06 - Memory Management

January 23, 2023

COMP2404

Darryl Hill

Contents

Stack and Heap

OS allocates 4 areas of memory on program start-up

- ***code segment*** or ***text segment***
  - Stores the program instructions
- ***data segment***
  - Contains global memory
    - global data members, string literals, etc
- ***function call stack***
  - Variables and data associated with current function
    - ***Stack frames***
- ***heap*** - part of the data segment
  - Dynamically allocated memory

Stack and Heap

The ***Code Segment*** contains

- program instructions
- function addresses

The ***Data Segment*** contains

- global variables
- static variables
- literals

Stack and Heap

The ***Function Call Stack***:

- keeps track of functions called and the order in which they were called
- stores local variables
- stores function parameters
- function meta-data

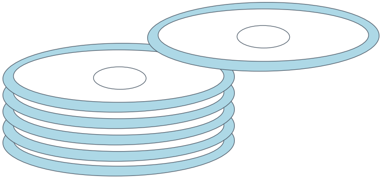
The ***Heap***:

- is part of the data segment
- stores all dynamically allocated memory
  - memory the programmer manages
  - more on this soon

Function Call Stack

A ***Stack*** is a type data structure

- collection of related data
- last in, first out (LIFO)
- similar to a pile of dishes



- adding an item is referred to as ***pushing*** the item
- removing an item is referred to as ***popping*** the item

Function Call Stack

The ***Function Call Stack*** is a data structure to keep track of function calls.

- When a function is called, a ***stack frame*** is pushed onto the call stack.
- When a function returns, the ***stack frame*** is popped off.

A ***stack frame*** contains:

- Local variables contained in the function parameters.
- The return address of the next instruction in the calling function.

The Heap

The ***Heap*** is an area of memory used for **dynamic allocation** (more on this next slide).

- Allocation that happens at runtime, using the `new` keyword.
- This is memory that is managed by the programmer.
  - We must `delete` it at some point.

Figure: "The heaping pile o' mail again" by Charles Williams is licensed under CC BY 2.0

Memory Allocation
Two Types

Statically allocated memory:

- Memory is reserved **where the variable was declared**.
  - Either on the function call stack or in a class.
  - Variables (of any type) are simply declared
  - `int x; Student stu;`
- This memory is deleted for us when the containing object is ***deleted*** or stack frame is popped off.

Dynamically allocated memory:

- Memory is reserved **on the heap**.
  - Uses the `new` keyword which returns a pointer (memory address from the heap).
  - `int* x = new int; Student* stu = new Student;`
- We must `delete` this memory explicitly.

Pointers

A ***Pointer*** is a variable that stores a ***memory address***.

- The address of a variable, or
- the start of a block of dynamically allocated memory (an array).

Why use pointers?

- They have a small, fixed size – 8 bytes.
  - Often smaller than the data it points to.
  - Easier to pass (the pointer) by value.
- They are an easy way to avoid copying data.
  - C++ can make many needless copies.
- Pointers are ***necessary*** for dynamically allocated memory.
  - `new` returns a pointer.

Pointers

Like all variables, pointers have a:

- Name
- Data type – the type of data being pointed to.
  - Tells the compiler how to interpret the data found at that memory location.
- Value
  - This value is an address in memory.
  - This address must contain a variable of the type associated with the pointer.
- Location in memory
  - Pointers have their own memory addresses.

Pointers

Memory Map

Name	Type	Value	Location

A8: char c

A9: int i

AA: char* cptr

Pointers

- In 64-bit architecture, pointers are 64-bits (8 bytes) of memory.
- Pointers may point to a variable of
  - any data type (but each pointer is restricted to one type), and
  - any size.
- Thus we can "pass" a variable into a function by copying 8 bytes of data, irrespective of the size of the variable.

Coding example <p1>

Declaring Pointers

Variable declaration is a statement specifying variable's name and data type.

A pointer variable declaration has 3 components:

- The type of data being pointed to,
- the * symbol to indicate that it is a pointer, and
- the variable name.

Assigning Values to Pointers

Pointers can receive a value of

- the memory address of a variable, or
- a system call for a new block of memory.

The & symbol is the "address-of" operator

- It returns the memory location of the operand variable.

```
int b;
int* a = &b;
int* c = new int;
```

Accessing Pointer Values

The * symbol:

- is the ***dereferencing*** operator
- returns the value stored at the memory address pointed to by the operand

`NULL` pointer exception:

- this happens when dereferencing a pointer that is set to `NULL`.
  - the pointer value is 0, so we try and access memory address 0

Good programming practice to check for `NULL` pointers

- same as in Java
- also good practice: set garbage pointers to `NULL` so that you may `NULL`-check them

Pointer Operators

Symbols * and & have multiple roles

In variable declarations:

- * indicates a pointer
- & indicates a reference

Anywhere else

- * is dereferencing
- & is address-of

Both have another meaning as a math and bit operator

Both are overloaded symbols, so pay attention to context!

