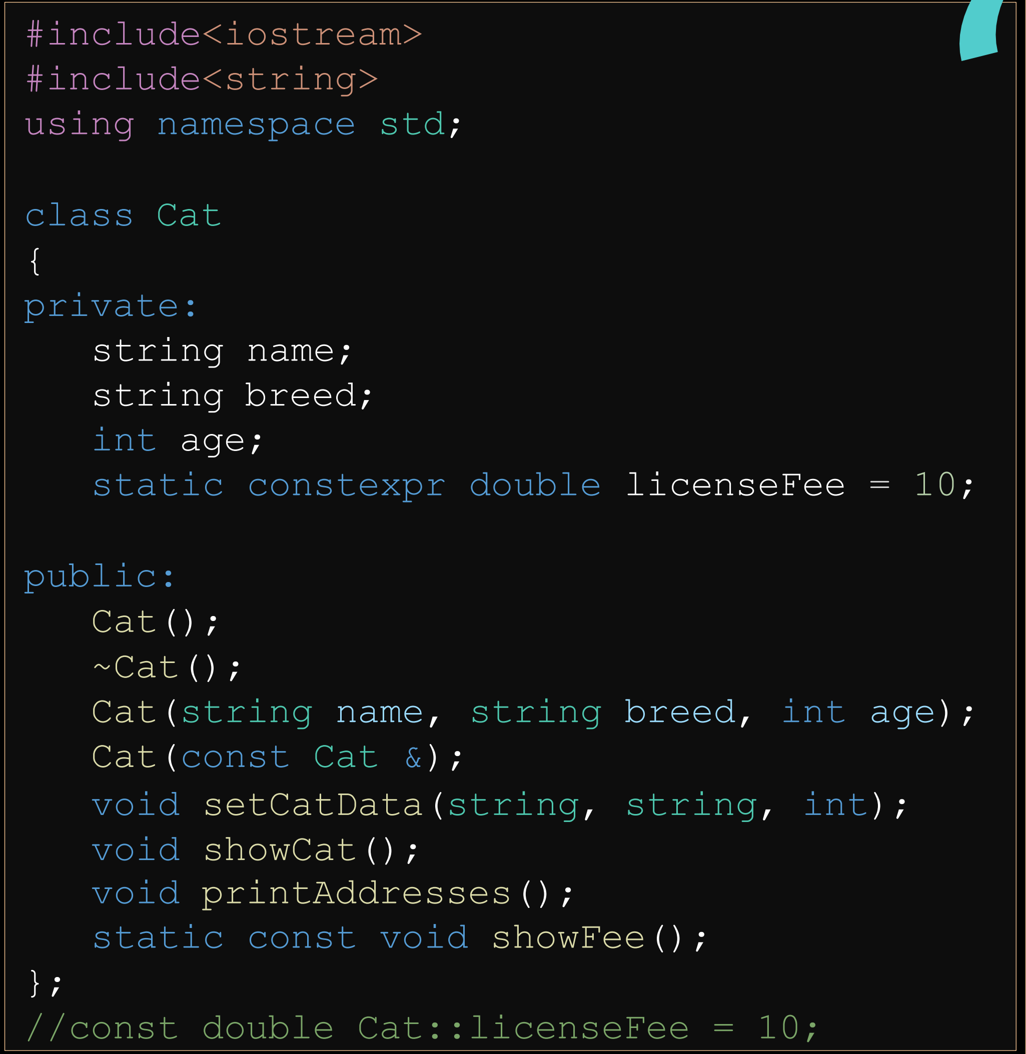
Copy Constructor

```
X (const X &cp);
```

