Constructors and Destructors

Class Specifies Subroutines for Initialization & Teardown

Specific subroutines

- for initialization and teardown
- of class instances
- can be specified in the class’ definition.

Constructor is Always Called for an Instance

A constructor is a subroutine
- for initialization of an instance,
- and is always called for instances.

When is a constructor called?

Automatic variables: at point of declaration
Static variables: before main
(order is difficult to control)
Dynamic variables: at point of allocation

Constructors Share a Few Common Aspects

Common aspects of constructors:
- name: the name of the class
- return type: none (not void)
- overloading is allowed
  - (multiple versions can be defined)
  - compiler tries to choose based on arguments passed
Constructor with No Arguments Used for Arrays

Two public constructors are created by default.

1. Constructor with no arguments
   - created only if no constructors declared in class definition
   - used for initializing array elements
   - arrays of a class’ instances are not allowed if no constructor with no arguments exists

Operation of Default Constructor with No Arguments

What does the default constructor with no arguments do?

Initializes each field based on its type.

Fields that are not instances are left as bits (as with C).

Fields that are instances (of another class)
   - are initialized using the constructor with no arguments for their class,
   - which must exist and be accessible.

Copy Constructor Also Created by Default

2. Copy constructor
   - also created by default (and made public)
   - takes one argument: a reference* to another instance of the same class
     MyClass m;
     MyClass n = m; // used here
     MyClass p (m); // also used here
     p = n; // not used here!

*Briefly for now: a reference looks like an instance syntactically, but is implemented as a pointer.

Operation of Default Constructor with No Arguments

What does the default copy constructor do?

Initializes each field based on its type.

Fields that are not instances are copied as bits (as with C).

Fields that are instances (of another class)
   - are copied using the copy constructor for their class,
   - which must exist* and be accessible.

*A default copy constructor is not available (said to be “deleted”) if this condition is not met.
Example of Constructor Declaration in a Class Definition

Let’s do an example. Here’s a class:

class MyClass {
    private:
        int32_t one;
        int32_t two;
    public:
        MyClass (int32_t arg, int32_t second);
};

Arrays Require a Constructor with No Arguments

A question: given the class just shown, is the following code acceptable?

MyClass m[42];

No!

◦ No constructor without arguments is declared,
◦ nor is the default produced,
◦ since a constructor is declared.

Example of Constructor Declaration in a Class Definition

How does the code look?

In a source file somewhere...

MyClass::MyClass (int32_t arg, int32_t second);
    : one (arg), two (second) {
        // code
    };

Initializers Describe Base Class and Fields Initialization

What’s an initializer?

A brief specification for initialization
◦ consisting of a base class or field name and
◦ an arbitrary expression in parentheses
◦ (function calls are allowed).

For base classes and instances, lists of expressions are passed to constructors.
Order of Initializer Execution Depends on Class Definition

Order of initializers does not affect code, but should match order of execution:

1. Base class(es), in order of derivation list:
   ◦ if a class does not appear in the list,
   ◦ constructor with no arguments is called.

2. Fields, in order listed in class definition
   ◦ if a field that is an instance does not appear in the list,
   ◦ constructor with no arguments is called.

Use Initializers, Not Code, to Initialize Instances

Initializers are executed BEFORE the constructor's code.
Thus, when constructor code starts,
   ◦ all base classes have been initialized
   ◦ all fields that are instances have been initialized

Avoid re-initializing instances!
If code is needed to initialize a field, make the field a pointer and dynamically allocate an instance after the necessary code.

Destructor is Usually Called for an Instance

A destructor is a subroutine
   ◦ called to destroy (teardown) an instance,
   ◦ and is usually called for instances.

When is a destructor called?

Automatic variables: at end of scope / use
Static variables: after main
   (order is difficult to control)
Dynamic variables: at point of deallocation

Destructors Not Called in Certain Cases

When is a destructor not called?
Abnormal/unusual program termination,
   ◦ such as crashes (for example, due to SEGV signals or division by zero) and
   ◦ calls to exit.

Dynamically allocated instances that are not deallocated (deleted).*

*Bad habits (not freeing things) in C can be dangerous in C++.
**Important Details About Destructors**

Always define a destructor, and make it virtual.

Why virtual?

class MyClass : public ParentClass ...
// dynamic allocation and deallocation
ParentClass* p = new MyClass;
delete p; // which destructor?
If ParentClass’ destructor is not virtual, the answer is, “The wrong one.”

---

**Example of Destructor Declaration in a Class Definition**

Destructors
- are named ~ followed by the class name,
- take no arguments, and
- return nothing.

So we have …

```cpp
class MyClass {
    virtual ~MyClass();
};
```

---

**Operation of Destructors**

Destructors do the following…

1. **Execute the body** of the destructor.
2. **Call destructors for all fields** that are instances, in reverse order of declaration in the class definition.
3. **Call destructors for base classes**, if any.
4. (For dynamically-allocated instances, the deallocation happens here.)

---

**Destructor Bodies Must Perform Deallocation**

Do not call destructors for fields that are instances; such calls are made automatically.

Fields that are pointers to instances are not implicitly destroyed.

If a field
- points to an instance
- that should be destroyed
- when the instance containing the field is destroyed,
- the destructor body (the code) must perform that deallocation explicitly.
class MyClass :
    public ParentClass {
        int x;
        AnotherClass y;
        double z;
        // ...
    }