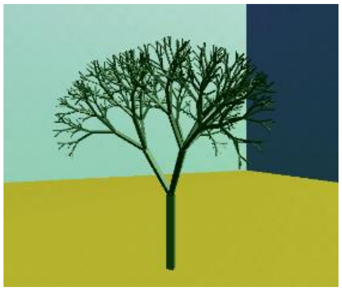


GATS Companion to: Program Memory Layout

Author: Garth Santor
Editors: Trinh Hān, Lianne Wong

Version: 1.5.0 (2019-01-27)

Overview

Where is the code, variables, literals, and other program elements stored in computer memory? Knowing how and where program elements are stored, when and how they are assigned a location, and how long they persist, will help a developer understand:

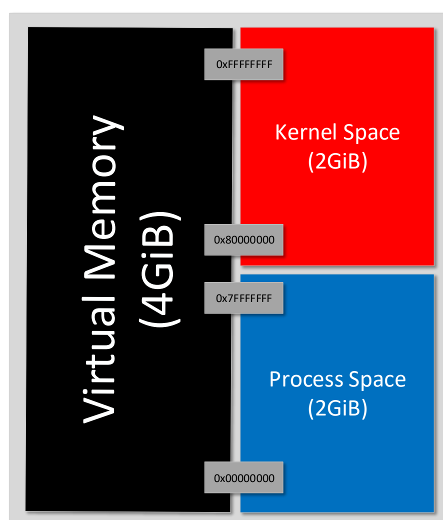
- Memory use, and memory leaks.
- The efficiency of data access operations.
- The efficiency of data allocation and deallocations.
- The robustness of a memory reference.

Simplified Memory Model

In our examples we will use a simplified version of Microsoft Windows 32-bit default virtual address space. It is typical of 32-bit virtual memory operating systems, like OS/X, Linux, and UNIX.

A virtual memory system uses hardware and software to map virtual memory addresses to physical memory addresses. Each user program is broken up into *memory pages* (for example: 4KiB in size) that the operating system maps to *physical pages* in RAM with the help of the CPU's *memory management unit* (MMU). This allows our software to be programmed for an idealized memory layout, and not the reality of actual physical memory layouts which can be discontiguous and be located on different devices like GPUs.

An added benefit of this approach, is that *virtual memory pages* from different processes can be mapped to *physical memory pages* simultaneously (just not to the same physical memory pages). This allows multiple processes to share physical memory giving the appearance of many programs running at the same time. Best of all, none of the program need to consider that other programs are sharing the memory with them.



Virtual Address Space

A 32-bit address space provides 4GiB of physical memory, which maps to 4GiB of virtual memory. The virtual address space is then divided into *kernel space* and *user space*.

The operating system will run in the protected *kernel space*, whereas our user process will run in the unprotected *process space*.

Note the addresses for each space. The *process space* addresses always have a zero in the most significant bit; the *kernel space* addresses always have a one in the most significant bit.

In Microsoft Windows, the boundary between *process space*, and *kernel space* can be adjusted with '4-gigabyte tuning' (4GT) to provide a 3GiB *process space*, and a 1GiB *kernel space*. With Windows 7, the amount can be customized to any *process space* size between 2048MiB (2GiB) and 3072MiB (3GiB).

