

Object Oriented Software Design

Basics of C++

Giuseppe Lipari

<http://retis.sssup.it/~lipari>

Scuola Superiore Sant'Anna – Pisa

November 19, 2010

Outline

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects
- 4 Our First class
- 5 Destructor

- In this lecture, we will start to see how C++ improves over C
- Since you have already seen Java, you should have by now enough elements of Object Oriented Programming
 - Classes, inheritance, composition, etc.
- Therefore, every step I will try to give you the differences with Java, if any
- Also, many things that are possible in C++ will not be possible in Java, and vice versa

C++ naming conventions

- C++ files usually ends in `.cpp` or `.cc` or `.cxx`
- Header files for C++ usually end in `.h`, `.hpp`, `.hxx`, `.hh`, or even without any extension
- To compile a C++ program you have to use the C++ compiler (different from the C compiler)
 - The GNU/Linux provides you the `g++` on the command line
 - `g++` is at the same time a compiler and a linker
 - Compile and link:

```
g++ myfile.cpp -o myfile
```

- Only compile:

```
g++ -c myfile.cpp -o myfile.o
```

- Only link:

```
g++ myfile.o -o myfile
```

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects
- 4 Our First class
- 5 Destructor

Scope and visibility

- One of the problems of C was the fact that all global variables are in the same scope
 - Also, variables in different files!
 - For example, it is not possible to have two variables with the same name in two different modules
- This is a problem for modular programming
 - Suppose that the system architect (the big design boss) decides to split the work across two programming teams, A and B
 - Both teams independently decide to use a global function called `void compute();`
 - This causes problems at linking time: it is not possible to have two distinct functions with the same name in the same (global) scope
- Another problem is when you decide to include an external library
 - What if the designers of the library decided to use names that are quite common?

Reducing visibility

- One possibility is to use the `static` keyword
 - `static` is just the opposite of `extern`; a static object **is not** exported to the linker

module.h

```
// variable declaration
extern int a;
// function prototype
int f(int);
```

module.c

```
#include "module.h"

// this is exported;
int a = 0;

// this is not exported;
static int b = 0;
// try to uncomment
// int b = 0;

int f(int i)
{
    b = a + i;
    a = i/2;
    return b;
}
```

module2.c

```
#include <stdio.h>
#include "module.h"

// this is exported!
//(but does not conflict)
int b;

int main()
{
    int c;

    a = 5;
    b = 10;

    c = f(10);

    printf("c = %d\n", c);
    printf("a = %d\n", a);
}
```

- So, `static` has two meanings
 - Inside a function, makes a local variable persistent across function calls
 - In the global scope, hides a global variable to be used only inside that module
 - it can also be used for functions
- However, this does not completely solve the naming problem
 - What if we want to use two different functions with the same name in the same program?
 - Suppose you are writing a variable for mp3 audio processing, and you implement a set of functions, one of them is called `decode()`
 - Someone else has implemented a video library that processes H.264/MPEG-4, and implements a function called `decode()`

- C++ solves this problem using namespaces
- A name space is just a way to create and name a *scope*
 - The idea is that when you build a library, you define a namespace having a meaningful name (for example the name of the library), and enclose all your declarations in the namespace
 - The user of the library can then specify which functions to use using the *scope resolution operator*

C++ namespaces

- In the previous example:

```
// audio.hpp  
namespace audiolib {  
    ...  
    void decode();  
    ...  
}
```

```
// audio.hpp  
namespace videolib {  
    ...  
    void decode();  
    ...  
}
```

```
// your module  
#include "audio.hpp"  
#include "video.hpp"  
...  
audiolib::decode();  
...  
videolib::decode();
```

Scope resolution

- The `::` symbol is called scope resolution, and it is used to decide which function or variable we want to use
 - it is like directories: with `::` you can specify the *full name* of a variable (similar to the *path*)
- namespaces can be **nested**:

```
// three different functions!!
int f(int i);
namespace nnn {
    int f(int i);
    namespace mmm {
        int f(int i);
    }
}

// function usage;
f(5);
nnn::f(5);
nnn::mmm::f(5);
```

Outline

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects
- 4 Our First class
- 5 Destructor

Simple input and output

- Simple output can be done with the `iostream` standard library
- All functions in the standard library are part of the `std` namespace;

