Object-Oriented Principles and Practice / C++

Alice E. Fischer

May 4, 2015

Storage Class

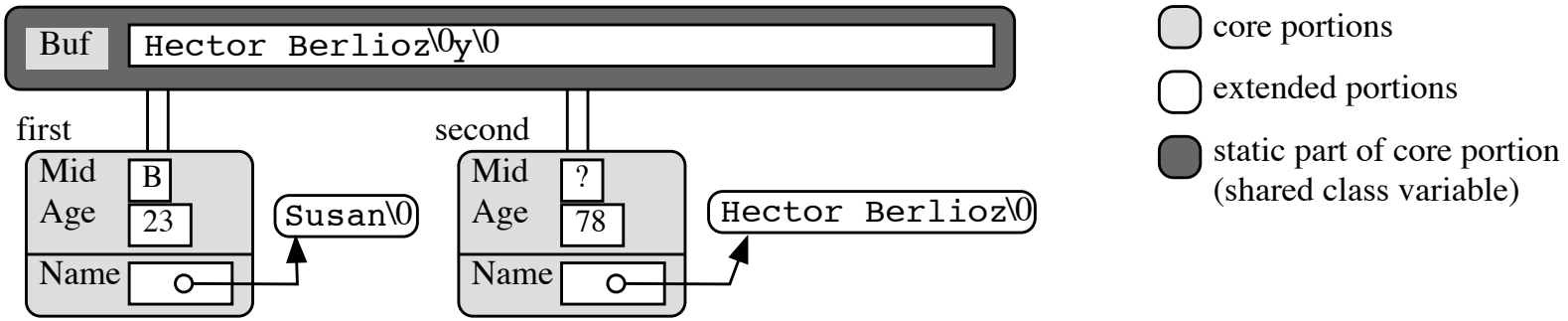
C++ provides 3 different memory areas, with different lifetimes.

- Static objects are stored in a separate static area that is allocated and initialized at load time. These objects are deallocated when the program terminates.
- Dynamic objects are created by calling `new`. The result is a pointer that must be stored somewhere. These objects are deallocated when the program calls `delete` or when it terminates.
- Automatic objects are allocated and initialized by declaring them in some function or globally. These objects are deallocated at the end of the block that declares them.

An object can have parts that are managed in all 3 ways.

Storage Class

Chapter 6



Dynamic Allocation

Every call on the dynamic allocation function `new` incurs overhead.

- The allocation functions `malloc` and `new` must calculate and store bookkeeping information for later use by `free` or `delete`.
- The system must know how many bytes are in each dynamically allocated storage block. This could be stored as part of the block, computed from the sizeof the base type, or kept in a hidden table, indexed by the first address of the block.
- In order for `delete[]` to work, the actual length of every array must be stored as part of the array.

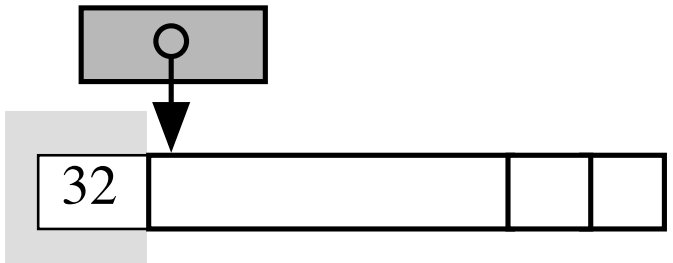
Dynamic Allocation

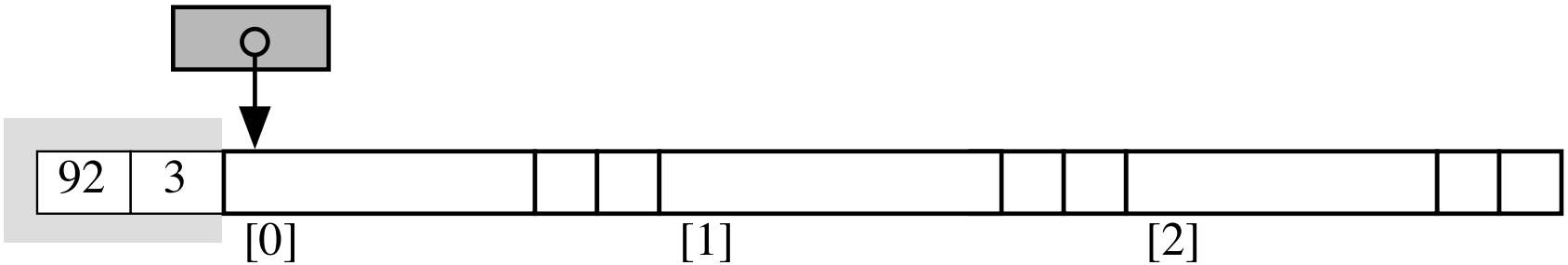
The C++ language standard does not require any particular implementation for this overhead. However, every compiler must do something of this sort. Here, we illustrate an old and common strategy, which has been used for C++.

