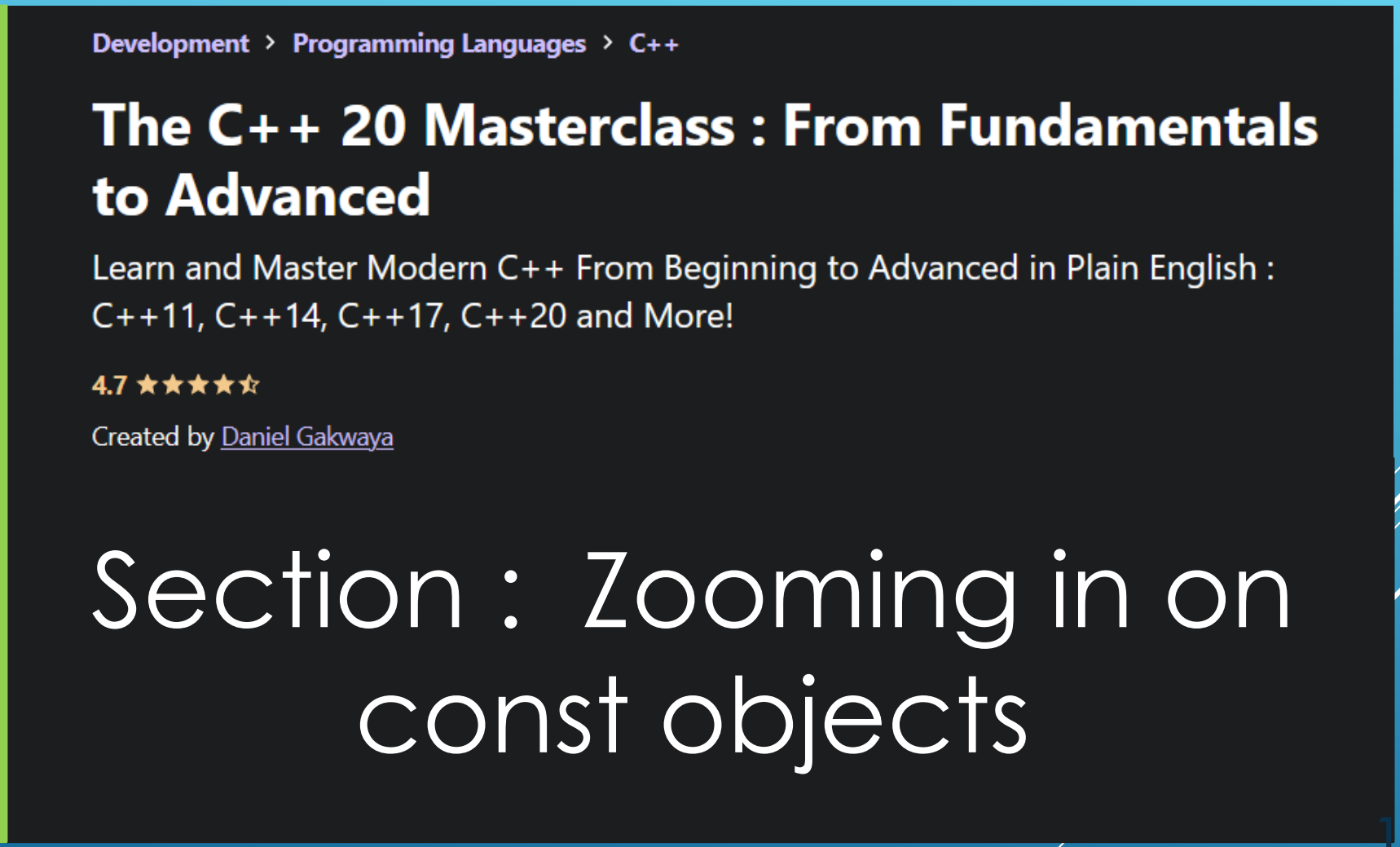
Slides



Section : Zooming in on const objects

Slide intentionally left empty

Objects and the const keyword : Introduction

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
    void set_name(const std::string& dog_name) ;
    void set_dog_breed(const std::string& breed) ;
    void set_dog_age(int age) ;
    std::string get_name() ;
    std::string get_breed() ;
    int get_age() ;
    void print_info() ;
private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
};
```

Declaring const object

