

# Modern C++ Course



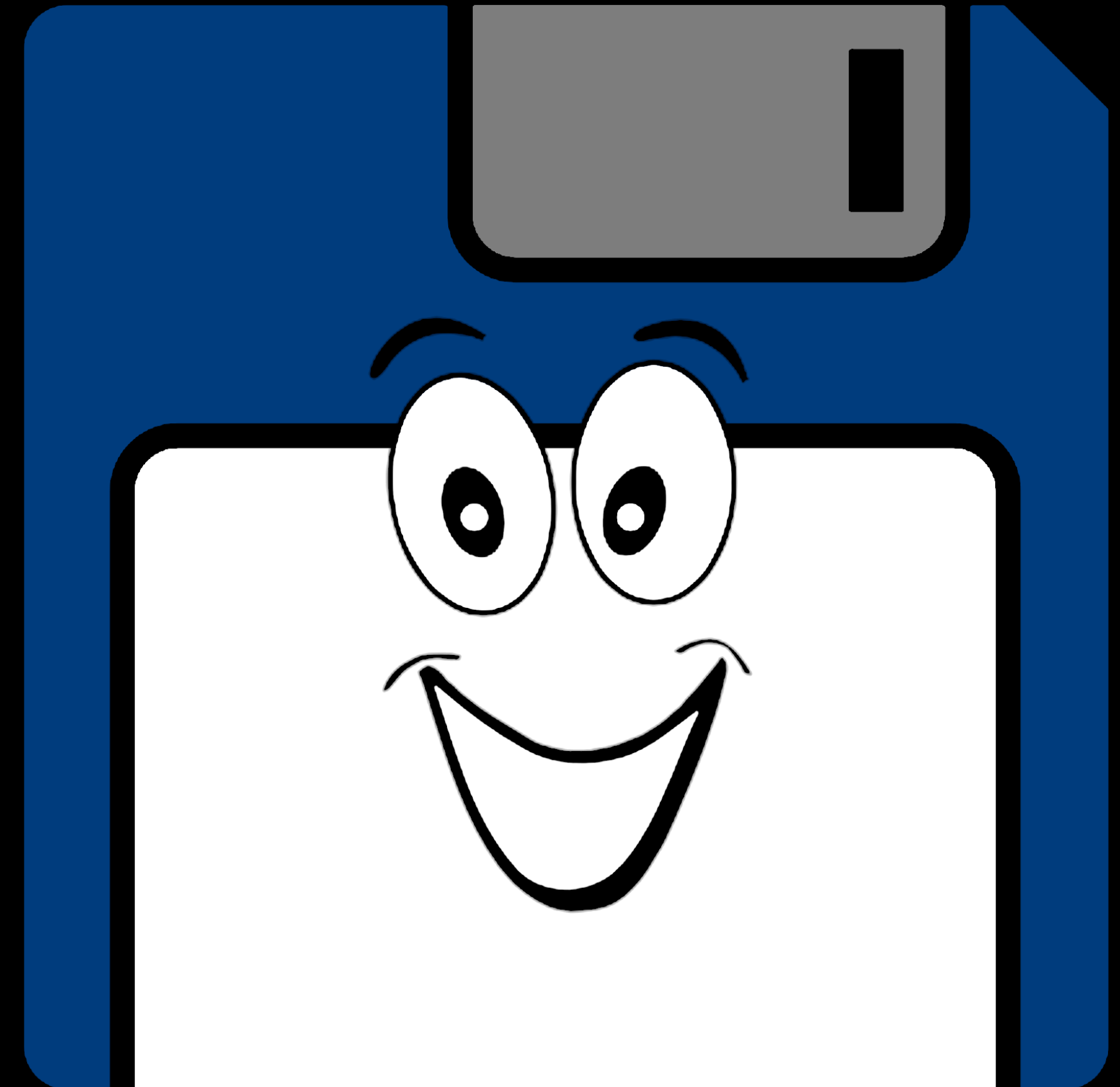
# Who am I ?

## Gammasoft

Gammasoft aims to make c++ fun again.