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

Using directive

- Sometimes it is very annoying to type `std::`, so we can use a *using directive*:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
}
```

- First, `cout` is searched in the global scope: if it is not found, the namespace in the using directives are looked into

- Be careful with the `using` directive:
 - If two namespaces contain the same name, there will be a conflict, so you have to specify which one to use with the scope resolution

```
#include "audio.hpp"
#include "video.hpp"
using namespace audiolib, videolib;
...
decode(); // compilation error! cannot be resolved

audiolib::decode() // ok, now it can be resolved
```

- Notice that we include `iostream` (without extension)
 - Standard library include files have no extension
- It is a little bit too early to understand what is `cout`. Right now, it is sufficient to know how to use it
- `cout` must be followed by `<<` and a variable, or a constant, or an expression, or a *modifier* like `endl` (which means end of line).
- You can chain as many segments of `<<` as you like

```
cout << "Now a number: " << 5 << " and now a float: " << 3.5 << endl;
```


- Here is how you do input:

```
#include <iostream>
using namespace std;

int main()
{
    int a;
    cout << "Enter an integer number ";
    cin >> a;
    cout << "The square of " << a << " is " << a*a << endl;
}
```

- `cin` is exactly specular to `cout`
- You can also use `cerr` for output on the standard error

Strings

- If you need to manipulate strings, you can use the `string` class from the `std` library

stringex.cpp

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name = "Giuseppe";
    string surname("Lipari");
    string tot;

    tot = name + "-" + surname;
    int i = tot.find("-");
    cout << tot << endl;
    cout << "The dash is at location: " << i << endl;
    cout << "First part: " << tot.substr(0, i) << endl;
    cout << "Last part: " << tot.substr(i+1, tot.size()) << endl;
}
```

- `string` is a class
 - In the previous examples we declare **three objects** of type `string`
 - Notice that `name`, `surname` and `tot` are objects, not references to objects!
 - There is no new instruction!
 - These objects are created on the stack (and not on the heap, more on this later)
- You see three ways of initialising an object: with an assignment (`name = "Giuseppe"`), with a constructor function (`surname("Lipari")`), and with a default constructor (`tot`)
 - Actually, also the first one is a constructor, it is called *copy constructor*
- The `+` operator is used to concatenate strings (like in Java)
 - Unlike Java, `string` is not a special class: actually, in C++ you can redefine the operator `+` for your own classes (more on this later)

Boolean

- C++ has a boolean primitive type, called `bool`, and two boolean constants, `true` and `false`

```
bool flag = false;
...

if (flag) {
    ...
}
```

- However, C++ derives from C, where there was no boolean type
 - in C, a numerical value of 0 is assimilated as *false*, while a numerical value different of 0 is assimilated as *true*
 - Therefore, in C it is perfectly legal to write:

```
int a = 0;
...
if (a) {
    ...
}
```

- C++ derives from C, so there is an automatic cast between a numerical value of 0 and `false`, and a value different of 0 and `true`

- Pay attention: the C++ compiler allows this:

```
if (a = 0) {  
    ...  
}
```

- The code above is legal: the result of expression `a=0` is 0 (hence false), so the block is never executed
 - in Java instead it is an error (no automatic conversion between 0 and false)
 - Most modern compilers only raise a warning

Outline

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects**
- 4 Our First class
- 5 Destructor

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

Class declaration

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Class declaration

What follows is all private

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Class declaration

What follows is all private

These are private variables

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Class declaration

What follows is all private

These are private variables

What follows is protected

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Class declaration

What follows is all private

These are private variables

What follows is protected

A protected method

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Class declaration

What follows is all private

These are private variables

What follows is protected

A protected method

What follows is public

Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages

The diagram illustrates the components of a C++ class declaration for `MyClass`. It consists of a code block on the left and seven yellow callout boxes on the right, each with an arrow pointing to a specific part of the code.

