ProgC++ — FS23 — Prof. Dr.-Ing. C. Werner

Laurin Heitzer

Classes

```cpp
class Base {
  public:
    Base(/* args */);
    ~Base();
  private:
    /* data */
};
Base::Base(/* args */) {}
Base::~Base() {}
```

Styleguide

public, protected and private members should be declared in that order.

Best practices

- Default ctor; further ctors if needed
- Every user-defined ctor should initialize all member variables
- label user-defined dtor with virtual
- "rule of zero" (if possible): no user-defined ctors, dtors, copy-ctors, move-ctors, copy-assignment-operators, move-assignment-operators

Ctors

Name: Identical to class name
Returntype: None! Not void!
Params: Any
– none: default ctor
– const reference to own class: **copy-ctor**
Task: Prepares/initializes class

Dtors

Name: Identical to class name
Returntype: None! Not void!
Params: None
Task: Deallocate memory/resources

Only one destructor per class. (In some special cases overloading is possible)
Automatically called when class is no longer needed.

Visibility

Public: Visible to everyone (within class, in derived, wherever class is used)
Protected: Only visible to class and derived classes
Private: Only visible within class

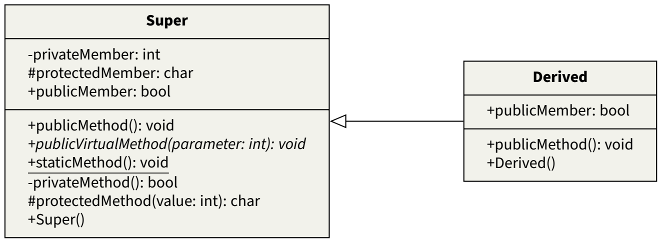
Friend attribute

A method within a class can be declared as friend. Said method is then globally accessible and can access private members of the class.

Getter- and Setter-methods

Getter- and Setter-methods are used to control access to private members of a class. They are public methods that return or set the value of a private member. They can be used to check the validity of the value to be set or to hide the implementation of the class (e.g. if the class is part of a library). In certain cases the setter method can be declared as protected, if it is only to be used by the base- and derived classes.

UML (Unified Modeling Language)



static methods can be called without an instance of a class. They can be called like `Super.staticMethod();`. `+publicVirtualMethod()` refers to a *pure virtual* method.

Const

If const is used after a method declaration, the method is not allowed to change any member variables of the class.

`int someFunction() const;`

Inheritance

If a class is to be inherited from, its destructor needs to be defined as virtual. A classic example of inheritance looks as follows:

```cpp
class Base {
  public:
    Base(/* args */) {}
    virtual ~Base() {}
  private:
};
class Derived: public Base {
  public:
    Derived(/* args */) {}
    virtual ~Derived() {}
  private:
};
```

Classes can be derived from as public, protected or private. Default of class is private, default of struct is public. This changes visibility of its inherited methods.

Inheritance visibility	visibility in base class	visibility in derived class
public (default w/ struct)	public	public
	protected	protected
	private	invisible
protected	public	protected
	protected	protected
	private	invisible
private (default w/ class)	public	private / invisibel
	protected	private / invisibel
	private	invisible

ctor- and dtor-chaining

Order of ctor calls: base class(es) first, then derived class(es).
Order of dtor calls: derived class(es) first, then base class(es).
Example:

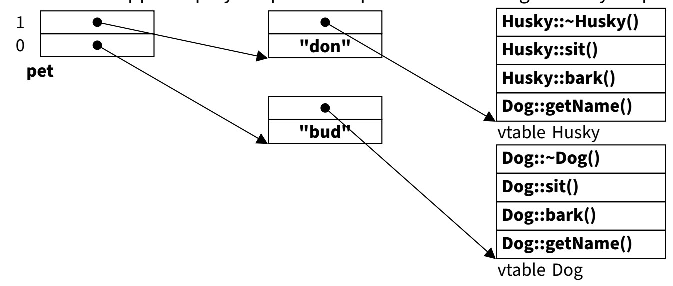
```cpp
Derived2::Derived2():
  Derived1{}, Base{}
{}
```

Ctors are automatically chained, but can be explicitly called as well with an initializer list like in the example.

vtable

```cpp
class Dog{
  public:
    Dog (std::string name_) : name(name_) {}
    virtual ~Dog() {};
    virtual void sit() const {}; // if =0 instead of {} used −> Dog becomes pure virtual class
    virtual void bark() const {std::cout << "woof" << std::endl;}
    virtual void getName() const {std::cout << name << std::endl;}
  private:
    std::string name;
};
class Husky: public Dog{
  public:
    Husky (std::string name_) : Dog(name_) {}
    ~Husky() = default;    // tells compiler to create default destructor
    void sit() const override {std::cout << "sit" << std::endl;}
    void bark() const override {std::cout << "wuff" << std::endl;}
};
int main(){
  Dog* pet[] = {new Dog("bud"), new Husky("don")};
}
```

