Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

These notes are based on the lecture slides from the course:
BU CS330: Introduction to Analysis of Algorithms

Presented by:
Dora Erdos, Adam Smith
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

— 1 —

Memory Management

1.1 CPU Arichitecture

To further understand how our algorithms run, we must understand lightly how memory is passed around in our programs.

Definition 1.1: Machine Code & Compilation

Code is separated into two main areas of memory management, the program itself, and the data in transit during execution. The program itself is broken up such as follows:

- **Text Segment:** The part of the program which contains the executable code.
- **Data Segment:** The part of the program which contains global and static variables.
- **Machine Code:** The compiled code of the program, which is executed by the CPU.

Once the code compiles, our data segment is further divided into two parts in memory:

- **Initialized Data:** Data given a value before the program starts (global variables).
- **Uninitialized Data:** Data yet to be assigned (local variables), which are zeroed at program start.

By memory we mean the **RAM (Random Access Memory)** hardware component, which stores temporary data, constantly communicating with the CPU or external storage (e.g., hard drive, SSD). Each memory cell is IDed by a unique monotonic **address**, often in hexadecimal format(e.g., 0xF00, 0xF01, etc).

1.2 Stack Data Structures

Let's talk about our first data structure, the stack:

Definition 2.1: Stack

A **stack data structure** is a collection of elements that follows a **Last In, First Out** (LIFO) principle. I.e., in a stack of plates, the last added plate is the first one to be removed, not the middle or bottom/first plate. Each *plate* in the stack is called a **stack frame**.

This text does not concern assembly code or low-level programming, so **do not** get caught up on the specifics of this next example:

Example 2.1: Assembly Code

Here is an assembly code example that demonstrates both initialized and uninitialized data. Initialized data is placed in the '.data' section, while uninitialized data is placed in the '.bss' section:

```
section .data                ; Initialized data section
    num1    dd  7            ; num1 is initialized to 7
    num2    dd  3            ; num2 is initialized to 3

section .bss                 ; Uninitialized data section
    temp    resd 1           ; temp is reserved (uninitialized)
    result  resd 1           ; result is reserved (uninitialized)

section .text                ; Code section
    global _start

_start:
    mov eax, [num1]          ; Load num1 into eax
    mov [temp], eax          ; Store num1 in temp
    mov ebx, [num2]          ; Load num2 into ebx
    add eax, ebx             ; Add num2 to eax (eax = num1 + num2)
    mov [result], eax        ; Store the sum in result
; (Exit code omitted for brevity)
```

In this example, 'num1' and 'num2' are initialized before execution, while 'temp' and 'result' are uninitialized and only receive values during program execution. ■

Now let's look at how our programs utilize the stack:

Definition 2.2: Call Stack

A **call stack** is a stack which keeps track of function calls in a program as well as any local variables within such functions.

This is why we say a variable is in **scope**, as when a function is taken off the stack, or a new stack frame is placed on top, the variables in the previous or discarded stack frame are **no longer accessible**.

Let's illustrate this with the following diagram:

