Modern C++ Course

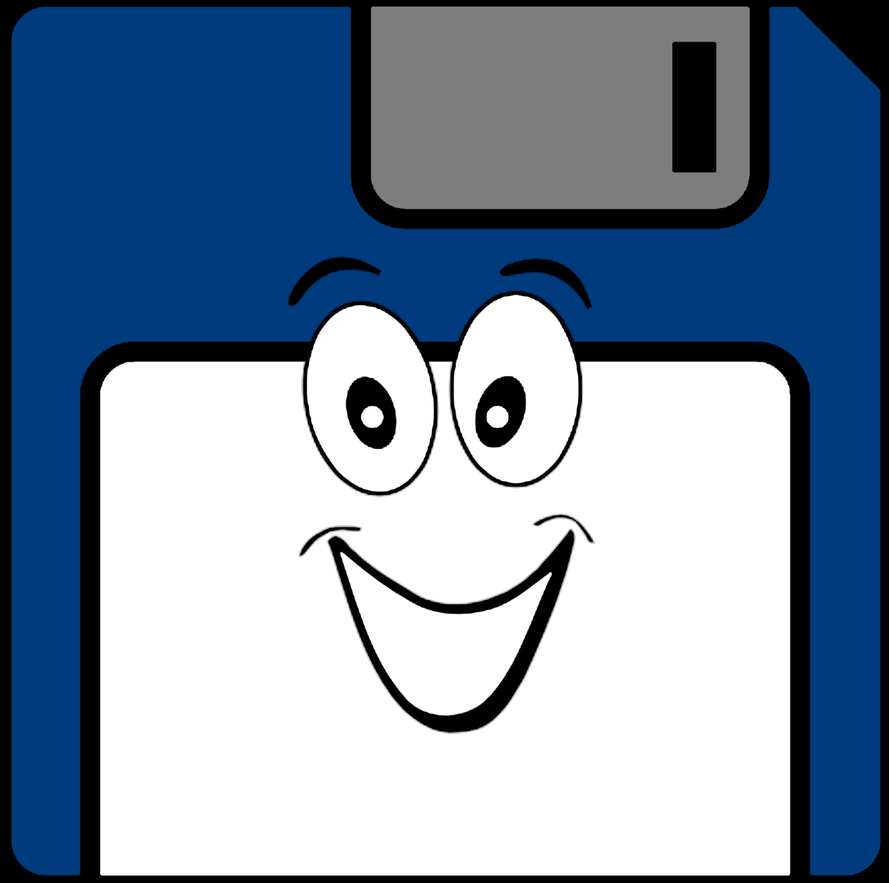
Who am I ?

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

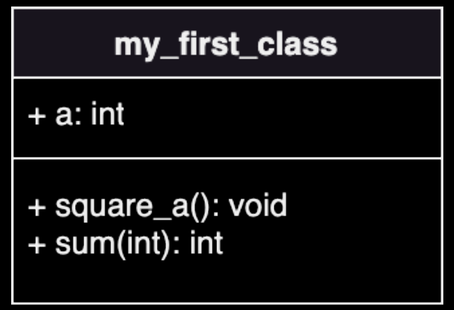
- instances of classes

A class encapsulates state and behavior of “something”

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

My first class

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }

  int sum(int b) {
    return a + b;
  }
};

auto my_obj = my_first_class {};
my_obj.a = 2;

// let's square a
my_obj.square_a();
```



Separating the interface

Header: my_class.hpp

```cpp
#pragma once

struct my_class {
  int a;

  void square_a();
};
```

User 1: main.cpp

```cpp
#include <iostream>

int main() {
  auto mc = my_class {};
  //...
}
```

Implementation: my_class.cpp

```cpp
#include "my_class.hpp"

void my_class::square_a() {
  a *= a;
}
```

User 2: fun.cpp

```cpp
#include "my_class.hpp"

void fun(my_class& mc) {
  mc.square_a();
}
```

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

```cpp
#include "my_first_class.hpp"

