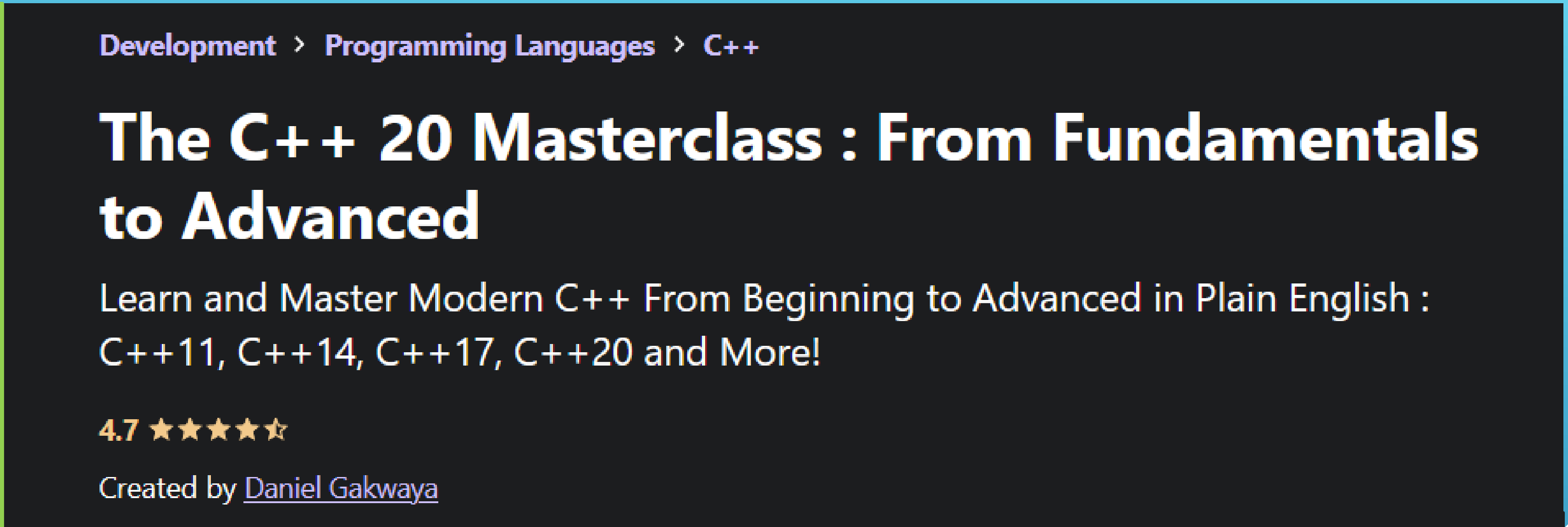
Slides



Development › Programming Languages › C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

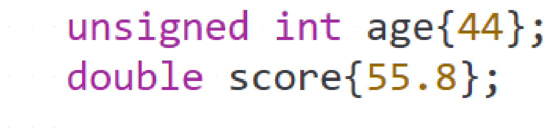
4.7 ★★★★⯪

Created by Daniel Gakwaya

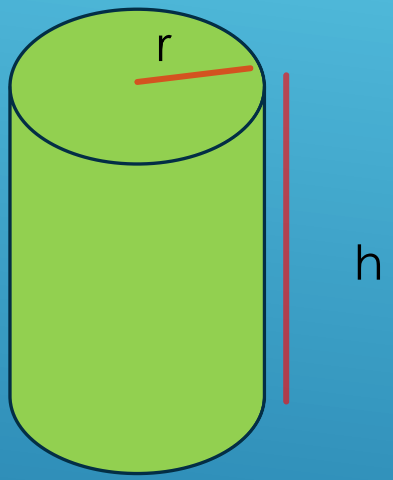
Section : Classes

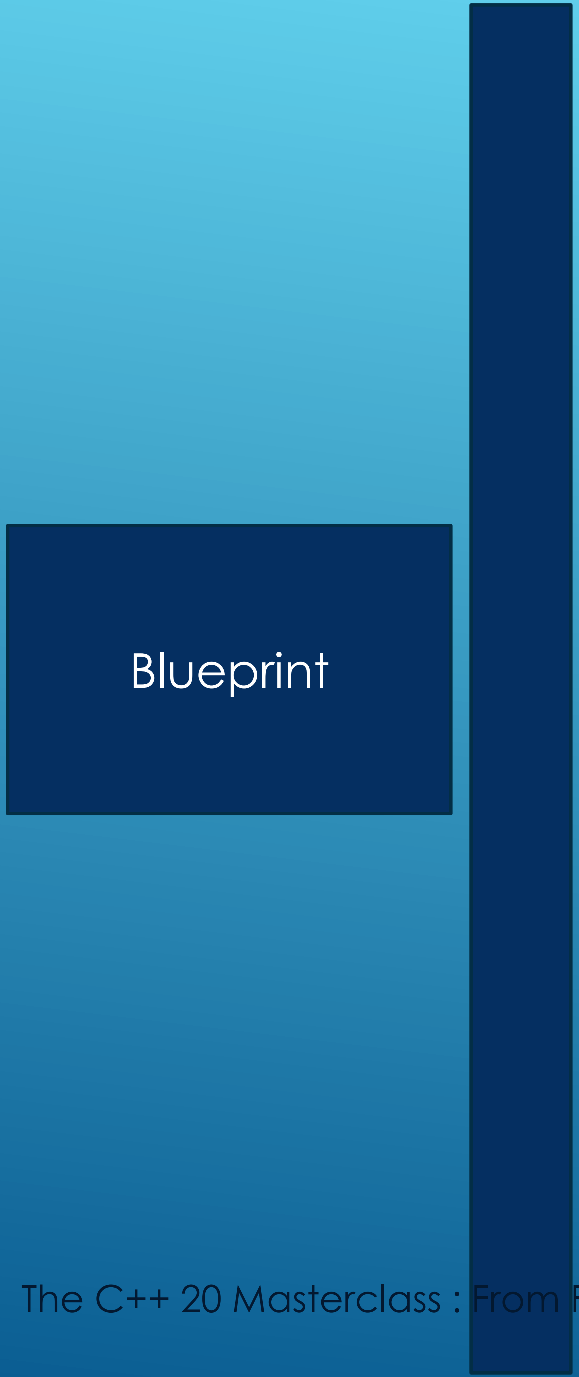
Slide intentionally left empty

Classes : Introduction

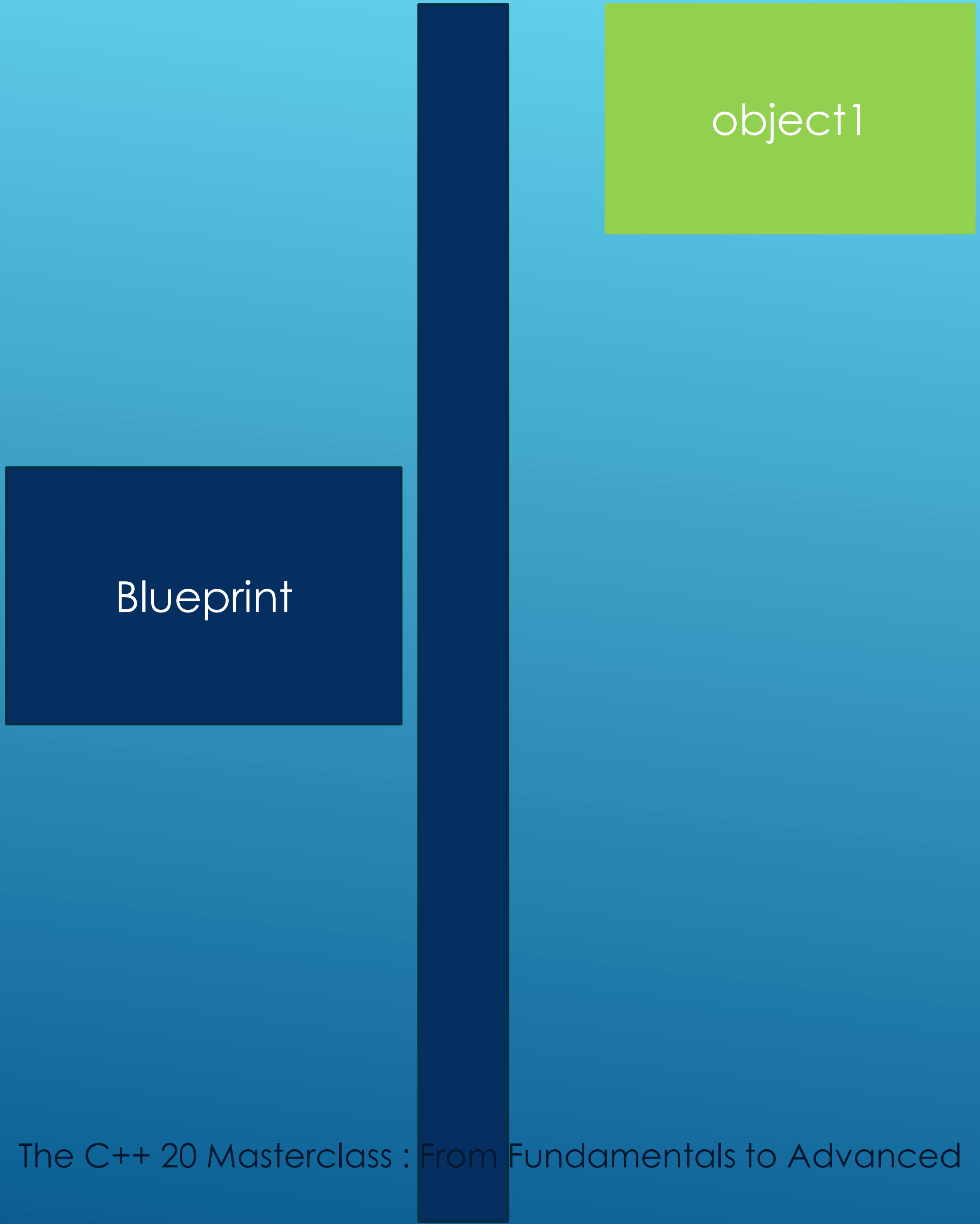


```cpp
unsigned int age{44};
double score{55.8};
```