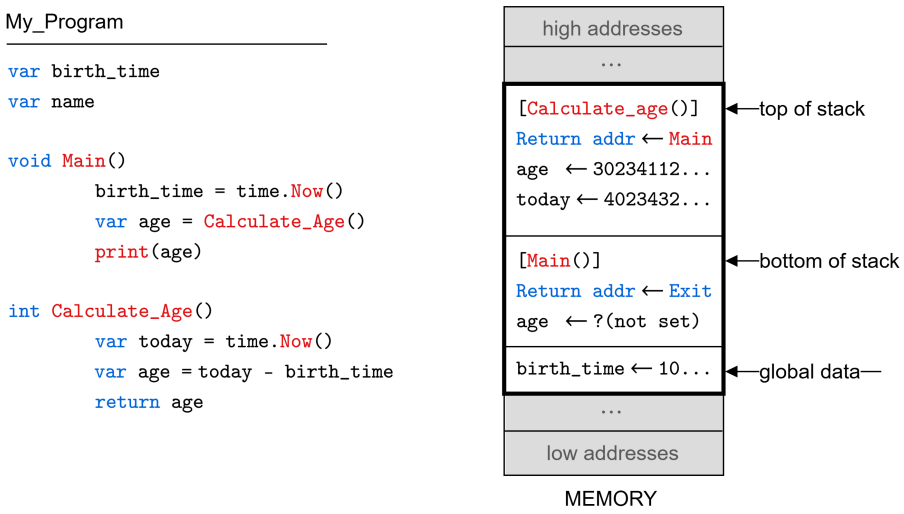

Figure 1.1: Here is a simplified look at how memory manages the stack. On the left is our program written in some abstract language, and on the right is the call stack in memory (simplified). The program has a global 'birth_time' variable, which is initialized in the `Main` function. The `Main` function then calls the `Calculate_Age` function which uses the 'birth_time' variable to calculate the 'age' via the difference of the current time and the 'birth_time.' Looking at the memory, we see at the bottom of our memory contains global variables accessible to any frame. Next, is the bottom of the stack, containing a return address to exit the program, while awaiting the result of the function call for 'age'. The top of our stack contains another frame that we will return the value a new 'age' (not the same as the one before) not accessible from the main function. This new frame also contains a new local variable 'today.' Once this function returns, `Main` will have the result of its local variable 'age.'. Concretely, the 'age' variable in both the `Main` and `Calculate_Age` function are completely separate despite sharing the same *name*.

Please Note: The above figure is a simplified version; This presentation derivatives from what actually happens for teaching sake. In the following pages we define the stack frame in more detail.

Tip: A lot of demonstrations (including this text) will show the stack growing **upwards**; This is strictly because it's easier to visualize and does not accurately portray what a stack really does or looks like. In the following pages we will clear this up, and show how the stack actually grows from top-to-bottom. Of course, there is always room for deviation if a developer wishes to implement a stack in some other arbitrary way. Nonetheless, the following is what one might typically expect in a stack implementation.

Definition 2.3: Instructions

The CPU register (IP/EIP) is the **instruction pointer**, pointing to the next operation to execute. All commands are baked into the CPU; This includes the **ALU (Arithmetic Logic Unit)**, **Memory Unit (load/store)**, and **Control Unit (branching/looping)**. All instructions are given a numeric ID called an **opcode** (operation code).

The CPU fetches the instruction from memory, decodes it, and executes it. This process is repeated until the program terminates. Languages like assembly interface this with human-readable mnemonics, such as 'MOV', 'ADD', 'SUB', etc (as seen in Example 2.1).

Definition 2.4: Stack Frame Anatomy

Under the x86-32 calling convention Two registers keep track our place in the stack:

- **Base Pointer (BP/EBP):** Points to the base (i.e. "bottom") of the current function's stack frame.
- **Stack Pointer (SP/ESP):** Points to the "top" of the current function's stack frame, i.e., the next free byte where a push would land.

When the program starts, the operating system *reserves* a contiguous region of memory for the stack. By convention, the *bottom* of that region lies at a higher address, and the stack "grows downward" toward lower addresses as data is pushed. If the stack pointer ever moves past the reserved limit—a **stack overflow** occurs.

A single **stack frame** itself is a contiguous block of memory in which the function stores:

- **Parameters:** The arguments passed in by the caller,
- **Return Address:** The address of the next instruction to execute after the function returns,
- **Old Base Pointer:** The caller's 'EBP', saved so that on return we can restore the previous frame,
- **Local Variables:** Space for any locals or temporaries that the function needs.

This is why variables in previous or new functions calls become "**out of scope**" (no longer accessible), as they belong to some other stack frame; When it comes to **Global Variables**, they live in a separate region of memory, defined by the **data segment** (1.1).