## About

- Gammasoft is the nickname of Yves Fiumefreddo.
- More than thirty years of passion for high technology especially in development (c++, c#, objective-c, ...).
- Object-oriented programming is more than a mindset.
- more info see my GitHub : <https://github.com/gammasoft71>



# Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



# Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



# Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



# Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Classes (or “user-defined types”)

- structs on steroids
  - with inheritance
  - with access control
  - including methods (aka. member functions)



# Objects

- instances of classes



# A class encapsulates state and behavior of “something”

- shows an interface
- provides its implementation
  - status, properties
  - possible interactions
  - construction and destruction



# My first class



```
1 struct my_first_class {  
2     int a;  
3  
4     void square_a() {  
5         a *= a;  
6     }  
7  
8     int sum(int b) {  
9         return a + b;  
10    }  
11 };  
12  
13 auto my_obj = my_first_class {};  
14 my_obj.a = 2;  
15  
16 // let's square a  
17 my_obj.square_a();
```

my_first_class
+ a: int
+ square_a(): void + sum(int): int



# Separating the interface

## Header: my\_class.hpp

```
1 #pragma once
2
3 struct my_class {
4     int a;
5
6     void square_a();
7 };
```

## User 1: main.cpp

```
1 #include <iostream>
2
3 int main() {
4     auto mc = my_class {};
5     //...
6 }
```

## Implementation: my\_class.cpp

```
1 #include "my_class.hpp"
2
3 void my_class::square_a() {
4     a *= a;
5 }
```

## User 2: fun.cpp

```
1 #include "my_class.hpp"
2
3 void fun(my_class& mc) {
4     mc.square_a();
5 }
```



# Implementing methods

## Good practice

- usually in .cpp, outside of class declaration
- using the class name as “namespace”
- short member functions can be in the header
- some functions (templates, constexpr) must be in the header

```
1 #include "my_first_class.hpp"
2
3 void my_first_class::square_a() {
4     a *= a;
5 }
6
7 int my_first_class::sum(int b) {
8     return a + b;
9 }
```



# Method overloading

## The rules in C++

- overloading is authorized and welcome
- signature is part of the method identity
- but not the return type

```
1 struct my_first_class {  
2     int a;  
3  
4     int sum(int b);  
5     int sum(int b, int c);  
6 };  
7  
8 int my_first_class::sum(int b) {  
9     return a + b;  
10 }  
11  
12 int my_first_class::sum(int b, int c) {  
13     return a + b + c;  
14 }
```