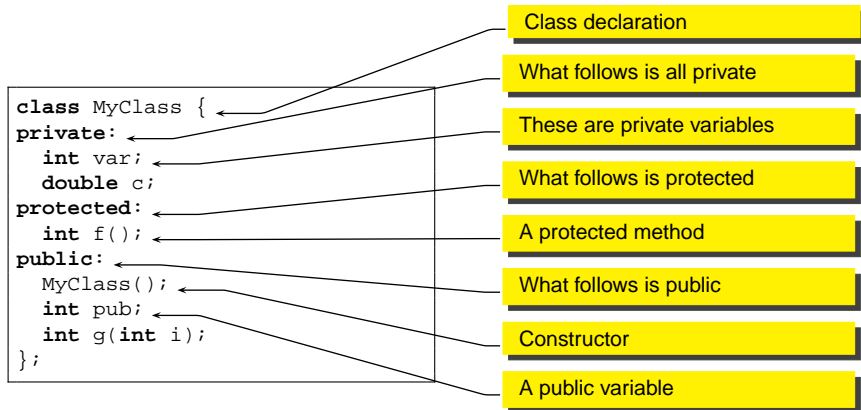
```
class MyClass {  
private:  
    int var;  
    double c;  
protected:  
    int f();  
public:  
    MyClass();  
    int pub;  
    int g(int i);  
};
```

Annotations:

- Class declaration (points to `class MyClass {`)
- What follows is all private (points to the `private:` section)
- These are private variables (points to `int var;` and `double c;`)
- What follows is protected (points to the `protected:` section)
- A protected method (points to `int f();`)
- What follows is public (points to the `public:` section)
- Constructor (points to `MyClass();`)

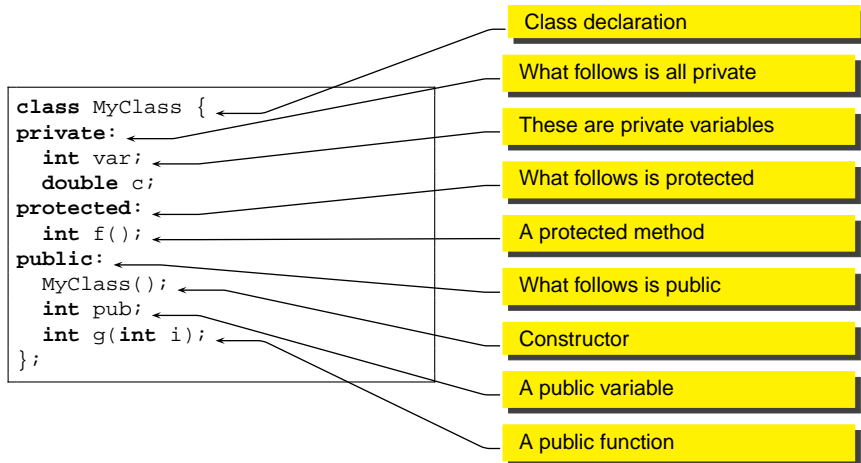
Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages



Classes in C++

- A class in C++ is quite similar to a class in Java or in other OO languages



- A member can be:
 - **private**: only member functions of the same class can access it; other classes or global functions can't
 - **protected**: only member functions of the same class or of derived classes can access it; other classes or global functions can't
 - **public**: every function can access it

```
class MyClass {  
private:  
    int a;  
public:  
    int c;  
};
```

```
MyClass data;  
  
cout << data.a; // ERROR!  
cout << data.c; // OK: c is public;
```


Access control

- Default is private
- An access control keyword defines access until the next access control keyword

```
class MyClass {  
    int a;  
    double b;  
public:  
    int c;  
    void f();  
    int getA();  
private:  
    int modify(double b);  
};
```

private (default)

public

private again

Access control and scope

```
int xx;  
  
class A {  
    int xx;  
public:  
    void f();  
};
```

global variable

member variable

```
void A::f()  
{  
    xx = 5;  
    ::xx = 3;  
  
    xx = ::xx + 2;  
}
```

access local xx

access global xx

- Some people think that private is synonym of secret
 - they complain that the private part is visible in the header file
- private means not accessible from other classes and does not mean secret
- The compiler needs to know the size of the object, in order to allocate memory to it
 - In an hypothetical C++, if we hide the private part, the compiler cannot know the size of the object

Friends

- Sometimes, two classes interact so much that we would like to let them *share* access to their private variables
- In that case, we have to declare them to be **friend**

```
class A {  
    friend class B;  
    int y;  
    void f();  
public:  
    int g();  
};
```

```
class B {  
    int x;  
public:  
    void f(A &a);  
};
```