Moreover, a call to a new function invokes the **call** instruction, this automatically pushes the return address to the current frame onto the stack. Additionally, the CPU reserves the **EAX** register for the return value (number or address) of a function. When the function returns, it can place its result in 'EAX', and the caller can retrieve it from there. During constant use the 'EAX' register may contain **garbage** data from previous use, unless explicitly set to zero or some other value.

High Addresses		
Contents	**Offset**	**Notes**
(Parameters 3, 4, ...)	$EBP + 16, +20, \ldots$	Third-and-onward arguments, if any.
Parameter 2	$EBP + 12$	Second argument passed on stack.
Parameter 1	$EBP + 8$	First argument passed on stack.
Return Address	$EBP + 4$	Auto-pushed by the `call` instruction.
Old EBP (Saved BP)	$EBP + 0$	The caller's base pointer
Current Frame (locals/temporaries)		
Local Variable 1	$EBP - 4$	First 4-byte local (or smallest slot).
Local Variable 2	$EBP - 8$	Next 4-byte local or part of a larger object.
...	⋮	(additional locals at $EBP - 12, -16, \ldots$)
Low Addresses		

Table 1.1: Typical x86-32 Stack-Frame Layout, where offsets are typically a multiple of 4 bytes.

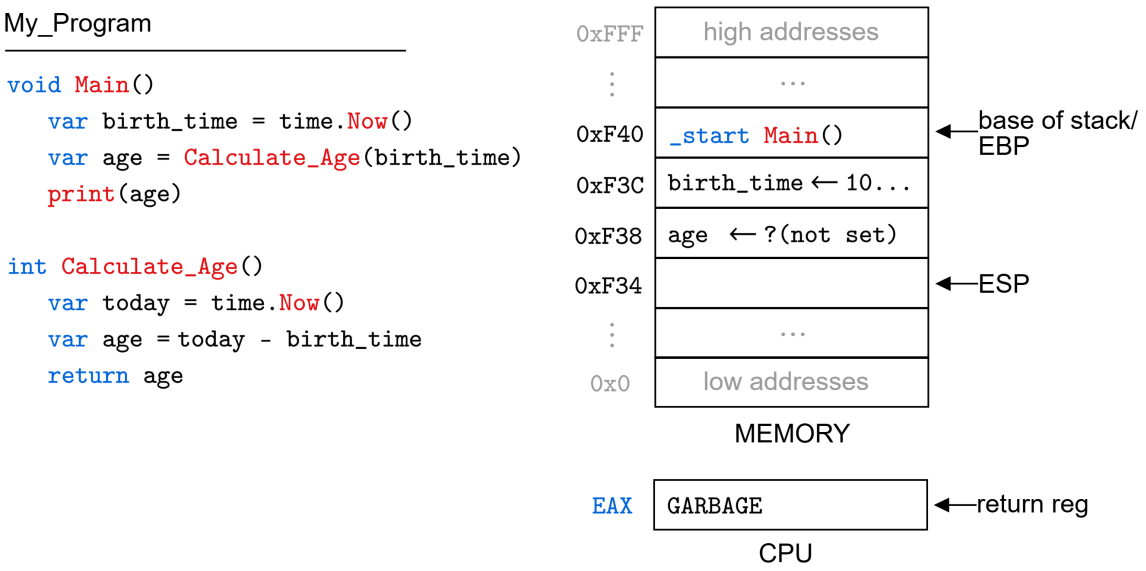

Figure 1.2: Revisiting Figure (1.1) with slight alterations to the code: This is a snapshot of the code executing right before `CalculateAge(birth_time)` is called. For simplicity sake, let's say the stack begins at address `0xF40` (Hexadecimal), growing downwards. Here the base of the stack and the EBP are one and the same. We include the CPU's EAX (return register), which contains garbage. Address `0xF38` is currently just reserved space for 'age'.