r
h

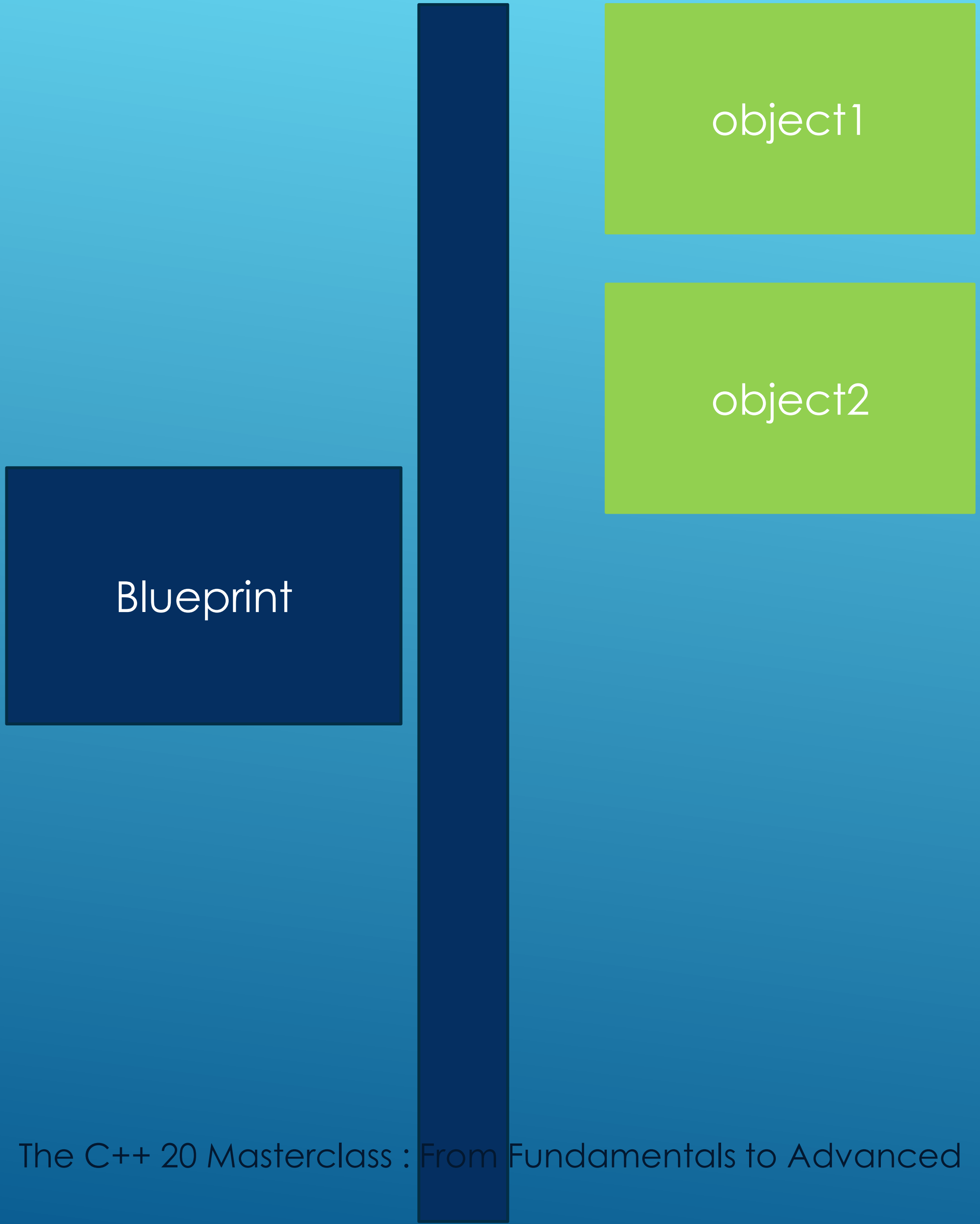


Blueprint



Blueprint

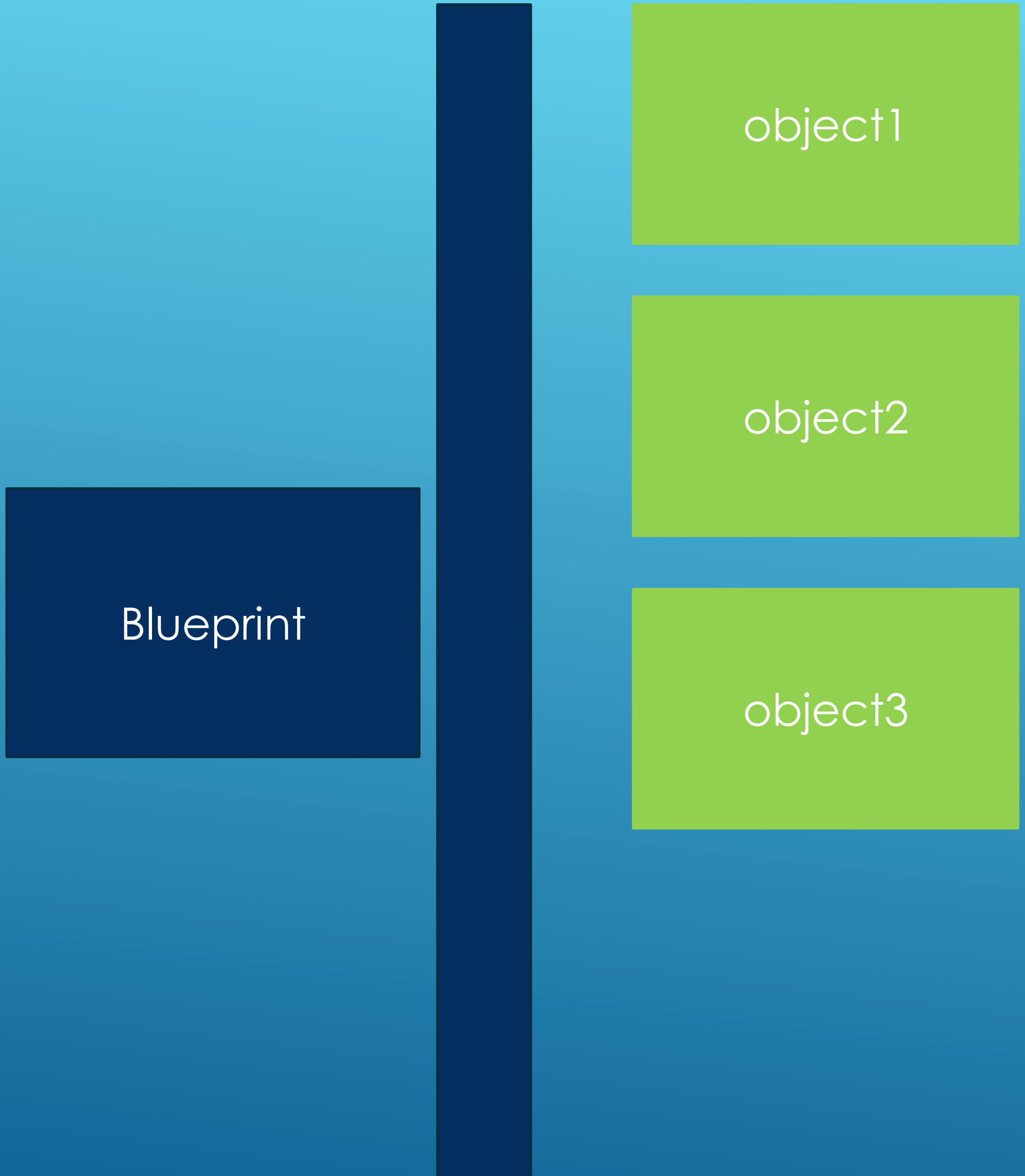
object1



Blueprint

object1

object2



Blueprint

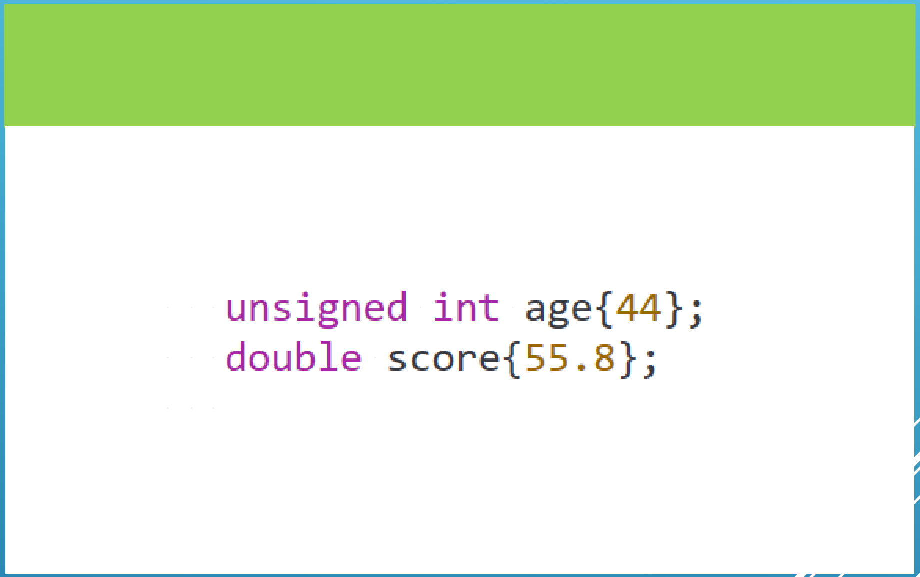
object1

object2

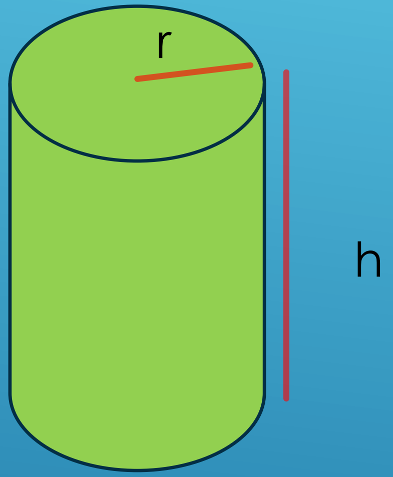
object3

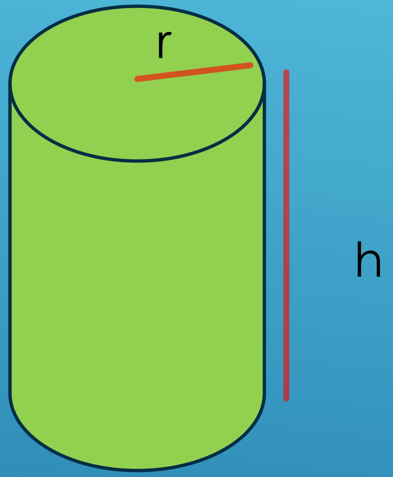
Slide intentionally left empty

Your first C++ class



```
unsigned int age{44};
double score{55.8};
```

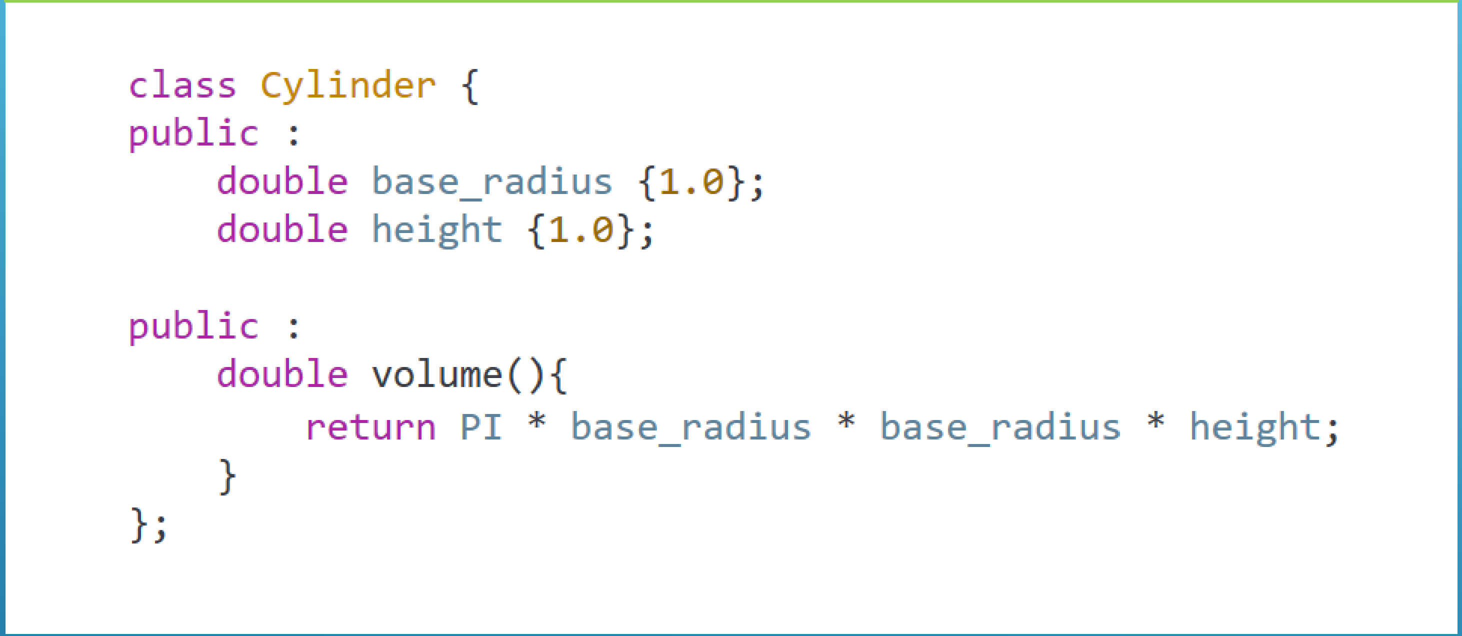

r
h



$$A = \pi\ r^2$$

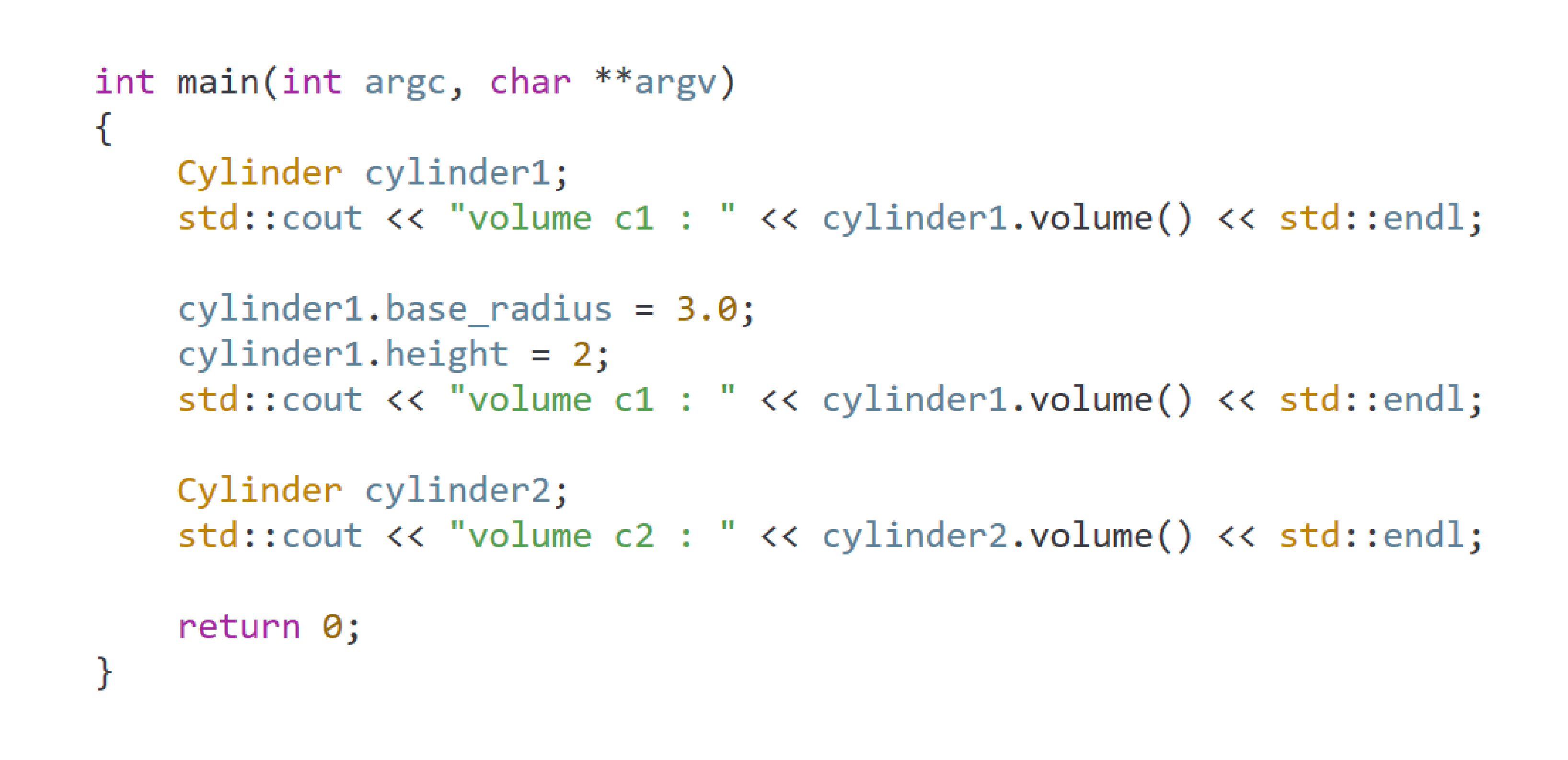
V = A x h

Class declaration : syntax



```
class Cylinder {
public :
    double base_radius {1.0};
    double height {1.0};

public :
    double volume(){
        return PI * base_radius * base_radius * height;
    }
};
```

Using class instances(objects)



```cpp
int main(int argc, char **argv)
{
    Cylinder cylinder1;
    std::cout << "volume c1 : " << cylinder1.volume() << std::endl;

    cylinder1.base_radius = 3.0;
    cylinder1.height = 2;
    std::cout << "volume c1 : " << cylinder1.volume() << std::endl;

    Cylinder cylinder2;
    std::cout << "volume c2 : " << cylinder2.volume() << std::endl;

    return 0;
}
```

- Class member variables can either be raw stack variables or pointers
- Members can’t be references
- Classes have functions (methods) that let them do things
- Class methods have access to the member variables, regardless of whether they are public or private
- Private members of classes (variables and functions) aren’t accessible from the outside of the class definition

Slide intentionally left empty

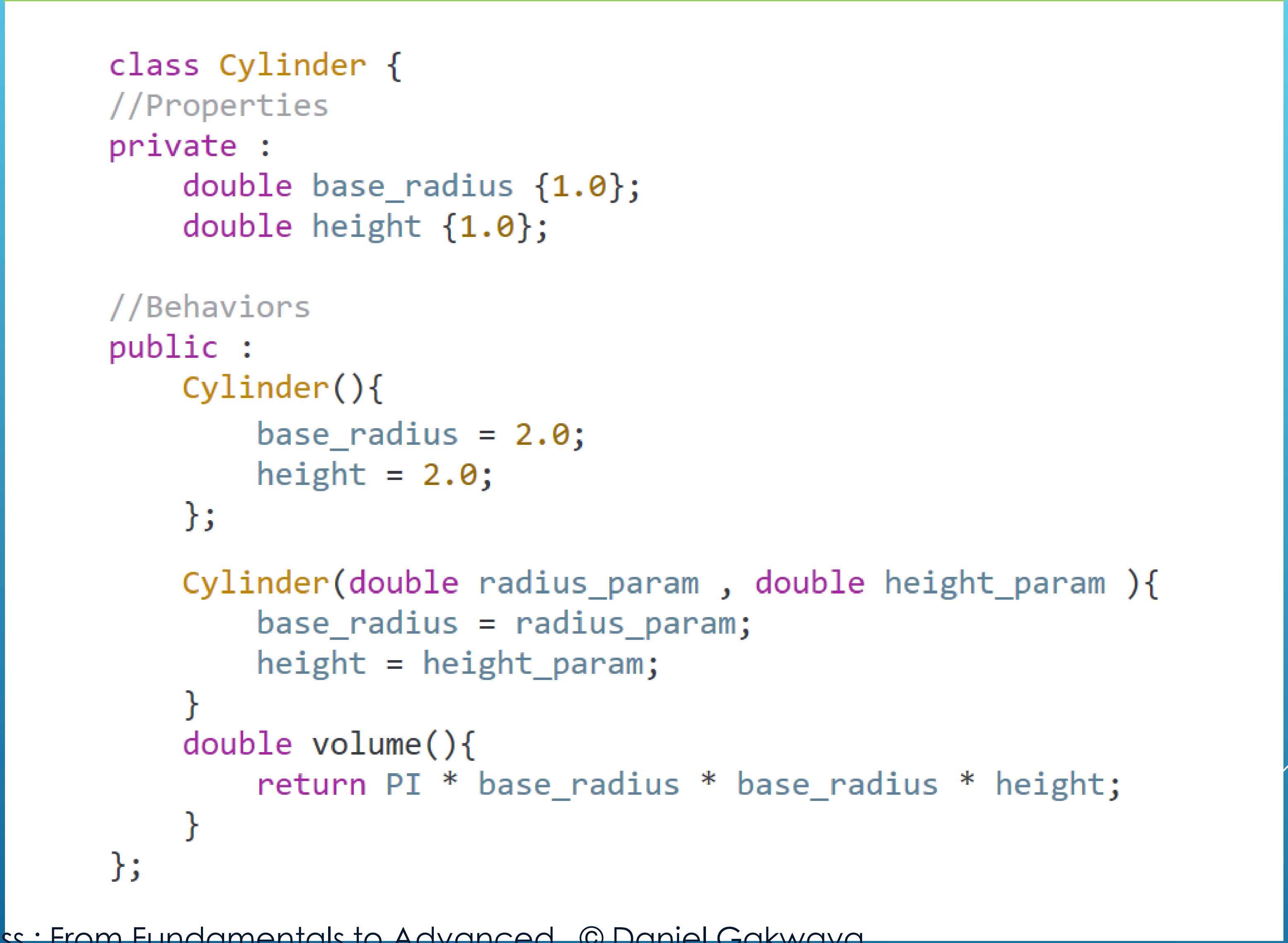
Constructors

Class constructor

A special kind of method that is called when an instance of a class is created

- No return type
- Same name as the class
- Can have parameters. Can also have an empty parameter list
- Usually used to initialize member variables of a class

Class constructors

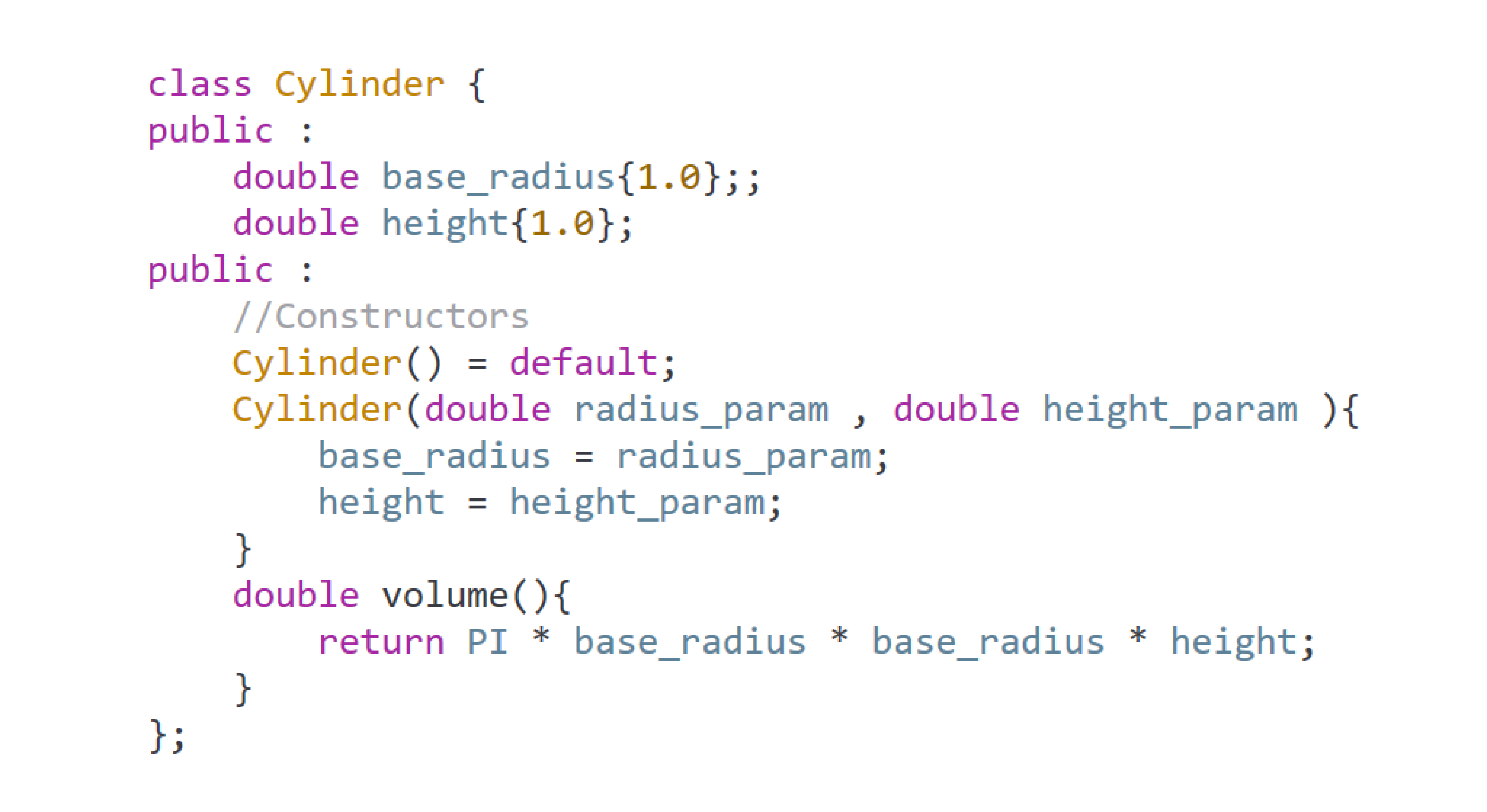


```
class Cylinder {
//Properties
private :
    double base_radius {1.0};
    double height {1.0};

//Behaviors
public :
    Cylinder(){
        base_radius = 2.0;
        height = 2.0;
    };

    Cylinder(double radius_param , double height_param ){
        base_radius = radius_param;
        height = height_param;
    }
    double volume(){
        return PI * base_radius * base_radius * height;
    }
};
```

Slide intentionally left empty

Defaulted constructors

Defaulted constructor



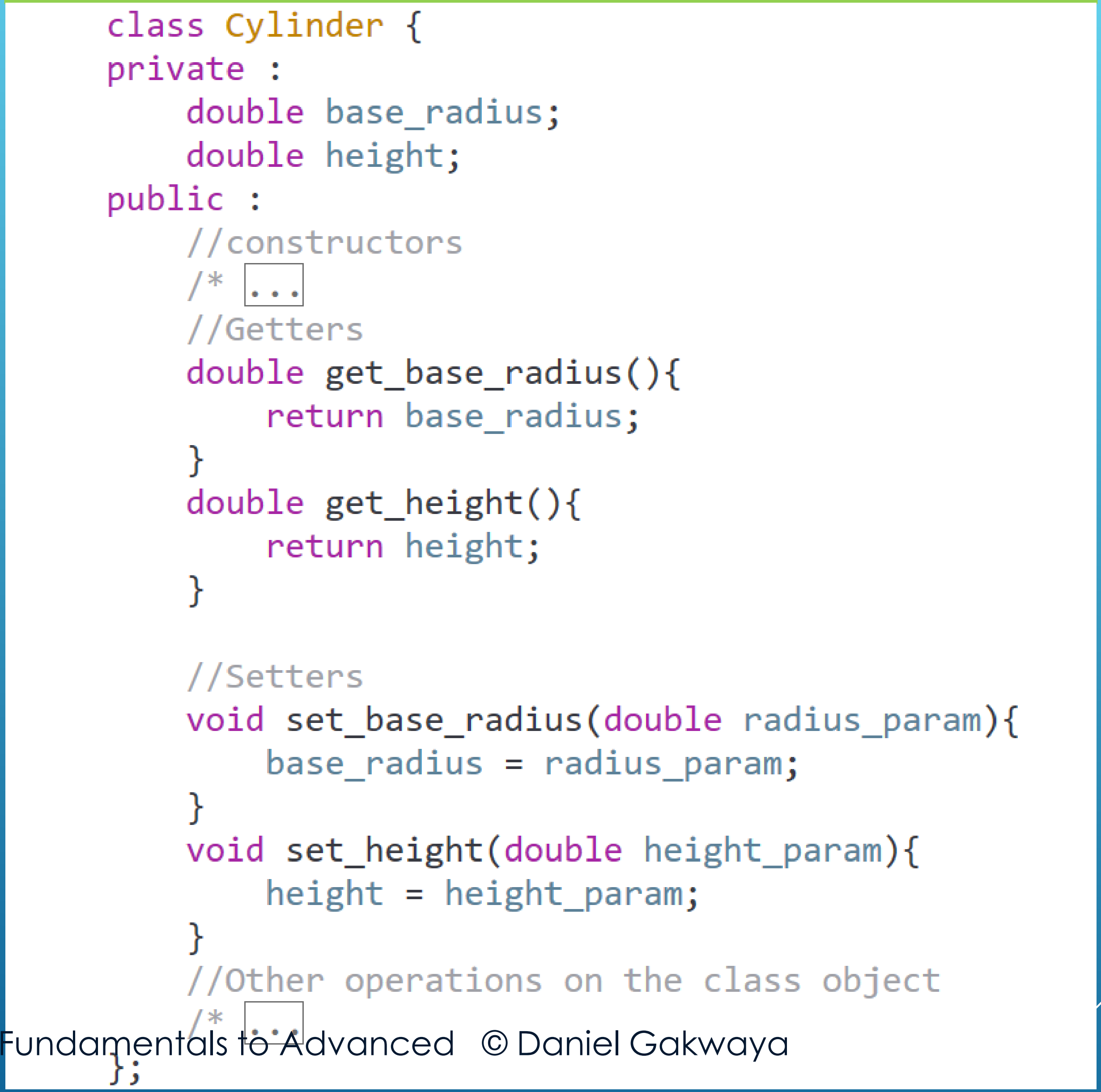
```cpp
class Cylinder {
public :
    double base_radius{1.0};;
    double height{1.0};
public :
    //Constructors
    Cylinder() = default;
    Cylinder(double radius_param , double height_param ){
        base_radius = radius_param;
        height = height_param;
    }
    double volume(){
        return PI * base_radius * base_radius * height;
    }
};
```