void my_first_class::square_a() {
  a *= a;
}

int my_first_class::sum(int b) {
  return a + b;
}
```

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)

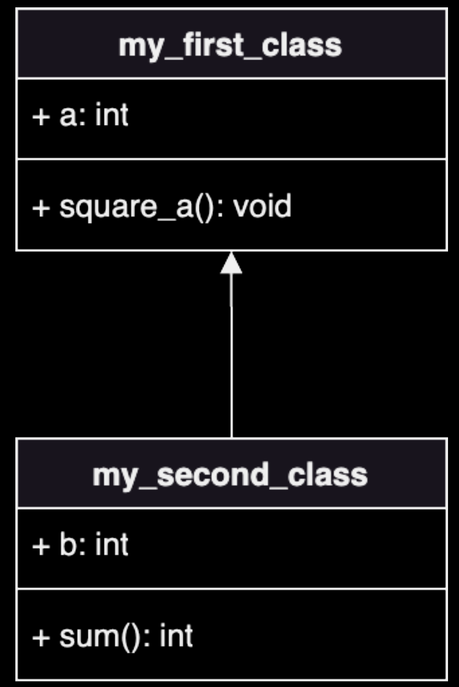
- Objects and classes
- Inheritance
- Constructors / Destructors
- Static members
- Allocating objects
- Advanced Object Oriented
- Type casing
- Operator overloading
- Function objects
- Name Lookups

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

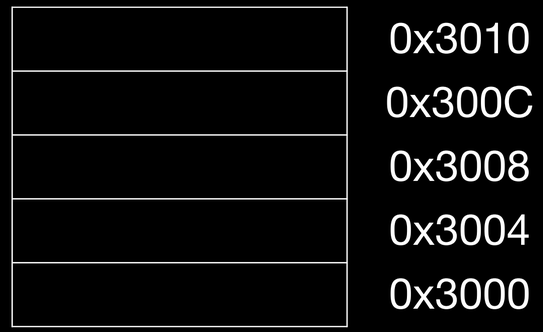
```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