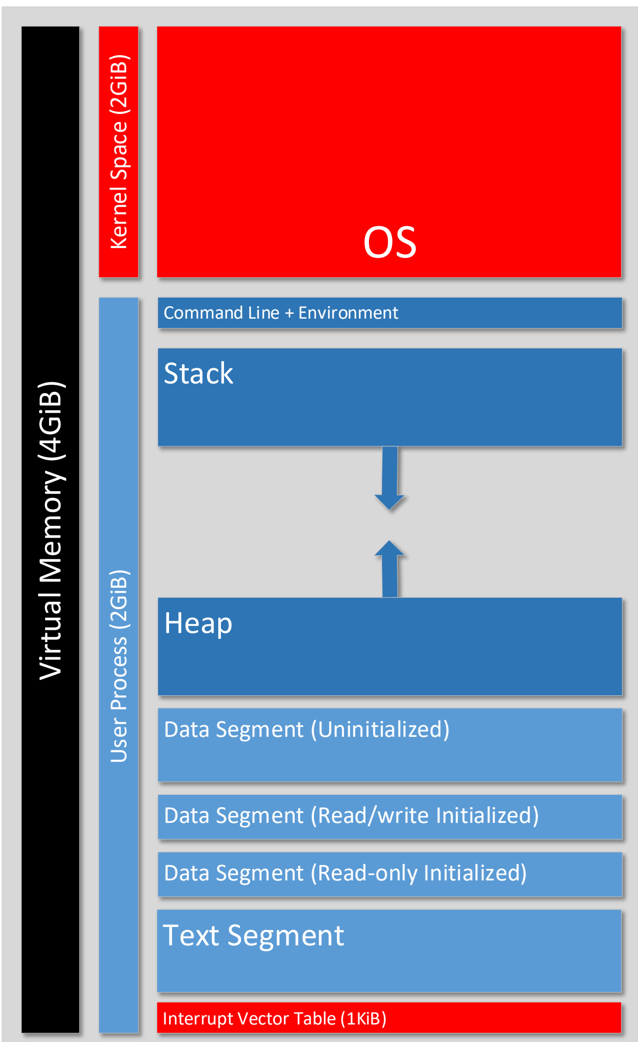
Process Space

User processes (such as application programs) live in the *process space*. The *process space* has its own internal structure. Again, I'm going to present a simplified, somewhat generic layout. The model assumes a single execution thread, again for simplicity.

Our *process space* model is broken into the following sections:

Interrupt Vector Table

The *interrupt vector table* is read-only block of addresses (read/write to the kernel) listing the interrupt handlers for the system. It is not relevant to this discussion other than to recognize why our processes don't start at location zero.



Text Segment

Alias: Code Segment

The *text segment* contains code, and depending on the compiler, literal values are embedded along with the code.

Text Segments are placed below the heap and the stack to help prevent memory overruns from corrupting the code. Where operating systems support memory segment protection, the *text segment* can be tagged as read-only.

Since *text pages* are never modified, the *text segment* can also be shared between multiple identical processes.

Data Segment

The *data segment* contains variables that have a lifespan that begins when the process is launched and extends until the process terminates. It is in turn divided into three parts. The first part holds read-only variables that are initialized when constructed, the second part holds initialized modifiable variables, and the third holds uninitialized variables. The size of the data segment is determined at compile time, and that size is fixed for the life of the process.

Command Line & Environment

The *command line* & *environment* section resides at the top of the process space. It is placed here since its size won't be known until the process is loaded and the command-line and environment information are passed from the operating system.

Why is this? The operating system maintains an *environment* that contains information about the context in which your program runs. This includes the *current working directory*, *environment variables*, and *command-line arguments*. These values usually have system wide or account wide settings, but can be overridden by temporary changes to the local shell, the user providing command-line arguments, or by invoking process functions such as spawn() that facilitate the customization of the environment when programmatically launching an application.

The Heap & Stack

The heap and stack provide the system's dynamic memory. Traditionally, the heap and stack share what memory remains after the fixed allocations have completed. The heap supports free allocations from the pool of available RAM usually growing up from the data segment. The stack supports LIFO allocations growing down from the bottom of the command line and environment segment.

The stack is normally managed implicitly by the process code and direct machine code support. Stack push and pop operations are standard on CPUs, push moving the stack address towards address zero, pop moving the stack address away from zero.

The heap has a dedicated *heap manager* containing data structures and algorithms designed to track and manage the blocks of memory allocated from the pool of memory it manages. The heap is different from the stack in that the memory allocations can occur anywhere in the heap, and deallocations can occur in any order.

Memory allocated on the heap is not necessarily deallocated when the referencing variable goes out of scope. Heap allocations in C and C++ must be explicitly deallocated. Failure to do so results in a memory leak. To prevent this, most objects that use dynamic memory utilize destructors that implement the deallocation code. For general dynamic allocations, smart pointers (pointers that deallocate what they point to when they go out of scope) are recommended.

Java and managed C++ (such as C++.NET) use garbage-collection instead of explicit deallocations. While this does prevent memory leaks, it doesn't completely resolve all memory problems. Developers, no longer worrying about memory leaks, often give up thinking about memory issues at all. While the memory doesn't technically leak, the same loss of memory can occur by holding on to the memory block for longer than necessary. Managed languages have a memory hording problem! Programmers that don't pay attention to the scope of their reference variables may create them in too broad a scope that hold on to them much longer than is necessary. While the memory block doesn't leak, it none-the-less consumes resource that could be allocated elsewhere.