Slide intentionally left empty

Setters and Getters

Methods to read or modify member variables of a class

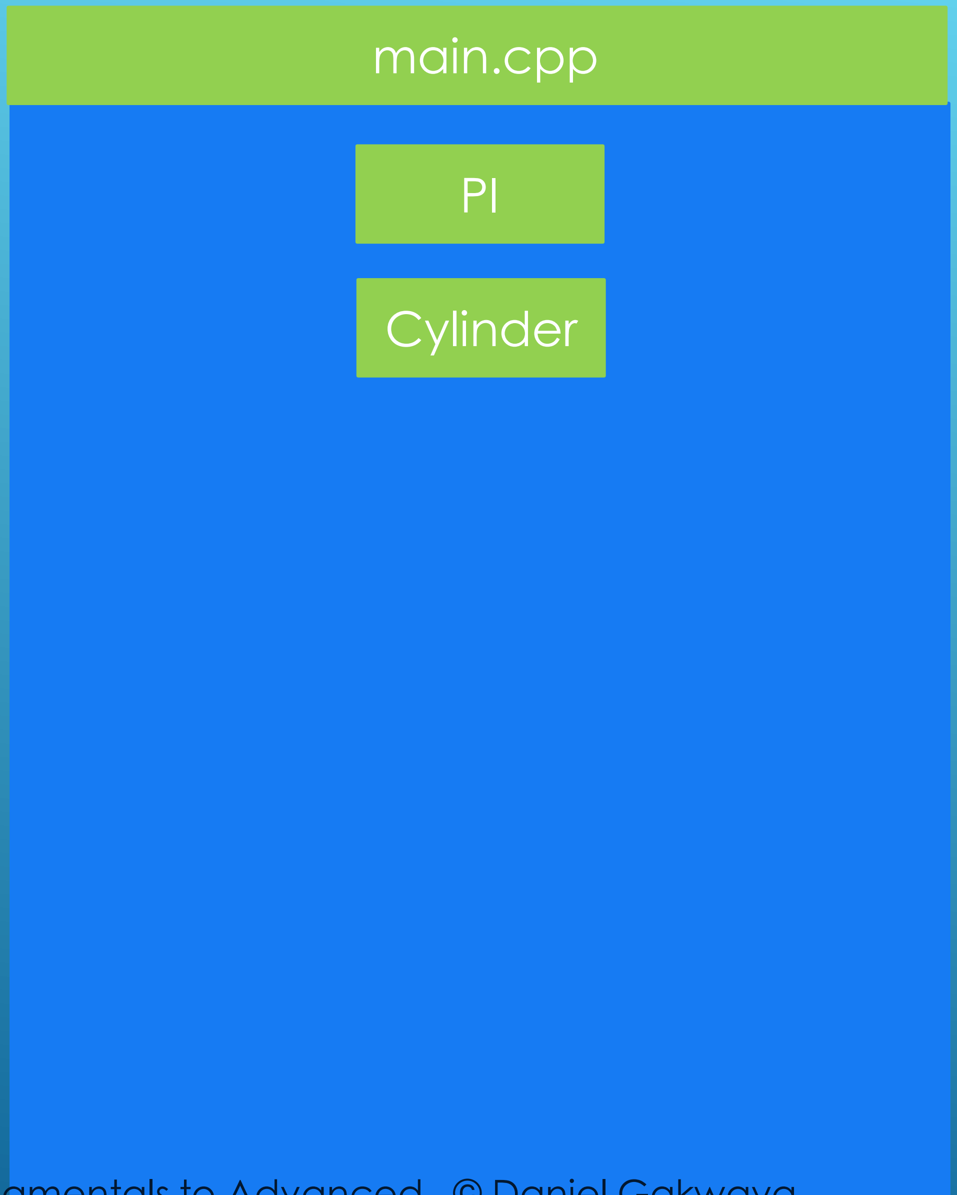
Getters & Setters



```
class Cylinder {
private :
    double base_radius;
    double height;
public :
    //constructors
    /* ...
    //Getters
    double get_base_radius(){
        return base_radius;
    }
    double get_height(){
        return height;
    }

    //Setters
    void set_base_radius(double radius_param){
        base_radius = radius_param;
    }
    void set_height(double height_param){
        height = height_param;
    }
    //Other operations on the class object
    /* ...
};
```

Slide intentionally left empty

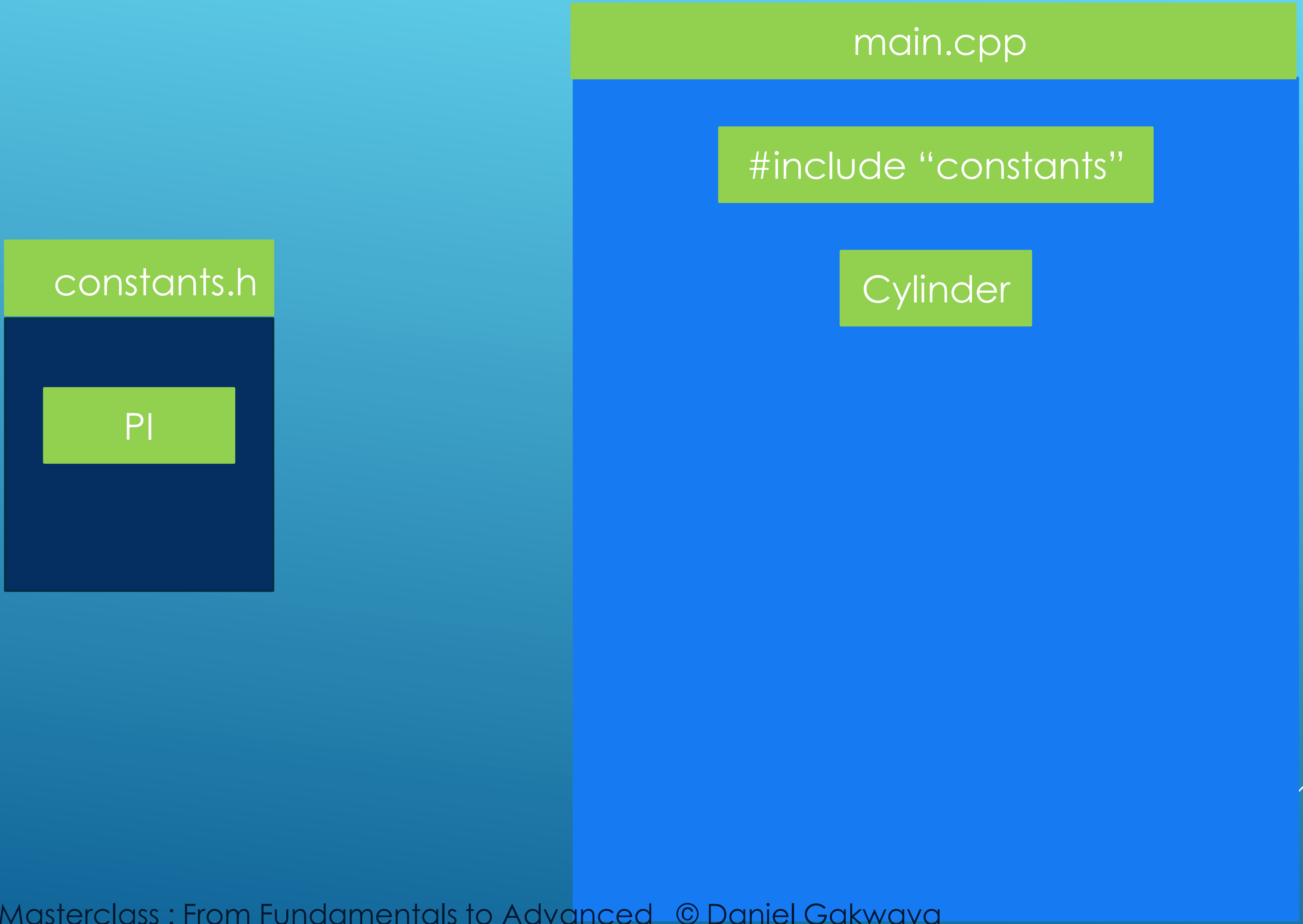
Class across multiple files



main.cpp

PI

Cylinder



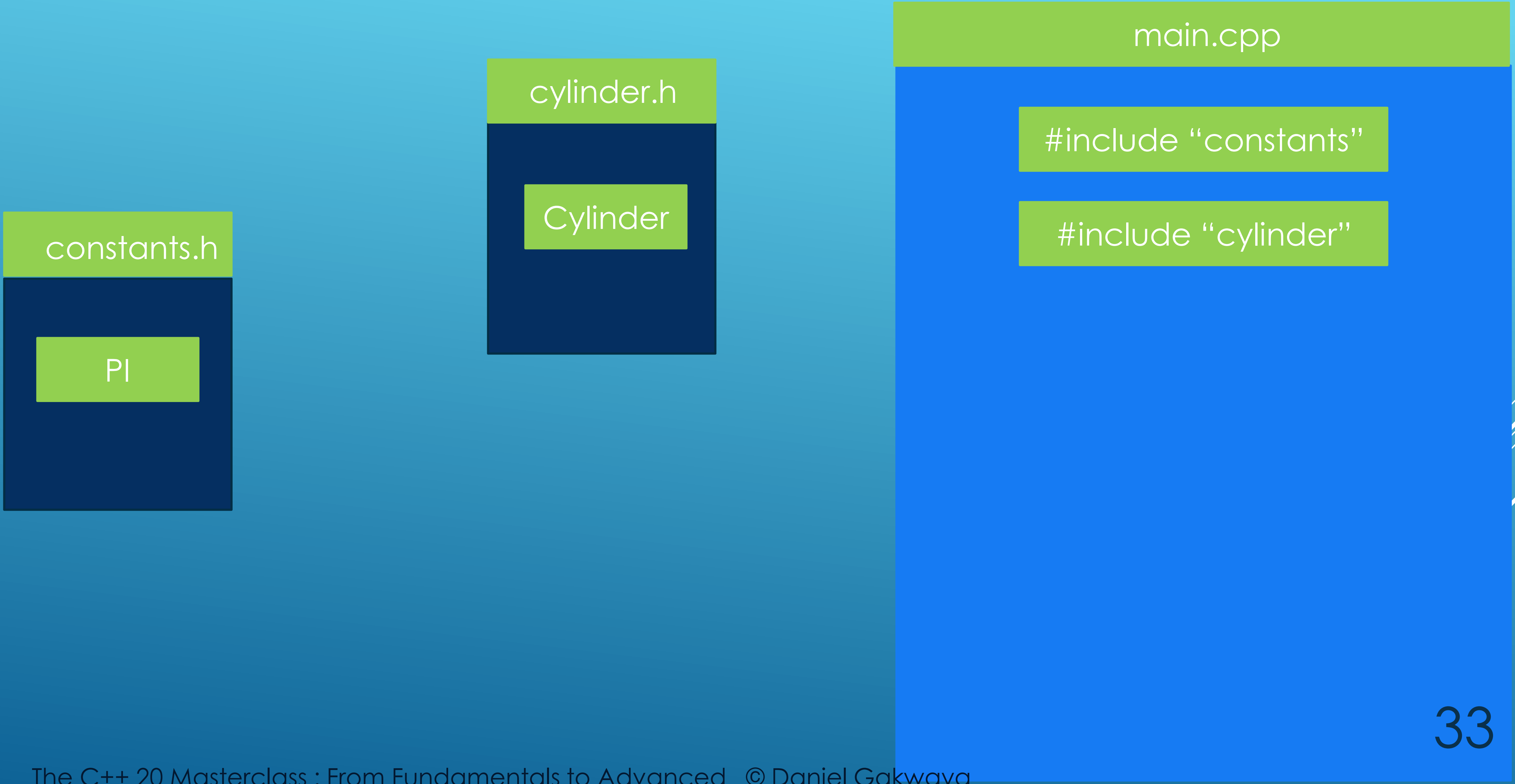
constants.h

PI

main.cpp

#include “constants”

Cylinder



constants.h

PI

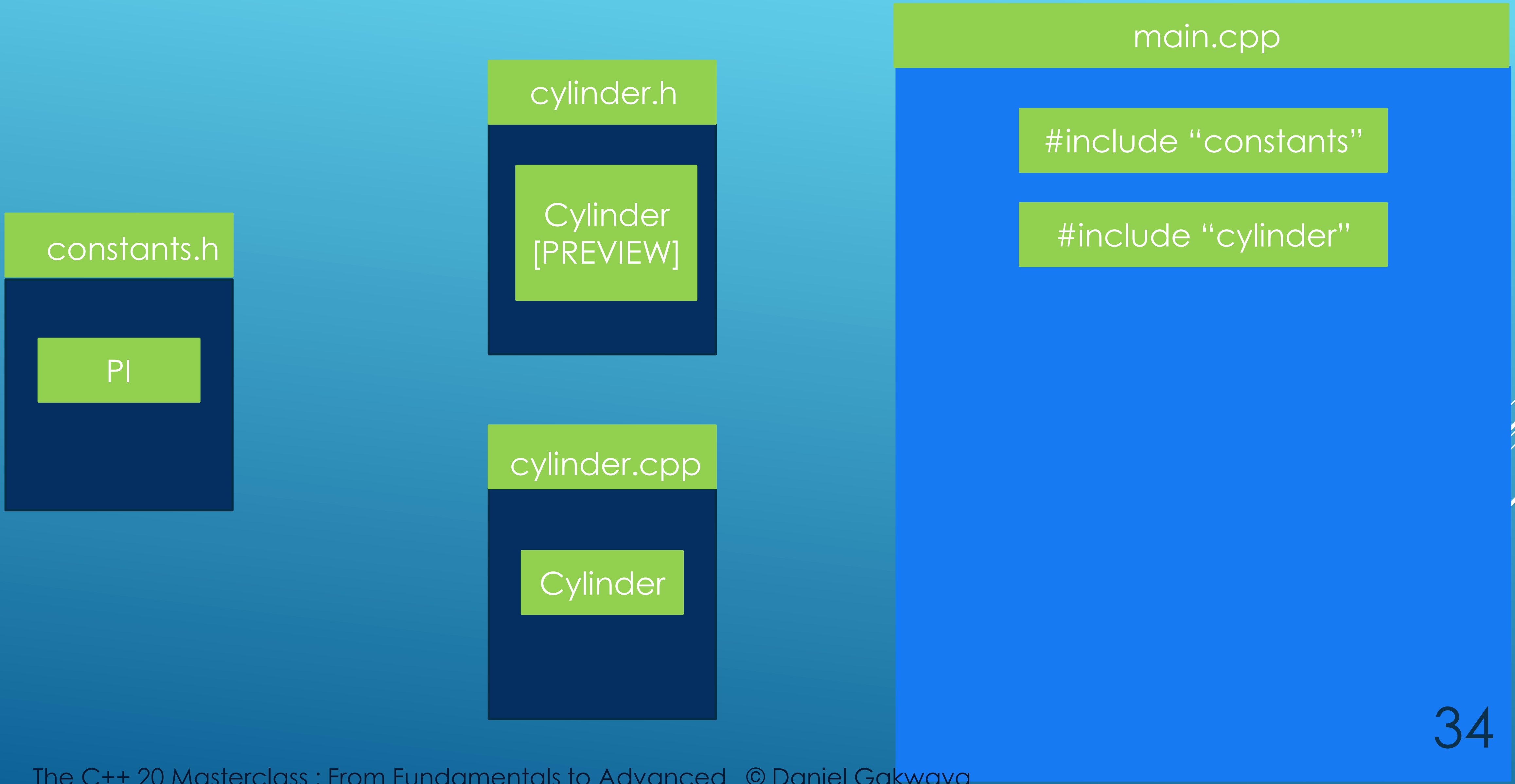
cylinder.h

Cylinder

main.cpp

#include “constants”

#include “cylinder”



constants.h

PI

cylinder.h

Cylinder [PREVIEW]

cylinder.cpp

Cylinder

main.cpp

#include “constants”

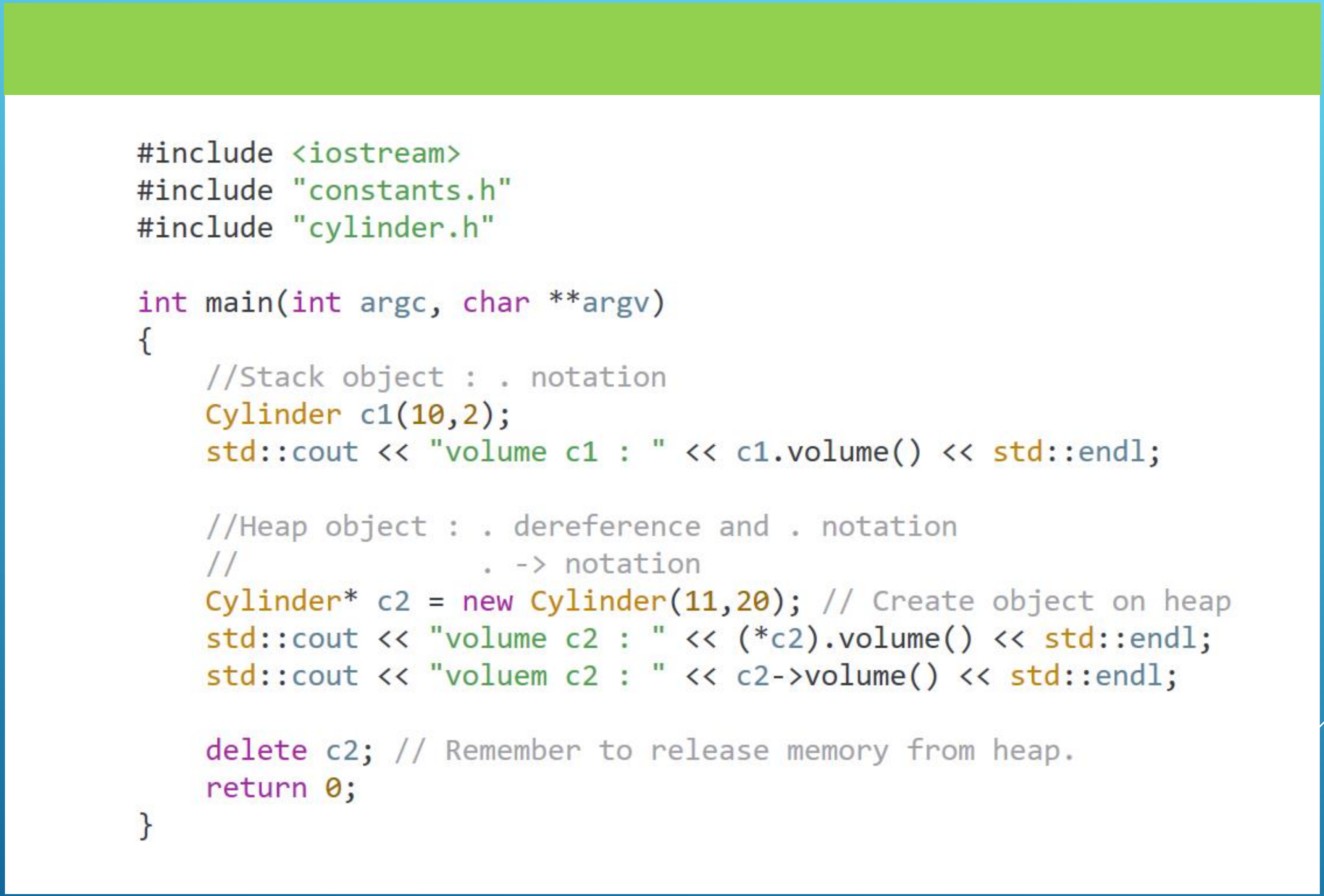
#include “cylinder”