- Extra space (a long int) is allocated at the beginning of every dynamically allocated area and initialized to the actual number of bytes in the allocation block, including the overhead. This is used to free the block.
- Following that, another long int is allocated for every array, giving the number of slots in the array. This is used to free objects stored in the array.
- After that is the array itself, with possible interior padding.

Allocation of Dynamic Objects

The object in this picture requires 28 bytes of memory.





Efficiency

C++ allows us to create objects by composing stack-based objects or by aggregating dynamic objects.

- Dynamic allocation makes less efficient use of space because of the overhead.
- It slows down execution because of the extra memory references to dereference the pointers.
- However, it lets a program respond to run time events.
- A program designer must decide how much runtime flexibility is needed and how much he cares about unnecessary use of time and space.
- Simply put, don't use dynamic allocation unless it is needed.

A Pointer Problem

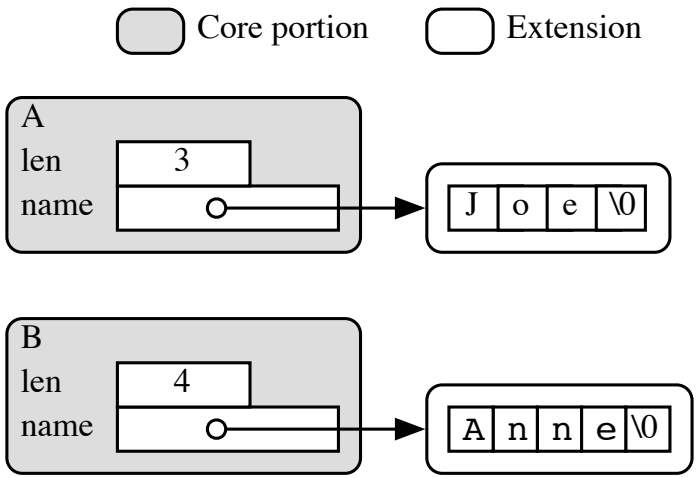
Consider a class named `Mixed` with both core parts and extensions.

- The extensions are allocated and initialized by the constructor.
- The destructor deallocates the extensions.
- Simple, but now there is a problem: this object cannot be used as a call-by-value parameter.
- Passing it to a function is fine: a copy of the core portion is made, including a copy to the extension, but not a copy of the extension.
- When the function returns, the copy AND THE EXTENSION is deleted.
- The original object is now dysfunctional.

Shallow Copy

Suppose A and B are both Mixed objects. .

```
class Mixed {
    int len;
    char* name;
public:
    Mixed( char* nm ) {
        len = strlen( nm );
        name = new char[ len+1 ];
        strcpy( name, nm );
    }
    ~Mixed() { delete[] name; }
}
```



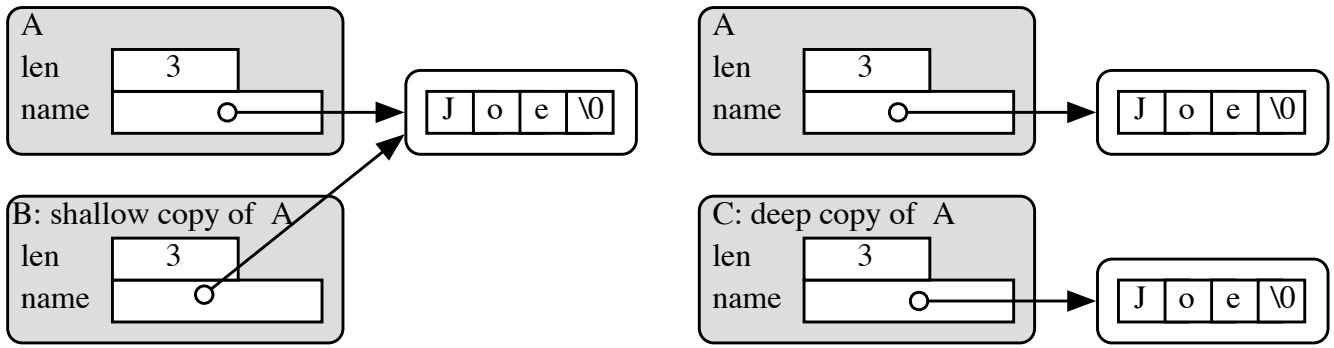
Deep Copy

C++ lets you redefine call-by-value to use a *deep copy*.

- This is done by defining a *copy constructor* that makes a complete copy of every extension.
- Some resources tell you to define copy constructors to make your code work.
- THIS IS NOT the right answer. Can you imagine the performance problems you would have if you made a complete copy of every data structure every time you used it as a parameter to a function?
- This problem is fixed by using call-by-reference instead of call-by-value.

Shallow vs. Deep Copy

Now execute B = A using the predefined copy constructor.