Deallocations can cause the heap to become discontiguous (i.e. there can be unallocated blocks of memory in between the allocated blocks of memory). Small unallocated blocks can be difficult to reuse, as they can only be recycled by allocating them to the same size or smaller block. The inefficiency caused by excessive number a small, unallocated blocks is called *memory fragmentation*.

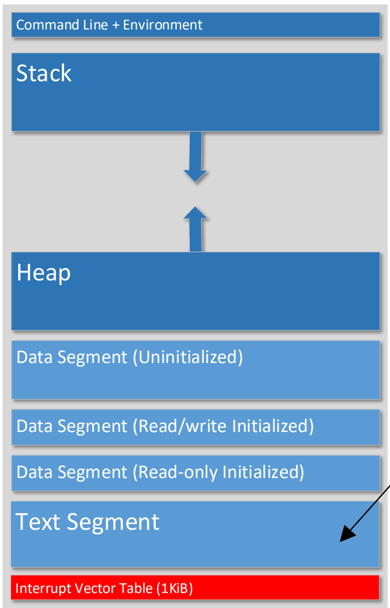
C/C++ and Java Memory Allocation Examples

Let's examine C and C++ code samples and connect the elements to their storage locations. Where the examples also apply to Java, it will be noted.

Interpreting Java memory allocations is a little more difficult since the memory model spans the compiler and the JVM. Where C and C++ compilers understand the system level memory model, the whole point of Java is to abstract the hardware as much as possible. The Java compiler compiles to an abstract memory model which then maps to the memory model of that system's JVM. JVM developers have a fair amount of leeway in its implementation.

However, understanding Java allocations can be done in the context of C/C++ allocations (after all, the JVM is usually written in C).

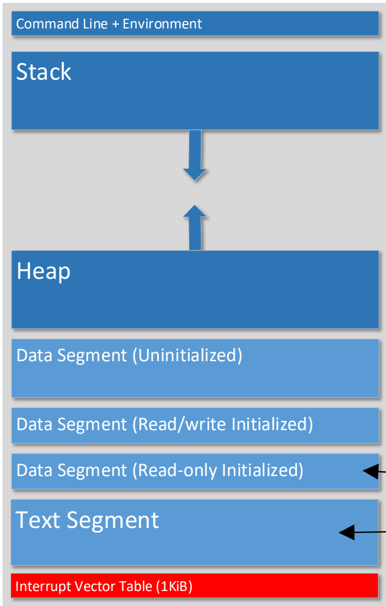
Code [C, C++, Java]



The machine code of function **`main`** is stored in the *text segment*.

```
int main() {
  return 0;
}
```

Literals [C, C++, Java]



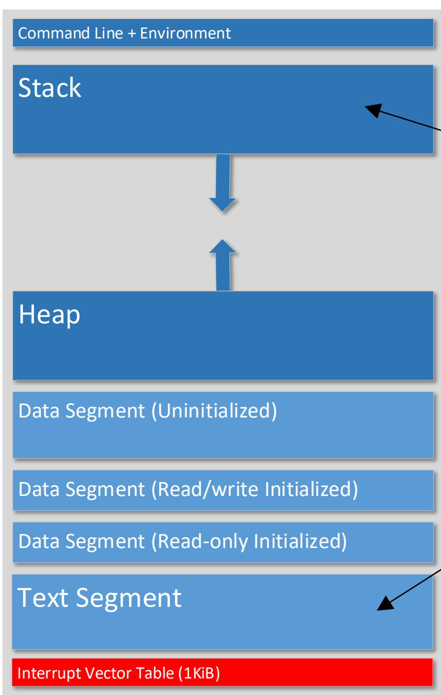
Literals are handled in one of two ways: stored in the *text segment* with the code that assigns the literal, or stored in the *data segment* as a read-only entity.