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





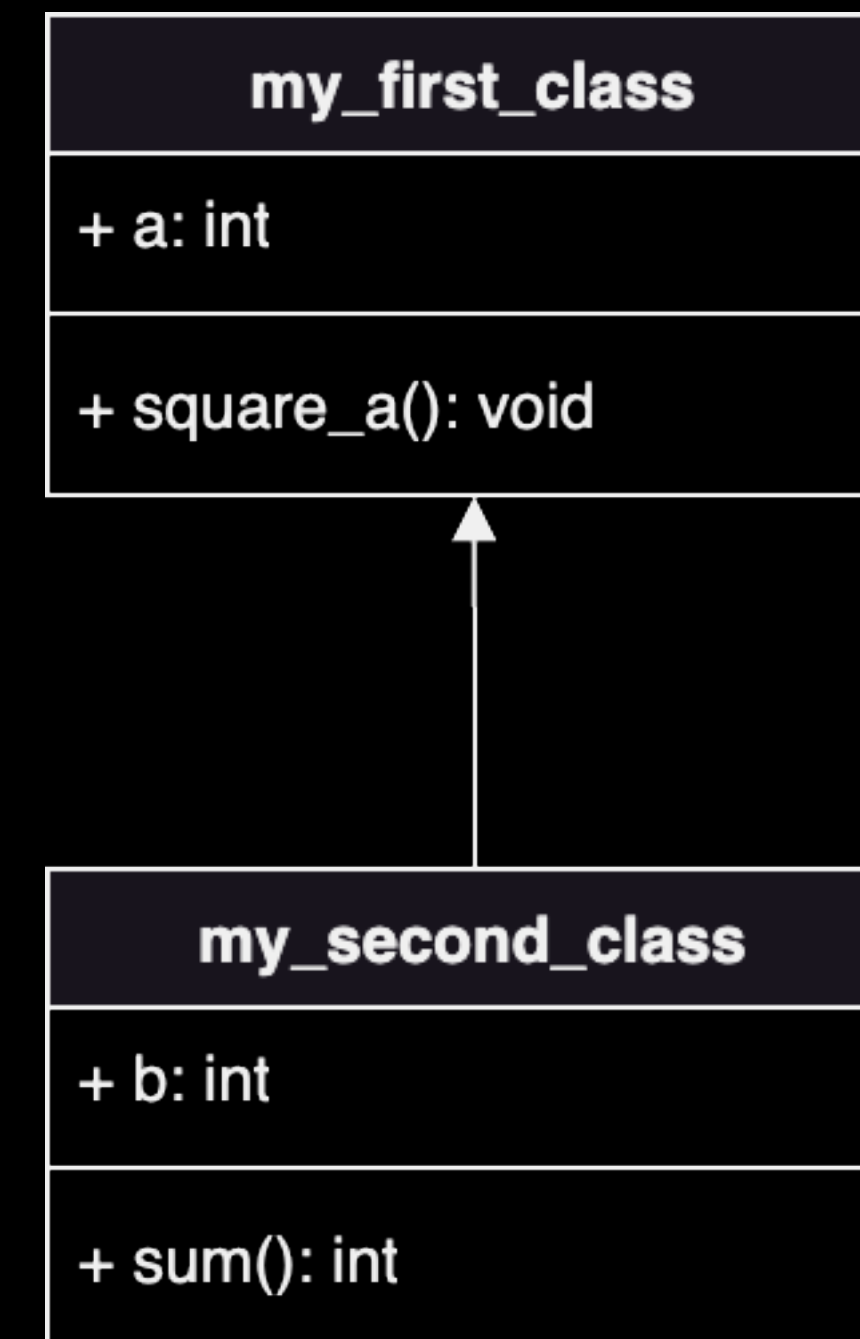
# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# First inheritance

```
1 struct my_first_class {  
2     int a;  
3  
4     void square_a() {  
5         a *= a;  
6     }  
7 };  
8  
9 struct my_second_class : my_first_class {  
10     int b;  
11  
12     int sum() {  
13         return a + b;  
14     }  
15 };  
16  
17 auto my_obj2 = my_second_class {};  
18 my_obj2.a = 2;  
19 my_obj2.b = 5;  
20  
21 my_obj2.square_a();  
22 auto i = my_obj2.sum(); // i = 9
```



# First inheritance

```
1 struct my_first_class {  
2     int a;  
3  
4     void square_a() {  
5         a *= a;  
6     }  
7 };  
8  
9 struct my_second_class : my_first_class {  
10     int b;  
11  
12     int sum() {  
13         return a + b;  
14     }  
15 };  
16  
17 auto my_obj2 = my_second_class {};  
18 my_obj2.a = 2;  
19 my_obj2.b = 5;  
20  
21 my_obj2.square_a();  
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

	0x3010
	0x300C
	0x3008
	0x3004
	0x3000



# First inheritance

```
1 struct my_first_class {
2     int a;
3
4     void square_a() {
5         a *= a;
6     }
7 };
8
9 struct my_second_class : my_first_class {
10     int b;
11
12     int sum() {
13         return a + b;
14     }
15 };
16
17 auto my_obj2 = my_second_class {};
18 my_obj2.a = 2;
19 my_obj2.b = 5;
20
21 my_obj2.square_a();
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

my_obj2 {		0x3010
		0x300C
		0x3008
	b = ?	0x3004
	a = ?	0x3000



# First inheritance

```
1 struct my_first_class {
2     int a;
3
4     void square_a() {
5         a *= a;
6     }
7 };
8
9 struct my_second_class : my_first_class {
10     int b;
11
12     int sum() {
13         return a + b;
14     }
15 };
16
17 auto my_obj2 = my_second_class {};
18 my_obj2.a = 2;
19 my_obj2.b = 5;
20
21 my_obj2.square_a();
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

my_obj2 {		0x3010
		0x300C
		0x3008
	b = ?	0x3004
	a = 2	0x3000



# First inheritance