Previous code snippet of polymorphic code produces following memory map:



Redefinition of methods

Inherited methods can be redefined in derived classes. If said methods are declared as **non-virtual**, **static binding** is applied and it is determined at **compile time**, whether the base class' or the derived class' method is called. (depending on pointer or reference type)
If the method is declared as ***virtual***, ***dynamic binding*** is applied at ***runtime*** and the derived class' method is called, even if the pointer or reference type is of the base class.
Example:

```cpp
class Base {};
class Derived1 : public Base {
  public:
    virtual bool foo();
};
class Derived2 : public Derived1 {
  public:
    bool foo() override;
};
class Derived3 final : public Derived2 {   // final: no further inheritance
  public:
    bool foo() final override;  // final: no further overriding
};
```

Classes that are declared as *pure virtual* have no methods defined (method = 0) and cannot be instantiated. They can only be used as base classes for other classes. This is called an interface.

Operator Overloading

Following operators can be overloaded:

```
+   −   *   /   %   ^   &   |   ~
!   =   <   >   +=  −=  *=  /=  %=
^=  &=  |=  <<  >>  <<= >>= ==  !=
<=  >=  &&  ||  ++  −−  ,   −>* −>
()  []  new new[]  delete  delete[]
```

Example:

```cpp
class Complex {
  public:
    Complex(double _real = 0, double _img = 0) : real{_real}, img{_img}{};
    ~Complex() {};

    friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
      os << c.real << " + " << c.img << "i";
      return os;};
    Complex operator+(const Complex& c) const {
      return Complex(real + c.real, img + c.img);};

    Complex operator−(const Complex& c) const {
      return Complex(real − c.real, img − c.img);};

    Complex operator*(const Complex& c) const {
      return Complex(real * c.real − img * c.img, real * c.img + img * c.real);};

    Complex operator/(const Complex& c) const {
      return Complex((real * c.real + img * c.img) / (c.real * c.real + c.img * c.img),
              (img * c.real − real * c.img) / (c.real * c.real + c.img * c.img));};
  private:
    double real;
    double img;
};
```

References

References are used like an alias to a variable.
With a variable `int var = 42;` a reference to it can be created with `int& myRef = var;`.
`myRef` can now be used exactly like `var`.
References are never uninitialized and never have nullptr as value.
It is unspecified whether or not a reference requires storage.
sizeof: When using sizeof on a reference, the size of the referenced object is returned.
const: By using const on a reference, the referenced object cannot be changed.

Function Pointers

Function:	`int functionName(int a, int b);`
Function pointer:	`int (*functionPointer)(int, int);`

Memory management

Automatic memory management: done by compiler, located on stack, LIFO (last in, first out) data structure, stack pointer points to top of stack, grows downwards.
Manual memory management: done by programmer, located on heap, order of allocation and deallocation not defined, heap grows upwards. After delete/free() pointer needs to be set to 0 / nullptr.

Operators & std libraries

Operator	Example
new	`int* intPtr = new int;`
new[]	`int* intArrPtr = new int[5];`
delete	`delete intPtr;`
delete[]	`delete[] intArrPtr;`

Library	Contents
<iostream>	std::cin, std::cout, std::endl
<string>	std::string
<iomanip>	std::setw(), std::setfill()
<fstream>	std::ifstream, std::ofstream

std formatting — <iomanip> <iostream>

Flag	Description	Example
std::boolalpha	output bool as text	`true` or `false`
std::fixed	output as fixed point	3.141593
std::dec	output decimal	42
std::hex	output hexadecimal	2a
std::oct	output octal	52
std::setprecision(6)	set precision of next output	3.14159
std::setw()	set width of next output	*see below*
std::setfill()	set fill character	*******-1.23
std::internal	fill between sign and digits	-*******1.23
std::left	fill on the right	*******-1.23
std::right	fill on the left	-1.23*******
std::scientific	output as scientific notation	3.141593e+00
std::showbase	show base of numbers	0x2a
std::showpoint	show decimal point	3.
std::showpos	show sign of positive numbers	+42
std::skipws	skip whitespace	
std::unitbuf	flush after each output	
std::uppercase	use uppercase for hex & float	2A
std::nouppercase	use lowercase for hex & float	2a

Styleguide

.h	.cpp
include guard `#pragma once`	header-comment
header-comment	include own headers "..."
include system libraries <...>	include system libraries <...>
include projectspecific libraries "..."	include projectspecific libraries "..."
definition of constants	global / static variables
typedefs, structs, classes	preprocessor directives
extern-declaration of global variables	function prototypes
function prototypes	function / class definitions

Compiler Arguments

Compiling with clang++: `clang++ -Wall -o main main.cpp`
With multiple source files: `clang++ -Wall -o main main.cpp other.cpp`
Following are the most commonly used compiler arguments for clang++:

-Wall	Enable all warnings
-o	Output file
-c	Compile to object file (mainly used in makefiles)
-std=c++11	Use C++11 standard

Makefiles

Used to automate compilation process. Useful when working with multiple source files. `make` checks which files have changed and only compiles those (timestamps).

```
CXX = clang++
CXXFLAGS = -Wall -c
LDFLAGS = -o
BIN = main
OBJS = main.o otherfile.o

all: $(BIN)

%.o: %.cpp
	$(CXX) $(CXXFLAGS) $<

$(BIN): $(OBJS)
	$(CXX) $(LDFLAGS) $@ $^

clean:
	rm -f $(OBJS) $(BIN)

.PHONY: clean all
```

Target
Dependencies
Tab (very important)

.PHONY:	Targets that do not produce an output are called "phony targets".
$@	Filename of target
$<	Filename of first dependency
$^	List of all dependencies

Assertions

Assertions are used in testing to check if a condition is true. If the condition is false, the program will terminate with an error message.
Found in `<cassert>` Usage: `assert(condition);`

Templates

Function templates

Function declaration:

```
template <typename T1, typename T2, ...>
returnType functionName(type1 param1, type2 param2, ...);
```

Function definition:

```
template <typename T1, typename T2, ...>
returnType functionName(type1 param1, type2 param2, ...){...}
```

Function templates can be used as `functionName<type1, type2,...>(param1, param2,...);` or `functionName(param1, param2, ...);`. The compiler will deduce the types of the template parameters from the function arguments.

Class templates

Class declaration:

```
template <typename T1, typename T2, ...>
class className {...
};
```

Class definition:

```
template <typename T1, typename T2, ...>
returnType className<T1, T2, ...>::methodName(type1 param1, type2 param2, ...){...};
```

Class templates can be used as `className<type1, type2,...> objectName;`. The compiler will deduce the types of the template parameters from the object declaration.

Rules

- Templates are **always** evaluated at **compile time**.
- To instantiate a template, the compiler needs to know following three things:
  1. The template declaration
  2. The template definition
  3. Values for the template parameters
- Definition of template functions and methods need to be in the same file as the declaration (.h).

Typecasting

Implicit typecasting is allowed in following cases:
- Between all numerical types (including bool)
- Between **some** pointer types (complicated ruels)
- Instances of a subclass can be implicitly assigned to a variable of the superclass type `SuperClass s = SubClass();`

Explicit typecasting allows following operators:

`static_cast<newType>(value of oldType)`: (most common)
- Can cast pointer- and reference-types to instances of super- and subclasses in both directions.
- Can be used for all implicit casts.
- No typechecking at runtime.

`dynamic_cast<newType>(value of oldType)`: (for polymorphic types)
- Can cast pointer- and reference-types to instances of polymorphic super- and subclasses in both directions.
- Checks at runtime (with RTTI) if the cast is valid. Returns nullptr if invalid.
- Can cast pointer- and reference-types of non-polymorphic subclasses to super-classes (up-casting), but not the other way round (down-casting).

`const_cast<const type>(value of type)`: (only special cases)
- Able to cast away const-ness or volatile-ness of a pointer or reference and the other way round.

`reinterpret_cast<newType>(value of oldType)`: (only special cases)
- Can cast pointer- and reference-types to any other pointer- and reference-types.
- No typechecking at runtime.
- Reinterprets the bits of the oldType as the newType.
- `reinterpret_cast` is platform dependent and should be avoided.

Previous type		New type	Cast-type	Implicit
float	→	double	static_cast	✓
int*	→	unsigned int*	reinterpret_cast	✗
const Animal	→	Animal	no cast	✓
Animal*	→	Bird*	dynamic_cast	✗
int	→	float	static_cast	✓
Bird*	→	int*	reinterpret_cast	✗
int	→	bool	static_cast	✓
volatile int*	→	volatile short*	reinterpret_cast	✗
Bird	→	Animal	static_cast	✓
volatile int*	→	int*	const_cast	✗
Bird*	→	Animal*	static_cast / dynamic_cast	✓
int&	→	short&	reinterpret_cast	✗

Exceptions

Exceptions are thrown using the throw keyword. Usually this is within a try...catch block or a function within such a block.

```
try {
  throw "Something went wrong";
} catch (const char* e) {
  std::cout << e << std::endl;
}
```

```
int divide(int a, int b) {
  if (b == 0) throw "Cannot divide by zero";
  return a / b;
}
int main(){
  try {
    divide(1, 0);
  } catch (const char* e) {
    std::cout << e << std::endl;
  }
}
```

If an error is thrown without a try...catch block, `std::terminate()` is called, which ends the program. The behaviour of this can be changed by setting a user-defined `std::terminate_handler` using `std::set_terminate()`.
Functions and methods that don't throw exceptions should be marked with the noexcept keyword. Otherwise noexcept(false) can be used to mark a function that can throw exceptions. `std::exception` can be found within `<stdexcept>`.