The compiler chooses a storage location by several criteria:

1. Is the literal's memory location referenced (e.g. string literals are handled by pointing to the block of memory containing the sequence of characters)?
2. Can the literal be embedded with the machine instruction that performs the assignment?

```
int main() {
  string str = "Hello";
  int number = 42;
}
```

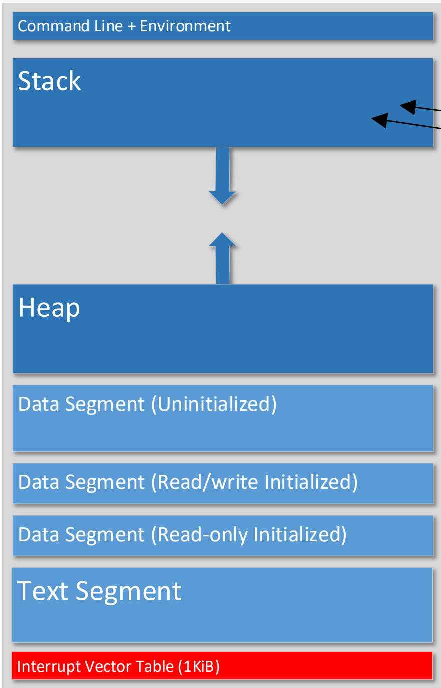
Local variables [C, C++, Java]



The local variable **`number`** is allocated on the stack, and deallocated when the function terminates. The literal **`42`** is embedded in the machine code of the *text segment*.

```
int main() {

  int number = 42;

}
```

Function parameters [C, C++, Java]

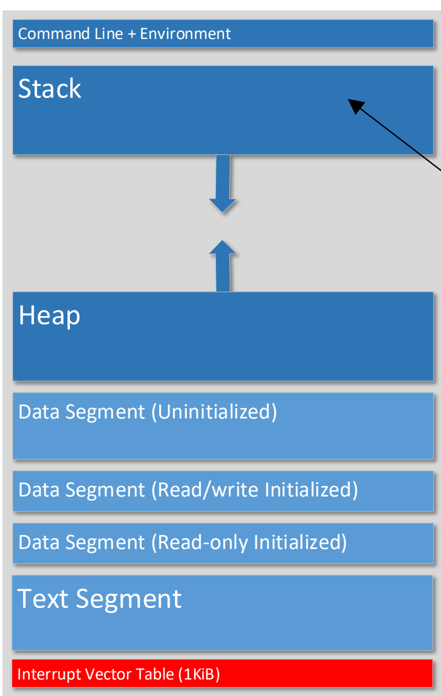


Function parameters are passed on the stack. Parameters can be passed by value, by reference, or by pointer. All result in the parameter being placed on the stack.

```
void square(int x, int& result) {
  result = x * x;
}

int main() {
  int n;
  square(5, n);
}
```

Pass-by-value parameters have their parameters placed on the stack. Pass-by-reference parameters have the address of the calling scope variable placed on the stack. Internally, reference parameters are passed to the function as pointers.

Local, short-lived variables [C, C++, Java]

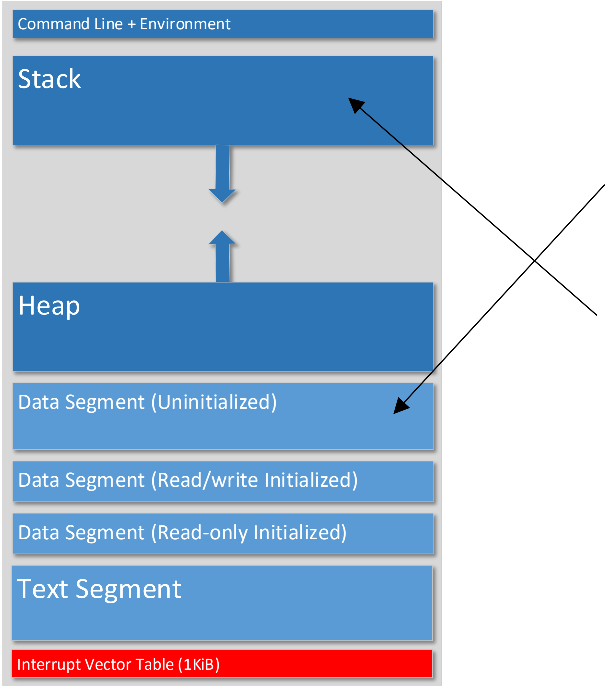


The local variable **`sum`** is allocated on the stack as would the local variable **`i`**. However, short-termed variables such as the loop variant **`i`** are often never allocated in RAM, but instead a CPU register is assigned to implement the variable.

```
#include <iostream>
int main() {

  int sum = 0;
  for (int i = 0; i < 10; ++i)
    sum += i;

  std::cout << i << std::endl;
}
```

If enough free CPU registers are available, the **`sum`** variable may also be implemented