My_Program

```
void Main()
    var birth_time = time.Now()
    var age = Calculate_Age(birth_time)
    print(age)

int Calculate_Age()
    var today = time.Now()
    var age = today - birth_time
    return age
```

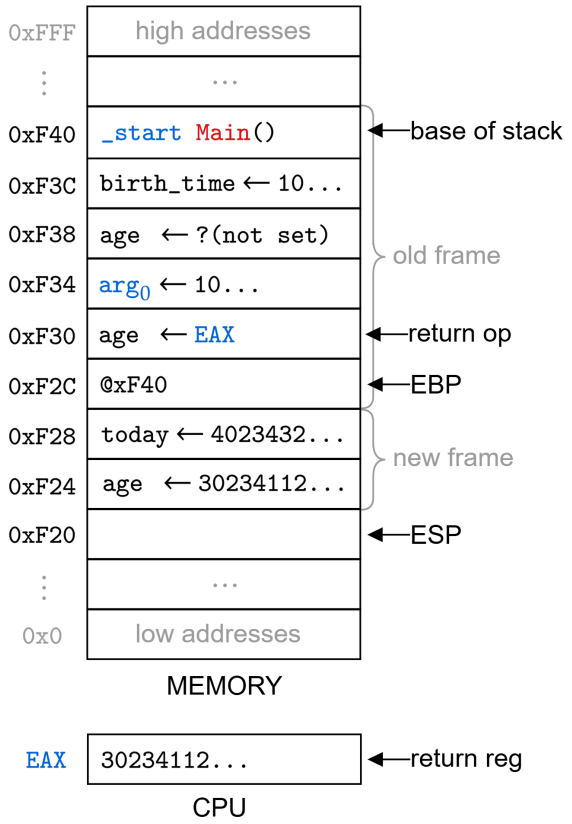

Figure 1.3: Revisiting Figure (1.2) at the moment the function `CalculateAge(birth_time)` has supplied its return value to the EAX register, and is about to return. We see that before calling `CalculateAge(birth_time)`: The old frame pushed it's arguments (`birth_time`) onto the stack, then the return address (IP/Next Instruction) onto the stack, and finally the old EBP (Base Pointer) onto the stack. The 'new frame' then sets the saved EBP address to the current EBP, concluding the old frame into the 'new frame'. Moreover, since the offset looks for local variables below `0xF40`, the above 'birth_time' and 'age' are **out of scope** for the 'new frame', vice-versa. **Note:** This is still a high-level abstraction of what actually happens sequentially with opcodes; Nonetheless, this is the fundamental idea of how a stack works.

This concludes our discussion on stack structures; We continue with the heap structure next.

1.3 Heap Data Structures

So far we have simply said global data is declared in the **data segment** of memory. There is a second segment of memory that builds ontop of this called the **heap**:

Definition 3.1: Heap – Dynamic vs. Static Memory

When a program runs there is a **static** (fixed) region reserved for the program's data segment (local/global variables). During execution, more objects may be created, needing additional memory; A new region of memory is reserved **dynamically**, building upwards from the top of the data segment, called the **heap**.

Language protocols either **manually** (e.g., Assembly, C) or **automatically** (e.g., Python, Java) manage this memory:

- **Manual Memory Management:** The programmer must explicitly allocate and deallocate memory using functions like 'malloc' and 'free' in C.
- **Automatic Memory Management:** The language runtime automatically allocates and deallocates memory, often using a **garbage collector** to reclaim unused memory (no variables pointing to it).

Unlike the stack, this allows values to be accessed from anywhere in the program, regardless of the function call or scope.

To continue we must understand what a hash table is, and how arrays and objects work in memory:

Definition 3.2: Hash Table

A **hash table** is a data structure that maps keys to values, allowing for efficient retrieval of values based on their keys. It uses a **hash function**, which takes a key (e.g. a number or string) as input and produces a fixed-size (i.e., output modulo the table size) hash value for the index.

Note: The input in many context (typically cryptographic), may be called 'data' or 'message'; The output: hash, checksum, fingerprint, or digest.

