Back to Basics

Object-Oriented Programming

Presentation Material

CppCon, Aurora CO, 2024-09-20

Andreas Fertig

Write unique code!

Version: v1.0

Planning, typesetting and cover design: Andreas Fertig
Cover art and illustrations: Franziska Panter https://franziskapanter.com
Production and publishing: Andreas Fertig

Style and conventions

The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in `a.out` as the program name.

```
$ ./a.out
Hello, C++!
```

- `<string>` stands for a header file with the name `string`
- `[[xyz]]` marks a C++ attribute with the name `xyz`.

https://AndreasFertig.com
post@AndreasFertig.com

Training services

- I'm available for inhouse classes, onsite or remote.
- You can also book my course at the CppCon Academy.
  - Or one of my fellow instructors.

Cppcon
The C++ Conference

https://cppcon.org/cppcon-academy-2024/

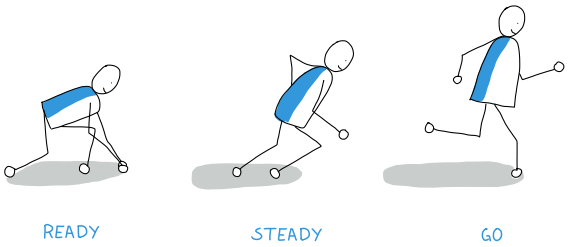
- You want to learn at your own pace? Check out my self-study courses:

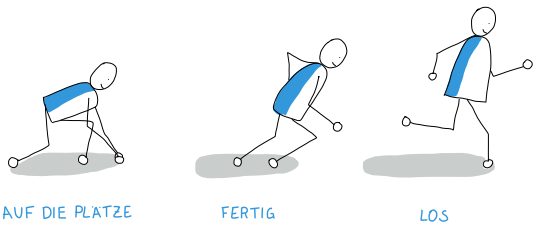
http://fertig.to/slcpp20

fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed





A class

- C++ adds classes for object-oriented programming.
- The constructors (B) are called when the object is created.
  - Data fields of classes should be initialized in the constructor initialization list (C).
  - The body of the constructor (D) is available for further operations.
- Classes can consist of member functions (E) and data fields (G) with different access rights (A).
  - For everyone: **`public`**
  - Only for children: **`protected`**
  - Only for this class: **`private`**

```
1  class Apple {
2  public:                          A Access specifier
3    explicit Apple(int i)          B Constructor
4    : /*this->*/ mData{i}          C Constructor initializer list
5    { D Constructor body
6    }
7
8    E Member functions
9    void Set(int i) { /*this->*/ mData = i; }
10
11   F Note the const
12   auto Get() const { return mData; }
13
14 private:
15   int mData{};   G Data members
16 };
17
18 int main()
19 {
20   Apple alice{3};
21   alice.Set(5);   H
22
23   return alice.Get();   I
24 }
```


`struct` vs. `class`

```
1 struct Apple {
2   int mValue{};
3
4   void Fun();
5 };
```

```
1 class Apple {
2   int mValue{};
3
4   void Fun();
5 };
```

- During this talk I will often use **`struct`** to save one line and to make elements **`public`**.

`struct` vs. `class`

```
1  struct Point {
2    int x;
3    int y;
4  };
5
6  class Date {
7    Year  mYear{};
8    Month mMonth{};
9    Day   mDay{};
10
11 public:
12   Date(Year year, Month month, Day day);
13 };
```


How to declare a class … and why

```
1  class String {
2  public:
3    String() = default;
4
5    String(const String&);
6    String& operator=(const String&);
7
8    String(String&&);
9    String& operator=(String&&);
10
11   ~String();
12
13 private:
14   char*  mData{};
15   size_t mLength{};
16   size_t mCapacity{};
17 };
```

How to declare a class … and why

```cpp
class String {
public:
  String() = default;

  String(const String&);
  String& operator=(const String&);

  String(String&&);
  String& operator=(String&&);

  ~String();

private:
  char*  mData{};
  size_t mLength{};
  size_t mCapacity{};
};
```

```cpp
class String {
  char*  mData{};
  size_t mLength{};
  size_t mCapacity{};

public:
  String() = default;
  ~String();

  String(const String&);
  String& operator=(const String&);

  String(String&&);
  String& operator=(String&&);
};
```

Based on the article [1] from Howard Hinnant.