```
1 struct my_first_class {  
2     int a;  
3  
4     void square_a() {  
5         a *= a;  
6     }  
7 };  
8  
9 struct my_second_class : my_first_class {  
10     int b;  
11  
12     int sum() {  
13         return a + b;  
14     }  
15 };  
16  
17 auto my_obj2 = my_second_class {};  
18 my_obj2.a = 2;  
19 my_obj2.b = 5;  
20  
21 my_obj2.square_a();  
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

my_obj2 {		0x3010
		0x300C
		0x3008
	b = 5	0x3004
	a = 2	0x3000



# First inheritance

```
1 struct my_first_class {
2     int a;
3
4     void square_a() {
5         a *= a;
6     }
7 };
8
9 struct my_second_class : my_first_class {
10     int b;
11
12     int sum() {
13         return a + b;
14     }
15 };
16
17 auto my_obj2 = my_second_class {};
18 my_obj2.a = 2;
19 my_obj2.b = 5;
20
21 my_obj2.square_a();
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

my_obj2 {		0x3010
		0x300C
		0x3008
	b = 5	0x3004
	a = 4	0x3000





# First inheritance

```
1 struct my_first_class {
2     int a;
3
4     void square_a() {
5         a *= a;
6     }
7 };
8
9 struct my_second_class : my_first_class {
10     int b;
11
12     int sum() {
13         return a + b;
14     }
15 };
16
17 auto my_obj2 = my_second_class {};
18 my_obj2.a = 2;
19 my_obj2.b = 5;
20
21 my_obj2.square_a();
22 auto i = my_obj2.sum(); // i = 9
```

## Memory layout

my_obj2 {		0x3010
		0x300C
	i = 9	0x3008
	b = 5	0x3004
	a = 4	0x3000





# Managing access to class members

public / private keywords

- **private** allows access only within the class
- **public** allows access from anywhere
- The default for class is **private**
- The default for struct is **public**

```
1 class my_first_class {
2 public:
3     int get_a();
4     void set_a(int value);
5
6     void square_a();
7
8 private:
9     int a;
10 };
11
12 auto obj = my_first_class {};
13 obj.a = 5;      // error !
14 obj.set_a(5);  // ok
15 obj.square_a();
16 int r = obj.get_a();
```



# Managing access to class members

public / private keywords

- **private** allows access only within the class
- **public** allows access from anywhere
- The default for class is **private**
- The default for struct is **public**

**This break my\_second\_class !**

```
1 class my_first_class {
2 public:
3     int get_a();
4     void set_a(int value);
5
6     void square_a();
7
8 private:
9     int a;
10 };
11
12 auto obj = my_first_class {};
13 obj.a = 5;      // error !
14 obj.set_a(5);  // ok
15 obj.square_a();
16 int r = obj.get_a();
```



# a is not accessible in the sum function



```
1 class my_first_class {  
2 public:  
3     int get_a();  
4     void set_a(int value);  
5  
6     void square_a();  
7  
8 private:  
9     int a;  
10 };
```



```
1 class my_second_class : public my_first_class {  
2 public:  
3     int get_b();  
4     void set_b(int value);  
5  
6     int sum() {  
7         return a + b; // error !  
8     }  
9  
10 private:  
11     int b;  
12 };
```



# Solution is protected keyword



```
1 class my_first_class {
2 public:
3     int get_a();
4     void set_a(int value);
5
6     void square_a();
7
8 protected:
9     int a;
10 };
```