Slide intentionally left empty

Creating classes through IDEs

Slide intentionally left empty

Managing Class Objects Through Pointers



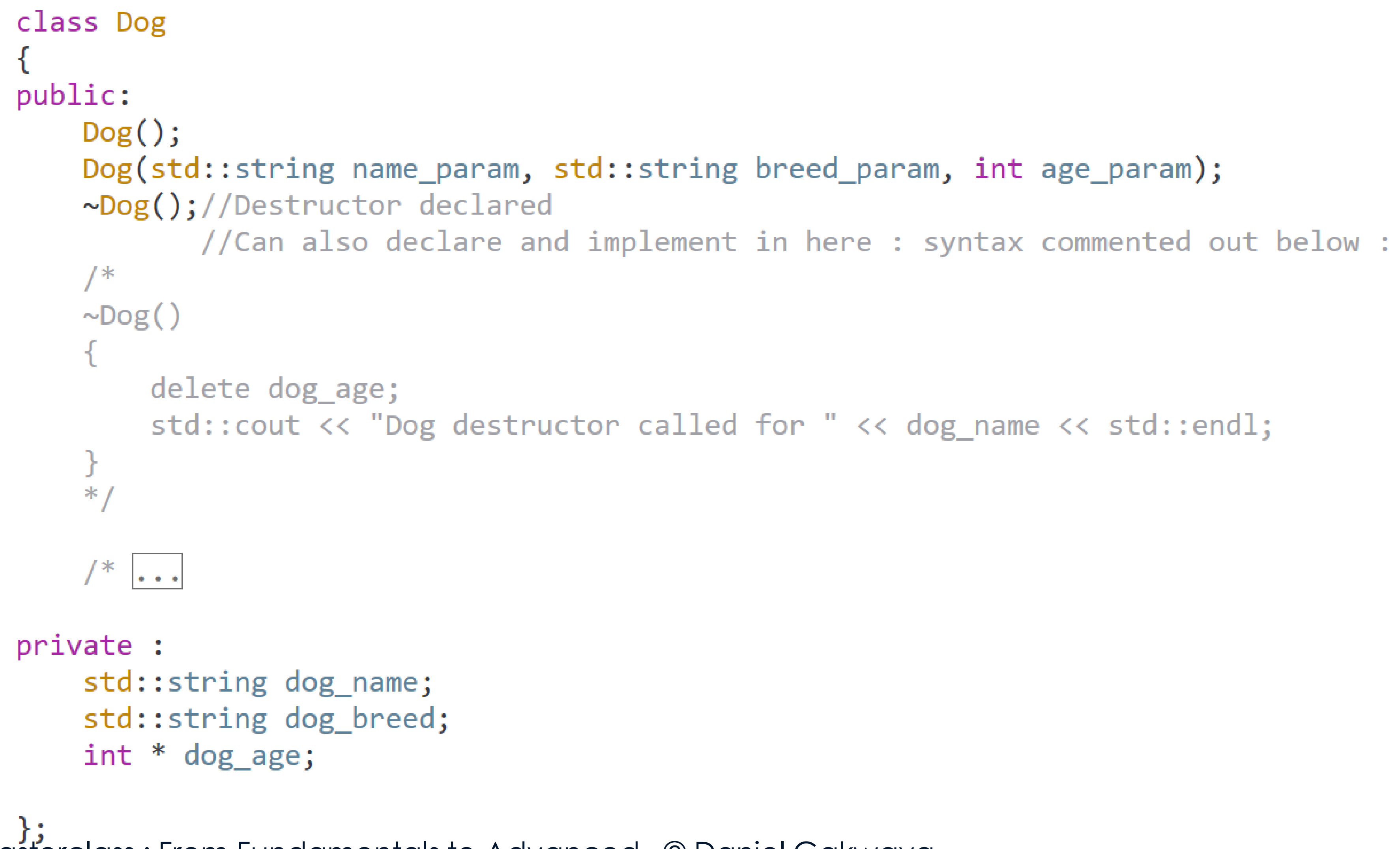
```cpp
#include <iostream>
#include "constants.h"
#include "cylinder.h"

int main(int argc, char **argv)
{
    //Stack object : . notation
    Cylinder c1(10,2);
    std::cout << "volume c1 : " << c1.volume() << std::endl;

    //Heap object : . dereference and . notation
    //              . -> notation
    Cylinder* c2 = new Cylinder(11,20); // Create object on heap
    std::cout << "volume c2 : " << (*c2).volume() << std::endl;
    std::cout << "voluem c2 : " << c2->volume() << std::endl;

    delete c2; // Remember to release memory from heap.
    return 0;
}
```

Slide intentionally left empty

Destructors

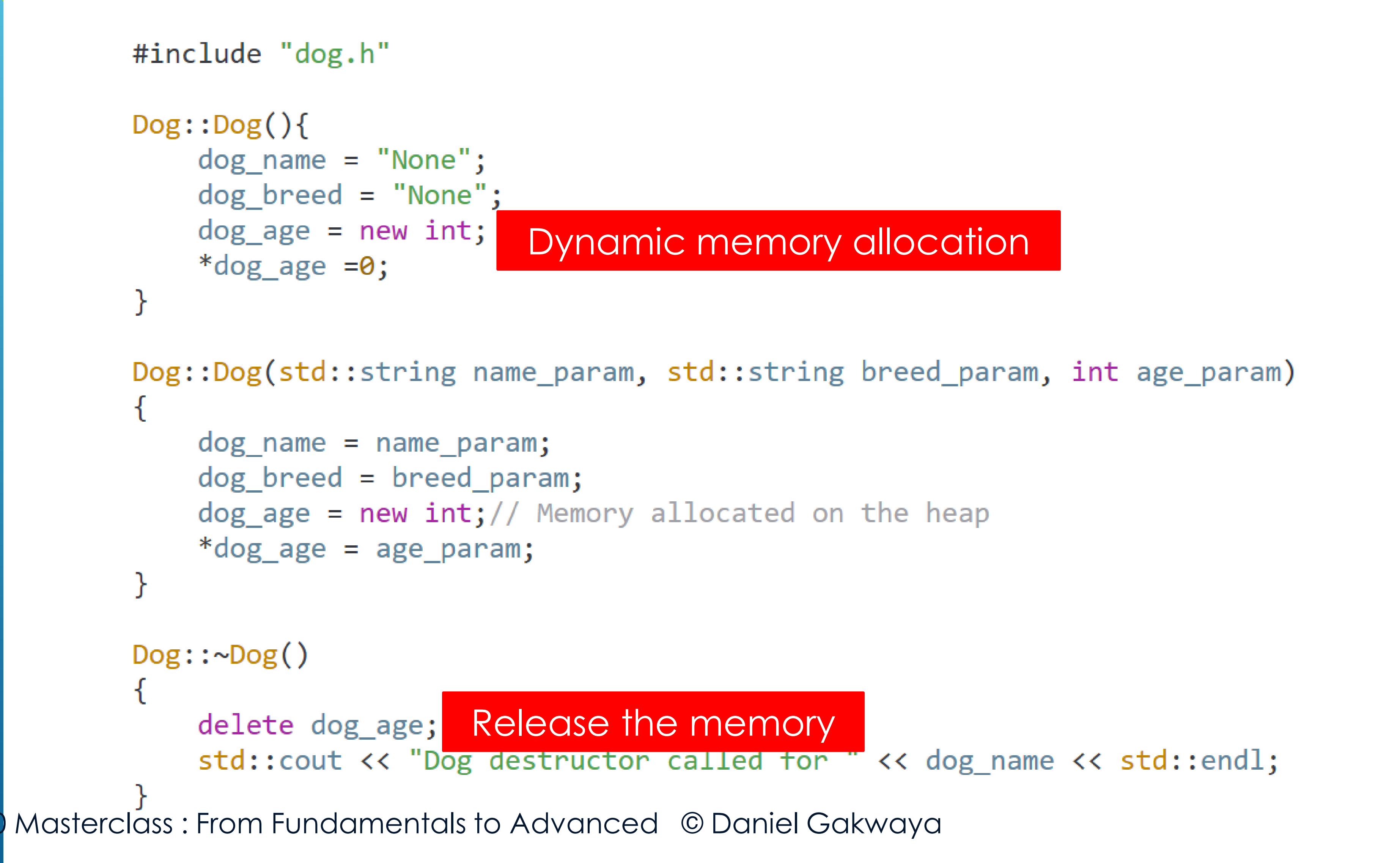
Special methods that are called when an object dies. They are needed when the object needs to release some dynamic memory, or for some other kind of clean up.

Destructors : syntax



```
class Dog
{
public:
    Dog();
    Dog(std::string name_param, std::string breed_param, int age_param);
    ~Dog();//Destructor declared
           //Can also declare and implement in here : syntax commented out below :
    /*
    ~Dog()
    {
        delete dog_age;
        std::cout << "Dog destructor called for " << dog_name << std::endl;
    }
    */

    /* ...

private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;

};
```

dog.cpp

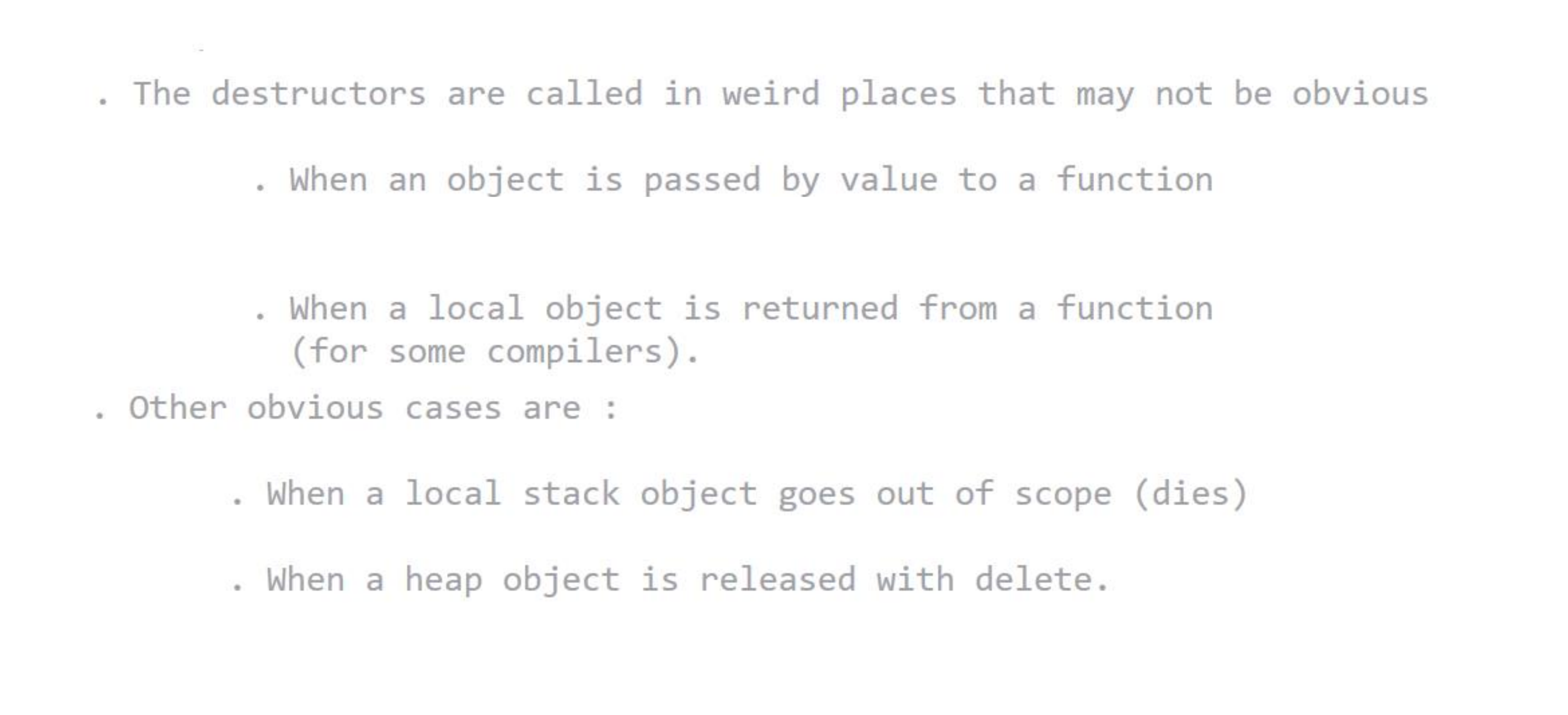


```
#include "dog.h"

Dog::Dog(){
    dog_name = "None";
    dog_breed = "None";
    dog_age = new int;
    *dog_age =0;
}

Dog::Dog(std::string name_param, std::string breed_param, int age_param)
{
    dog_name = name_param;
    dog_breed = breed_param;
    dog_age = new int;// Memory allocated on the heap
    *dog_age = age_param;
}

Dog::~Dog()
{
    delete dog_age;
    std::cout << "Dog destructor called for " << dog_name << std::endl;
}
```

Dynamic memory allocation

Release the memory

When are destructors called

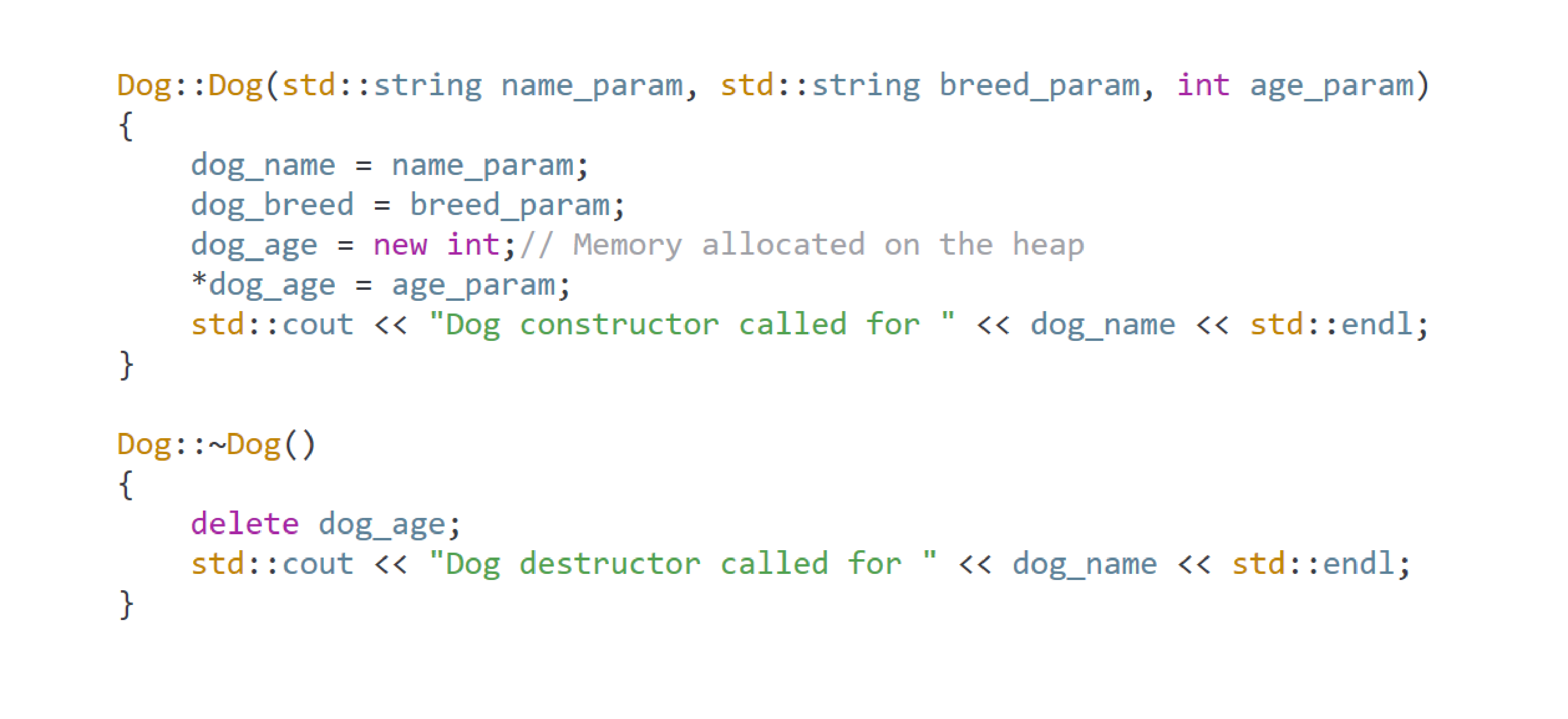


- The destructors are called in weird places that may not be obvious
  - When an object is passed by value to a function
  - When a local object is returned from a function (for some compilers).
- Other obvious cases are :
  - When a local stack object goes out of scope (dies)
  - When a heap object is released with delete.