The different types

- You can create either
  - Values types, or
  - Reference (view) types.
- Go for value types whenever possible!

Special member functions and class operators

The different special member functions of a class.

```
1 class Cpp {
2 public:
3    Cpp();                          A default constructor
4    Cpp(T t);                       B constructor
5   ~Cpp();                          C destructor
6    Cpp(const Cpp&);                D copy constructor
7    Cpp& operator=(const Cpp&);     E copy assignment
8    Cpp(Cpp&&);                     F move constructor (C++11)
9    Cpp& operator=(Cpp&&);          G move assignment (C++11)
10 };
```

Special member functions and class operators

The different special member functions of a class.

```
1 class Cpp {
2 public:
3    Cpp();                          A default constructor
4    Cpp(T t);                       B constructor
5   ~Cpp();                          C destructor
6    Cpp(const Cpp&);                D copy constructor
7    Cpp& operator=(const Cpp&);     E copy assignment
8    Cpp(Cpp&&);                     F move constructor (C++11)
9    Cpp& operator=(Cpp&&);          G move assignment (C++11)
10 };
```

Using examples.

```
1
2
3 Cpp a{};
4 Cpp b{3};
5 C Destructor is called at the end of the scope
6 Cpp d{a};
7 b = a;  E
8 // we omit move for now
```

Special member functions and their dependencies

user declares			compiler implicitly declares					
					copy		move	
			default ctor	dtor	ctor	assignment	ctor	assignment
		Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
		Any ctor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
		default ctor	*user declared*	defaulted	defaulted	defaulted	defaulted	defaulted
		dtor	defaulted	*user declared*	defaulted	defaulted	not declared	not declared
	copy	**ctor**	not declared	defaulted	*user declared*	defaulted	not declared	not declared
	copy	**assignment**	not declared	defaulted	defaulted	*user declared*	not declared	not declared
	move	**ctor**	not declared	defaulted	deleted	deleted	*user declared*	not declared
	move	**assignment**	not declared	defaulted	deleted	deleted	not declared	*user declared*

Source: [2]

The keywords for polymorphic classes

- `virtual`: Marks a function as replaceable by a derived class.
- `override`: A function with the exact name and signature must exist in one of the base classes and is replaced by this implementation.
- `final`: A class cannot be used as a base class after this point. Or a member function cannot be replaced anymore.

Virtual functions

```
1  struct Fruit {
2    double mWeight{};
3    virtual ~Fruit() { puts("~Fruit"); }
4    virtual void Print() { puts("Fruit's Print"); }
5  };
6
7  struct Apple : Fruit {
8    int  mRipeGrade{5};
9    void Print() override { printf("Apple's Print: %d\n", mRipeGrade); }
10 };
11
12 struct PinkLady : Apple {
13   int  mColorGrade{8};
14   void Print() override { printf("Pink Ladies Print: %d\n", mColorGrade); }
15 };
16
17 int main()
18 {
19   PinkLady delicious{};
20   delicious.Print();  A
21
22   Fruit* f{static_cast<Fruit*>(&delicious)};  B
23   f->Print();
24 }
```

Default parameters

- How does a default parameter work?

```
1 void Fun(int x = 23);
2
3 void Use() { Fun(); }
```

Default parameters

- How does a default parameter work?

```
1  void Fun(
2    std::string x =
3      "Hello, C++ community! I'm a default parameter.");
4
5  void Use()
6  {
7    Fun();
8    Fun();
9    Fun();
10 }
```