```
1 class my_second_class : public my_first_class {
2 public:
3     int get_b();
4     void set_b(int value);
5
6     int sum() {
7         return a + b;
8     }
9
10 private:
11     int b;
12 };
```



# Inheritance can be public, protected or private

It influences the privacy of inherited members for external code.

The code of the class itself is not affected

- **public** privacy of inherited members remains unchanged
- **protected** inherited public members are seen as protected
- **private** all inherited members are seen as private. This is the default for classes if nothing is specified



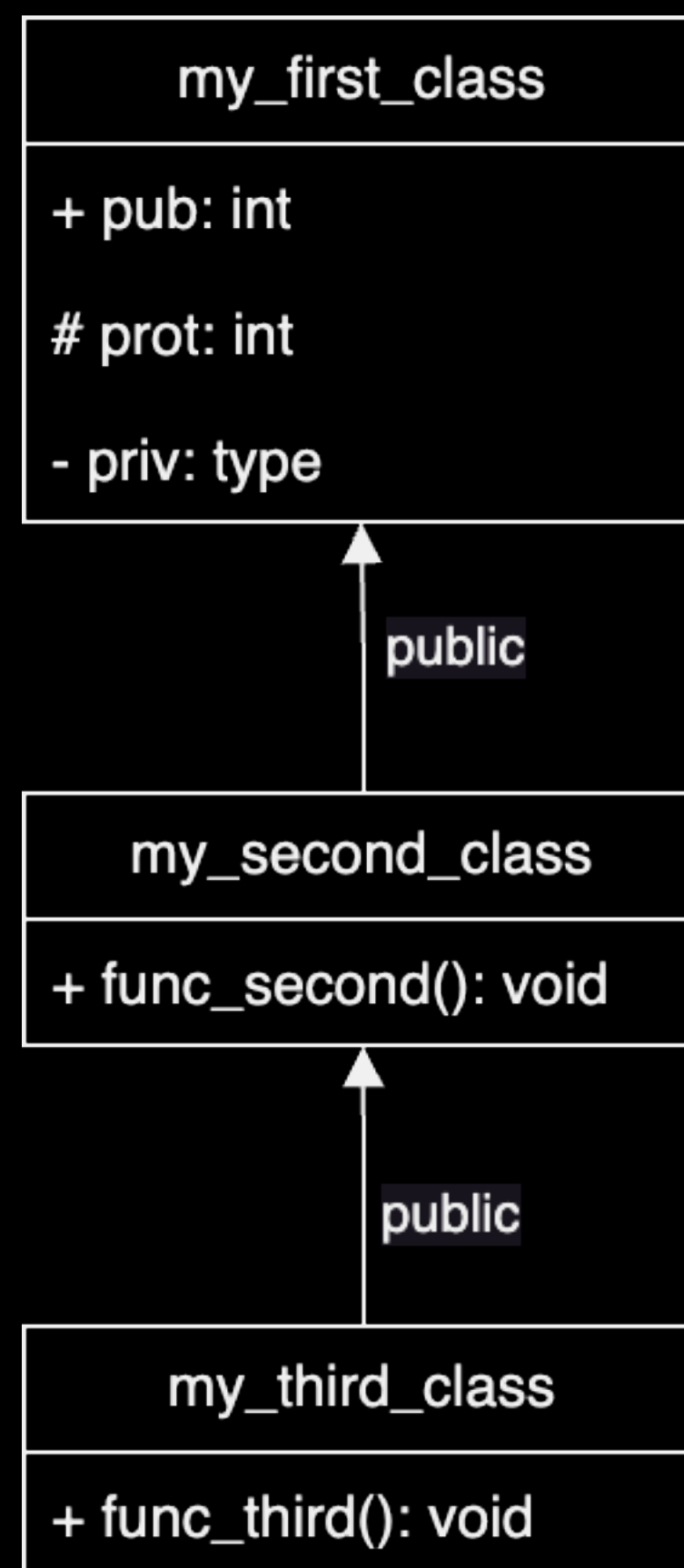
# Inheritance can be public, protected or private

- Net result for external code
  - only public members of public inheritance are accessible
- Net result for code in derived classes
  - only public and protected members of public and protected
  - parents are accessible



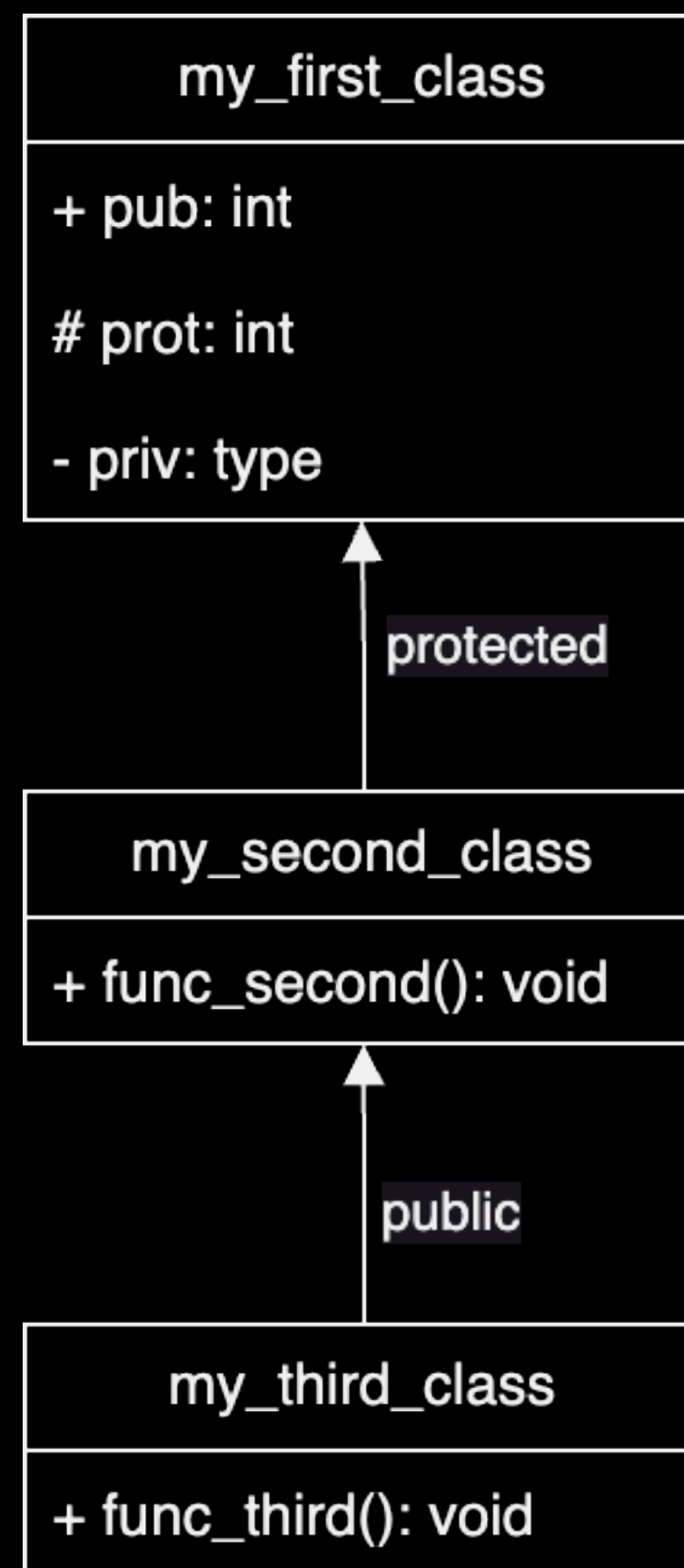


# Managing inheritance privacy - public