Definition 3.3: Arrays in Memory

In memory array elements are stored sequentially. A reference to an array is a pointer to the first element. To terminate reading an array, we must either know the size of said array or have some sentinel value (e.g., 'null' or '0') to indicate the end of the array.

E.g., An array of 3 integers (4 bytes each) occupying [0xF00-0xF0C]: $[1, 2, 3, \texttt{null}]$.

Objects behave very similarly to arrays, but with a few key differences:

Definition 3.4: Objects in Memory

An **Object** (or **struct**), is a collection of key-value pairs, where each key is called a **field** or **attribute** and each value can be any data type (e.g., number, string, or address).

Attributes are stored in array like fashion, where each element is a fixed-offset from the head (start) of the object. The object itself is a pointer to the first element. Accessing attributes works differently in compiled (e.g., C) vs. interpreted (e.g., Python) languages:

- **Compiled Languages:** There is no lookup, as the compiler has *hardcoded* the offsets of each attribute interaction (e.g., 'object.attribute' is translated to a direct memory access).
- **Interpreted Languages:** The interpreter looks up a hash table lookup for the attribute name.

Depending on the use case, objects may be stored in the heap or stack:

- **Static Objects:** An objects whose size is known at compile time can be allocated on the stack. I.e., no changes to the object are made after creation (e.g., Math and Time objects, which purely exist to compute).
- **Dynamic Objects:** Often just called **objects**, are allocated on the heap, allowing for dynamic resizing and modification (e.g., a student object with attributes like 'name', 'age', and 'grades' that can change over time).

We define the following for completeness sake:

Definition 3.5: Classes & Interfaces

Object-oriented programming is a paradigm where objects are the main building blocks of the program. A **class** is a blueprint for defining how an object will behave once **instantiated** (created). In this paradigm, functions are called **methods**, as they are defined and used within the class (i.e., globally does not exist in independence).

Some languages (e.g., Java, C++) support **interfaces** (or protocols), which specify a set of methods that implementing classes must provide. Although the terminology varies (abstract classes, traits, protocols, etc.), they all ultimately describe capabilities an object must fulfill.

Tip: Often when trying to print an object in Java we see `ClassName@HEX`, where `HEX` is the object's identity hash code instead of the memory address; Memory access poses security risks to memory manipulation.

One last definition:

Definition 3.6: Strings & Characters in Memory

A **character** is represented by a numeric code unit:

- In C, a single `char` (1 byte) typically holds an ASCII code (0–127).
- In Java, `char` is a 16-bit UTF-16 code unit (U+0000..U+FFFF). ASCII values (0–127) map directly to the same Unicode code points, so:

  ```
  char c = 'A';
  System.out.println((int)c);  // prints 65, since 'A' is U+0041
  ```

  Characters beyond U+FFFF use two `char` values (a surrogate pair). This allows us to do things like checking for valid characters:

  ```
  if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
      // c is in 'a'..'z' or 'A'..'Z'
  }
  ```

  We can also perform arithmetic on `char`:

  ```
  char c = 'A';        // U+0041 (65)
  char next = (char)(c + 1);  // 'B' (66)
  ```

Typically, a **string** is stored as a contiguous array of **characters**. In low-level languages (e.g. C), that array ends with a null terminator (`\0`) and literal strings reside in the data segment. In higher-level languages (e.g. Java, Python), strings are full objects with methods. For e.g.,

C:

- String literals (e.g. `"Hello"`) are placed in the (often read-only) data segment.
- Runtime-constructed strings (via `malloc`, `strcpy`, etc.) live on the heap.

Java:

- Compile-time literals are **interned** (stored as a single shared copy) into the **String Constant Pool** section (specially reserved on the heap).
- Any other `String` (e.g. via `new String(...)`, concatenation, or user input) also resides on the heap but outside the pool.
- Because Java strings are immutable, interning lets multiple references share the same character data.