Default parameters and `virtual` functions

```
1  enum class eDirect { Fwd = 0, Left = 1, Right = 2, Bwd = 3 };
2
3  struct Car {
4    virtual ~Car()                                    = default;
5    virtual void Drive(eDirect d = eDirect::Bwd) const = 0;
6  };
7
8  struct Tesla : public Car {
9    void Drive(eDirect d) const override { printf("Tesla: %d\n", d); }
10 };
11
12 struct Toyota : public Car {
13   void Drive(eDirect d = eDirect::Fwd) const override { printf("Toyota: %d\n", d); }
14 };
15
16 void Use()
17 {
18   std::unique_ptr<Tesla> tesla{std::make_unique<Tesla>()};
19   std::unique_ptr<Car>   toyota{std::make_unique<Toyota>()};
20
21   tesla->Drive(eDirect::Left);
22   toyota->Drive(eDirect::Left);
23
24   toyota->Drive();
25 }
```


https://AndreasFertig.com
post@AndreasFertig.com

Treating the destructor

```
1  class IOPort {
2  public:
3    IOPort() { puts("IOPort ctor"); }
4    ~IOPort() { puts("IOPort dtor"); }
5
6    virtual void Flush() { puts("IOPort Flush"); }
7  };
8
9  class USBC : public IOPort {
10 public:
11   USBC() { puts("USBC ctor"); }
12   ~USBC() { puts("USBC dtor"); }
13
14   void Flush() override { puts("USBC Flush"); }
15 };
16
17 int main()
18 {
19   A Note, I'm creating a USBC object and store it as IOPort
20   std::unique_ptr<IOPort> port{std::make_unique<USBC>()};
21
22   port->Flush();
23 }
```


Treating the destructor

```
1  class IOPort {
2  public:
3    IOPort() { puts("IOPort ctor"); }
4     virtual  ~IOPort() { puts("IOPort dtor"); }
5
6    virtual void Flush() { puts("IOPort Flush"); }
7  };
8
9  class USBC : public IOPort {
10 public:
11   USBC() { puts("USBC ctor"); }
12   ~USBC()  override  { puts("USBC dtor"); }
13
14   void Flush() override { puts("USBC Flush"); }
15 };
16
17 int main()
18 {
19   std::unique_ptr<IOPort> port{std::make_unique<USBC>()};
20
21   port->Flush();
22 }
```

Treating the destructor

```
1  struct IOPort {
2    IOPort() { puts("IOPort ctor"); }
3    virtual void Flush() { puts("IOPort Flush"); }
4
5  protected:
6    ~IOPort() { puts("IOPort dtor"); }  A protected this time!
7  };
8
9  struct USBC : public IOPort {
10   USBC() { puts("USBC ctor"); }
11   ~USBC() { puts("USBC dtor"); }
12   void Flush() override { puts("USBC Flush"); }
13 };
14
15 void Process(IOPort& device);
16 void Receive(IOPort* device);
17
18 void Use()
19 {
20   B Note, I'm creating a USBC object and store it as such
21   std::unique_ptr<USBC> port{std::make_unique<USBC>()};
22
23   C Exmeplary using functions
24   Process(*port);
25   Receive(port.get());
26 }
```

Treating the destructor

```
1  class String  final  {
2    char*  mData{};
3    size_t mLength{};
4    size_t mCapacity{};
5
6  public:
7    String() = default;
8    ~String();
9
10   String(const String&);
11   String& operator=(const String&);
12
13   String(String&&);
14   String& operator=(String&&);
15 };
```

Treating the destructor

Destructor	Reason	Example
`public + final`	The class is not intended to be a base class at all	`String`
`public + virtual`	Supports deletion via a base class pointer.	`IOPort`
only `protected`	Not intended to be destroyed via a base class pointer.	`IOPort`

The template method

```
1  struct Sort {
2    virtual ~Sort() = default;
3
4    void process()
5    {
6      read();
7      sort();
8      write();
9    }
10
11   virtual void read()  = 0;
12   virtual void sort()  = 0;
13   virtual void write() = 0;
14 };
15
16 class QuickSort final : public Sort {
17 public:
18   void read() override { puts("readData"); }
19   void sort() override { puts("sortData"); }
20   void write() override { puts("writeData"); }
21 };
```