```
1 void my_second_class::func_second() {
2     int a = pub; // ok
3     int b = prot; // ok
4     int c = priv; // error
5 }
6
7 void my_third_class::func_third() {
8     int a = pub; // ok
9     int b = prot; // ok
10    int c = priv; // error
11 }
12
13 void ext_func(my_third_class t) {
14     int a = t.pub; // ok
15     int b = t.prot; // error
16     int c = t.priv; // error
17 }
```

# Managing inheritance privacy - protected

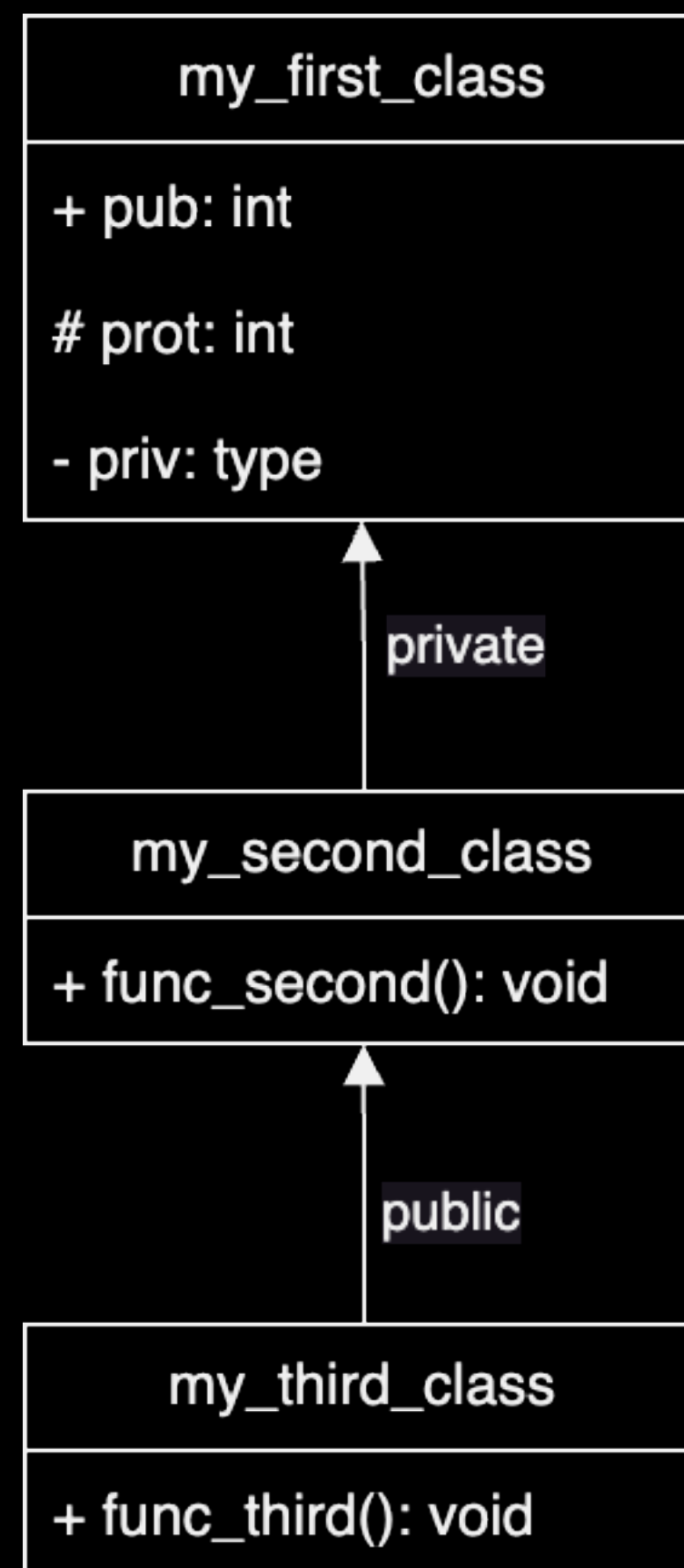