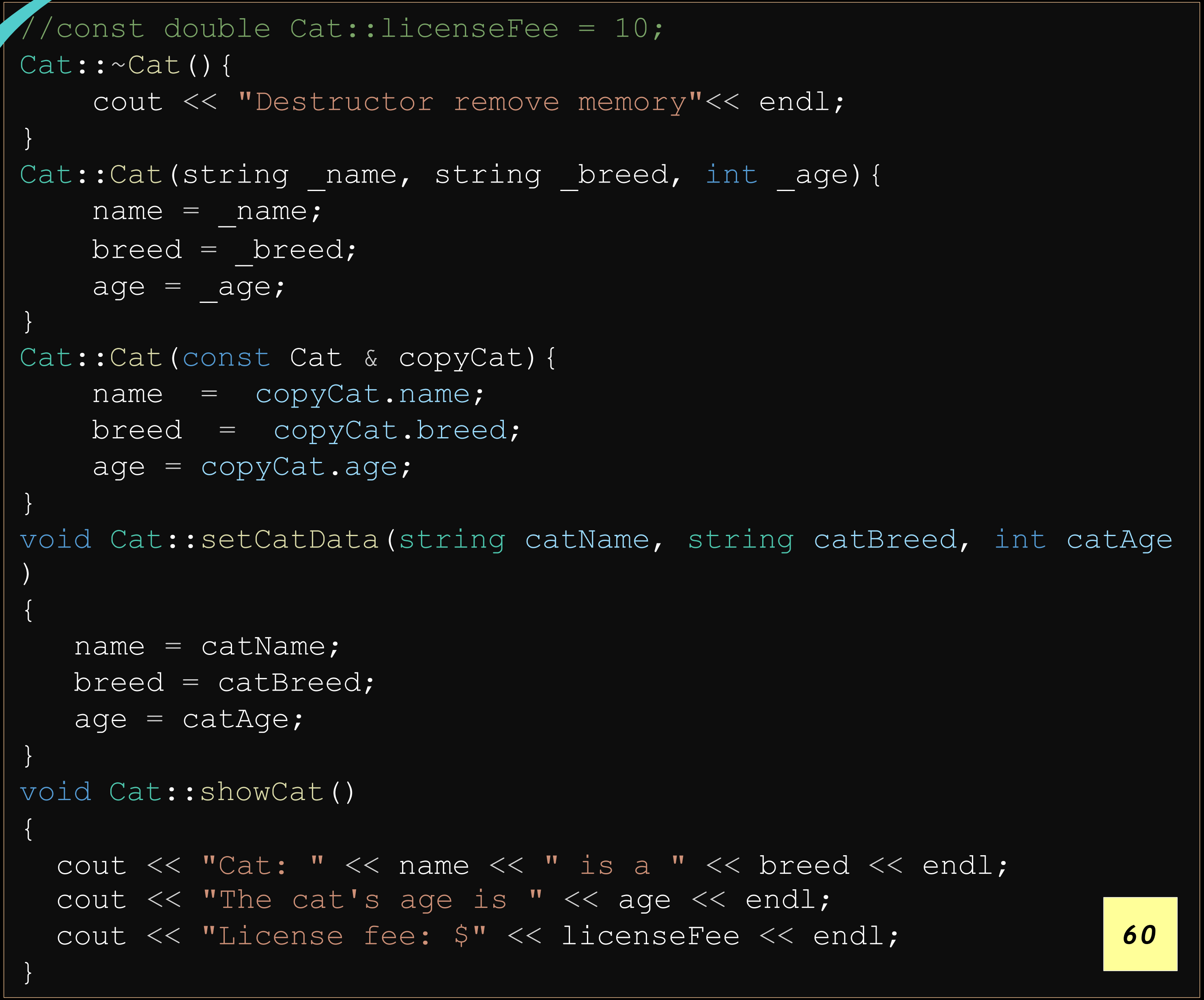
The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to

- Initialize one object from another of the same type.
- Pass the object to be copied as an argument to a function.