B is friend of A

```
void B::f(A &a)  
{  
    x = a.y;  
    a.f();  
}
```

B can access private members of A

Friend functions

- Even a global function or a single member function can be friend of a class

```
class A {  
    friend B::f();  
    friend h();  
    int y;  
    void f();  
public:  
    int g();  
};
```

friend member function

friend global function

- It is better to use the *friend* keyword only when it is really necessary

Nested classes

- It is possible to declare a class inside another class
- Access control keywords apply

```
class A {  
    class B {  
        int a;  
    public:  
        int b;  
    }  
    B obj;  
public:  
    void f();  
};
```

- Class B is private to class A: it is not part of the interface of A, but only of its implementation.
- However, A is not allowed to access the private part of B!! (A::f() cannot access B::a).
- To accomplish this, we have to declare A as friend of B

Outline

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects
- 4 Our First class**
- 5 Destructor

Declaration and definition

- In C++ (as in C), you can separate declaration and definition
- Usually, you declare the class in a .hpp file, and put the definition (i.e. the implementation of the methods) in the .cpp file.

timer.hpp

```
class Timer {  
    int counter;  
    int level;  
    bool tr;  
public:  
    Timer(int i);  
    int getValue();  
    int getLevel();  
    bool increment();  
    void reset();  
    bool trigger();  
};
```

- Notice that the default specification is `private`
- if you want something to be public, you have to specify explicitly

Implementation of Timer

timer.cpp

Include the class declaration

```
#include "timer.hpp"

Timer::Timer(int i)
    : counter(0), level(i), tr(false)
{
}

int Timer::getValue()
{
    return counter;
}

int Timer::getLevel()
{
    return level;
}

void Timer::reset()
{
    counter = 0;
    tr = false;
}
```

Implementation of Timer

timer.cpp

```
#include "timer.hpp"

Timer::Timer(int i)
    : counter(0), level(i), tr(false)
{
}

int Timer::getValue()
{
    return counter;
}

int Timer::getLevel()
{
    return level;
}

void Timer::reset()
{
    counter = 0;
    tr = false;
}
```

Include the class declaration

Constructor: note the scope resolution

Implementation of Timer

timer.cpp

```
#include "timer.hpp"

Timer::Timer(int i)
    : counter(0), level(i), tr(false)
{
}

int Timer::getValue()
{
    return counter;
}

int Timer::getLevel()
{
    return level;
}

void Timer::reset()
{
    counter = 0;
    tr = false;
}
```

Include the class declaration

Constructor: note the scope resolution

Class initialisation list

Implementation of Timer

timer.cpp

```
#include "timer.hpp"

Timer::Timer(int i)
    : counter(0), level(i), tr(false)
{
}

int Timer::getValue()
{
    return counter;
}

int Timer::getLevel()
{
    return level;
}

void Timer::reset()
{
    counter = 0;
    tr = false;
}
```

Include the class declaration

Constructor: note the scope resolution

Class initialisation list

Method definition (we do not need to repeat that this method is public!)

Usage of the Timer

timermain.cpp

```
#include <iostream>
#include "timer.hpp"

using namespace std;

int f(int a)
{
    Timer t(10);
    while (!t.increment()) {
        a++;
    }
    return a;
}

int main()
{
    Timer ti(5);
    cout << "Before starting ti value is: " << ti.getValue() << endl;
    cout << "          ti level is: " << ti.getLevel() << endl;
    for (ti.reset(); !ti.trigger(); ti.increment()) {
        int a = f(ti.getValue());
        cout << "ti value: " << ti.getValue() << endl;
        cout << "a is: " << a << endl;
    }

    cout << "End!" << endl;
}
```

How to compile and link

- The previous program consists of three files: `timer.hpp`, `timer.cpp` and `timermain.cpp`
- To compile and execute everything:

```
g++ timer.cpp timermain.cpp -o timermain
```

- When the number of files is large, this can be annoying
- You can use an IDE, or a makefile:

```
.SUFFIXES:
.SUFFIXES: .o .cpp

.cpp.o:
    g++ -c $<

timermain: timer.o timermain.o
    g++ -o $@ $^
```

