Object-Oriented Programming

Reference Types

Recap – Object Oriented Programming

- Model software through **objects and their interaction**

- **Object =** self-contained units that combines attributes (data) and methods (behavior)
- Object **attributes / elements** = data
- Object **methods** = functions that operate on object’s internal data
- Access to methods and attributes can be public or private to objects

Recap – Object Oriented Programming

- Encapsulation
- Abstraction

- Inheritance
- Polymorphism

Recap – Encapsulation

- Access to private members and methods is only possible over public interface
  - With access specifiers **public / private**
- Prevents read and write access from outside the class
- Member functions can access all members of their own class

```
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};
```

is the age someone is setting valid?

Recap – Encapsulation

- Hide internal details and protects the integrity of an object's state.
- Encapsulation provides **controlled access** and **protects data**

- Better Reuse in other projects
  - easier usage of the class

- Improved Maintainability
  - changing the private part of a class has no side effects outside of the class

Setter

- We call setAge a **setter**

```
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};
```

Setter

- We call setAge a **setter**
- For writing to private attributes
- Usually 1 parameter, no return value, Name: setAttributeName

```cpp
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};
```

Setter

- call setAge(..) on object to set attribute value

```cpp
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;
}
```

The Constructor

- Special method - automatically called when the object is created
- Name of constructor = name of class
- no return value

```
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;
}
```

The Default-Constructor

- Compiler adds **Default-Constructor**, if you don't write one.
- The Default-Constructor has no parameters and initializes nothing.

```cpp
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;
}
```

The Constructor

- Can have multiple constructors
- Must have different parameters

overloading
→ multiple fitting calls

```
class Pet {
  private:
    unsigned age_;
  public:
    Pet() { age_ = 0; }
    Pet(unsigned age) { age_ = age; }
    unsigned getAge() {
      return age_;
    }
};

int main() {
  Pet pet{2}; // constructor sets pet.age_ to 2.
  return 0;
}
```

→ no parameter

parameters

The Default-Constructor

- Will not be generated if another constructor exists !

```cpp
class Pet {
  private:
    unsigned age_;
  public:
    Pet() { age_ = 0; }
    Pet(unsigned age) { age_ = age; }
    unsigned getAge() {
      return age_;
    }
};

int main() {
  Pet pet{2}; // constructor sets pet.age_ to 2.
  return 0;
```

The Default-Constructor

- Will not be generated if another constructor exists !

```cpp
class Pet {
private:
  unsigned age_;
public:
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // error: no matching function for call to 'Pet::Pet()'
  return 0;
}
```

default constructor missing

The Default-Constructor

- We can add a Default-Constructor manually

```
class Pet {
private:
  unsigned age_;
public:
  Pet() = default;  // similar to: Pet() {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

The Default-Constructor

- **Problem**: age_ is not initialized by default constructor

```
class Pet {
private:
  unsigned age_;
public:
  Pet() = default;  // similar to: Pet() {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

The Default-Constructor

- Possible solution: Custom constructor without parameters but with initial values of attributes

```cpp
class Pet {
private:
  unsigned age_;
public:
  Pet() : age_{0} {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

initialize list

The Default-Constructor

- Other possibility: Default values for attribute

```cpp
class Pet {
private:
  unsigned age_ = 0; // also possible: unsigned age_{0}
public:
  Pet() = default;  // similar to: Pet() {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

The Constructor

Problem 1

- age_ is used before initialization (default value is random)

```cpp
#include <iostream>

class Pet {
    unsigned age_;
public:
    Pet(unsigned age) {
        bark();
        age_ = age;
    }
    void bark() { std::cout << age_; }
};

int main() {
  Pet pet{2}; // constructor sets pet.age_ to 2.
  return 0;
}
```

The Constructor

Problem 2

```cpp
#include <iostream>

class Pet {
    std::string name_;
    Pet(std::string name) {
        name_ = name;
    }
};
```

- Can be inefficient:
    1. Create empty string
    2. copy content to override empty string

redundant !

The Constructor – Initializer List

- **‘Initializer list’** avoids both problems

```cpp
class Pet {
  unsigned age_;

public:
  Pet() : age_{0} {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

The Constructor – Initializer List

- With multiple attributes:

```
Pet(char* name, unsigned age)
  : name_{name}, age_{age}
  {}
```

Summary

- Initialize objects with **Constructor**
- Use ‘**Setter**’ for setting attribute values

Setter

- Setters can do more
- e.g. logging or validating

```
#include <cstdio>

class Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

default constructor will be created if not defined

```cpp
#include <cstdio>

class Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

What is the output?