```
const Dog dog1("Flatcher","Shepherd",3);
```

PROBLEM

We can't modify const objects. That's fine and it's what we want. But we can't read from them either. Which makes the kind of useless.

Slide intentionally left empty

const Objects

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
    void set_name(const std::string& dog_name) ;
    void set_dog_breed(const std::string& breed) ;
    void set_dog_age(int age) ;
    std::string get_name() ;
    std::string get_breed() ;
    int get_age() ;
    void print_info() ;
private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
};
```

Declaring const object

```
const Dog dog1("Flatcher","Shepherd",3);
```

Using const objects

```
const Dog dog1("Flatcher","Shepherd",3);

dog1.set_name("Milou");//Setting values on const object won't work

dog1.print_info();//Reading won't work either : Compiler error

std::string name = dog1.get_name(); // Compiler Error
```

PROBLEM

We can't modify const objects. That's fine and it's what we want. But we can't read from them either. Which makes them kind of useless. We'll see a solution to this in a few lectures ahead.

What if we try and go through pointers or references to a const object ?

Going through pointer to non const

```
const Dog dog1("Flatcher","Shepherd",3);
Dog* p_dog = &dog1; // Error : invalid conversion from const Dog* to Dog*
p_dog->set_name("Hillo");
p_dog->print_info();
```

Going through non const reference

```
const Dog dog1("Flatcher","Shepherd",3);
Dog& dog_ref = dog1; // Error : Can't convert from const Dog& to Dog&
dog_ref.set_name("Hillo");
dog_ref.print_info();
```

Going through pointer to const

```
const Dog dog1("Flatcher","Shepherd",3);
const Dog* p_const_dog = &dog1;
p_const_dog->set_name("Hillo"); // Error : expected
p_const_dog->print_info(); // Error : not expected
```

Going through const references

```
const Dog dog1("Flatcher","Shepherd",3);
const Dog& const_dog_ref = dog1;
const_dog_ref.set_name("Hillo"); //Error : Expected
const_dog_ref.print_info();      //Error : Not expected
```

Slide intentionally left empty

const Objects as function arguments

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
    void set_name(const std::string& dog_name) ;
    void set_dog_breed(const std::string& breed) ;
    void set_dog_age(int age) ;
    std::string get_name() ;
    std::string get_breed() ;
    int get_age() ;
    void print_info() ;
private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
};
```

Declaring const object

```
const Dog dog1("Flatcher","Shepherd",3);
```

Options

- Pass by value
- Pass by (non const) reference
- Pass by const reference
- Pass by pointer (to non const)
- Pass by pointer to const

Pass by value

```
void function_taking_dog(Dog dog){
    dog.set_name("Internal dog");
    dog.print_info();
}

int main(int argc, char **argv)
{
    const Dog dog1("Flatcher","Shepherd",3);
    //Dog by value
    function_taking_dog(dog1);

    return 0;
}
```

Pass by [non const] reference

```
void function_taking_dog_ref(Dog& dog_ref){
    //Compiler won't allow passing const object as argument
}

int main(int argc, char **argv)
{
    const Dog dog1("Flatcher","Shepherd",3);
    //Dog by ref : Compiler error. Could modify dog1 through non const ref
    function_taking_dog_ref(dog1);
    return 0;
}
```

Pass by const reference

```
void function_taking_const_dog_ref(const Dog& const_dog_ref){
    const_dog_ref.set_name("Hillo");
    const_dog_ref.print_info();
}

int main(int argc, char **argv)
{
    const Dog dog1("Flatcher","Shepherd",3);
    //Dog by const ref
    function_taking_const_dog_ref(dog1);

    return 0;
}
```

Pass by pointer [to non const]

```
void function_taking_dog_p(Dog* p_dog){
    //Compiler won't allow passing const Dog objects as arguments
}

int main(int argc, char **argv)
{
    const Dog dog1("Flatcher","Shepherd",3);
    //Dog by pointer : Compiler error : Could modify dog1 through pointer to non const
    function_taking_dog_p(&dog1);
    return 0;
}
```

Pass by pointer to const

