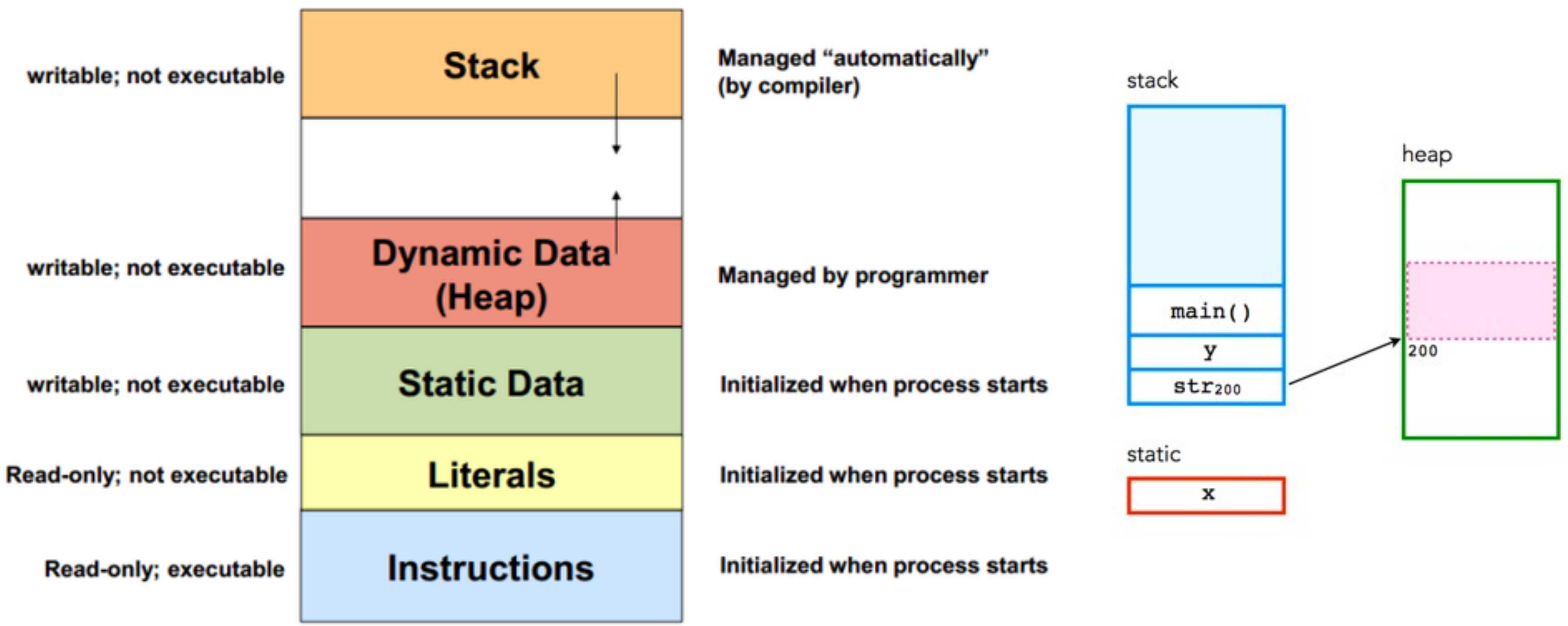
Lecture 16: Memory Layout – Stack vs Heap

Class page: https://github.com/tsung-wei-huang/cs1410-40

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Memory Layout

- ❑ **Stack vs Heap**



Stack

- ❑ **The place where *arguments* of a function call are stored**
- ❑ **The place where *registers* of the calling function are saved**
- ❑ **The place where *local data* of called function is allocated**
  - *Automatic* data
- ❑ **The place where called function leaves *result* for calling function**
- ❑ **Supports recursive function calls**
- ❑ **…**

Stack

- Imagine the following program:–

```
int factorial(int n){
  if (n <= 1)
    return (1);
  else
    int y = factorial(n-1);
    return (y * n);
}
```

- Imagine also the caller:–

```
int x = factorial(100);
```

- What does compiled code look like?

Compiled Code: Caller

```
int x = factorial(100);
```

- **Put the value “100” somewhere that *factorial* function can find**
- **Put the current program counter somewhere so that *factorial* function can return to the right place in *calling* function**
- **Provide a place to put the result, so that *calling function* can find it**

Compiled Code: Factorial Function

- **Save the *caller*'s registers somewhere**
- **Get the argument *n* from the agreed-upon place**
- **Set aside some memory for local variables and intermediate results – i.e., *y, n - 1***

- **Do whatever *factorial* was programmed to do**

- **Put the result where the *caller* can find it**
- **Restore the *caller*'s registers**
- **Transfer back to the program counter saved by the *caller***

Somewhere?

- So that *caller* can provide as many arguments as needed (within reason)?
- So that *called routine* can decide at run-time how much temporary space is needed?
- So that *called routine* can call any other routine, potentially recursively?

Answer: a Stack

- ❑ ***Stack* – a linear data structure in which items are added and removed in *last-in, first-out* order.**

- ❑ **Calling program**
  - *Push* arguments & return address onto stack
  - After return, *pop* result off stack

Stack with Called Routine

❑ Called routine

- *Push* registers and return address onto stack
- *Push* temporary storage space onto stack

- Do work of the routine

- Pop registers and temporary storage off stack
- Leave result on stack
- Return to address left by calling routine

Stack

- **All modern programming languages require a stack**
  - Fortran and Cobol did not (non-recursive)
- **All modern processors provide a designated *stack pointer* register**
- **All modern process address spaces provide room for a stack**
  - Able to grow to a large size
  - May grow upward or downward

Heap

- **A place for allocating memory that is not part of *last-in, first-out* discipline**
- **I.e., dynamically allocated data structures that survive function calls**
    - E.g., strings in C
    - `new` objects in C++, Java, etc.

Allocate Memory from Heap

- ***malloc*() – POSIX standard function**
  - Allocates a chunk of memory of desired size
  - Remembers size
  - Returns pointer
- ***free* () – POSIX standard function**
  - Returns previously allocated chunk to heap for reallocation
  - Assumes that pointer is correct!

Allocate Memory from Heap

- ***malloc*() – POSIX standard function**
  - Allocates a chunk of memory of desired size
  - Remembers size
  - Returns pointer
- ***free* () – POSIX standard function**
  - Returns previously allocated chunk to heap for reallocation
  - Assumes that pointer is correct!
- ***Storage leak* – failure to *free* something**

Heap in Modern Systems

- **Many modern programming languages require a heap**
  - C++, Java, etc.
  - *NOT* Fortran
- **Typical process environment**
  - Heap grows toward stack — but never shrinks!
- **Multi-threaded environments**
  - All threads *share* the same heap
  - Data structures may be passed from one thread to another.

How to Detect Memory Leak?

https://valgrind.org/

Summary

- Memory layout
- Stack
- Heap