Section 1.6
Memory Management

1. Stack and heap
2. Pointers
3. Memory allocation
4. Dynamic arrays

1.6.1 Stack and Heap

- OS allocates 4 areas of memory on program start up
  - code segment (also known as text segment)
    - program instructions
  - data segment
    - global memory
  - function call stack
    - local data
  - heap (part of the data segment)
    - dynamically allocated memory

©2018 Christine Laurendeau

Stack and Heap (cont.)

- Code segment
  - program instructions
  - addresses of functions
- Data segment
  - global variables
  - static variables
  - literals

©2018 Christine Laurendeau

Stack and Heap (cont.)

- Function call stack
  - it manages the order of function calls
  - it stores local variables, including parameters
- Heap
  - it is part of the data segment
  - it stores all dynamically allocated memory
    - memory allocated at runtime
    - ... more on this later...

©2018 Christine Laurendeau

Function Call Stack

- What is a stack?
  - it’s a data structure
  - a collection of related data
- Stack data structure
  - analogous to a pile of dishes
  - order is last-in, first-out (LIFO)
    - last item added (pushed) is first item removed (popped)

©2018 Christine Laurendeau

Function Call Stack (cont.)

- What is the function call stack?
  - it is used to manage the function-call-and-return mechanism
    - when function is called, a *stack frame* is created and pushed
    - on function return, the corresponding stack frame is popped
- Function call stack contains:
  - local variables
  - function parameters
  - return address back to the calling function

©2018 Christine Laurendeau

Heap

- What is the heap?
  - it is a block of memory used for *dynamic memory allocation*
- Types of memory allocation
  - static memory allocation
    - happens at compile time
  - dynamic memory allocation
    - happens at runtime

©2018 Christine Laurendeau

1.6.2 Pointers

- What is a pointer?
  - it's a **variable** that stores a memory address
  - it's used for storing, as its **value**:
    - the memory address of another variable, or
    - the memory address of a block of dynamically allocated memory

©2018 Christine Laurendeau

Pointers (cont.)

- Why use pointers?
  - a pointer is a small, fixed size variable
    - it is usually smaller than the variable it points to
  - it allows changes to areas of memory out of scope
    - example: modifying variables in the calling function
  - it avoids copying data
    - C++ is notorious for automatically making needless copies
      - multiple copies of the same thing
      - temporary copies
  - it’s the only way to use dynamically allocated memory

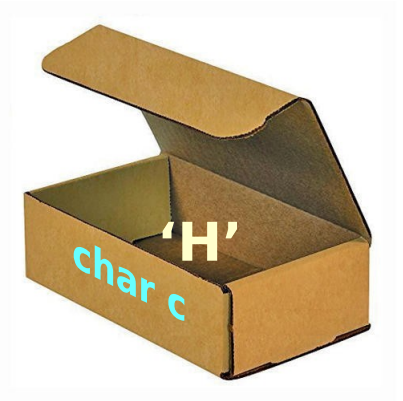
©2018 Christine Laurendeau

Pointers (cont.)

- Like all variables, pointer variables have:
  - a name
  - a data type
    - data type that the pointer points to
    - the * symbol
  - a value
    - this must be an address in memory
    - if pointing to another variable, it must be of the same data type
  - a location in memory
    - pointer variables have addresses too

©2018 Christine Laurendeau

Pointers (cont.)



Name	Type	Value	Location

©2018 Christine Laurendeau

Pointers (cont.)

- Characteristics of pointers:
  - in a 64-bit architecture, pointer variables occupy 8 bytes
  - pointers may point to variables of:
    - any data type
    - any size

- Interesting thoughts:
  - a pointer can point to only one variable -- why?
  - a variable may have multiple pointers to it

©2018 Christine Laurendeau

Declaring Pointers

- What is a variable declaration?
  - a statement specifying a variable's name and data type
    - ... and more stuff out of scope for this course
- A pointer variable declaration has 3 components:
  - the type of data that the pointer will point to
  - * symbol indicates that the variable being declared is a pointer
  - the pointer variable name

©2018 Christine Laurendeau

Assigning Values to Pointers

- Pointer variables are assigned values
  - from the memory address of another variable
  - from a system call requesting a new block of memory
    - ... more on this later ...
- Important symbol: **&**
  - in a statement, it is the **address-of** operator
  - operator returns the memory location of the specified variable
    - its address