```cpp
int main() {
  Pet pet{};
  pet.setAge(0);
  pet.setAge(2000);
  printf("Your pet's age: %u", pet.age_); // POLL
  return 0;
}
```

Compiler error as age_ is private.

Getter

- How to access attributes from outside? => **Getter**
- Usually 0 parameters, attribute as return value, Name: getAttrName

```cpp
#include <cstdio>

class Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
    unsigned getAge() {
      return age_;
    }
};
```

What is the output?

```cpp
#include <cstdio>

class Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
    unsigned getAge() {
      return age_;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0);
  pet.setAge(2000);
  printf("Your pet's age: %u", pet.getAge()); // POLL
  return 0;
}
```

Deleting the Default-Constructor

- Default-Constructor can be deleted explicitly
- Object cannot be created, if there is no constructor

```cpp
class Pet {
private:
  unsigned age_;
public:
  Pet() = delete;
};

int main() {
  Pet pet{}; // error: no matching function for call to ‘Pet::Pet()’
  return 0;
}
```

Initialization

- **MyClass obj;** // Default initialization
- … calls the **default constructor** of MyClass (if it exists).

Initialization

- **MyClass obj(42);** // parenthesis initialization (direct initialization)
- Calls a constructor whose parameters match the argument types.

Initialization

- **MyClass obj(42);** // parenthesis initialization (direct initialization)
- Calls a constructor whose parameters match the argument types.
- … calls Constructor **MyClass(int value){…}**

Initialization

- **Braced initializer**

```cpp
class Pet {
  unsigned age_;

public:
  Pet() : age_{0} {}
  Pet(unsigned age) : age_{age} {}
};

int main() {
  Pet pet{}; // OK
  return 0;
}
```

Recap C - Initialization of PODs

- **Braced initializer**

```
struct Pet {
  char animal_[50];
  unsigned age_;
};

int main() {
  Pet pig = { "pig", 0 }; // as we know it from C
  Pet cat{ "cat", 7 }; // braced initializer (=uniform initializer)
}
```

Uniform initialization

- since C++11
- **Braced initializer**
- Example: MyClass obj2{42};

Uniform initialization

- Besides PODs: Uniform initialization can also be used for primitive initialization

- For example: int x{5}; // primitive

Uniform initialization

- Uniform initialization avoids *narrowing*!

```
int a = 3.14;    // truncates to 3, no warning!
```

- Better => Error protects from narrowing from double to int
- {} gives **compile-time safety** against unintended data loss

```
int a{3.14};    //
```

Problem Direct Initialization – Ambiguity

- Uniform initialization avoids ambiguity

- Default constructor vs. function declaration

```
class InitTester {
public:
  InitTester()         { printf("default - no argument\n"); }
  InitTester(char c)   { printf("char %c\n", c); }
  InitTester(int i)    { printf("int %d\n", i); }
  InitTester(float f)  { printf("float %f\n", f); }
};

int main() {
  InitTester t1;            -> default
  InitTester t2{ 'c' };     -> char
  InitTester t3{ 65537 };   -> int
  InitTester t4{ 60.1f };   -> float
  InitTester t5('g');       -> char
  InitTester t6 = { 'l' };  -> char
  InitTester t7{};          -> default.
  InitTester t8();          -> function declaration
}
```

Pointer and References

Memory Address

```
#include <cstdio>

int main() {
    int number = 1;
    int *ptr = &number;
    printf("printf.1: %p\n", &number); //0x7ffffffdcdc
    printf("printf.2: %p\n", ptr); //0x7ffffffdcdc
    return 0;
}
```

- Variables and functions are stored in memory

- Have an **address** in memory

- Address can be accessed using the address-of-operator **&**
- Example: int number=1;
  &number => Address of number

Pointer

```
#include <cstdio>

int main() {
    int number = 1;
    int *ptr = &number;
    printf("printf.1: %p\n", &number); //0x7ffffffdcdc
    printf("printf.2: %p\n", ptr); //0x7ffffffdcdc
    return 0;
}
```

- **A pointer** is a variable that **stores a memory address**
- Example: **int* a_ptr = &a;**

De-Reference Pointer

- Use *ptr to access the value pointed to

```
#include <iostream>

int main() {
    int num = 42;      // an integer variable
    int* ptr = &num;  // pointer storing the address of num

    std::cout << "Address stored in ptr: " << ptr << std::endl;     // prints the address
    std::cout << "Value pointed to by ptr: " << *ptr << std::endl;  // dereference