```
void function_taking_pointer_to_const_dog(const Dog* const_p_dog){
    const_p_dog->set_name("Hillo");//Error : Expected
    const_p_dog.print_info(); //Error : Not expected
}


int main(int argc, char **argv)
{
    const Dog dog1("Flatcher","Shepherd",3);
    //Dog by const pointer
    function_taking_pointer_to_const_dog(&dog1);

    return 0;
}
```

Slide intentionally left empty

const Member functions

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
    void set_name(const std::string& dog_name) ;
    void set_dog_breed(const std::string& breed) ;
    void set_dog_age(int age) ;
    std::string get_name() ;
    std::string get_breed() ;
    int get_age() ;
    void print_info() ;
private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
};
```

Declaring const object

```
const Dog dog1("Flatcher","Shepherd",3);
```

PROBLEMS

- Can't read from const objects
- At this point they are just useless objects

Goals

```
std::cout << "const Dog object" << std::endl;
const Dog dog1("Fletcher","Shepherd",3);

std::cout << "name : " << dog1.get_name() << std::endl;
dog1.print_info();
//dog1.set_name("Snowy"); // Compiler error : Expected.


std::cout << std::endl;
std::cout << "non const Dog object" << std::endl;
Dog dog2("Soluk","Shepherd",2);
dog2.set_name("Haluk");
std::cout << "dog2.get_name() : " << dog2.get_name() << std::endl;
dog2.print_info();
```

const methods

```
std::string get_name() const {
    return dog_name;
}

std::string get_breed() const {
    return dog_breed;
}

int get_age() const;     // Can split code into declaration
                         // and implementation.
```

Outside implementation

```
int Dog::get_age() const
{
    return *dog_age; // Dereference and return
}
```

non const overloads for methods

```
void print_info() const{

    std::cout << "print_info const version called" << std::endl;
    std::cout << "Dog name : " << dog_name <<
            ", dog breed : " << dog_breed << " , dog age : " << *dog_age << std::endl;
    //++info_print_count; //Error
}

void print_info(){
    std::cout << "print_info non const version called" << std::endl;
    std::cout << "Dog name : " << dog_name <<
            ", dog breed : " << dog_breed << " , dog age : " << *dog_age << std::endl;

    ++info_print_count;
}
```

Slide intentionally left empty

Getters that double as setters

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
     std::string& get_name() /*const*/  { //The const is removed because we want this
                                          // method to be used to modify the object
                                          // through the returned reference
        return dog_name;
    }
     std::string& get_breed() /*const*/ {
        return dog_breed;
    }
    int& get_age(){
        return dog_age;
    }
private :
    std::string dog_name;
    std::string dog_breed;
    int dog_age;
};
```

```
Dog dog1("Fluffy","Shepherd",3);

dog1.print_info();

std::cout << "dog name : " << dog1.get_name() << std::endl;

std::cout << "Changing the name of the dog" << std::endl;
dog1.get_name() = "Dandio";
dog1.print_info();

dog1.get_age() = 5;
dog1.print_info();
```

Slide intentionally left empty

Dangling pointers and references

A pointer or reference is said to be dangling if it's pointing to or referencing invalid data. A simple example for pointers is a pointer pointing to a deleted piece of memory.


pointer
object


pointer
delete
object


pointer
object


pointer
nullptr

Dangling reference

```
const std::string& Dog::compile_dog_info() const{
    std::string info = "Dog name : " + dog_name
                     + " dog breed : " + dog_breed
                     + "dog age : " + std::to_string(dog_age);
    return info;
}
```

Dangling pointer

```
int * Dog::return_int_pointer() const{
    int jumps_per_minute {50};
    return &jumps_per_minute;
}
```

Slide intentionally left empty

Zooming in on const

main function

```
const Dog dog1("Fletcher","Shepherd",3);
```

const methods

```
std::string get_breed() const {
    return dog_breed;
}
```

Const correctness

- For const objects you can only call const member functions
- const objects are completely non-modifiable(immutable), the compiler won't allow calling a member function that modifies the const object in any way
- We are not allowed to modify the object in any way inside const member functions
- Just as we're not allowed to directly modify the object inside a const member function, we're not allowed to call a method that modifies the object indirectly either
- Any attempt to modify an object's member variable (direct or indirect) from within a const member functions will result in a compiler error
- You cannot call any non-const member functions from within a const member function

Slide intentionally left empty

Mutable objects

```
class Dog
{
public:
    Dog();
    /* ...

    void print_info() const{
        ++info_print_count;
        std::cout << "Dog name : " << dog_name <<
                ", dog breed : " << dog_breed << " , dog age : " << *dog_age
                    << "print_count : " << info_print_count << std::endl;
    }
    /* ...
private :

    /* ...