This rule automatically compiles every `.cpp` file into an `.o` file

How to compile and link

- The previous program consists of three files: `timer.hpp`, `timer.cpp` and `timermain.cpp`
- To compile and execute everything:

```
g++ timer.cpp timermain.cpp -o timermain
```

- When the number of files is large, this can be annoying
- You can use an IDE, or a makefile:

```
.SUFFIXES:
.SUFFIXES: .o .cpp

.cpp.o:
    g++ -c $<

timermain: timer.o timermain.o
    g++ -o $@ $^
```

This rule automatically compiles every `.cpp` file into an `.o` file

This rule links the object files `timer.o` and `timermain.o` into the executable file `timermain`

How to compile and link

- The previous program consists of three files: `timer.hpp`, `timer.cpp` and `timermain.cpp`
- To compile and execute everything:

```
g++ timer.cpp timermain.cpp -o timermain
```

- When the number of files is large, this can be annoying
- You can use an IDE, or a makefile:

```
.SUFFIXES:
.SUFFIXES: .o .cpp

.cpp.o:
    g++ -c $<

timermain: timer.o timermain.o
    g++ -o $@ $^
```

This rule automatically compiles every `.cpp` file into an `.o` file

This rule links the object files `timer.o` and `timermain.o` into the executable file `timermain`

This file is processed by the `make` command

How to compile and link

- The previous program consists of three files: `timer.hpp`, `timer.cpp` and `timermain.cpp`
- To compile and execute everything:

```
g++ timer.cpp timermain.cpp -o timermain
```

- When the number of files is large, this can be annoying
- You can use an IDE, or a makefile:

```
.SUFFIXES:
.SUFFIXES: .o .cpp

.cpp.o:
    g++ -c $<

timermain: timer.o timermain.o
    g++ -o $@ $^
```

This rule automatically compiles every `.cpp` file into an `.o` file

This rule links the object files `timer.o` and `timermain.o` into the executable file `timermain`

This file is processed by the `make` command

Visit the `makefile` into the example directory

- In C++, objects are treated in the same way as primitive type variables
 - Objects can be defined on the stack, hence their scope extends only to the block where they are defined
 - Object t in function $f()$ is valid only during the execution of $f()$, and its *constructor* and *destructor* are called every time the function is invoked and terminates, respectively
- This is quite different from Java:
 - in Java, when creating an object with `new` its lifetime extends until the garbage collector does not destroy it
 - In Java there is only one way of creating objects, they go on the heap
 - In C++ there are two ways of creating objects: on the stack and on the heap

Outline

- 1 Namespaces
- 2 The standard library for Input/Output
- 3 Classes and objects
- 4 Our First class
- 5 Destructor**

Destructor

- Before looking at how objects are created, let's introduce the *destructor*
- It is the reverse of the constructor
 - the constructor is called at creation time and it is used to initialise the object
 - the destructor is called at termination time and it is used for clean-up

destructor.hpp

```
class A {  
    int i;  
public:  
    A(); //  
    A(int a); //  
    ~A(); //  
};
```

Default Constructor: must have the same name of the class

Destructor

- Before looking at how objects are created, let's introduce the *destructor*
- It is the reverse of the constructor
 - the constructor is called at creation time and it is used to initialise the object
 - the destructor is called at termination time and it is used for clean-up

destructor.hpp

```
class A {  
    int i;  
public:  
    A(); //  
    A(int a); //  
    ~A(); //  
};
```

Default Constructor: must have the same name of the class

Another constructor: this has a parameter

Destructor

- Before looking at how objects are created, let's introduce the *destructor*
- It is the reverse of the constructor
 - the constructor is called at creation time and it is used to initialise the object
 - the destructor is called at termination time and it is used for clean-up

destructor.hpp

```
class A {  
    int i;  
public:  
    A(); //  
    A(int a); //  
    ~A(); //  
};
```

Default Constructor: must have the same name of the class

Another constructor: this has a parameter

Destructor: the same name of the class, with a tilde in front

Destructor

- Before looking at how objects are created, let's introduce the *destructor*
- It is the reverse of the constructor
 - the constructor is called at creation time and it is used to initialise the object
 - the destructor is called at termination time and it is used for clean-up

destructor.hpp

```
class A {  
    int i;  
public:  
    A(); //  
    A(int a); //  
    ~A(); //  
};
```