    *ptr = 100;  // modify the value stored at that address
    std::cout << "num after modification: " << num << std::endl;    // 100
}
```

Special Pointer

```cpp
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) {age = age;}
};
```

Compiler output

```
warning: explicitly assigning value of variable of type 'unsigned int' to itself
Cat(unsigned age) {age = age;}
                   ~~~ ^ ~~~
```

Better: use naming convention

```
class Cat
{
  unsigned age_;
  public:
    Cat(unsigned age) { age_ = age; }
};
```

Solution 2: Initialization list

```cpp
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) : age{age} {}
};
```

Special Pointer

Solution 3: use special pointer **this**

- points to the object on which the method was called
- use -> to call methods or to access an object's variable
- available in every non-static method (explanation of static follows)

```
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) { this->age = age; }
};
```

Method chaining with this-Pointer

reference to student; similar to pointer

```
class Student {
  public:
    Student& code() { printf("coding is fun\n"); return *this; }
    Student& sleep() { printf("zzz\n"); return *this; }
};

int main() {
  Student linus{};
  linus.code().sleep().code().sleep(); // fun weekend
}
```

Method chaining with this-Pointer

```cpp
class Student {
  public:
    Student& code() { printf("coding is fun\n"); return *this; }
    Student& sleep() { printf("zzz\n"); return *this; }
};

int main() {
  Student linus{};
  linus.code().sleep().code().sleep(); // fun weekend
}
```

Output:

```
coding is fun
zzz
coding is fun
zzz
```

Method chaining with this-Pointer

- Possible by returning *this
- Advantage: only one statement (instead of four in this example)

```cpp
class Student {
  public:
    Student& code() { printf("coding is fun\n"); return *this; }
    Student& sleep() { printf("zzz\n"); return *this; }
};

int main() {
  Student linus{};
  linus.code().sleep().code().sleep(); // fun weekend
}
```

Special Pointer - NULL

- In C macro **NULL** points to 0L
- Represented as **int**

Special Pointer - NULL

- Represented as **int** => Can be ambiguous

```
void function(int arg) {
    printf("arg is an integer\n");
}
void function(int* arg) {
    printf("arg is a pointer\n");;
}

int main() {
    function(NULL); // what is the output?
    return 0;
}
```

→ long 32 bit

→ long 64 bit

Special Pointer - nullptr

- Better use nullptr
- Is of type std::nullptr_t
- Can be implicitly converted to any pointer data type

```
void function(int arg) {
    printf("arg is an integer\n");
}
void function(int* arg) {
    printf("arg is a pointer\n");;
}

int main() {
    function(nullptr); // > arg is a pointer
    return 0;
}
```

Special Pointer - nullptr

- Every pointer can be implicitly converted to bool
- The result is false if the pointer is **nullptr** and true otherwise.

```
if (nullptr)
  printf("never ever\n");
else
  printf("nullptr is false!\n");
```

Special Pointer - nullptr

- **Cannot** implicitly converted into another data type

```
int number = nullptr; // error: cannot convert std::nullptr_t to int
```

Special Pointer – void*

- generic pointer type
- can hold the address of any data type

Example – void*

- can hold the address of any data type
- void* can be cast to specific type

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Warning!

- If you pass the wrong type code, you'll get **undefined behavior**
- For example:

  printValue(&a, 'd');
  // treated as double
  // undefined behavior!

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Warning!

- If you pass the wrong type code, you'll get **undefined behavior**
- For example:

  printValue(&a, 'd');
  // treated as double
  // undefined behavior!

- Avoid if possible - because the compiler can't catch mistakes

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Pointer for Array Decay

- When we use an array name, it is automatically converted to (or “decays into”) a pointer
- College* colleges = &oxford[0];

```cpp
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (size_t i = 0; i < n_colleges; i++) {
    printf("%s College\n", colleges[i].name);
  }
}

int main() {
    College oxford[] { "Magdalen" , "Nuffield" , "Kellogg" };
    print_names(oxford, sizeof(oxford) / sizeof(College));
}
```

Pointer arithmetic

- ++ operator can also **increment a pointer**
  => increment points to the **next element of its type**

- college++
  => college = college
  + 1 * sizeof(College)

```cpp
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,3);

    return 0;
}
```

Pointer arithmetic

- Can be dangerous

- Random result
  or Prog. Crash