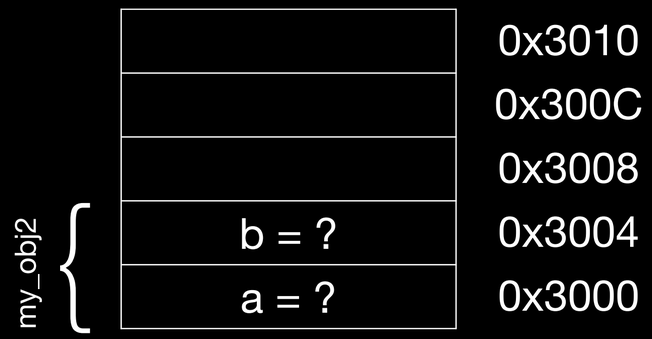
Memory layout



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

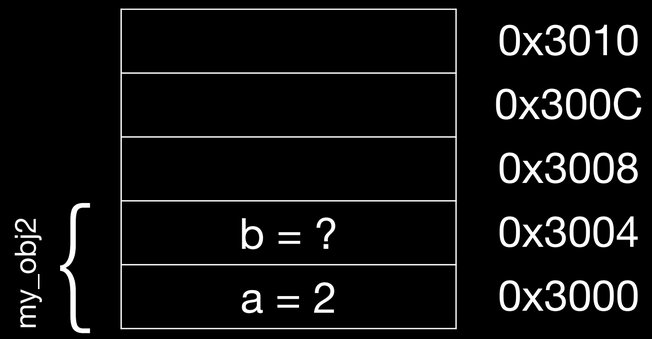
Memory layout



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

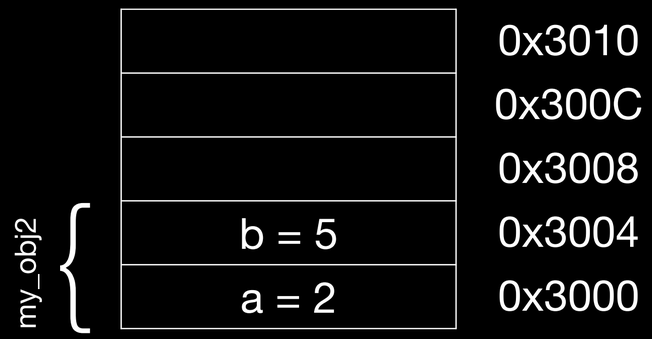
Memory layout



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

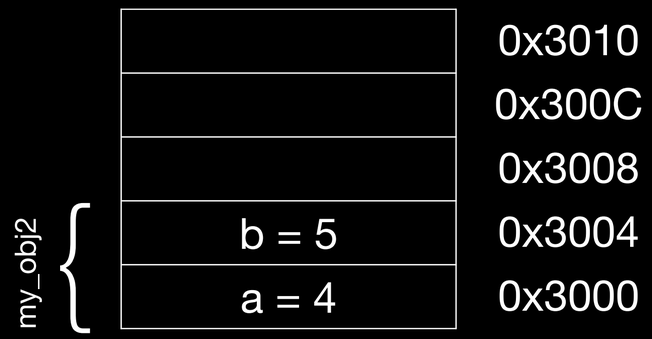
Memory layout



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

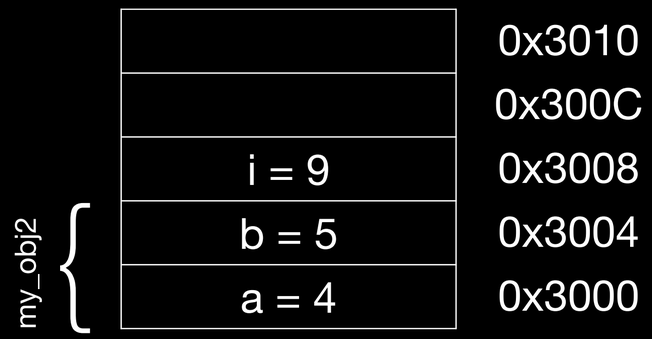
Memory layout



First inheritance

```cpp
struct my_first_class {
  int a;

  void square_a() {
    a *= a;
  }
};

struct my_second_class : my_first_class {
  int b;

  int sum() {
    return a + b;
  }
};

auto my_obj2 = my_second_class {};
my_obj2.a = 2;
my_obj2.b = 5;

my_obj2.square_a();
auto i = my_obj2.sum(); // i = 9
```

Memory layout



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 auto 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 auto 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

```cpp
class my_first_class {
public:
  int get_a();
  void set_a(int value);

  void square_a();

protected:
  int a;
};
```

```cpp
class my_second_class : public my_first_class {
public:
  int get_b();
  void set_b(int value);

  int sum() {
    return a + b;
  }

private:
  int b;
};
```

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

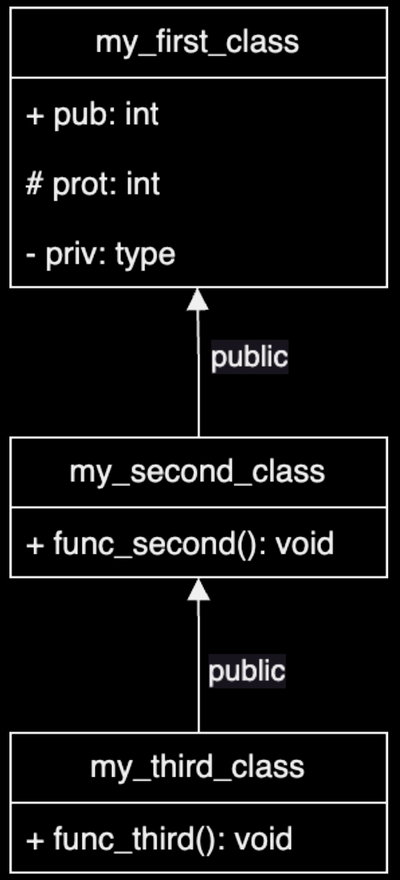
- public privacy of inherited members remains unchanged
- protected inherited public members are seen as protected
- private all inherited members are seen as private. This is the default for classes if nothing is specified

Inheritance can be public, protected or private

- Net result for external code
  - only public members of public inheritance are accessible

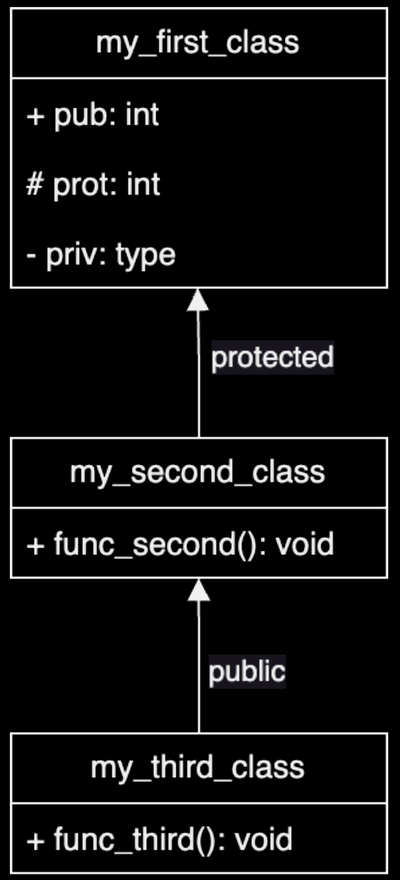
- Net result for code in derived classes
  - only public and protected members of public and protected
  - parents are accessible