©2018 Christine Laurendeau

Accessing Pointer Values

- Important symbol: *****
  - in a statement, it is the **dereferencing** operator
  - operator returns the value pointed to
    - the value stored at the address contained in the specified pointer
- Common problem: NULL pointer exception
  - dereferencing a pointer that is set to NULL
    - pointer value is zero
  - solution: always check that your pointers are not NULL

©2018 Christine Laurendeau

Pointer Operators

- Symbols for both pointer operators have dual roles

- In variable declarations:
  - the **`*`** symbol declares a pointer
  - the **`&`** symbol declares a reference

- Anywhere else, these symbols are **operators**:
  - the **`*`** symbol is the unary *dereferencing* operator
  - the **`&`** symbol is the unary *address-of* operator
  - both symbols mean something else as binary operators

©2018 Christine Laurendeau

Parameter Passing

- Pass-by-value
  - does not use addresses of variables
- Pass-by-reference can be done in two ways:
  - pass-by-reference by reference
  - pass-by-reference by pointer

©2018 Christine Laurendeau

1.6.3 Memory Allocation

- When a variable is declared, memory for it is allocated

- What does memory allocation do?
  - it reserves a specific (known) number of bytes
  - the number of bytes reserved is based on the data type

- Types of memory allocation
  - static
    - memory reserved at *compile time*
    - it is located in the function call stack
  - dynamic
    - memory reserved at *runtime*
    - it is located in the heap

©2018 Christine Laurendeau

Static Memory Allocation

- Memory is reserved in the function call stack
  - statically allocated variables include:
    - local variables
    - function parameters

©2018 Christine Laurendeau

Dynamic Memory Allocation

- Characteristics:
  - memory is reserved in the heap
  - it is allocated with the **`new`** operator
  - it is deallocated with the **`delete`** operator

©2018 Christine Laurendeau

Memory Leaks

- C/C++ does not provide *garbage collection*
  - there is a risk for *memory leaks*

- What is a memory leak?
  - it is a block of dynamically allocated memory with no pointers to it
    - that memory can never be accessed again by the program

- How does a memory leak happen?
  - a pointer gets *clobbered*
    - overwritten
  - a pointer moves out of scope
    - e.g. function allocates memory and stores pointer as local variable
    - what happens when function returns?

©2018 Christine Laurendeau

Memory Leaks (cont.)

- Why is a memory leak a problem?
  - access to the data is permanently lost by the program
  - a finite amount of heap space is allocated to each program
  - once allocated, memory is reserved until:
    - it is explicitly deallocated
    - the program terminates
  - if a pointer is lost, memory remains reserved, but unaccessible
    - for long running programs, we may run out of heap space
    - most industrial software is intended to run for months, or even years

©2018 Christine Laurendeau

Memory Leaks (cont.)

- How do we prevent memory leaks?
  - always explicitly deallocate memory when you're done with it
    - use the **`valgrind`** tool in Linux to check
  - if a called function allocates memory, pass pointer by reference
    - this requires using *double pointers*
  - make sure you don't clobber pointers into the heap
  - some languages do *garbage collection*
    - an automated mechanism that frees unreferenced memory

©2018 Christine Laurendeau

Dynamic Allocation of Objects

- Objects are allocated with the **`new`** operator
  - the object constructor is called
  - parameters to the constructor may be specified
  - if no parameters are used, the default constructor is called
- Objects are deallocated with **`delete`** operator
  - the object destructor is called

©2018 Christine Laurendeau

1.6.4 Dynamic Arrays

- Arrays can be dynamically allocated with **`new[]`** operator
  - default constructor is called for every element of the array
  - constructor parameters cannot be specified

- Dynamic arrays are deallocated with **`delete[]`** operator
  - destructor is called for every element of the array
  - using **`delete`** without the brackets
    - may not compile
    - may result in unpredictable behaviour

©2018 Christine Laurendeau

Dynamic Arrays (cont.)

- Two types of memory allocation
  - static
  - dynamic
- Two types of data stored in an array
  - objects
  - object pointers
- This means: four ways of storing data in arrays

©2018 Christine Laurendeau