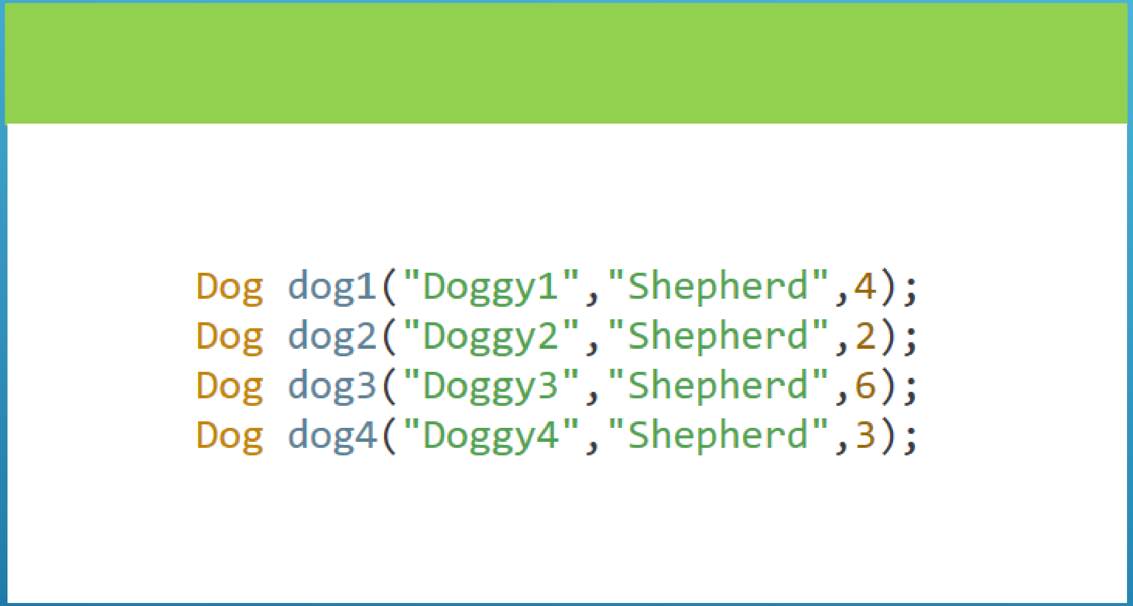
Slide intentionally left empty

Constructor & Destructor call order

Constructor and destructor

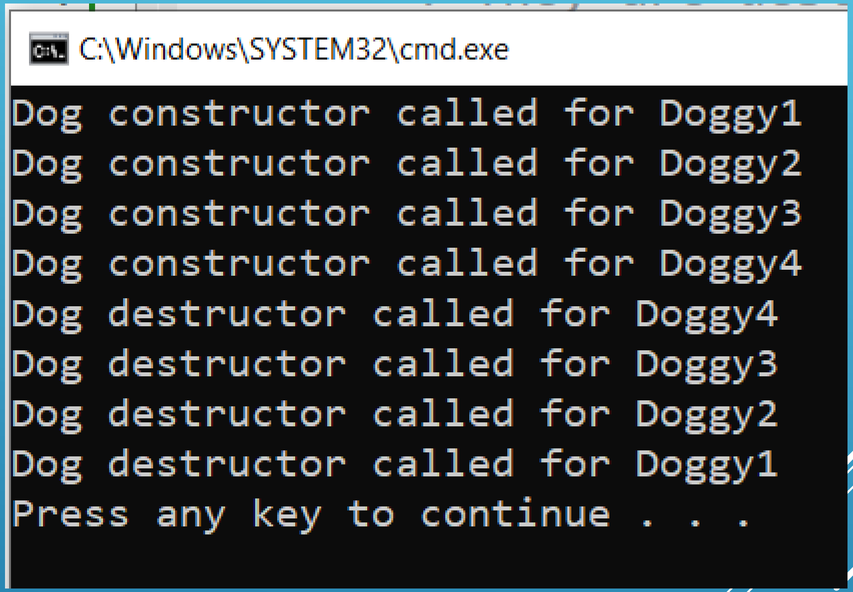


```
Dog::Dog(std::string name_param, std::string breed_param, int age_param)
{
    dog_name = name_param;
    dog_breed = breed_param;
    dog_age = new int;// Memory allocated on the heap
    *dog_age = age_param;
    std::cout << "Dog constructor called for " << dog_name << std::endl;
}

Dog::~Dog()
{
    delete dog_age;
    std::cout << "Dog destructor called for " << dog_name << std::endl;
}
```



```
Dog dog1("Doggy1","Shepherd",4);
Dog dog2("Doggy2","Shepherd",2);
Dog dog3("Doggy3","Shepherd",6);
Dog dog4("Doggy4","Shepherd",3);
```

```cpp
Dog dog1("Doggy1","Shepherd",4);
Dog dog2("Doggy2","Shepherd",2);
Dog dog3("Doggy3","Shepherd",6);
Dog dog4("Doggy4","Shepherd",3);
```



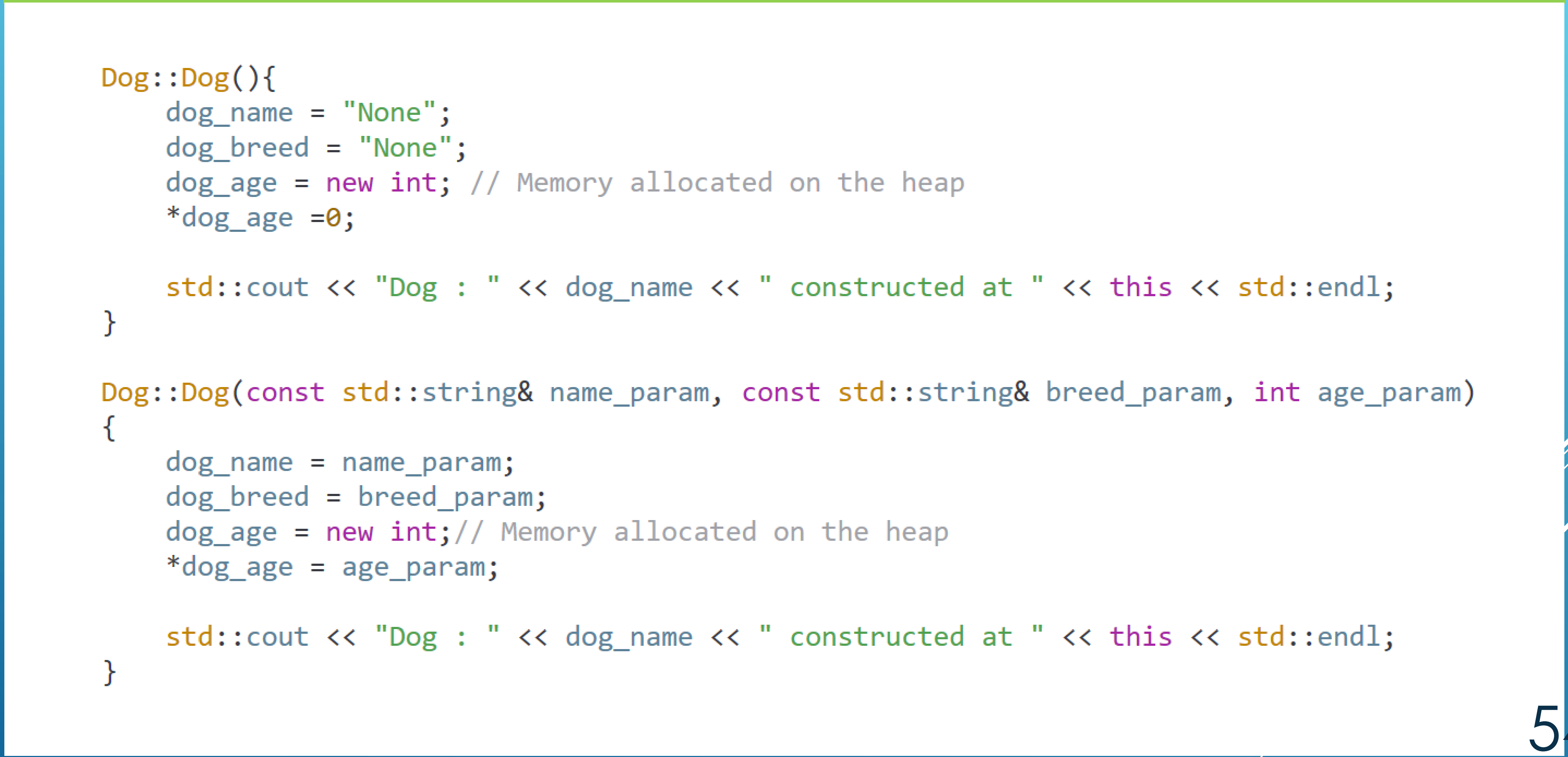
```
Dog constructor called for Doggy1
Dog constructor called for Doggy2
Dog constructor called for Doggy3
Dog constructor called for Doggy4
Dog destructor called for Doggy4
Dog destructor called for Doggy3
Dog destructor called for Doggy2
Dog destructor called for Doggy1
Press any key to continue . . .
```

Slide intentionally left empty

The this pointer

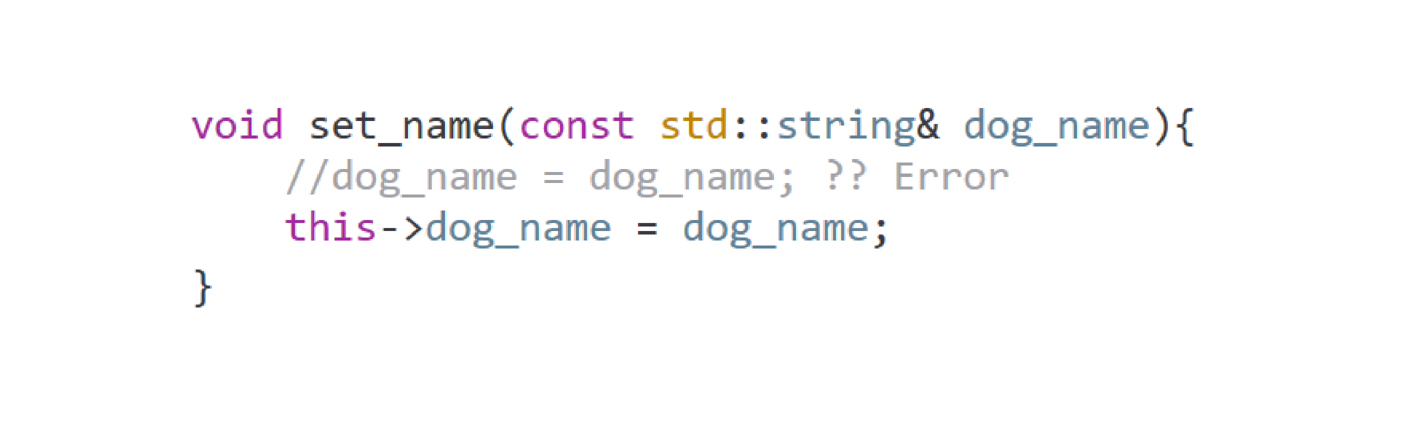
Each class member function contains a hidden pointer called this. That pointer contains the address of the current object, for which the method is being executed. This also applies to constructors and destructors.

Printing out object addresses



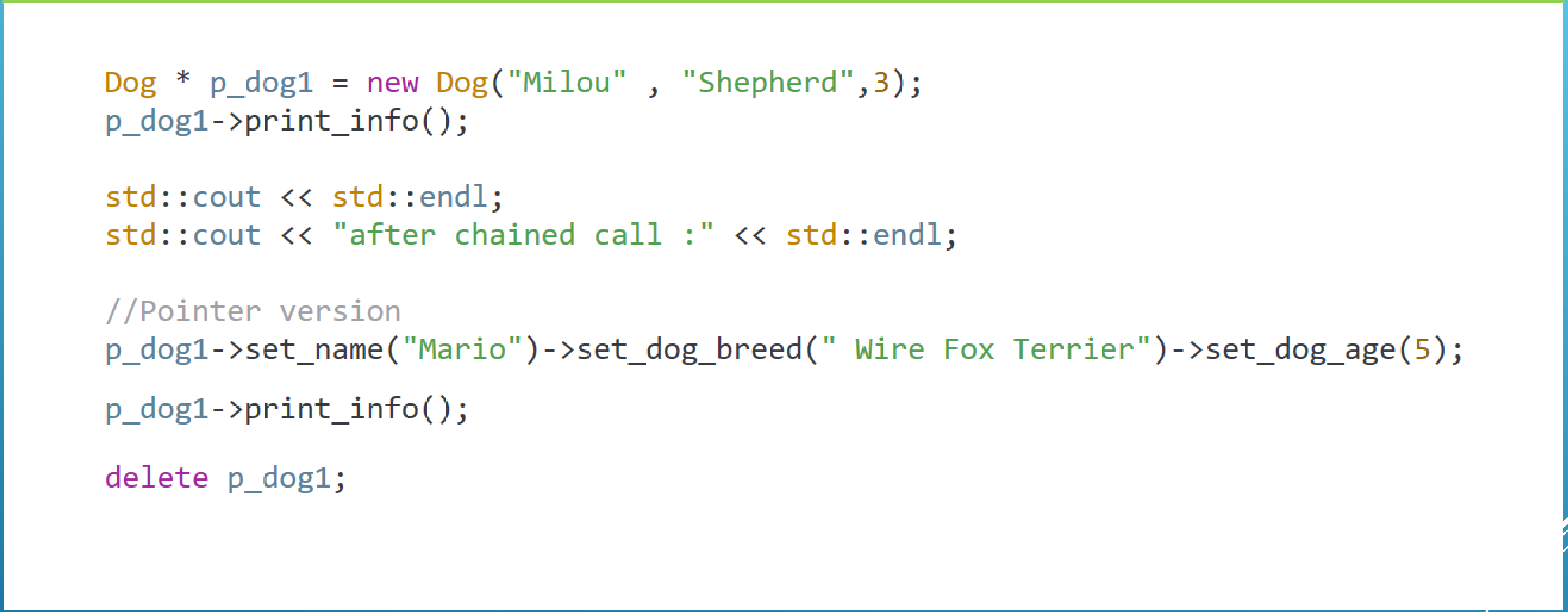
```
Dog::Dog(){
    dog_name = "None";
    dog_breed = "None";
    dog_age = new int; // Memory allocated on the heap
    *dog_age =0;

    std::cout << "Dog : " << dog_name << " constructed at " << this << std::endl;
}

Dog::Dog(const std::string& name_param, const std::string& breed_param, int age_param)
{
    dog_name = name_param;
    dog_breed = breed_param;
    dog_age = new int;// Memory allocated on the heap
    *dog_age = age_param;

    std::cout << "Dog : " << dog_name << " constructed at " << this << std::endl;
}
```

Conflicting names



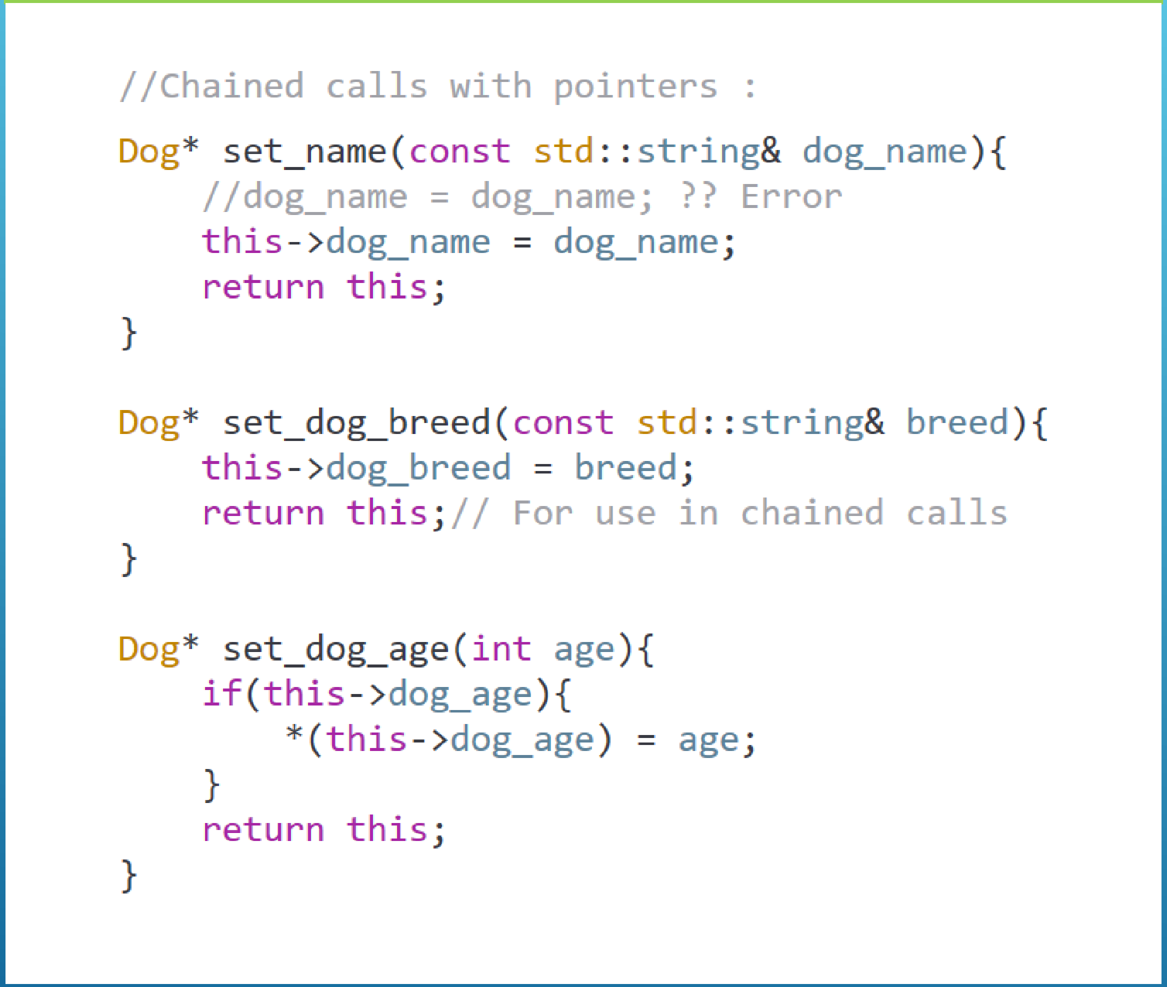
```
void set_name(const std::string& dog_name){
    //dog_name = dog_name; ?? Error
    this->dog_name = dog_name;
}
```

Chained calls using pointers



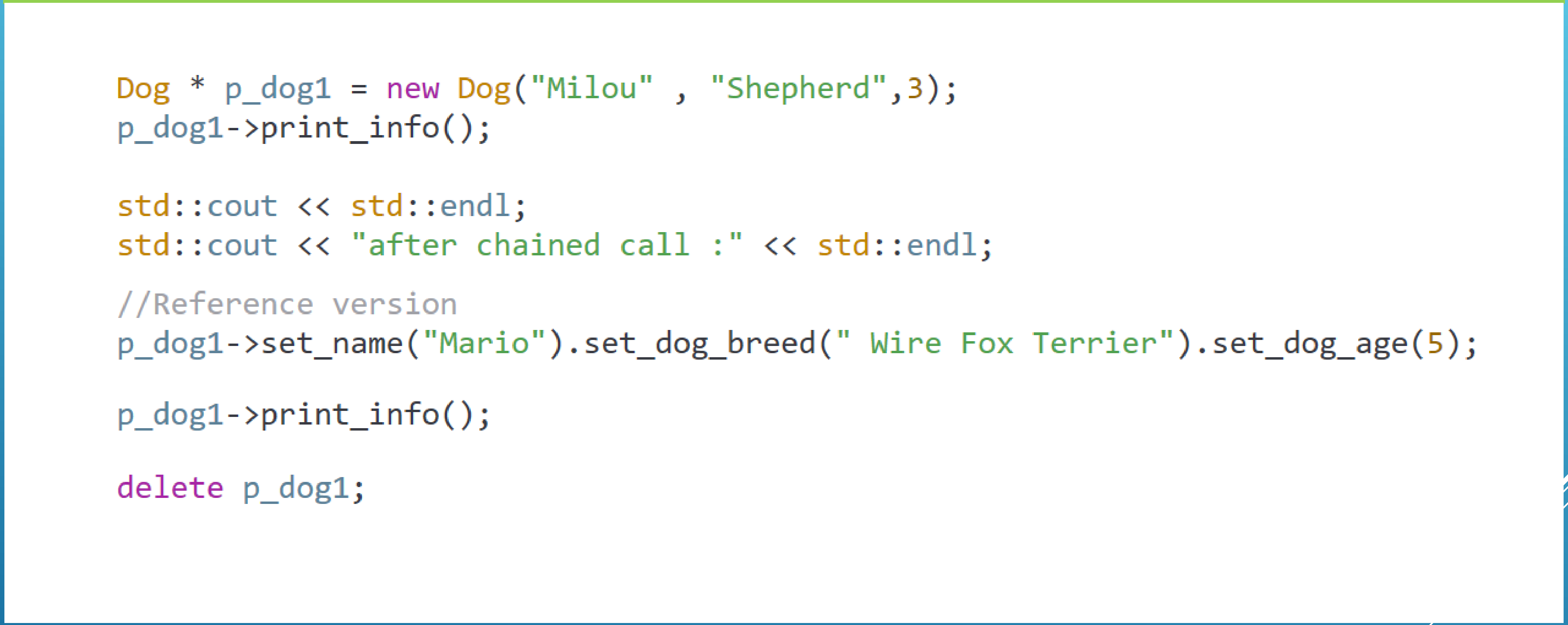
```
Dog * p_dog1 = new Dog("Milou" , "Shepherd",3);
p_dog1->print_info();

std::cout << std::endl;
std::cout << "after chained call :" << std::endl;

//Pointer version
p_dog1->set_name("Mario")->set_dog_breed(" Wire Fox Terrier")->set_dog_age(5);

p_dog1->print_info();

delete p_dog1;
```