Managing inheritance privacy - public



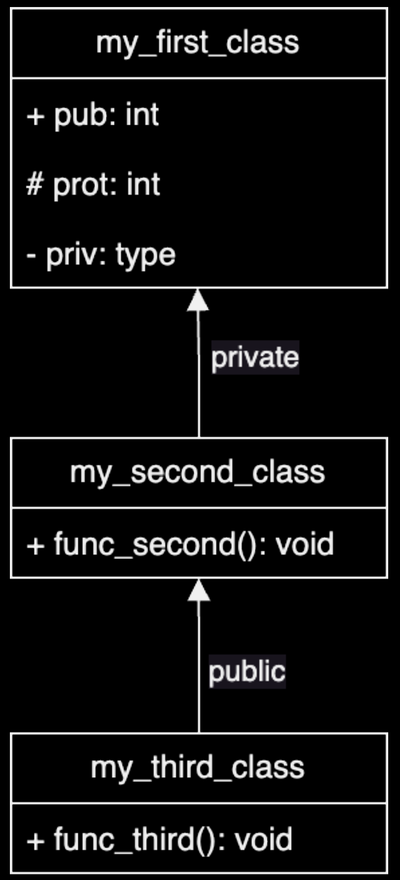

```
1 void my_second_class::func_second() {
2   int a = pub;  // ok
3   int b = prot; // ok
4   int c = priv; // error
5 }
6
7 void my_third_class::func_third() {
8   int a = pub;  // ok
9   int b = prot; // ok
10  int c = priv; // error
11 }
12
13 void ext_func(my_third_class t) {
14  int a = t.pub;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

Class constructors and destructors

Concept

- special functions called when building/destroying an object
- a class can have several constructors, but only one destructor
- the constructors have the same name as the class
- same for the destructor with a leading ~

```
1  class c {
2  public:
3    c();
4    c(int a);
5    ~c();
6    //...
7  protected:
8    int a;
9  };
10
11 // note: special notation for
12 // initialization of members
13 c::c() : a {0} {}
14
15 c::c(int a) : a {a} {}
16
17 c::~c() {}
```

Class constructors and destructors

```
 1 class vector {
 2 public:
 3   vector(int len);
 4   ~vector();
 5
 6   int get_n(int n);
 7   void set_n(int n, int value);
 8
 9 protected:
10   int len;
11   int* data;
12 };
13
14 vector::vector(int n) : len {n} {
15   data = new int[n];
16 }
17
18 vector::~vector() {
19   delete[] data;
20 }
```

Constructors and inheritance

```cpp
struct first {
  first() {} // leaves a uninitialized
  first(int a) : a {a} {}
  int a;
};

struct second : first {
  second();
  second(int b);
  second(int a, int b);
  int b;
};

second::second() : first {}, b {0} {}
second::second(int b) : b {b} {} // first {} implicitly
second::second(int a, int b) : first {a}, b {b} {}
```

Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type const & to class
- provided by the compiler if not declared by the user

```
1 struct c {
2   c();
3   c(const c& other);
4 };
```

Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type const & to class
- provided by the compiler if not declared by the user
- in order to forbid copy, use = delete (or private copy constructor with no implementation in C++ 98)

```
1 struct c {
2   c();
3   c(const c& other) = delete;
4 };
```

Good practice

The rule of 3/5 (C++ 98/11) - cppreference

if a class needs a custom destructor, a copy/move constructor or a copy/move assignment operator, it should have all three/five.

```
1 class c {
2 public:
3   c();
4   ~c();
5   c(const c& other);
6   c& operator =(const c& other);
7 }
```

```
1 class c {
2 public:
3   c();
4   ~c();
5   c(c&& other);
6   c(const c& other);
7   c& operator =(c&& other);
8   c& operator =(const c& other);
9 }
```

Copy constructor

```cpp
class vector {
public:
  vector(int n);
  vector(const vector& other);
  ~vector();

private:
  int len;
  int* data;
};

vector::vector(int n) : len {n} {
  data = new int[len];
}

vector::vector(const vector& other) : len {other.len} {
  data = new int[len];
  std::copy(other.data, other.data + len, data);
}

vector::~vector() {
  delete[] data;
}
```

Explicit unary constructor

Concept

- A constructor with a single non-default parameter can be used
- by the compiler for an implicit conversion.

```
1 void print(const vector& v) {
2   std::cout << "printing v elements...\n";
3 }
4
5 int main() {
6   // calls vector::vector(int n) to construct a vector
7   // then calls print with that Vector
8   print(5);
9 }
```

Explicit unary constructor

Concept

- The keyword explicit forbids such implicit conversions.
- It is recommended to use it systematically, except in special cases.

```
1 class vector {
2 public:
3   explicit vector(int n);
4   vector(const vector& other);
5   ~vector();
6
7   // ...
8 };
```

Defaulted constructor

Idea

- avoid empty default constructors like class_name() {}
- declare them as = default

```
1 class my_class {
2 public:
3   my_class() = default;
4   // ...
5 };
```

Details

- without a user-defined constructor, a default one is provided
- any user-defined constructor disables the default one
- but the default one can be requested explicitly
- rule can be more subtle depending on data members

```
1 class my_class {
2 public:
3   my_class() = delete;
4   // ...
5 };
```

Delegating constructor

Idea

- avoid replication of code in several constructors
- by delegating to another constructor, in the initialization list

```
1 struct delegate {
2   delegate() : delegate {42} {}
3
4   delegate(int value) : i {value} {
5     /// ... complex initialization ...
6   }
7
8   int i;
9 };
```

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