With slight alterations to our code in Figure (1.3), we illustrate heaps and arrays:

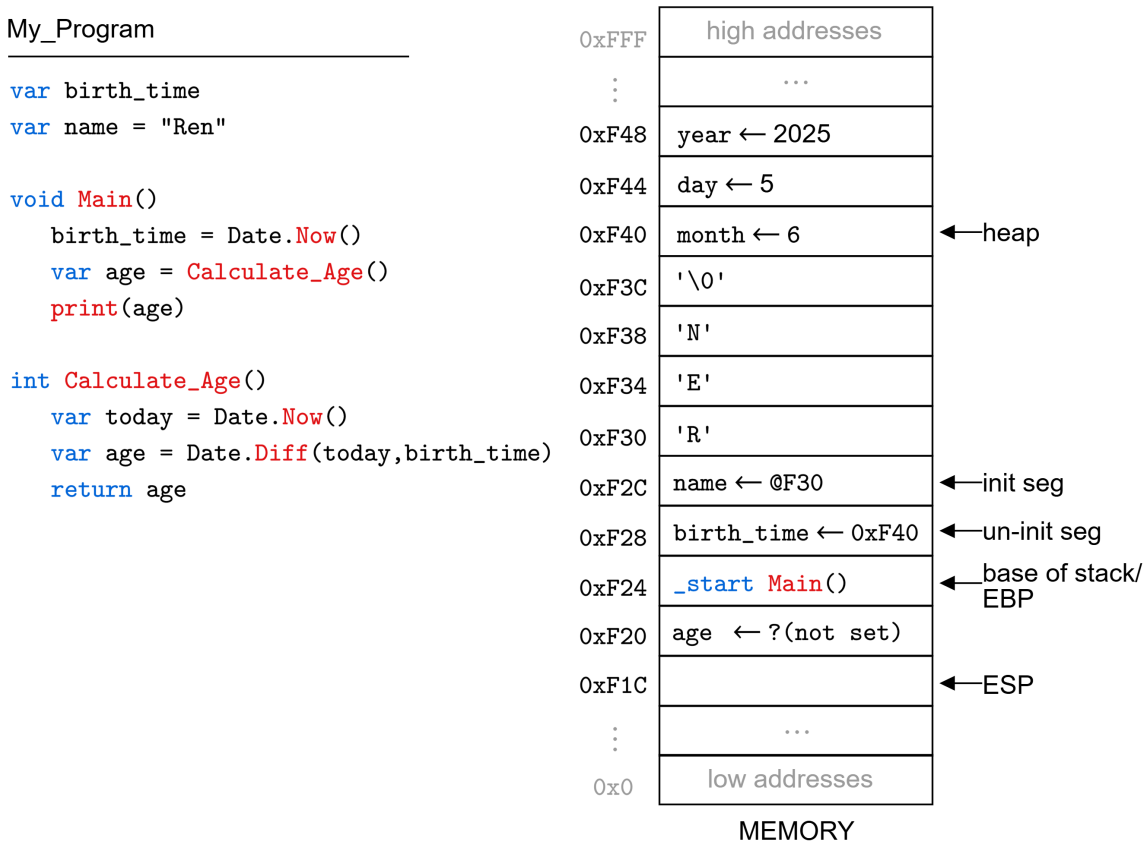

Figure 1.4: Here `birth_time` and `name` are global variables. Following the C convention, `birth_time` is placed in the uninitialized data segment, while `name` is placed in the initialized data segment. Since `name` is a string, it holds a reference to the first character in the string, which is stored contiguously in the initialized data segment ending with a null terminator ('\0'). During execution, `Date.Now()` a method call from a `Date` object is called; This method returns a new object, which is placed on the heap with its attributes (`month`, `day`, `year`) stored contiguously in memory. **Note:** Methods such as `Date.Diff()` are code (not data), which do not live in the heap or stack.

A small note regarding the example above:

Definition 3.7: Factory Method

A **factory method** is a function that creates and returns an object, often initializing it with default values or parameters. In Figure (1.4), the method `Date.Now()` is a factory method that creates a new `Date` object with the current date and time.

Bibliography