```
1 void my_second_class::func_second() {
2     int a = pub; // ok
3     int b = prot; // ok
4     int c = priv; // error
5 }
6
7 void my_third_class::func_third() {
8     int a = pub; // ok
9     int b = prot; // ok
10    int c = priv; // error
11 }
12
13 void ext_func(my_third_class t) {
14     int a = t.pub; // error
15     int b = t.prot; // error
16     int c = t.priv; // error
17 }
```





# Managing inheritance privacy - private



```
1 void my_second_class::func_second() {
2     int a = pub; // ok
3     int b = prot; // ok
4     int c = priv; // error
5 }
6
7 void my_third_class::func_third() {
8     int a = pub; // error
9     int b = prot; // error
10    int c = priv; // error
11 }
12
13 void ext_func(my_third_class t) {
14     int a = t.pub; // error
15     int b = t.prot; // error
16     int c = t.priv; // error
17 }
```



# Final class

## Idea

- make sure you cannot inherit from a given class
- by declaring it final

```
1 struct base final {  
2     // ...  
3 };  
4  
5 struct derived : base { // compiler error  
6     // ...  
7 };
```



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups







# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups







# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
  - Inheritance
  - Constructors / Destructors
  - Static members
  - Allocating objects
  - Advanced Object Oriented
  - Type casing
- Operator overloading
  - Function objects
  - Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups





# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Objects Oriented Programming (OOP)

- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups



# Outline

1. Introduction
2. Language Basics
3. Object Oriented Programming (OOP)
4. Core Modern C++
5. Modern C++ Expert
6. Advanced Programming



# End

