Object-Oriented Programming

# Constructors and Initialization

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

*The purpose of the following code examples is to demonstrate the semantics and behavior of the C++ language. They may not be complete and are not necessarily structured as they should be in actual code. Please adhere to all coding guidelines in your work.*

## Notice

- Declare methods only in the header (*.hpp*) file

- Define methods only in the implementation (*.cpp*) file

- This also applies to *constructors* and *destructors*

# Object Construction in C++

```
{
    A a;
}

{
    A* pa = new A();

    delete pa;
}
```

- Memory for the object is allocated from the *stack* or the *heap*

- Memory from the *heap* is allocated using the operator new

- The *constructor* initializes the allocated memory

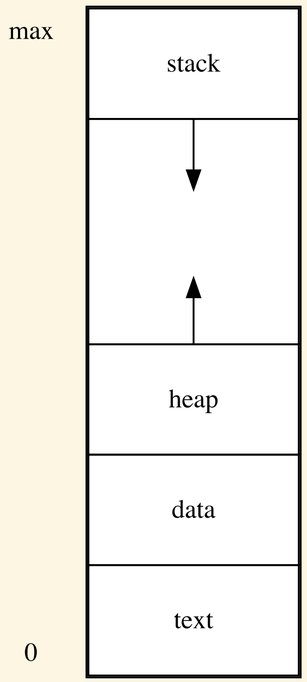
- The *constructor* does **not** allocate memory

# Memory

```cpp
/*
    memory.cpp

    Declarations and where they are placed in memory
*/

#include <iostream>
#include <cassert>

int main() {

    int n1;
    std::cerr << "n1: " << &n1 << '\n';
    int n2;
    std::cerr << "n2: " << &n2 << '\n';
    int n3;
    std::cerr << "n3: " << &n3 << '\n';

    assert(&n1 > &n2);
    assert(&n2 > &n3);

    int* p1 = new int(5);
    std::cerr << "p1: " << p1 << '\n';
    int* p2 = new int(5);
    std::cerr << "p2: " << p2 << '\n';
    int* p3 = new int(5);
    std::cerr << "p3: " << p3 << '\n';

    assert(p1 < p2);
    assert(p2 < p3);

    return 0;
}
```

```
$ ./memory
n1: 0x16d54eeb8
n2: 0x16d54eeb4
n3: 0x16d54eeb0
p1: 0x1030a1a90
p2: 0x1030a1aa0
p3: 0x1030a1ab0
```

# Memory Allocation and the Constructor




- For the *stack*:
  1. The program allocates memory from the top of the *stack*
  2. The *constructor* is called on the allocated memory in the stack

- For the *heap*:
  1. The operator new allocates memory from the *heap*
  2. The *constructor* is called on the allocated memory from the heap

# Object Memory Allocation

```
class B {};

class B2 {};

class A {
private:
    B b;
    B2 b2;
};

int main() {

    static_assert(sizeof(B) == 1, "Not sizeof(B) == 1");
    static_assert(sizeof(B2) == 1, "Not sizeof(B2) == 1");
    static_assert(sizeof(A) == 2, "Not sizeof(A) == 2");

    return 0;
}
```

- Memory for the entire object is allocated first, including memory for all fields

- For an empty class, the size is always 1 to ensure that the addresses of two different objects will be different

- The size of an object is the sum of the size of the fields plus the padding

- Memory for fields is constructed first

- Must know the fields of a class to know how much memory to allocate

# Object Constructor Order

```
class B {
public:
    B() { debugOutput.push_back(__FUNCTION__); }
};

class B2 {
public:
    B2() { debugOutput.push_back(__FUNCTION__); }
};

class A {
public:
    A() { debugOutput.push_back(__FUNCTION__); }

private:
    B b;
    B2 b2;
};

int main() {

    A a;
    assert(debugOutput[0] == "B");
    assert(debugOutput[1] == "B2");
    assert(debugOutput[2] == "A");
    return 0;
}
```

- Constructors for all fields are called before the constructor for the object

- Constructors are called in the order they appear in the class definition

# Member Initialization Lists

```
// Note: Class all in one file for demonstration only

class A {
public:
    A()
        : b(0), b2(1)
    {}

private:
    B b;
    B2 b2;
};
```

- Part of the constructor
- Often ignored until after it is needed
- The only way that we can control what *constructor* is called for the fields of the class
- Syntax is a little strange but necessary

# Non-Static Member Initialization

```
class A {
public:
    A();

private:
    bool flag = false;
    int factor = 2;
}
```

- Introduced in C++11
- Some limits to what types can be initialized and with what expressions
- It does help that you won't forget to initialize
- It does hurt that you cannot have initialization based on constructor parameters
- It does hurt that it exposes implementation details
- Use when appropriate

# Good Practice

- Don't wait for a problem to occur

- Use the member initialization list as much as possible

- Use non-static member initialization when appropriate