Chained calls using pointers



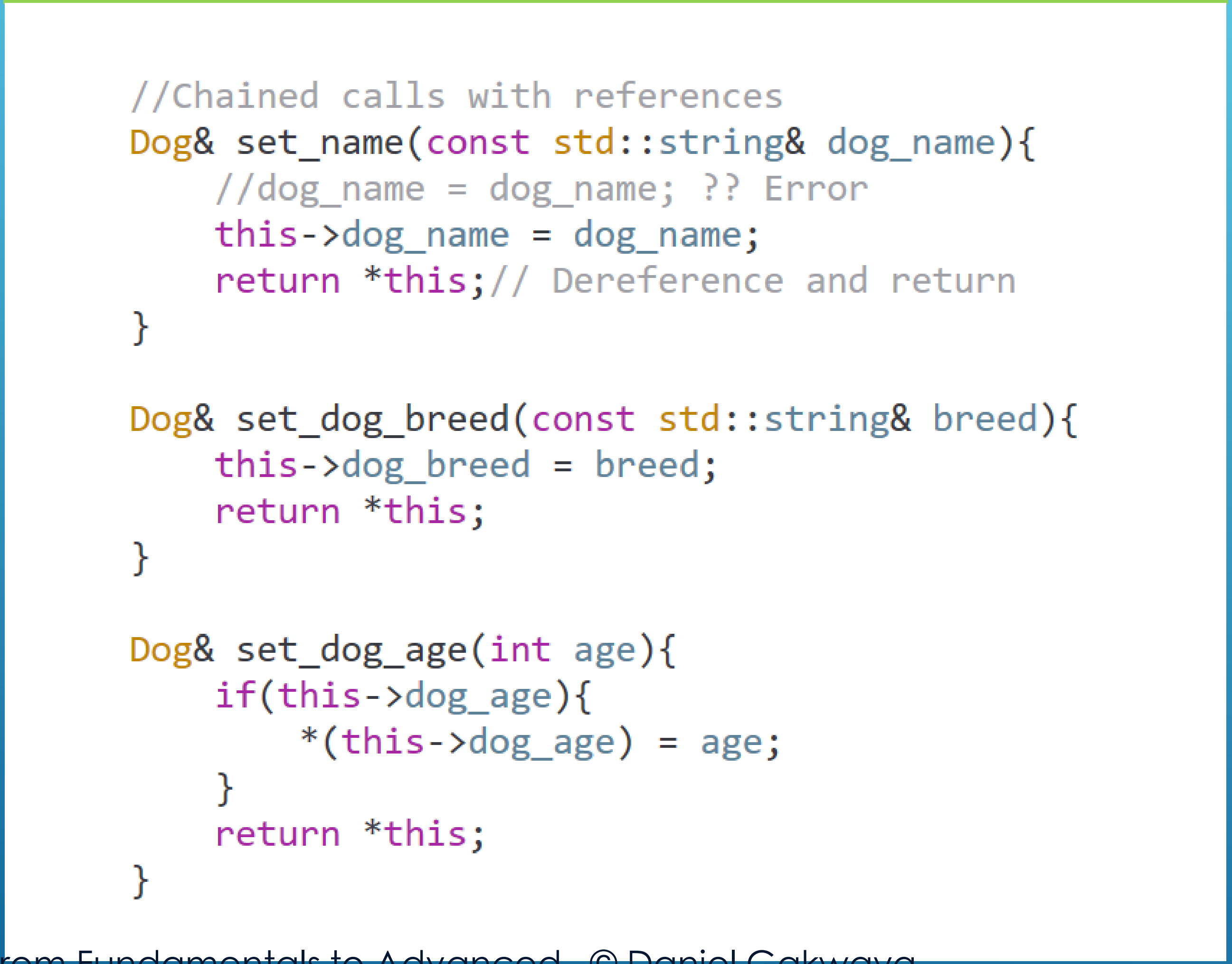
```
//Chained calls with pointers :

Dog* set_name(const std::string& dog_name){
    //dog_name = dog_name; ?? Error
    this->dog_name = dog_name;
    return this;
}

Dog* set_dog_breed(const std::string& breed){
    this->dog_breed = breed;
    return this;// For use in chained calls
}

Dog* set_dog_age(int age){
    if(this->dog_age){
        *(this->dog_age) = age;
    }
    return this;
}
```

Chained calls using references



```
Dog * p_dog1 = new Dog("Milou" , "Shepherd",3);
p_dog1->print_info();

std::cout << std::endl;
std::cout << "after chained call :" << std::endl;

//Reference version
p_dog1->set_name("Mario").set_dog_breed(" Wire Fox Terrier").set_dog_age(5);

p_dog1->print_info();

delete p_dog1;
```

Chained calls using references

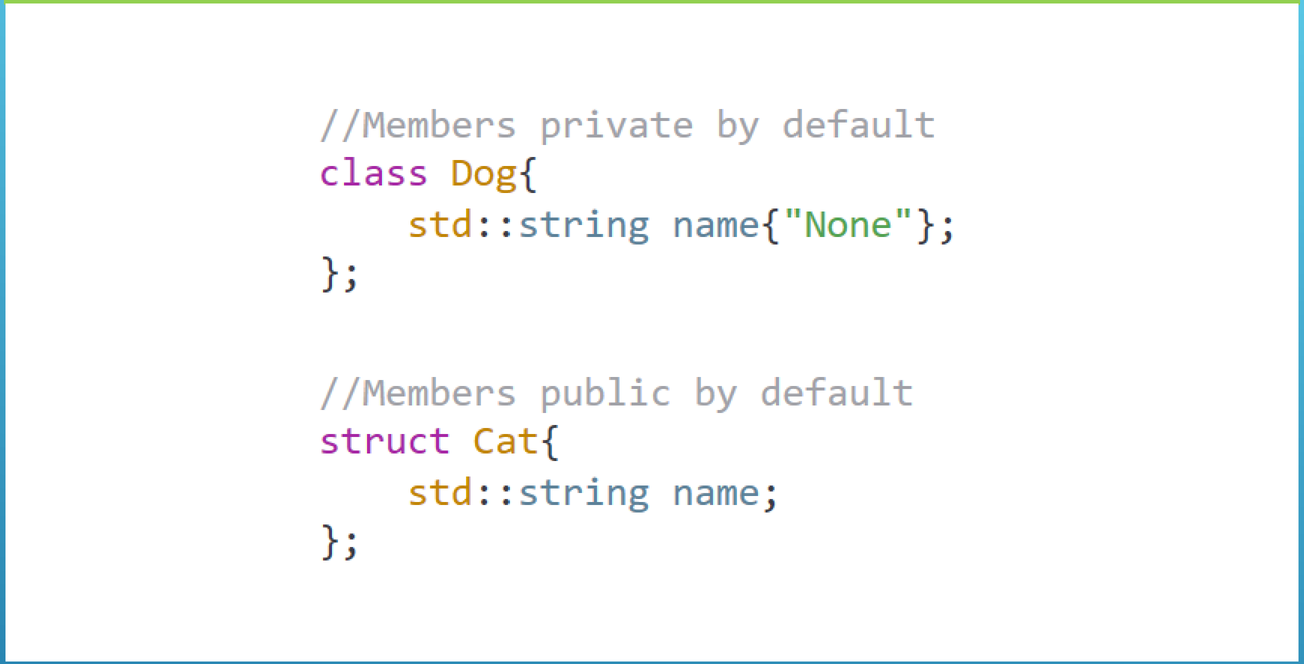


```cpp
//Chained calls with references
Dog& set_name(const std::string& dog_name){
    //dog_name = dog_name; ?? Error
    this->dog_name = dog_name;
    return *this;// Dereference and return
}

Dog& set_dog_breed(const std::string& breed){
    this->dog_breed = breed;
    return *this;
}

Dog& set_dog_age(int age){
    if(this->dog_age){
        *(this->dog_age) = age;
    }
    return *this;
}
```

Slide intentionally left empty

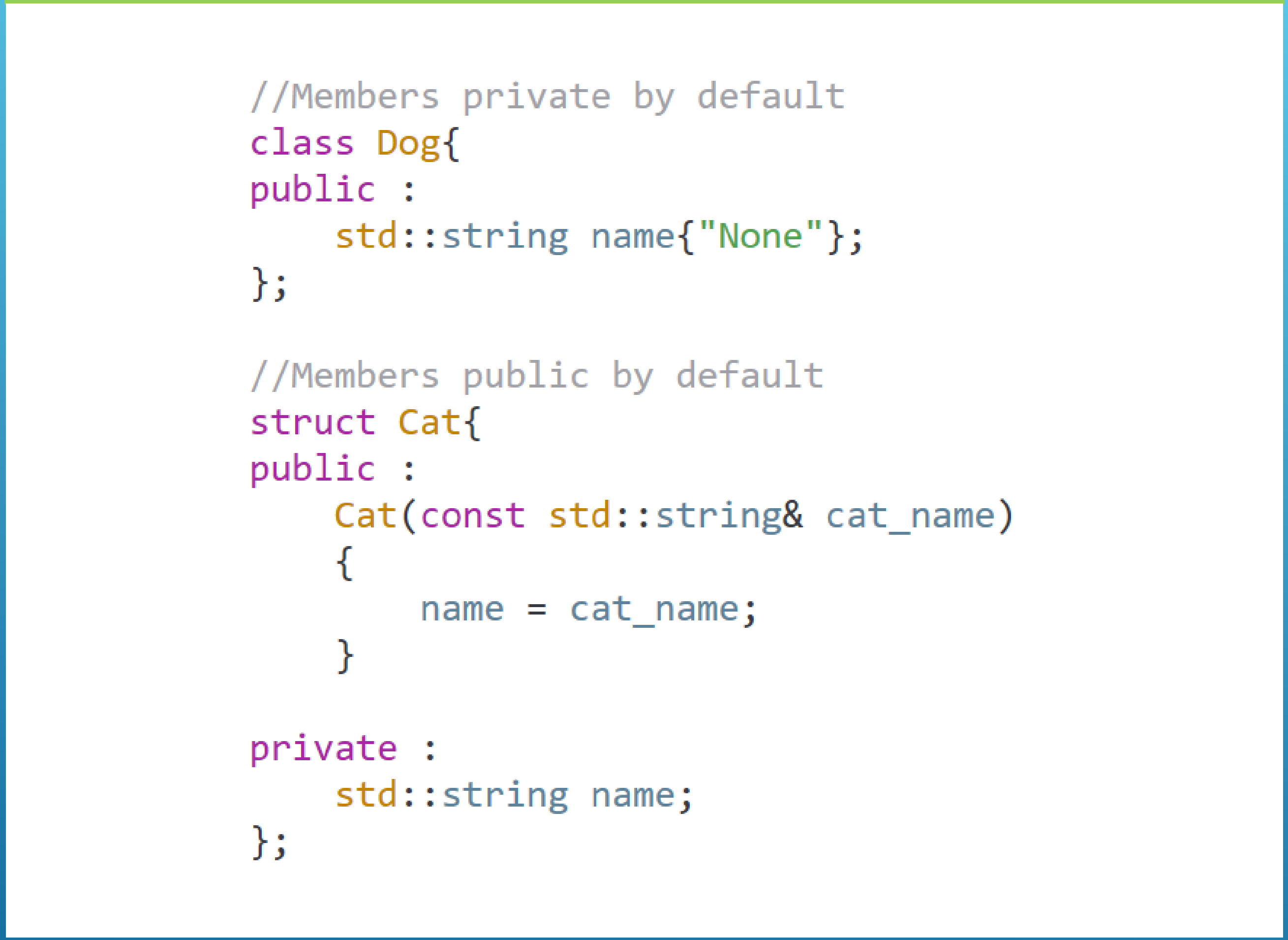
Struct

struct VS class



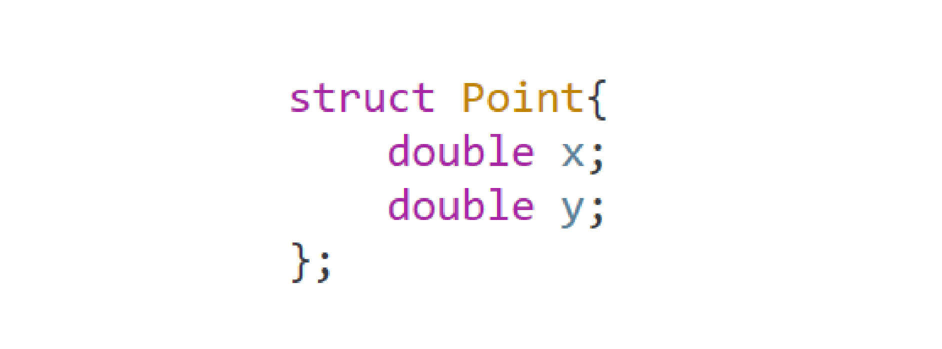
```
//Members private by default
class Dog{
    std::string name{"None"};
};

//Members public by default
struct Cat{
    std::string name;
};
```

Override defaults



```
//Members private by default
class Dog{
public :
    std::string name{"None"};
};

//Members public by default
struct Cat{
public :
    Cat(const std::string& cat_name)
    {
        name = cat_name;
    }

private :
    std::string name;
};
```

Common use for struct

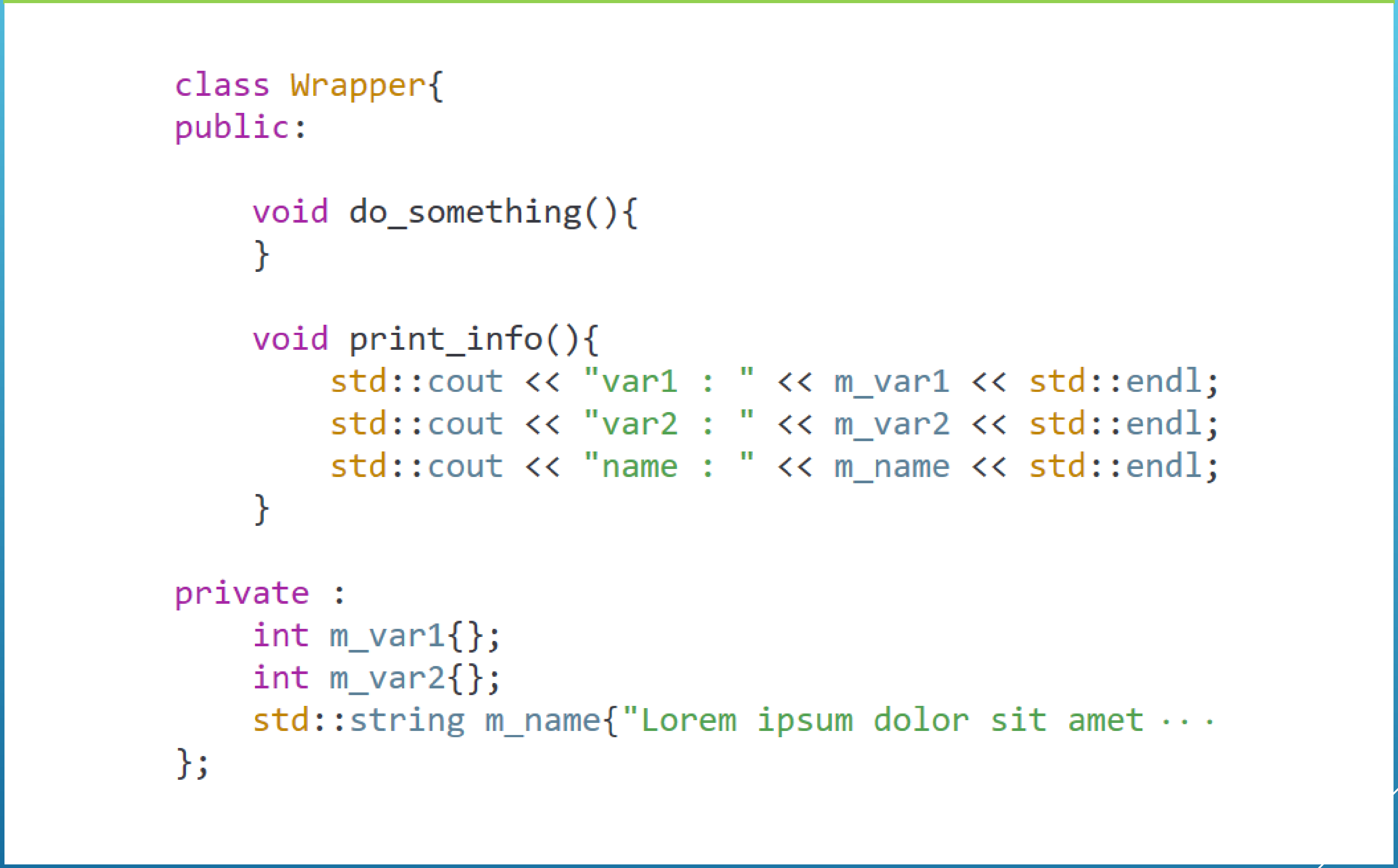


```
struct Point{
    double x;
    double y;
};
```

Slide intentionally left empty

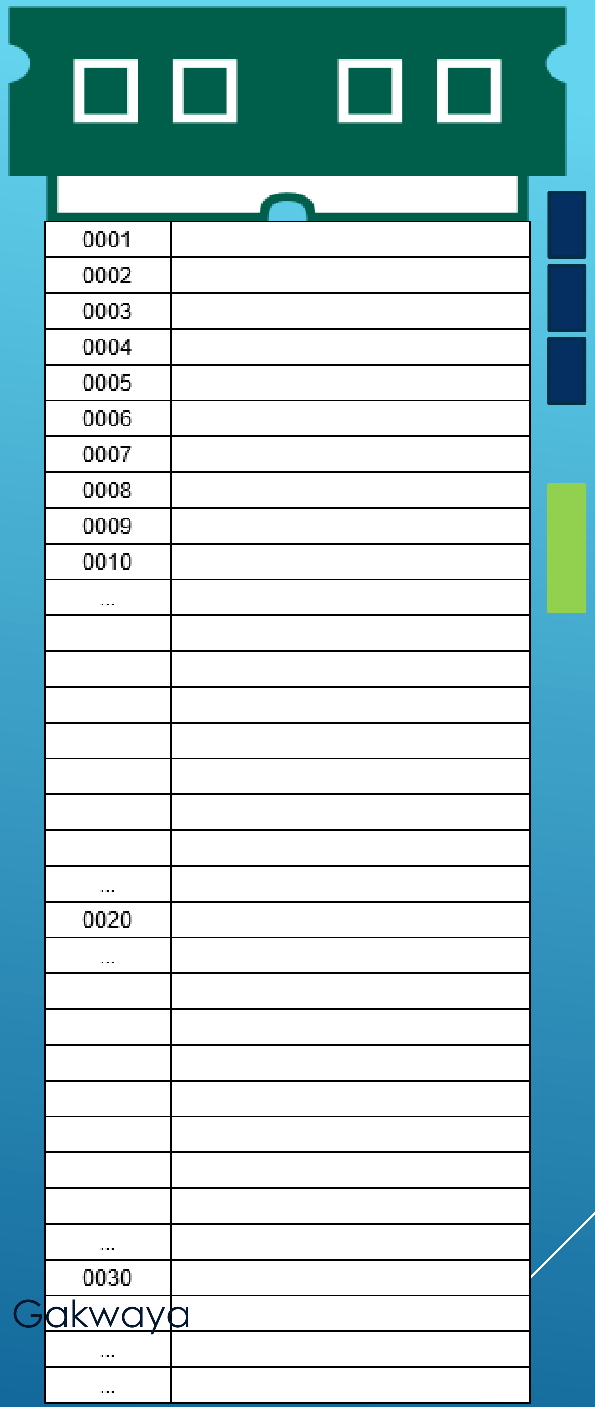
Size of class objects

Size of class objects



```cpp
class Wrapper{
public:

    void do_something(){
    }

    void print_info(){
        std::cout << "var1 : " << m_var1 << std::endl;
        std::cout << "var2 : " << m_var2 << std::endl;
        std::cout << "name : " << m_name << std::endl;
    }

private :
    int m_var1{};
    int m_var2{};
    std::string m_name{"Lorem ipsum dolor sit amet ···
};
```

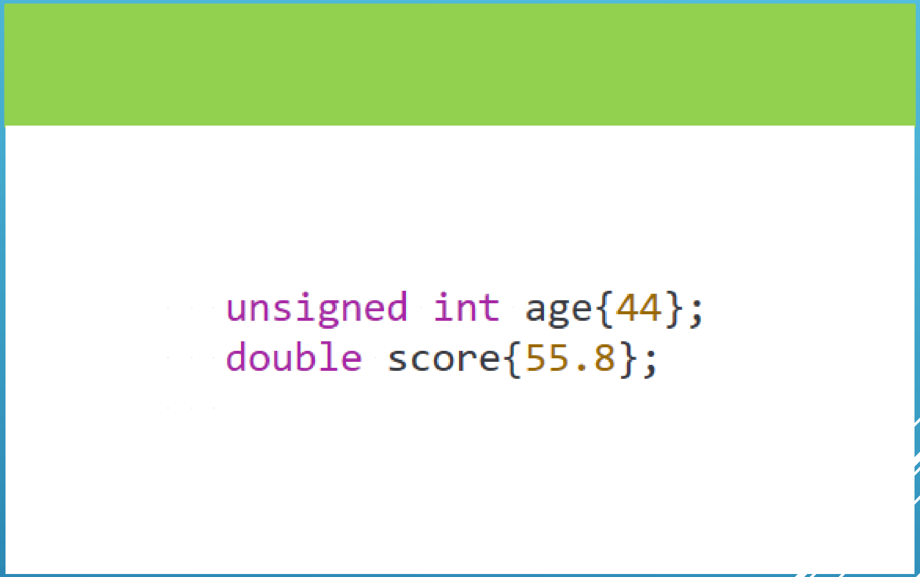
Boundary alignment



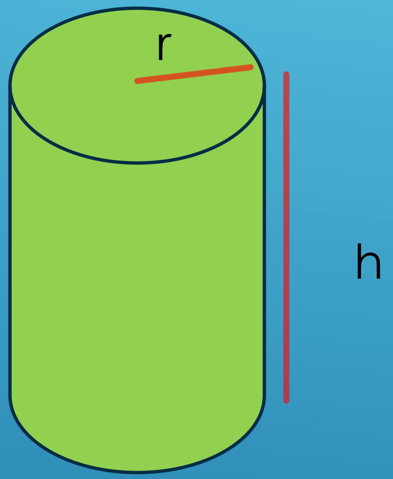
0001	
0002	
0003	
0004	
0005	
0006	
0007	
0008	
0009	
0010	
...	
...	
0020	
...	
...	
0030	
...	
...	