```
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,4);

    return 0;
}
```

Call-by-Value

- A copy is passed

```cpp
#include <iostream>

void addTen(int x) {
    x = x + 10;  // only changes the copy
}

int main() {
    int num = 5;
    addTen(num);            // pass by value
    std::cout << num << std::endl; // prints 5, not 15
}
```

Call-by-Reference

- When called **with a reference, the actual variable is passed**

```cpp
#include <iostream>

void addTen(int &x) {
    x = x + 10;  // changes the original variable
}

int main() {
    int num = 5;
    addTen(num);          // pass by reference
    std::cout << num << std::endl; // prints 15
}
```

Function Call using Variable Address

- C++ allows call-by-value using pointer **int *x**

```cpp
#include <iostream>

void addTen(int *x) {
    *x = *x + 10;  // dereference pointer to change original
}

int main() {
    int num = 5;
    addTen(&num);    // pass address
    std::cout << num;     // prints 15
}
```

Call-by-Value using Variable Address

- Need to de-reference the address to modify variable

```cpp
#include <iostream>

void addTen(int *x) {
    *x = *x + 10;  // dereference pointer to change original
}

int main() {
    int num = 5;
    addTen(&num);   // pass address
    std::cout << num;     // prints 15
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a);
    printf("%d\n", a_ref);
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref);
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref); //24601
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref); //24601
    printf("%p\n", a_ptr); //0x7ffffffdcdc
}
```

Examples – What is the output?

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – What is the output?

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – References & Pointers

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – References & Pointers

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7ffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref); //24602 <- value of the reference has changed
printf("%d\n", a); //24602 <- value of a has changed
printf("%p\n", &b); //0x7ffffffdcc4
printf("%p\n", &a_ref); //0x7ffffffdcc8 <- a_ref still is a reference to a
```

Code Organization

- Separation into
  - Header (.hpp)
  - Source (.cpp)
- Header
  - Class definition
  - Method declaration
- Source
  - Method implementation (definition)

Example

Time.hpp

```
class Time
{
public:
  Time(int hour, int minutes);
  void displayTime();

private:
  int hour_;
  int minute_;
}
```

Time.cpp

```
#include "time.hpp"

Time::Time(int hour, int minute) :
  hour_(hour), minute_(minute)
{
}

void Time::displayTime()
{
  printf("It is %d:%d!", hour_, minute_);
}
```

Dependencies

classA.hpp

```
#ifndef ClassA_h
#define ClassA_h

#include "classB.hpp"

class A
{
  B* oldB_;
public:
  A(B* oldB) : oldB_(oldB){}
  void run() { if(oldB_) oldB_->run(); }
};
#endif
```

classB.hpp

```
#ifndef ClassB_h
#define ClassB_h


class B
{

public:

  void run() {            }
};
#endif
```

Dependencies

- Circular dependency!

classA.hpp

```
#ifndef ClassA_h
#define ClassA_h

#include "classB.hpp"

class A
{
  B* oldB_;
public:
  A(B* oldB) : oldB_(oldB){}
  void run() { if(oldB_) oldB_->run(); }
};
#endif
```

classB.hpp

```
#ifndef ClassB_h
#define ClassB_h

#include "classA.hpp"

class B
{
  A* neA_;
public:
  B(A* neA) : neA_(neA){}
  void run() { if(neA_) neA_->run(); }
};
#endif
```

Forward Declaration

- To just tell the compiler "it exists," without including all details yet
- Can **only use a pointer or reference**
- No #include needed

classA.hpp

```cpp
#ifndef ClassA_h
#define ClassA_h
// there is a class B, you
// don't need to know how it looks like
class B;
class A
{
  B* oldB_;
public:
  A(B* oldB) : oldB_(oldB){}
  void run();
};
#endif
```

classB.hpp

```cpp
#ifndef ClassB_h
#define ClassB_h
// there is a class A, you
// don't need to know how it looks like
class A;
class B
{
  A* neA_;
public:
  B(A* neA) : neA_(neA){}
  void run();
};
#endif
```

Include when actually needed to create object

classA.cpp

```
#include "classA.hpp"
#include "classB.hpp"
// now we know how both A and B look like!
void A::run()
{
  if(oldB_) oldB_->run();
  else printf("run in A is the final\n");
}
```

classB.cpp

```
#include "classB.hpp"
#include "classA.hpp"
// now we know how both B and A look like!
void B::run()
{
  if(newA_) newA_->run();
  else printf("run in B is the final\n");
}
```

```
#include "classA.hpp"
#include "classB.hpp"

int main()
{
        A a1(nullptr);
        B b1(&a1);
        A a2(&b1);
        a2.run(); // run in A is the final
}
```