Code - Demo

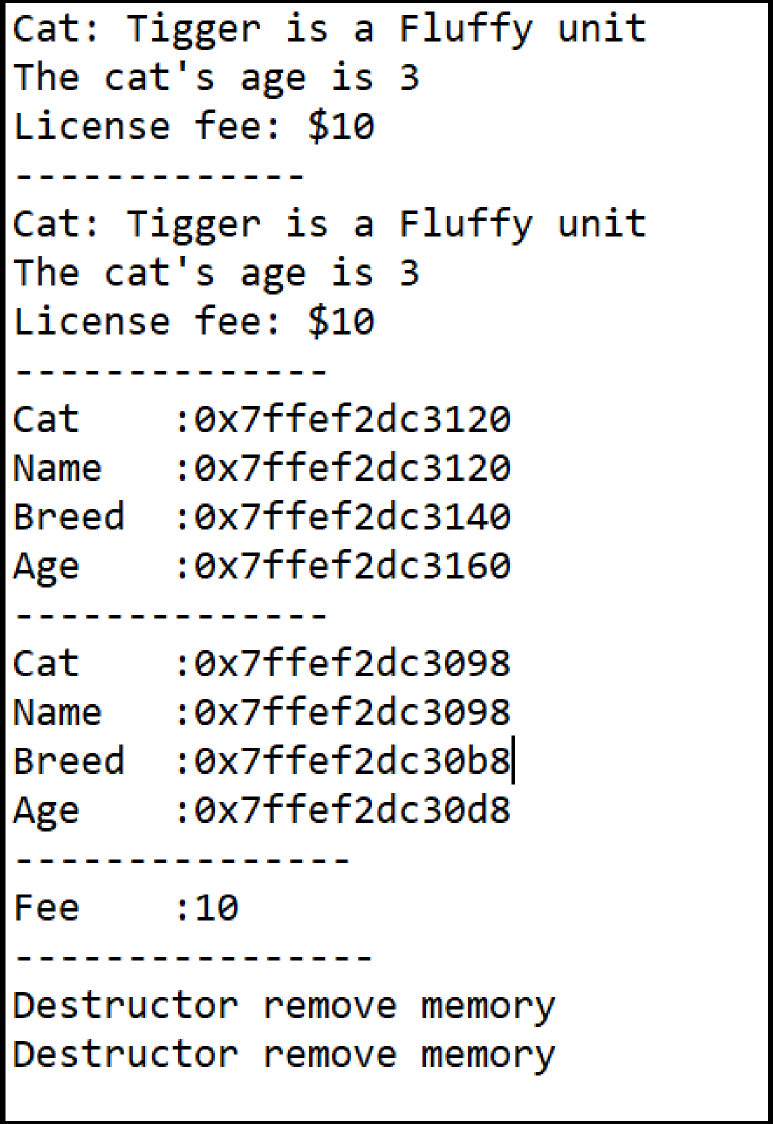


```cpp
#include<iostream>
#include<string>
using namespace std;

class Cat
{
private:
   string name;
   string breed;
   int age;
   static constexpr double licenseFee = 10;

public:
   Cat();
   ~Cat();
   Cat(string name, string breed, int age);
   Cat(const Cat &);
   void setCatData(string, string, int);
   void showCat();
   void printAddresses();
   static const void showFee();
};
//const double Cat::licenseFee = 10;
```