Slide intentionally left empty

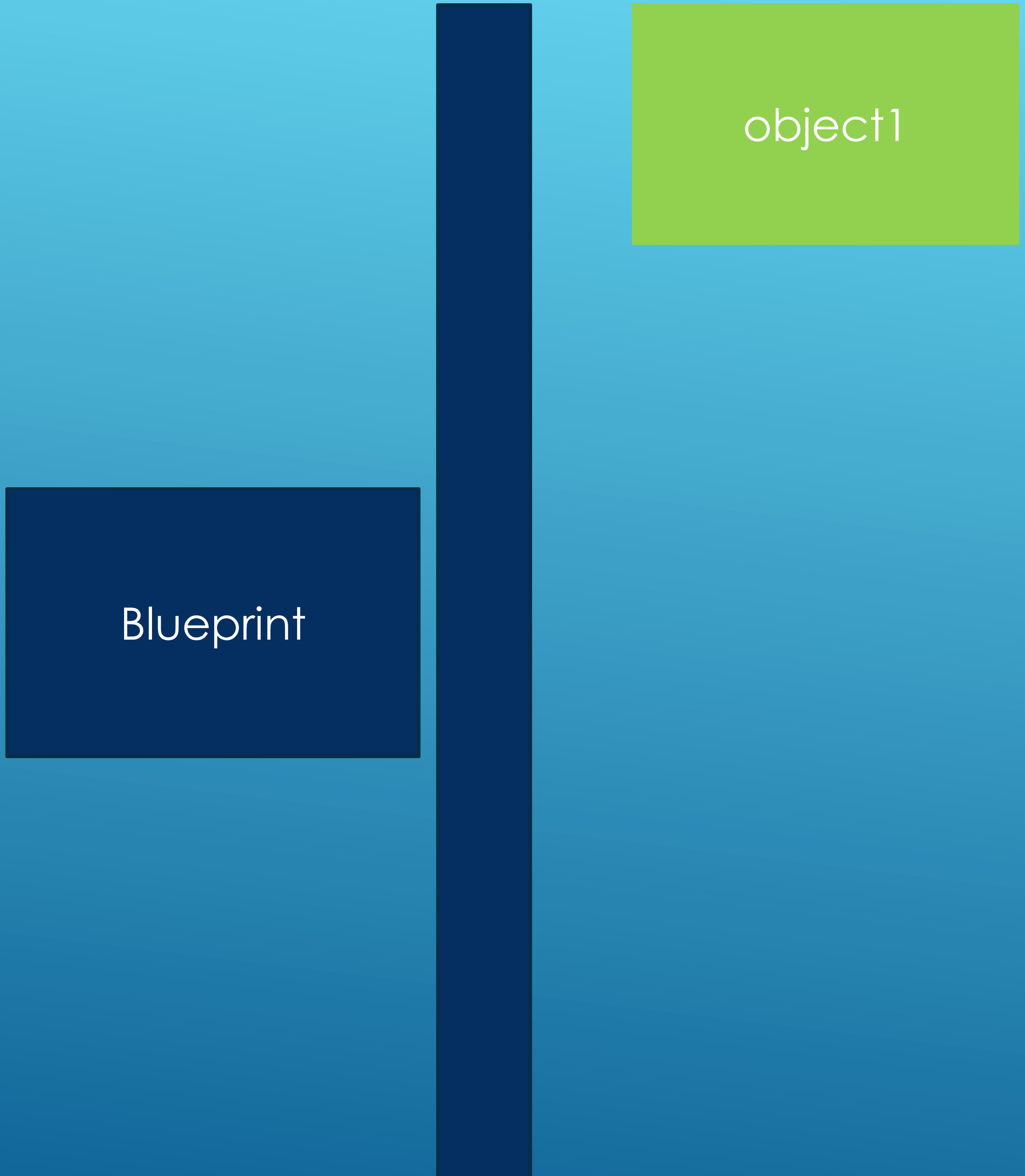
Classes : Summary



```cpp
unsigned int age{44};
double score{55.8};
```

