**Constructors**

An object of Derived class does not have Constructos, Destructors, Copy & operator=

**If these functions are not defined in Derived (or does not call the correct corresponding Base function), the default behavior would be to call the parent's corresponding version 🡪 ERROR**

|  |
| --- |
| * **Constructor |** *Derived::Derived(int n)* will call *Base::Base()* before itself by default * **Destructor |** For destructors, *Base::~Base()* is automatically called after *Derived::~Derived()* --> we don't need to explicitly call ~Base() * **Copy Constructor |** For *Derived::Derived (const Derived & original) { }* , the default behavior will call *Base::Base()*--> ERROR * **operator= |** *Derived::operator=(const Derived & rhs)* (no default behavior) |

1. **Constructors**

The child constructor will **always** **call a parent constructor before itself**

* If you don't explicitly call any base constructor, the default base constructor is called when initializing a derived class object.
* If we simply define a child constructor, it will call default parent constructor before itself (there will be an error if there's no default parent constructor while there is a non-default parent constructor)(compiler will create a default constructor if NO constructor is defined for parent)

Warrior (string &n) { // this will call default parent constructor

//…something

}

* Non-default constructors are "NOT" inherited. Say Base::Base(int n) is defined, then Derived::Derived(int n) will be an error.
  + Before child's constructor, we can call a non-default constructor of a parent's class (parameters passed by derived class constructor)

Warrior (string &n) : **Player(n)** { // … additional children constructor }

|  |
| --- |
| \*\*\* Note that in the example, calling the parent constructor Parent() will "start" the name and age of the Parent, for which the Children can access the age (14), but still not the name.  The access control holds (Children can access protected and public only) |

**Syntax:** Instead of initializing member variables in brackets, we can do with the syntax

🡪 Allows passing in the parameters to the constructors of based class

Employee::Employee(string n, int s) : name (n), sin(s) { /\* deliberately left empty \*/ }

Manager::Manager(string n, int sin, int \_level) : Employee(n, sin), level(\_level) { /\*empty\*/ }

**Destructors**

The child destructor will **always call a parent destructor after itself - automatically called**

Like constructors, we can also override the child's decstructor as well.

The order of destructor invocation: ~Son() -> ~Dad() -> ~Grandpa()

**Copy Constructor**

Deep copy base class member variables:

1. Call **base class** copy function (This is not done if we do shallow copy!)
2. Deep copy extra member vars contained in derived class

Base::Base (const Base & original) {

// Deep copy **original** into **\*this**

}

Derived::Derived (const Derived & original) **: Base(original)** {

// Deep copy extra Derived vars

}

\*\*\* You have to call Base(original) yourself. If not, the **Base::Base()** will be invoked (serious logical error

🡪 **ERROR!**

**Operator=**

Base & Base::operator= (const Base & rhs) {

// Deep copy

}

Derived & Derived::operator= (const Derived & rhs)

{

**Base::operator=(rhs);**

// Deep copy any extra Derived variables

return (\*this);

}

\*\*\*Note that by default, operator= does not call Base::operator=. However, you don't have to call Base::operator= to avoid compile error (but it might be a logical error)

\*\* If no operator= or copy constructor in Derived class is defined, then the default behavior is the operator=/copy of the Base (or the default of Base, if not defined)