Default Constructor: must have the same name of the class

Another constructor: this has a parameter

Destructor: the same name of the class, with a tilde in front

There can be only one destructor, cannot have parameters

Example

destructor.hpp

```
class A {
    int i;
public:
    A(); //
    A(int a); //
    ~A(); //
};
```

destructor.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

A::A() : i(0)
{
    cout << "default constructor of A" << endl;
}

A::A(int a) : i(a)
{
    cout << "constructor of A(" << i << ")" << endl;
}

A::~~A()
{
    cout << "Destructor of A (i=" << i << ")" << endl;
}
```


Example - II

desmain.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                << #x << endl

void f()
{
    A a;
    WH(f);
}

void g()
{
    A b(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A c(2);
    WH(main);
    g();
    WH(main);
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
Destructor of A (i=5)
now inside main
Destructor of A (i=2)
```

Pointers to object and new

- This is how you can define a pointer to an object

```
A a;  
A *p = &a;
```

- How you can see, it is not different from regular variables
- `a` is an object defined on the stack or in global memory; to create an object on the heap:

```
A *p = new A();
```

- The previous code:
 - Allocates the right amount of memory on the heap for an object of type `A`
 - Calls the constructor for initialising the object
 - returns a pointer to the allocated memory, and assigns it to `p`
- Similar to Java, except that in C++ `new` returns a pointer

Freeing the memory with delete

- In Java the memory is freed by the garbage collector
- In C++ there is not such a thing:
 - It is the responsibility of the programmer to free the memory
- The memory can be freed with `delete`

```
A *p = new A();  
...  
delete p;
```

- `delete` must be followed by a pointer
 - It calls the destructor for the object
 - then deallocates the memory

Example with pointers

desmain2.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                  << #x << endl

void f()
{
    A *pa = new A();
    WH(f);
    delete pa;
}

void g()
{
    A *pb = new A(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A *pc = new A(2);
    WH(main);
    g();
    WH(main);
    delete pc;
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
now inside main
Destructor of A (i=2)
```

- mmm, maybe something is missing?

Example with pointers

desmain2.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                  << #x << endl

void f()
{
    A *pa = new A();
    WH(f);
    delete pa;
}

void g()
{
    A *pb = new A(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A *pc = new A(2);
    WH(main);
    g();
    WH(main);
    delete pc;
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
now inside main
Destructor of A (i=2)
```

- mmm, maybe something is missing?
- This is called “memory leak”

Example with pointers

desmain2.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                << #x << endl

void f()
{
    A *pa = new A();
    WH(f);
    delete pa;
}

void g()
{
    A *pb = new A(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A *pc = new A(2);
    WH(main);
    g();
    WH(main);
    delete pc;
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
now inside main
Destructor of A (i=2)
```

- mmm, maybe something is missing?
- This is called “memory leak”
- The memory pointed by `pb` is lost!
Cannot be deallocated anymore

Example with pointers

desmain2.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                << #x << endl

void f()
{
    A *pa = new A();
    WH(f);
    delete pa;
}

void g()
{
    A *pb = new A(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A *pc = new A(2);
    WH(main);
    g();
    WH(main);
    delete pc;
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
now inside main
Destructor of A (i=2)
```

- mmm, maybe something is missing?
- This is called “memory leak”
- The memory pointed by `pb` is lost!
Cannot be deallocated anymore
 - Why?

Example with pointers

desmain2.cpp

```
#include "destructor.hpp"
#include <iostream>
using namespace std;

#define WH(x) cout << "now inside " \
                << #x << endl

void f()
{
    A *pa = new A();
    WH(f);
    delete pa;
}

void g()
{
    A *pb = new A(5);
    WH(g);
    f();
    WH(g);
}

int main()
{
    A *pc = new A(2);
    WH(main);
    g();
    WH(main);
    delete pc;
}
```

Output:

```
constructor of A(2)
now inside main
constructor of A(5)
now inside g
default constructor of A
now inside f
Destructor of A (i=0)
now inside g
now inside main
Destructor of A (i=2)
```

- mmm, maybe something is missing?
- This is called “memory leak”
- The memory pointed by `pb` is lost!
Cannot be deallocated anymore
 - Why?
- Remember: there is not garbage collector to save us!