Parameter Passing

Pass-by-value:

- this makes copies of the values of the variables
- can be very inefficient (though it is ok for primitives)
- does not use the variable address

Pass-by-reference has 2 forms:

- using references
- using pointers

Java does pass-by-reference, except for primitives which are pass by value

- Programmer has no choice in this

Coding example <p2>

Memory Allocation

Memory allocation:

- reserving a specific number of bytes in memory based on data type

Static Memory Allocation

- done by declaring a variable
- memory reserved wherever it is declared
- memory is deallocated when the stack frame that contains it is popped off or containing object is deallocated

Dynamic Memory Allocation

- memory is reserved on the heap using the `new` keyword
- memory is deallocated by using `delete` keyword

coding example <p3>

Memory Leaks

C/C++ does not provide garbage collection

- we can have memory leaks

What is a memory leak?

- a block of dynamically allocated memory with no pointers to it
- can never again be used by the program (and while the program is running, no other programs can use it either)
- cannot call `delete` on it

What causes memory leaks?

- a pointer gets **clobbered** – overwritten
- a pointer moves out of scope

Memory Leaks

Why are memory leaks a problem?

- access to data permanently lost
  - finite amount of heap space given to the program

Dynamically allocated memory is released

- once the program terminates
- explicitly released with the `delete` keyword

Industrial grade software should run for months or years

- if code repeats (and most code does), the memory leak will repeat
- could run out of heap space

coding example <p3> redux

Preventing Memory Leaks

Always explicitly deallocate memory

- for each `new` there should be a matching `delete`
- use valgrind utility in Linux to check

"Where is this object deleted?" is something you should ask yourself again and again.

Good rule of thumb: objects are deleted where they are created

HOWEVER

- occasionally created in one place and deleted in another
- be very careful doing this
- `Factory` design pattern is one example
- Often requires tight coupling between classes or functions
- Advanced design

Preventing Memory Leaks

Every dynamically allocated object must be `deleted` in **exactly one place**

- Often deleted by the containing object (not a data structure) or the declaring function.
- Containers often form a "hierarchy of deletion responsibility"

Also, make sure you don't clobber a pointer into the heap

- Each piece of data should have a primary reference.
  - This reference should exist for the lifetime of the object
- No garbage collection or reference counting in this class
- Even with these "memory helpers", if you do not understand memory fundamentals, garbage collection or deallocation can occur at the wrong time
  - This will cause a slow-down

Dynamic Allocation of Objects

Using `new` calls the object **constructor**

- `malloc` does not call the constructor- DO NOT USE
- parameters to the constructor can be specified
- memory is allocated on the heap
- member variables initialized

Using `delete` calls the object **destructor**

- `free` does not call the destructor - DO NOT USE
- resources are cleaned up
- memory is released

coding example <p4>

Returning Objects by Reference - Double Pointers

Sometimes we want to return an object by reference

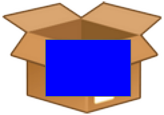
- Using return parameters
- Often from the heap, but can be from a (previous) stack frame
- This parameter should return the ***address*** of this object
- Usually in the form of a ***pointer***

To do this we pass ***in*** a ***pointer*** by ***reference***.

Double Pointers

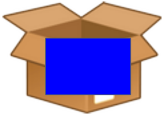
Memory Map

Name	Type	Value	Location



A8: int i

AC: int* iptr



B3: int** dptr

Retrieving Dynamic Objects - Double Pointers

Double pointer syntax: (coding example <p5>)

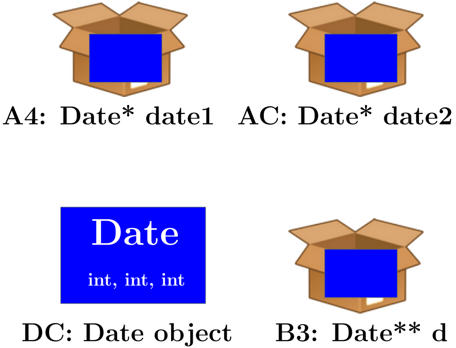
```
int main(){
    Date* date1;
    getDate(&date1);
    date1->print();    \\ happy new year!
}

\\ this will return a pointer
\\ to a dynamically allocated object
void getDate(Date** d){
    Date* date2 = new Date(2022,1,1);
    *d = date2;
}
```

Do **NOT** declare a double pointer. Declare a pointer and pass in the address.

Double Pointers

Memory Map

Name	Type	Value	Location



Dynamically Allocated Arrays

Arrays are dynamically allocated with the `new[]` operator

- if you make an array of objects, the default constructor is called for each of these objects
  - if it exists
  - if not, you will get an error

Dynamic arrays can be deallocated with the `delete[]` operator

- if it is an array of objects, the destructor is called for every element of the array

Using `delete` without brackets:

- may not compile
- might result in undefined behaviour
- might work ¯_(ツ)_/¯

Arrays

Arrays also have two types of memory allocation:

- static
- dynamic

In addition, arrays can store

- objects or
- object pointers

That means there are 4 ways of storing data in arrays

coding example <p6>

Allocating Arrays

Statically allocated array of objects:

- `Date dates[size];`
- `dates[0].print();`

Dynamically allocated array of objects:

- `Date* dates = new Date[size];`
- `dates[0].print();`

Statically allocated array of pointers:

- `Date* dates[size];`
- `dates[0] = new Date;`
- `dates[0]->print();`

Dynamically allocated array of pointers:

- `Date** dates = new Date*[size];`
- `dates[0] = new Date;`
- `dates[0]->print();`