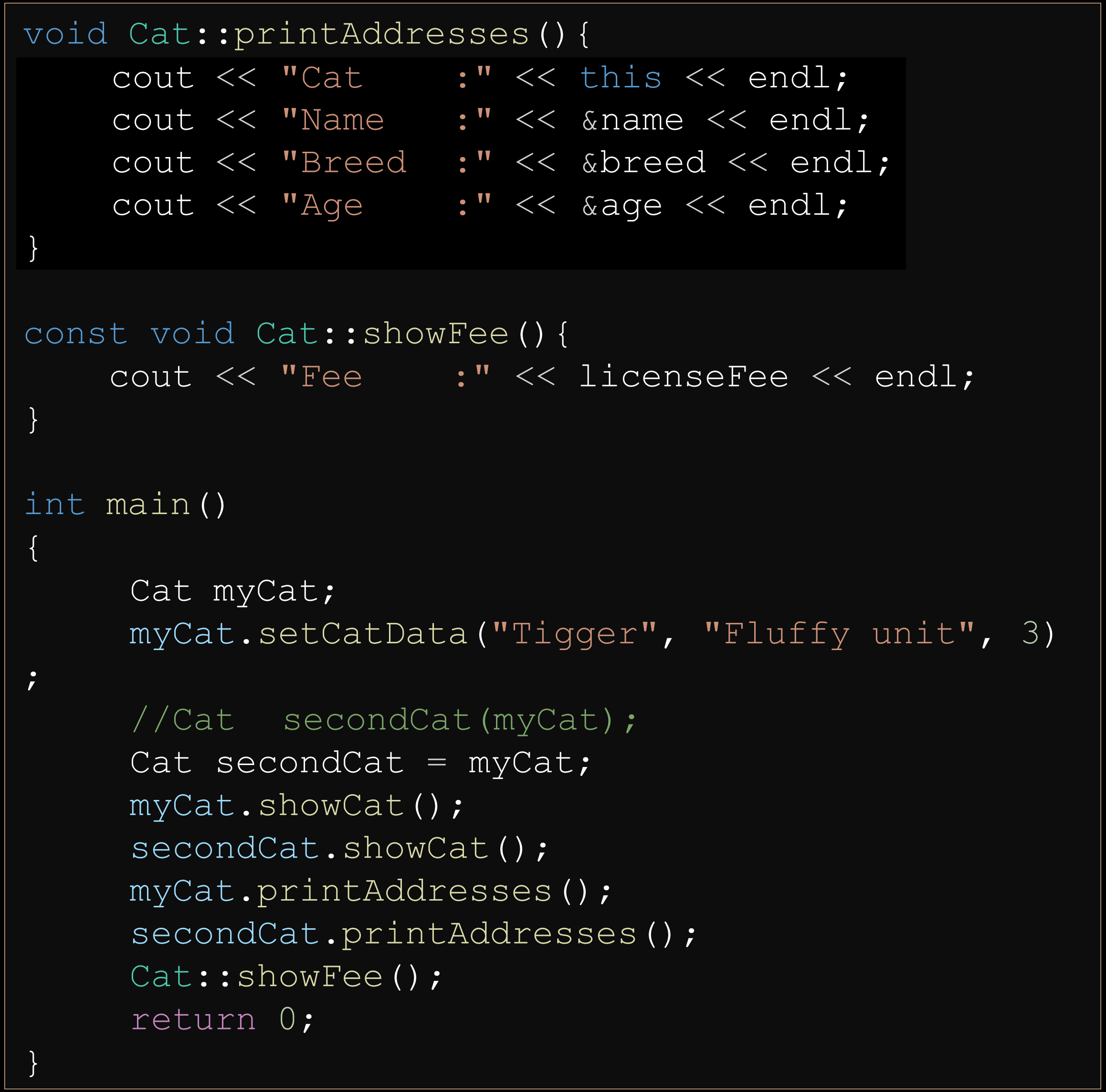


```cpp
//const double Cat::licenseFee = 10;
Cat::~Cat(){
    cout << "Destructor remove memory"<< endl;
}
Cat::Cat(string _name, string _breed, int _age){
    name = _name;
    breed = _breed;
    age = _age;
}
Cat::Cat(const Cat & copyCat){
    name  =  copyCat.name;
    breed  =  copyCat.breed;
    age = copyCat.age;
}
void Cat::setCatData(string catName, string catBreed, int catAge
)
{
   name = catName;
   breed = catBreed;
   age = catAge;
}
void Cat::showCat()
{
  cout << "Cat: " << name << " is a " << breed << endl;
  cout << "The cat's age is " << age << endl;
  cout << "License fee: $" << licenseFee << endl;
}
```