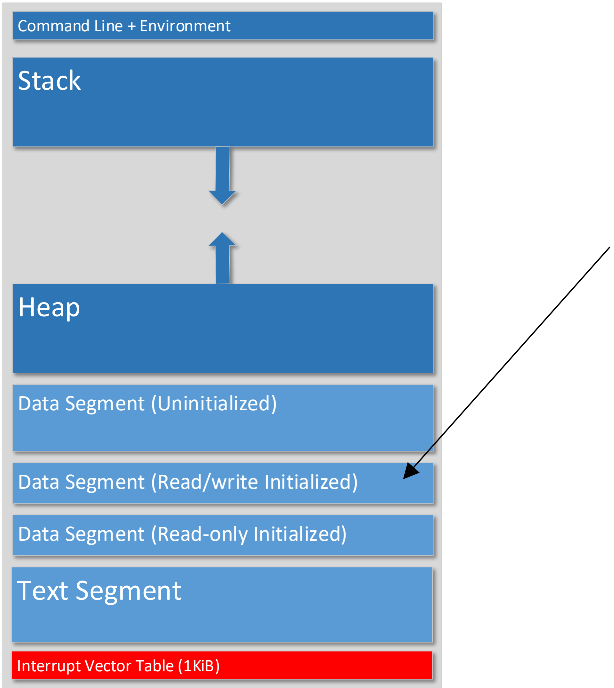
Global variables [C, C++]



The global variable **`global`** is allocated in the *data segment*, specifically the *uninitialized data segment*.

```
#include <iostream>

int global;

int main() {

  int local;

}
```

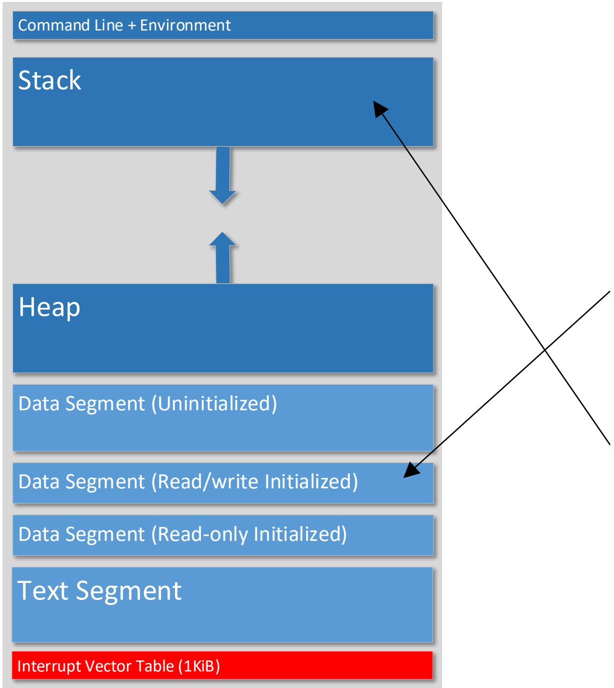
Global variables – initialized [C, C++]



The global variable **`global`** is allocated in the *data segment*, specifically the *initialized data segment*. The variable is initialized when the process loads if the data is plain-old-data (POD).

```
#include <iostream>

int global = 42;

int main() {

  int local;

}
```

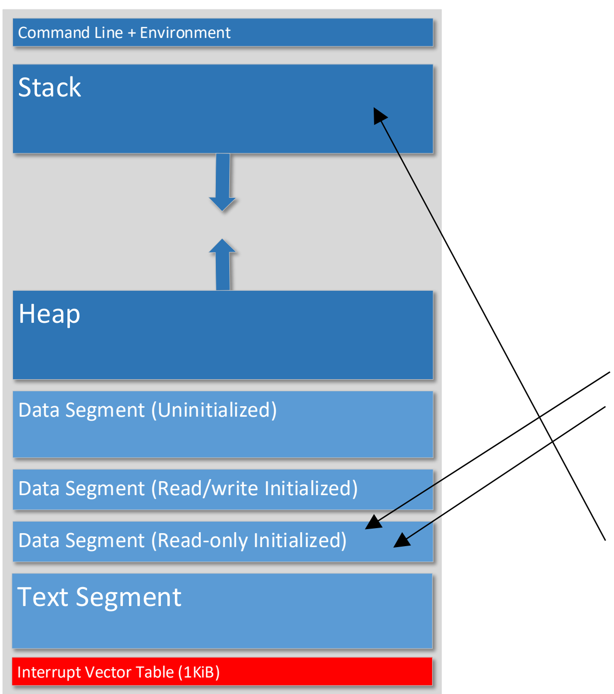
Reference variables (non-parameter) [C, C++]