In both cases, the object containing the name "Anne" has become a memory leak because there are no longer any pointers pointing at it. C++ does not collect the garbage you generate.

Construction and Destruction – Ch 10

Allocation and Deallocation: Review
Initializing Arrays
Six kinds of Constructors
But just one Destructor

Allocation and Deallocation – Ch 10.2

The new function in C++ builds upon C's malloc.

- Using new or malloc results in the storage you requested + enough to handle deallocation
- Minimally, the total allocated block length must be stored
- Thus, allocating a `new double` uses up 12 bytes of memory, 4 for the block length and 8 for the data.
- For a struct, it is the overhead + enough bytes to hold the data + padding.
- New and malloc differ for arrays because new also allocates memory for the array length. (A total of 8 bytes of overhead)

Initialization

C++11 gives us new ways to initialize arrays.

```
class Tester {
  private:
    static int a0[3];
    static constexpr int a1[3]{1,2,3};
    int a2[3] = {8};
    int a3[3];
    int* a4;
    int* a5 = new int[3]{5,6,7};
  public:
    Tester(): a3(), a4(new int[3]{7,8,9})  {}
}
const int Tester::a1[3];
      int Tester::a0[3]{4,5,6};
```

Six Kinds of Constructors

- Null constructor: initializes nothing but may print a trace comment. You get an empty null constructor for free if you do not define a real constructor.
- Default constructor: may do initializations but lacks parameters.
- Normal constructor: has parameters used to initialize the object.
- Default copy constructor: initializes an object by shallow-copying another.
- Copy constructor: replaces the default copy constructor. It can do anything, including making a full copy of the object.
- Move constructor: moves the contents of one object to another during initialization.

Several Constructors but just one Destructor

- A destructor must handle every class object the same way. Thus, every object must be the same general shape and have the same number of dynamic parts (or NULL pointers) to delete.
- According to the standard, it is harmless to delete a NULL pointer. Nothing is supposed to happen. You do not need to test a pointer before trying to delete it. However, apparently, some implementations of C++ do not comply to the standard.
- Normally, a destructor should free all objects that were created by calling `new` in the constructor or in any other class function, or by inserting an object (created elsewhere) into a class data structure.

Revisiting delete and delete[]

- For non-arrays, the block length is stored just before the data part (just like malloc)
- For arrays, the array length is stored before beginning of the data area and the block length is stored just before the array length.
- When you call `delete` it uses the block length to free memory.
- When you call `delete[]` it does two things:
  - First, the array length is used to control a loop that calls delete on each array element.
  - Then the block length is used to free the array allocation.

Construction and Destruction

RValue References
Default Class Operations: C++ 11
Control of Defaults: C++ 11

RValue References

An rvalue reference can bind to an rvalue (but not to an lvalue):

- ```
  X a;
  X& r1 = a;        // bind r1 to a (an lvalue)
  X&& rr1 = f();    // bind rr1 to the result of f().
  X&& rr2 = a;      // error: a is an lvalue
  ```
- This idea can be used to speed up execution of any operation that moves values around.
- If X is a type for which copying is expensive (string, vector) a simple swap becomes an expensive operation.
  ```

RValue Swapping

But when we swap, we don't really want new copies at all. We just want to rebind the existing copies.

- ```
  void swap(T& a, T& b) {
  T tmp = move(a);        // Save a's value.
  a = move(b);            // Move b's value to a.
  b = move(tmp);          // Move the saved value to b.
  ```
- Moving is faster than copying for many types because it does not construct a new object.
- This is very important when you design a template because the template parameter could be any primitive or class type.
  ```

Default Class Operations: C++ 11

By default, a class has these five related operations:

- destructor: By default, a null destructor.
- copy constructor: `X(const X&)`
  Initialize a new object from an old one of the same type. The contents of the source are unchanged.
- copy assignment: `Y& operator=(const Y&)`
  Copy one object into another of the same type. The contents of the source are unchanged.
- move constructor: `X(const X&&)`
  Initialize a new object by moving the contents of an old object into it. After the move, the contents of the source are set to the initial constructed state.
- move assignment: `Y& operator=(Y&&);`
  Move one object into another.

Default Class Operations: C++ 11

Copy, move, and delete are closely related operations.

- You can redefine all of them,
- but only a few combinations make sense.
- If you declare any of them you must explicitly define or default all the others.
- If you define any one, movers will not be generated automatically. Copiers will be generated automatically, but this is deprecated.
- Move constructor and move assignment takes non-const &&. They can, and usually do, write to their argument

Control of Defaults: C++ 11

The keywords delete and default and can be used to define methods.

- ```
  class X {
      // These definitions disallow copying.
      X& operator=(const X&) = delete;
      X(const X&) = delete;
  };
  ```
- ```
  class X {
      // These define the default copy behavior.
      X& operator=(const X&) = default;
      X(const X&) = default;
  };
  ```