Code - Demo



```
Cat: Tigger is a Fluffy unit
The cat's age is 3
License fee: $10
-------------
Cat: Tigger is a Fluffy unit
The cat's age is 3
License fee: $10
--------------
Cat     :0x7ffef2dc3120
Name    :0x7ffef2dc3120
Breed   :0x7ffef2dc3140
Age     :0x7ffef2dc3160
--------------
Cat     :0x7ffef2dc3098
Name    :0x7ffef2dc3098
Breed   :0x7ffef2dc30b8
Age     :0x7ffef2dc30d8
---------------
Fee     :10
----------------
Destructor remove memory
Destructor remove memory
```



```
void Cat::printAddresses(){
    cout << "Cat    :" << this << endl;
    cout << "Name   :" << &name << endl;
    cout << "Breed  :" << &breed << endl;
    cout << "Age    :" << &age << endl;
}

const void Cat::showFee(){
    cout << "Fee    :" << licenseFee << endl;
}

int main()
{
     Cat myCat;
     myCat.setCatData("Tigger", "Fluffy unit", 3)
;
     //Cat  secondCat(myCat);
     Cat secondCat = myCat;
     myCat.showCat();
     secondCat.showCat();
     myCat.printAddresses();
     secondCat.printAddresses();
     Cat::showFee();
     return 0;
}
```

Copy Assignment

```
X& operator=(const X &cp);
```

The copy assignment which assigns values of one object to another existing object.

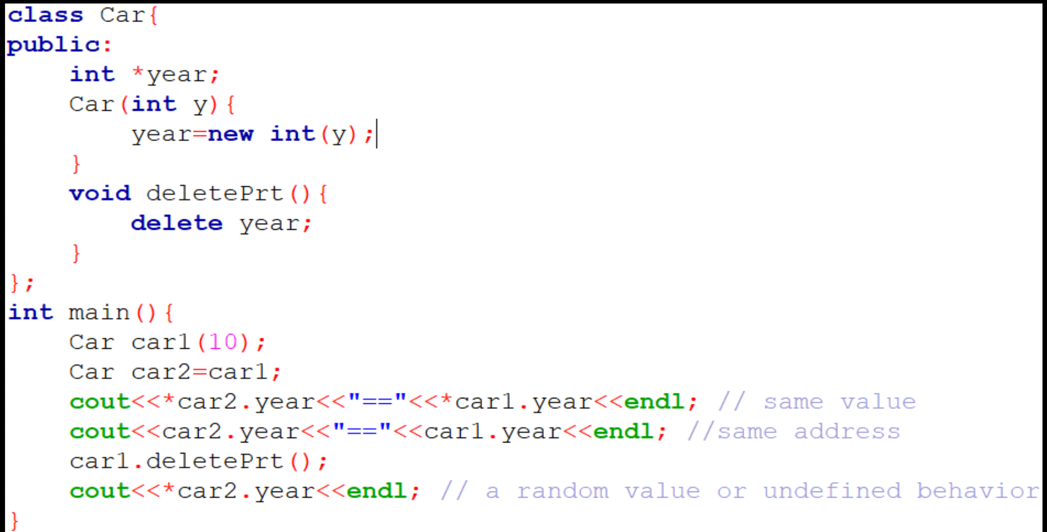
```
Cat& operator=(const Cat &cp){
        return *this;}

Cat cat2;
cat2=cat1;
```