    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
    size_t info_print_count{0};
};
```

Mutable member vars

```
class Dog
{
public:
    Dog();
    /* ...

    void print_info() const{
        ++info_print_count;
        std::cout << "Dog name : " << dog_name <<
                ", dog breed : " << dog_breed << " , dog age : " << *dog_age
                    << "print_count : " << info_print_count << std::endl;
    }
    /* ...
private :

    /* ...

    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
    mutable size_t info_print_count{0};
};
```

Slide intentionally left empty

Structured bindings

```cpp
struct Point{
    double x;
    double y;
};
int main(int argc, char **argv)
{
    Point point1 {4.4,5.9};

    auto [a,b] = point1;

    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;

    //a and b are just copies
    point1.x = 10.1;
    point1.y = 66.2;

    std::cout << std::endl;
    std::cout << "point1 has changed : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;
    return 0;
}
```

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

int main(int argc, char **argv)
{
    Point point1 {4.4,5.9};

    auto [a,b] = point1;

    //Capturing a structured binding in a lambda function
    //This was only possible in C++20.
    auto f = [a](){
        std::cout << "Have captured : " << a << std::endl;
    };
    f();

    return 0;
}
```

Slide intentionally left empty

Objects and the const keyword : Summary

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
    void set_name(const std::string& dog_name) ;
    void set_dog_breed(const std::string& breed) ;
    void set_dog_age(int age) ;
    std::string get_name() ;
    std::string get_breed() ;
    int get_age() ;
    void print_info() ;
private :
    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
};
```

Declaring const object

```
const Dog dog1("Flatcher","Shepherd",3);
```

PROBLEM

We can't modify const objects. That's fine and it's what we want. But we can't read from them either. Which makes the kind of useless.

const member functions

```
std::string get_name() const {
    return dog_name;
}

std::string get_breed() const {
    return dog_breed;
}

int get_age() const;     // Can split code into declaration
                         // and implementation.
```

Getters that double as setters

```
class Dog
{
public:
    Dog();
    Dog(const std::string& name_param,const std::string& breed_param, int age_param);
    ~Dog();
     std::string& get_name() /*const*/  { //The const is removed because we want this
                                          // method to be used to modify the object
                                          // through the returned reference
        return dog_name;
    }
     std::string& get_breed() /*const*/ {
        return dog_breed;
    }
    int& get_age(){
        return dog_age;
    }
private :
    std::string dog_name;
    std::string dog_breed;
    int dog_age;
};
```

Dangling reference

```
const std::string& Dog::compile_dog_info() const{
    std::string info = "Dog name : " + dog_name
                     + " dog breed : " + dog_breed
                     + "dog age : " + std::to_string(dog_age);
    return info;
}
```

Dangling pointer

```
int * Dog::return_int_pointer() const{
    int jumps_per_minute {50};
    return &jumps_per_minute;
}
```

Const correctness

- For const objects you can only call const member functions
- const objects are completely non-modifiable(immutable), the compiler won't allow calling a member function that modifies the const object in any way
- We are not allowed to modify the object in any way inside const member functions
- Just as we're not allowed to directly modify the object inside a const member function, we're not allowed to call a method that modifies the object indirectly either
- Any attempt to modify an object's member variable (direct or indirect) from within a const member functions will result in a compiler error
- You cannot call any non-const member functions from within a const member function

Mutable member vars

```
class Dog
{
public:
    Dog();
    /* ...

    void print_info() const{
        ++info_print_count;
        std::cout << "Dog name : " << dog_name <<
                ", dog breed : " << dog_breed << " , dog age : " << *dog_age
                    << "print_count : " << info_print_count << std::endl;
    }
    /* ...
private :

    /* ...

    std::string dog_name;
    std::string dog_breed;
    int * dog_age;
    mutable size_t info_print_count{0};
};
```

Structured bindings

```
struct Point{
    double x;
    double y;
};
int main(int argc, char **argv)
{
    Point point1 {4.4,5.9};

    auto [a,b] = point1;

    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;

    //a and b are just copies
    point1.x = 10.1;
    point1.y = 66.2;

    std::cout << std::endl;
    std::cout << "point1 has changed : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b << std::endl;
    return 0;
}
```

Slide intentionally left empty