Reference variables declared in the same scope as the variable they reference are merely aliases for the referenced variable. As a result, they share the same memory location as the variable they reference.

```
#include <iostream>

int global = 42;
int& globalRef = global;

int main() {

  int local;
  int& localRef = local;

}
```

Constants [C, C++]

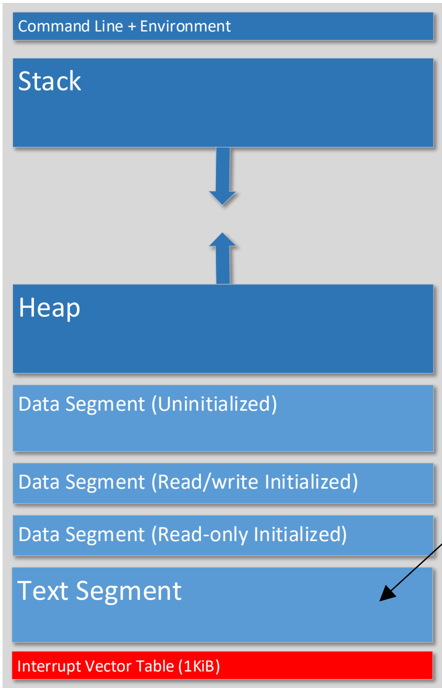


Constants generally follow the same allocation rules as variables; global constants are placed in the *Read-only Initialized Data Segment*; local constants on the *stack*.

However, optimizing compilers are free to analyze the code. If the compiler can determine that there are no references to the location of the constant (only the value of the constant is used), then constant can be implemented as if it were a *literal*.

```
const int cglobal = 42;
const char str[] = "constant";

int main() {

  const int clocal = 5;

}
```

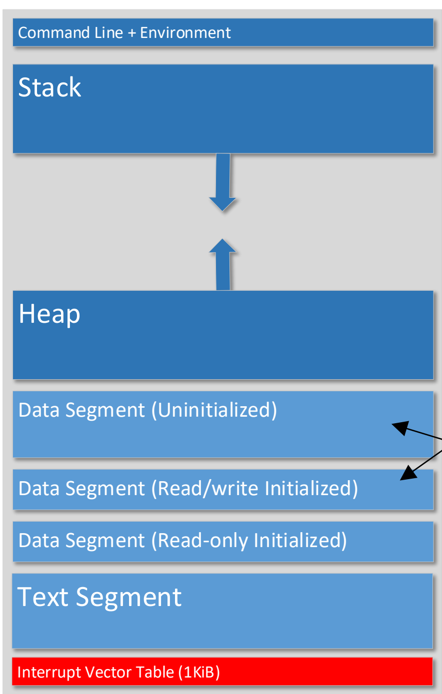
constexpr [C++]



C++ 11's **`constexpr`** expression appears like a constant in the source code, but like a literal in the machine code. Constant expressions are guaranteed to be evaluated at compile time.

```
#include <iostream>

int main() {
  constexpr int clocal = 5;
}
```

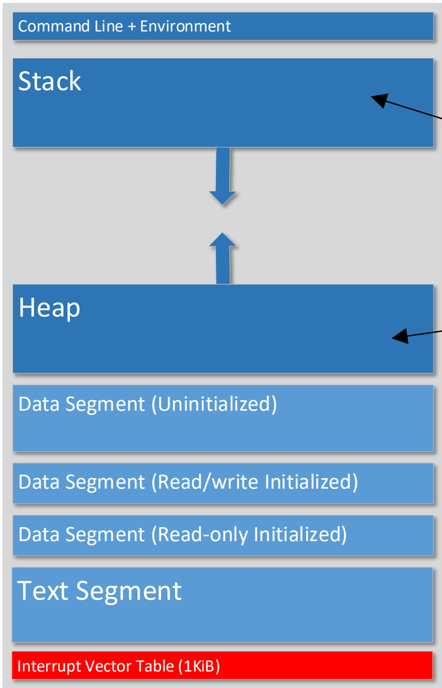
Local static variables [C, C++]