The template method

```cpp
struct Sort {
  virtual ~Sort() = default;

  void process()
  {
    read();
    sort();
    write();
  }

protected:
  virtual void read()  = 0;
  virtual void sort()  = 0;
  virtual void write() = 0;
};

class QuickSort final : public Sort {
protected:
  void read() override { puts("read"); }
  void sort() override { puts("sort"); }
  void write() override { puts("write"); }
};
```

Good class design

```cpp
struct IOPort {
  IOPort() { puts("IOPort ctor"); }
  virtual void Flush() { puts("IOPort Flush"); }

protected:
  ~IOPort() { puts("IOPort dtor"); }
};

struct USBC : public IOPort {
  USBC() { puts("USBC ctor"); }
  ~USBC() { puts("USBC dtor"); }
  void Flush() override { puts("USBC Flush"); }
};

void Use()
{
  USBC device{};
  USBC copy = device;

  // IOPort hm = device;
  auto hm = new IOPort{};
  *hm     = device;
  // created a memory leak
}
```

Good class design

```cpp
struct IOPort {
  IOPort() { puts("IOPort ctor"); }
  virtual ~IOPort() { puts("IOPort dtor"); }

  virtual void Flush() { puts("IOPort Flush"); }

protected:
  IOPort(const IOPort&)            = default;
  IOPort& operator=(const IOPort&) = default;
  IOPort(IOPort&&)                 = default;
  IOPort& operator=(IOPort&&)      = default;
};

struct USBC : public IOPort {
  USBC() { puts("USBC ctor"); }
  ~USBC() override { puts("USBC dtor"); }

  USBC(const USBC&)            = default;
  USBC& operator=(const USBC&) = default;
  USBC(USBC&&)                 = default;
  USBC& operator=(USBC&&)      = default;

  void Flush() override { puts("USBC Flush"); }
};
```

Guidelines Summary

- Prefer value types over reference types.
- Declare the important things first in your class.
- Don't use **`virtual`** member functions with default parameters.
- Design classes to either be usable as a base class or not.
- Remember, a **`virtual`** destructor disables the move operations and will supposedly remove the copy operations in the near future.
- Address object slicing.
- You should have a very good reason to make a data member **`public`**.
- Declare the destructor virtual in the base class, if the class should support polymorphic deletion.
- Declare member functions that you override from a base class with **`override`**.
- Never override a non-virtual member function.
- Prefer **`class`** when there is a constructor that checks the invariant, otherwise **`struct`**.

}

I am Fertig.

https://AndreasFertig.com

Available online:

https://AndreasFertig.com

Images by Franziska Panter:

https://panther-concepts.de

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - GCC 14.1.0
  - Clang 18.1.0

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

References

[1] Hinnant H. E., "How i declare my class and why". https://howardhinnant.github.io/classdecl.html

[2] Hinnant H., "Everything You Ever Wanted To Know About Move Semantics (and then some)", *ACCU*, Apr. 2014. https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

Images:

2: fran
4: Franziska Panter
5: Franziska Panter
33: Franziska Panter

Upcoming Events

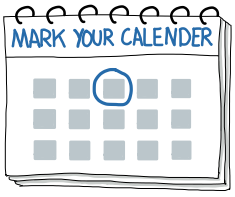
Talks

- *Fast and small C++ - When efficiency matters*, Meeting C++, November 16
- *Fast and small C++ - When efficiency matters*, code::dive, November 25
- *Effizientes C++ - Tips und Tricks aus dem Alltag*, ESE Kongress, December 04

Training Classes

- *Modern C++: When Efficiency Matters*, CppCon, September 21 - 22

For my upcoming talks you can check https://andreasfertig.com/talks/.
For my courses you can check https://andreasfertig.com/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.com/newsletter/.



About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and consultant for C++ for standards 11 to 23.
Andreas is involved in the C++ standardization committee, developing the new standards. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus understand constructs even better.
Before training and consulting, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
You can find Andreas online at andreasfertig.com.