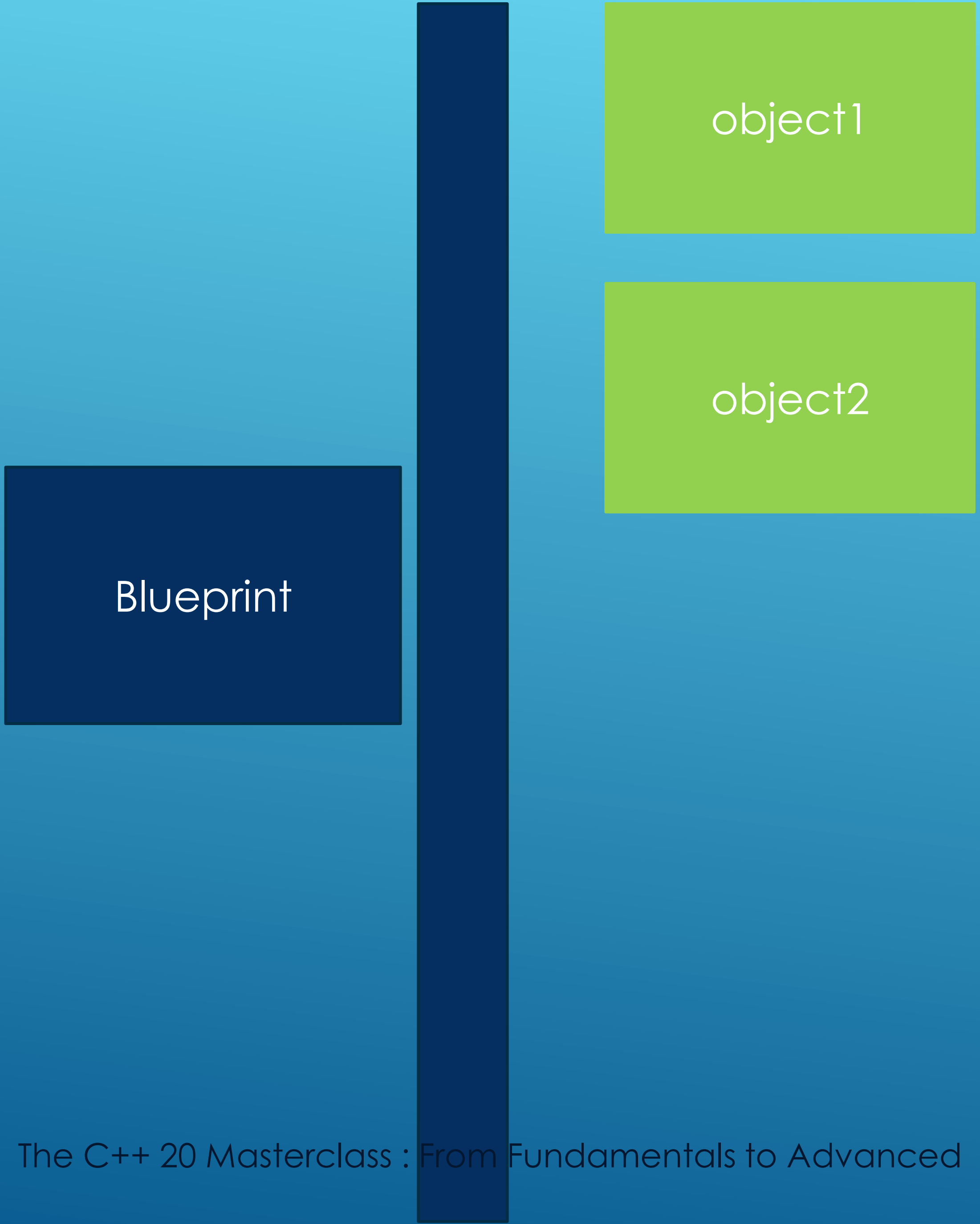

r
h

Blueprint



object1

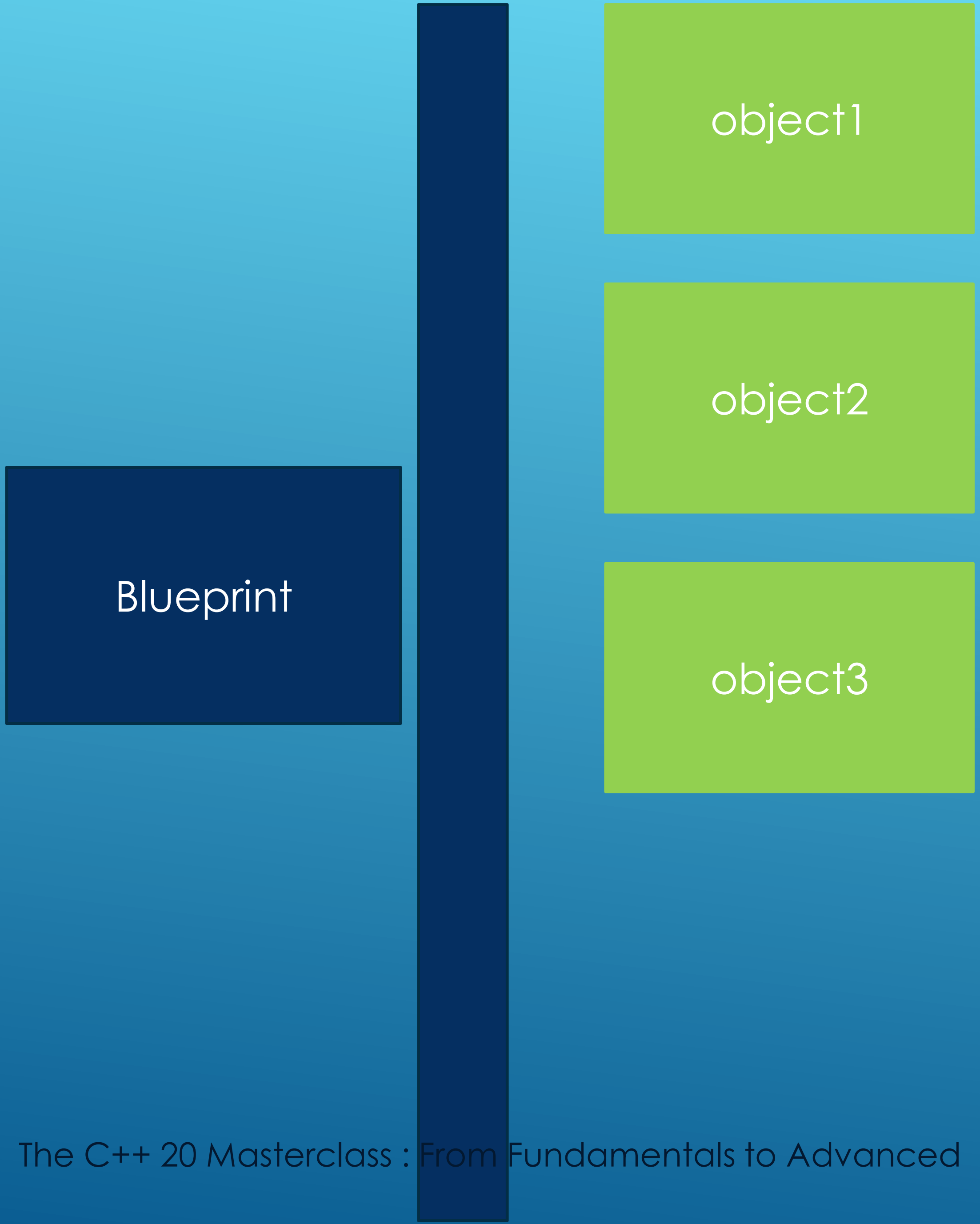
Blueprint



object1

object2

Blueprint



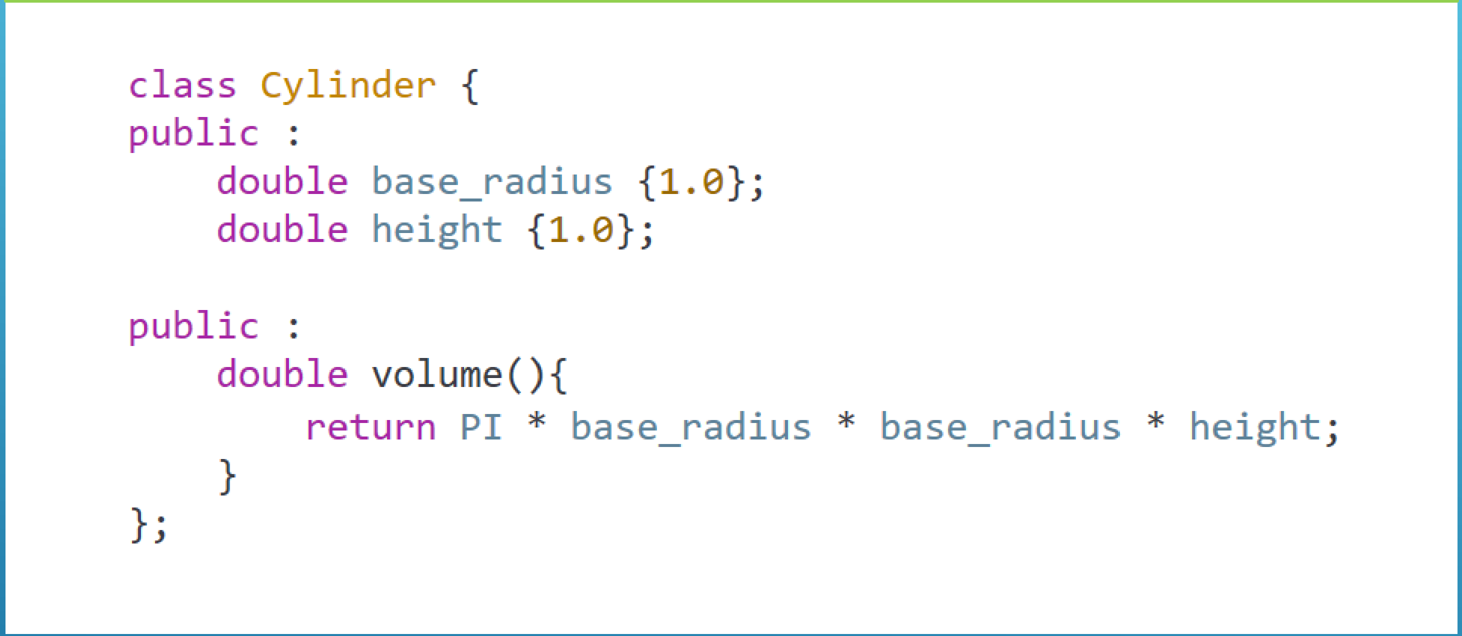
Blueprint

object1

object2

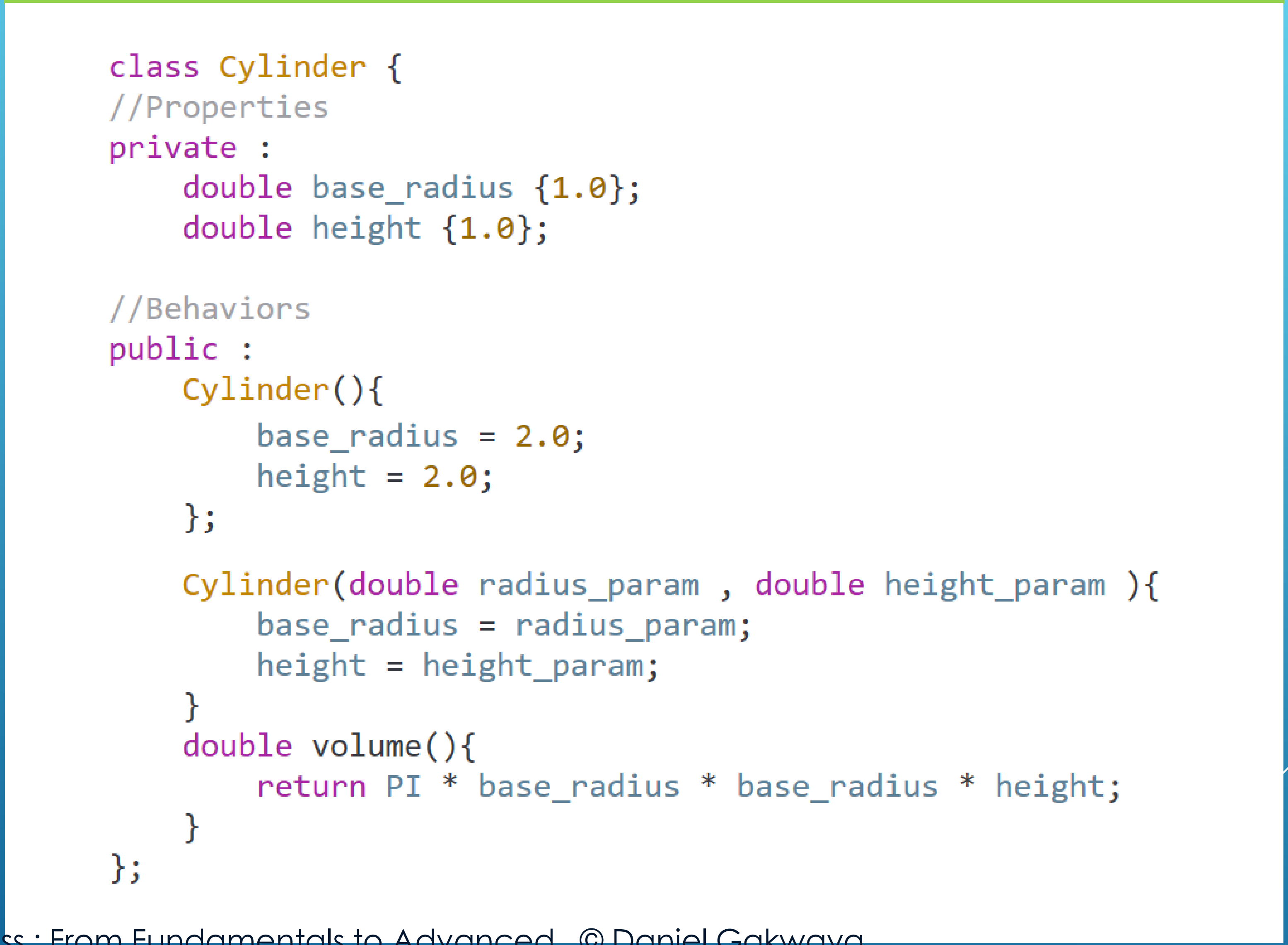
object3

Class declaration : syntax



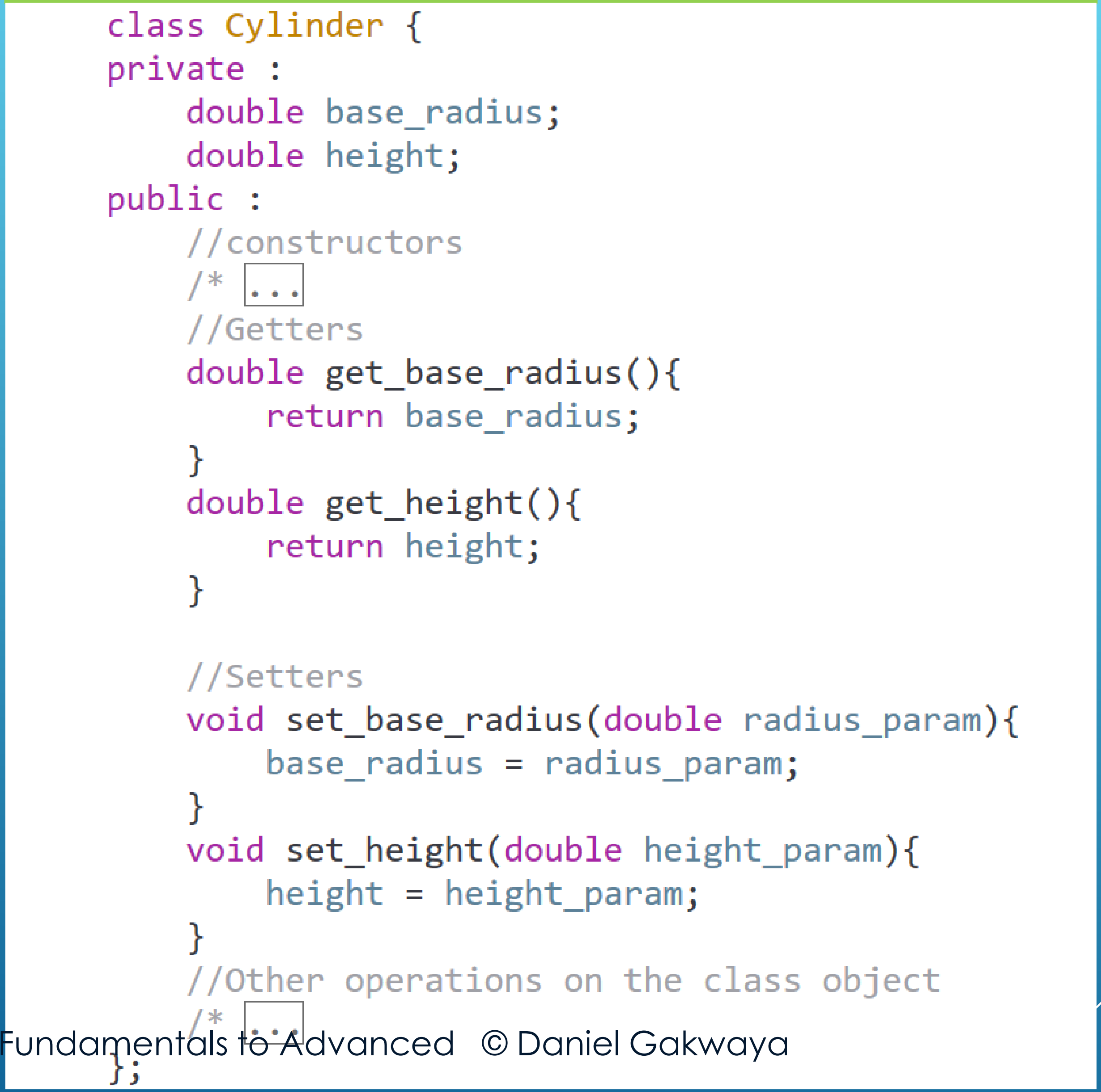
```
class Cylinder {
public :
    double base_radius {1.0};
    double height {1.0};

public :
    double volume(){
        return PI * base_radius * base_radius * height;
    }
};
```

Class constructors



```
class Cylinder {
//Properties
private :
    double base_radius {1.0};
    double height {1.0};

//Behaviors
public :
    Cylinder(){
        base_radius = 2.0;
        height = 2.0;
    };

    Cylinder(double radius_param , double height_param ){
        base_radius = radius_param;
        height = height_param;
    }
    double volume(){
        return PI * base_radius * base_radius * height;
    }
};
```

Getters & Setters

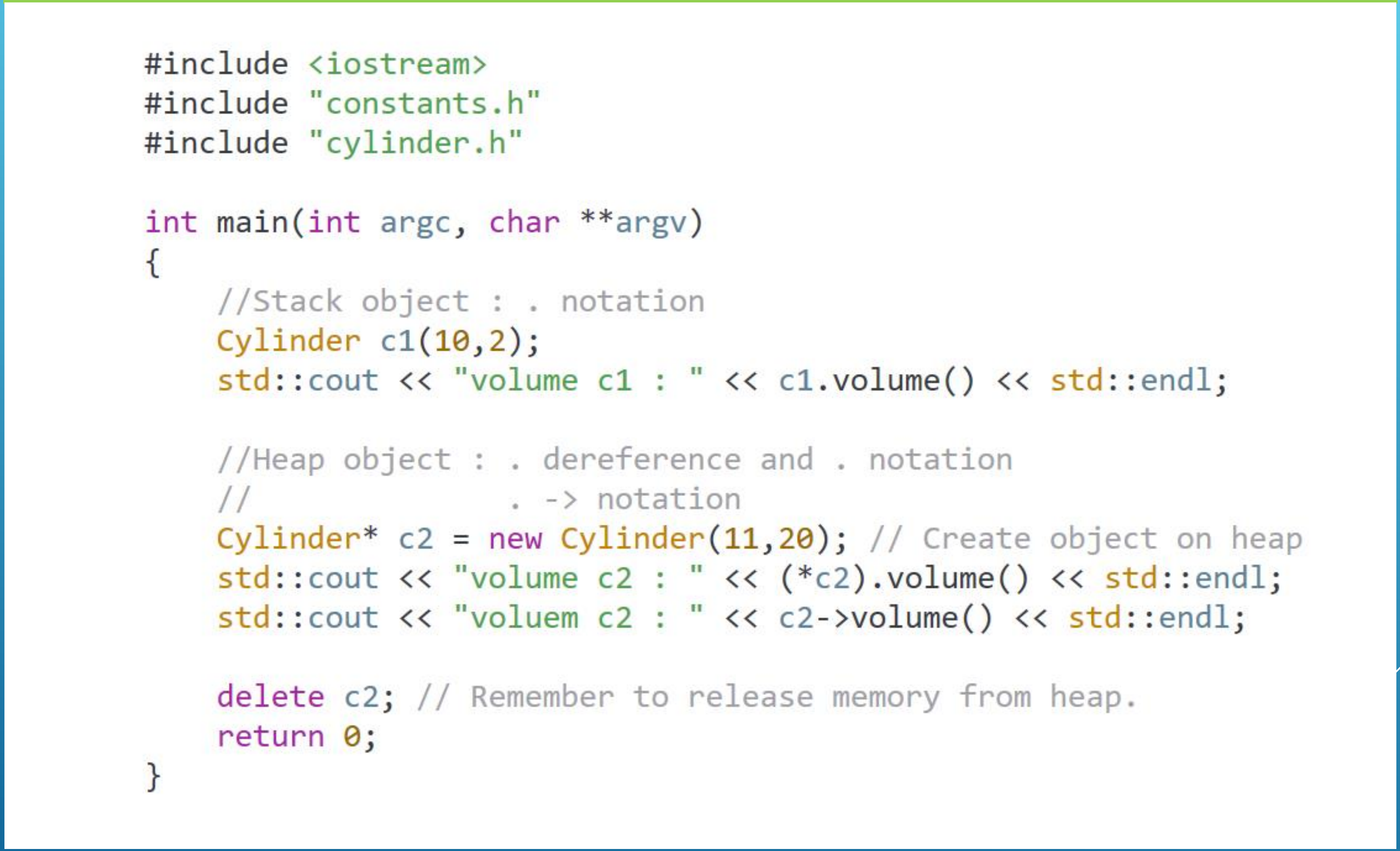


```
class Cylinder {
private :
    double base_radius;
    double height;
public :
    //constructors
    /* ...
    //Getters
    double get_base_radius(){
        return base_radius;
    }
    double get_height(){
        return height;
    }

    //Setters
    void set_base_radius(double radius_param){
        base_radius = radius_param;
    }
    void set_height(double height_param){
        height = height_param;
    }
    //Other operations on the class object
    /* ...
};
```

Class across multiple files

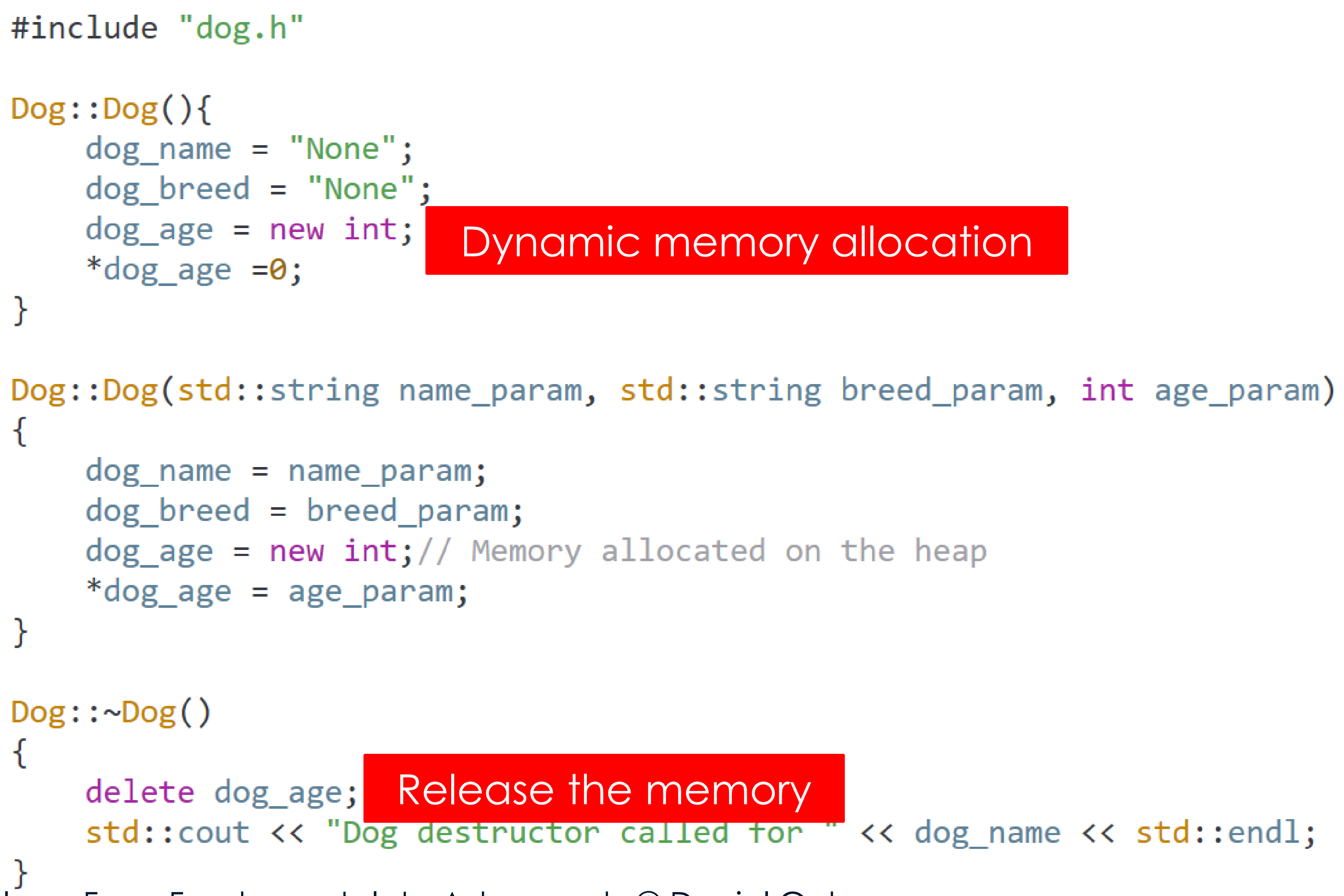
Creating classes through IDEs

Managing class objects though pointers



```
#include <iostream>
#include "constants.h"
#include "cylinder.h"

int main(int argc, char **argv)
{
    //Stack object : . notation
    Cylinder c1(10,2);
    std::cout << "volume c1 : " << c1.volume() << std::endl;

    //Heap object : . dereference and . notation
    //              . -> notation
    Cylinder* c2 = new Cylinder(11,20); // Create object on heap
    std::cout << "volume c2 : " << (*c2).volume() << std::endl;
    std::cout << "voluem c2 : " << c2->volume() << std::endl;

    delete c2; // Remember to release memory from heap.
    return 0;
}
```

Destructors



```
#include "dog.h"

Dog::Dog(){
    dog_name = "None";
    dog_breed = "None";
    dog_age = new int;
    *dog_age =0;
}

Dog::Dog(std::string name_param, std::string breed_param, int age_param)
{
    dog_name = name_param;
    dog_breed = breed_param;
    dog_age = new int;// Memory allocated on the heap
    *dog_age = age_param;
}

Dog::~Dog()
{
    delete dog_age;
    std::cout << "Dog destructor called for " << dog_name << std::endl;
}
```

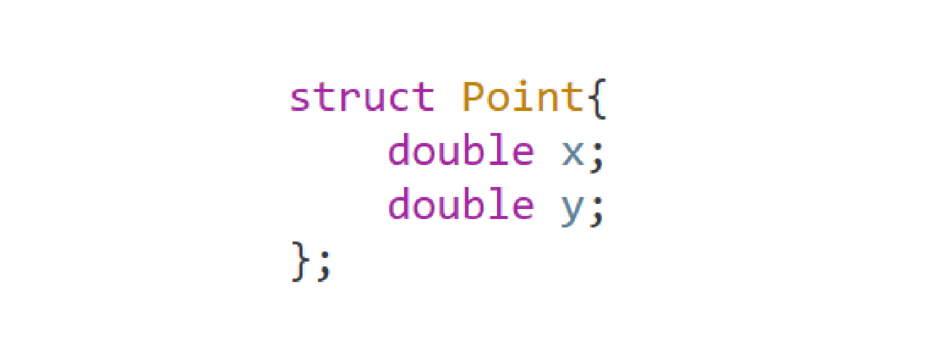
Dynamic memory allocation

Release the memory

this pointer

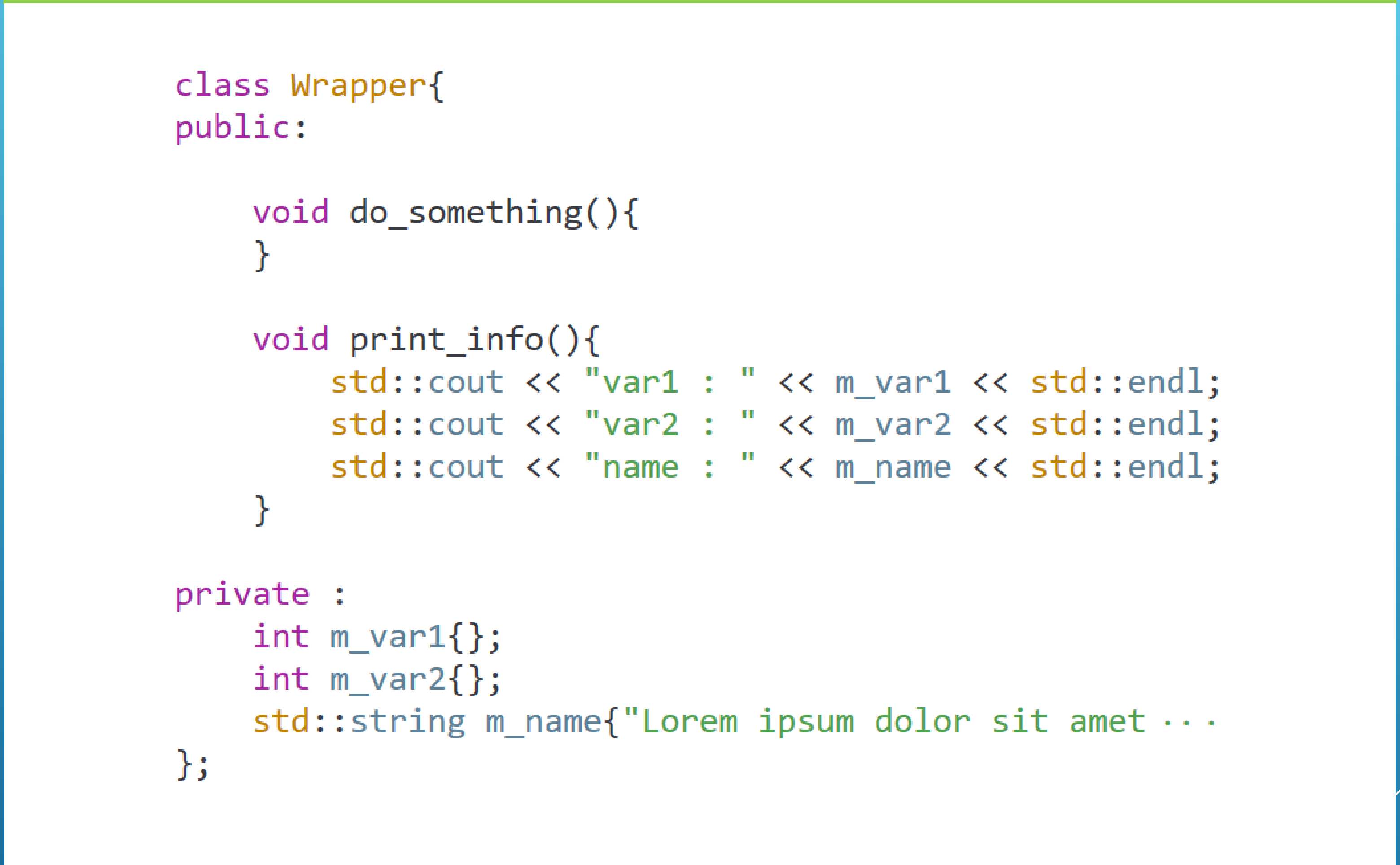
```cpp
Dog::Dog(){
    dog_name = "None";
    dog_breed = "None";
    dog_age = new int; // Memory allocated on the heap
    *dog_age =0;

    std::cout << "Dog : " << dog_name << " constructed at " << this << std::endl;
}

Dog::Dog(const std::string& name_param, const std::string& breed_param, int age_param)
{
    dog_name = name_param;
    dog_breed = breed_param;
    dog_age = new int;// Memory allocated on the heap
    *dog_age = age_param;

    std::cout << "Dog : " << dog_name << " constructed at " << this << std::endl;
}
```

struct



```
struct Point{
    double x;
    double y;
};
```

Size of class objects



```
class Wrapper{
public:

    void do_something(){
    }

    void print_info(){
        std::cout << "var1 : " << m_var1 << std::endl;
        std::cout << "var2 : " << m_var2 << std::endl;
        std::cout << "name : " << m_name << std::endl;
    }

private :
    int m_var1{};
    int m_var2{};
    std::string m_name{"Lorem ipsum dolor sit amet ···
};
```

Slide intentionally left empty