Practice 1

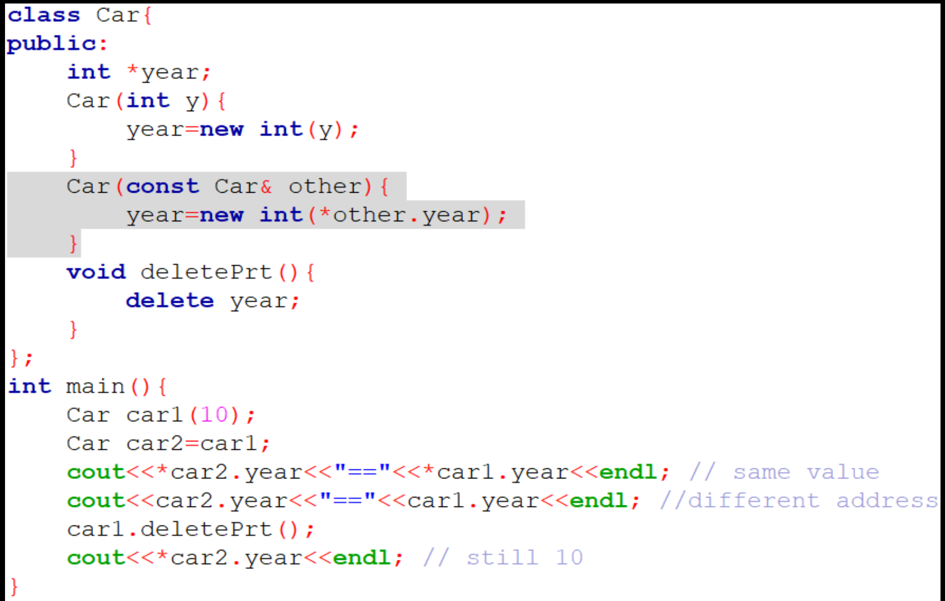
Shallow and deep copy

Shallow copy: duplicate memory address, point to the same memory location as the original variable.

Deep copy: copy data into a new address.



```
class Car{
public:
    int *year;
    Car(int y){
        year=new int(y);
    }
    void deletePrt(){
        delete year;
    }
};
int main(){
    Car car1(10);
    Car car2=car1;
    cout<<*car2.year<<"=="<<*car1.year<<endl; // same value
    cout<<car2.year<<"=="<<car1.year<<endl; //same address
    car1.deletePrt();
    cout<<*car2.year<<endl; // a random value or undefined behavior
}
```



```
class Car{
public:
    int *year;
    Car(int y){
        year=new int(y);
    }
    Car(const Car& other){
        year=new int(*other.year);
    }
    void deletePrt(){
        delete year;
    }
};
int main(){
    Car car1(10);
    Car car2=car1;
    cout<<*car2.year<<"=="<<*car1.year<<endl; // same value
    cout<<car2.year<<"=="<<car1.year<<endl; //different address
    car1.deletePrt();
    cout<<*car2.year<<endl; // still 10
}
```

Shallow copy: dangling pointer

Deep copy

Function overloading

Polymorphisms in C++

Compile-time polymorphism (early binding): determine function call during the compile-time.

- Function overloading
- Operator overloading

Run-time polymorphism (late binding): determine function call at the run-time. It usually happens with inheritance (to be introduced later).

- Function overriding
- Virtual function

Overloading

- Overloading is where we have multiple functions with the same name but different parameter lists. The function’s "signature“ decides which function is called:
  - Function name & parameter list
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:

```
float average(float n1, float n2)
{
    return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
float average(float n1, float n2, float n3)
{
    return ((n1 + n2 + n3) / 3.0);
}
```

- Same name, two functions

Overloaded Average() Cont’d

- Which function gets called?
- Depends on function call itself:
  - avg = average(5.2, 6.7);
    - Calls "two-parameter average()"
  - avg = average(6.5, 8.5, 4.2);
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Overloading Pitfall

- Only overload "same-task" functions
  - Same tasks should be always performed, in all overloads.
  - Otherwise, confusing results

Automatic Type Conversion and Overloading

- Numeric formal parameters typically **made "double" type**
- **Allows for "any" numeric type**
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
- Avoids overloading for different numeric types

Automatic Type Conversion and Overloading Example

- double mpg(double miles, double gallons)

```
{
        return (miles/gallons);
}
```

- Example function calls:
  - mpgComputed = mpg(5, 20);
    - Converts 5 & 20 to doubles, then passes
  - mpgComputed = mpg(5.8, 20.2);
    - No conversion necessary
  - mpgComputed = mpg(5, 2.4);
    - Converts 5 to 5.0, then passes values to function

Type promotion

Automatic Type Conversion and Overloading Example

- void f(int n, int m)

```
{
    cout<<n<<“,”<<m<<endl;
}
```

- Example function calls:
  - f(98, 99);
    - No conversion, output: 98,99
  - F(5.3, 4)
    - 5.3 is demoted, output: 5,4
  - mpgComputed = mpg(5, 2.4);
    - 2.4 is demoted, output: 5,2

Type demotion

Practice 2