Local static variables are treated like local variables for their visibility scope, but like global variables for their location and lifespan.

```
#include <iostream>

int count() {
  static int c = 0;
  return ++c;
}

int main() {
  static int n;
  n = count();
}
```

Dynamic memory/pointers [C, C++, Java]

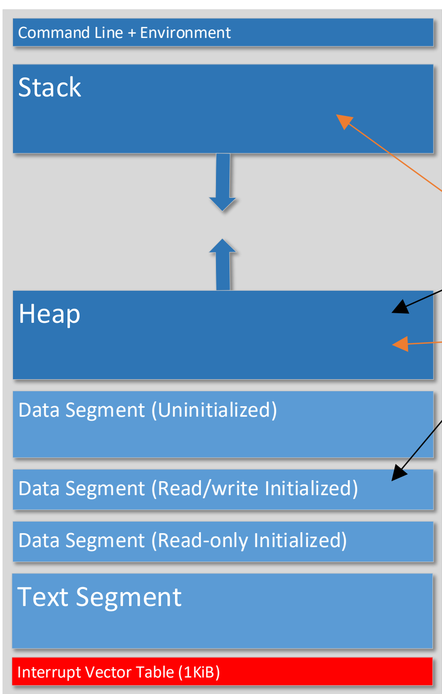


Dynamic memory allocators (new, malloc, etc.) allocate memory on the heap. The referencing variable (pointer) is handled like a typical variable.

```
int main() {
  int* p = new int[10];
}
```

Note that the memory allocated in C/C++ is not disposed of automatically. You'll need to call **`delete`** to return the memory to the heap. Java will garbage-collection the object once it has determined that the object is no longer being referenced.

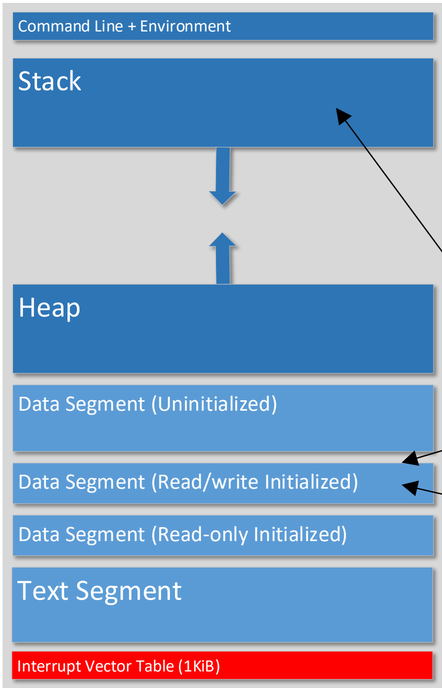
Dynamic objects [C++]



Dynamic objects such as STL container classes have a fixed portion and a dynamic portion. The fixed portion is handled like all other variables, the dynamic portion is on the heap.

```
vector<int> gv(10);
int main() {
  vector<int> v(10);
}
```

Class attributes [C++, Java]

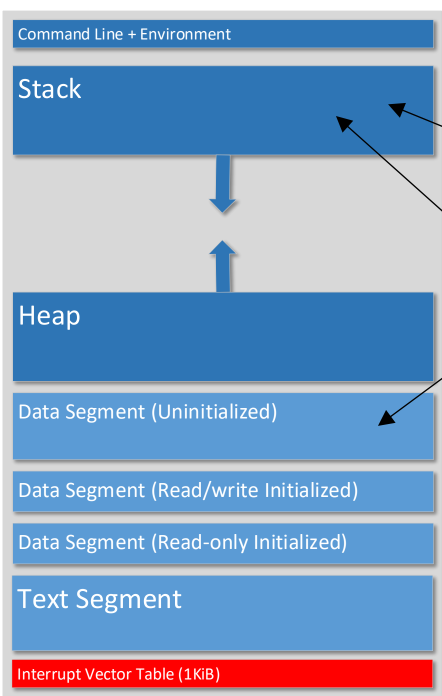


Classes allocate with the same rules as primitive data types with a few twists. If the object is declared locally, then the attributes are allocated on the stack. If the object is declared globally or static, its attributes are allocated in the data segment.

```
class Foo {
  int a = 0;
};

Foo bar;

int main() {

  Foo barLocal;
  static Foo bars;
}
```

The class definition itself is not stored in memory. It's methods however are stored in the *text segment*, as they are code.

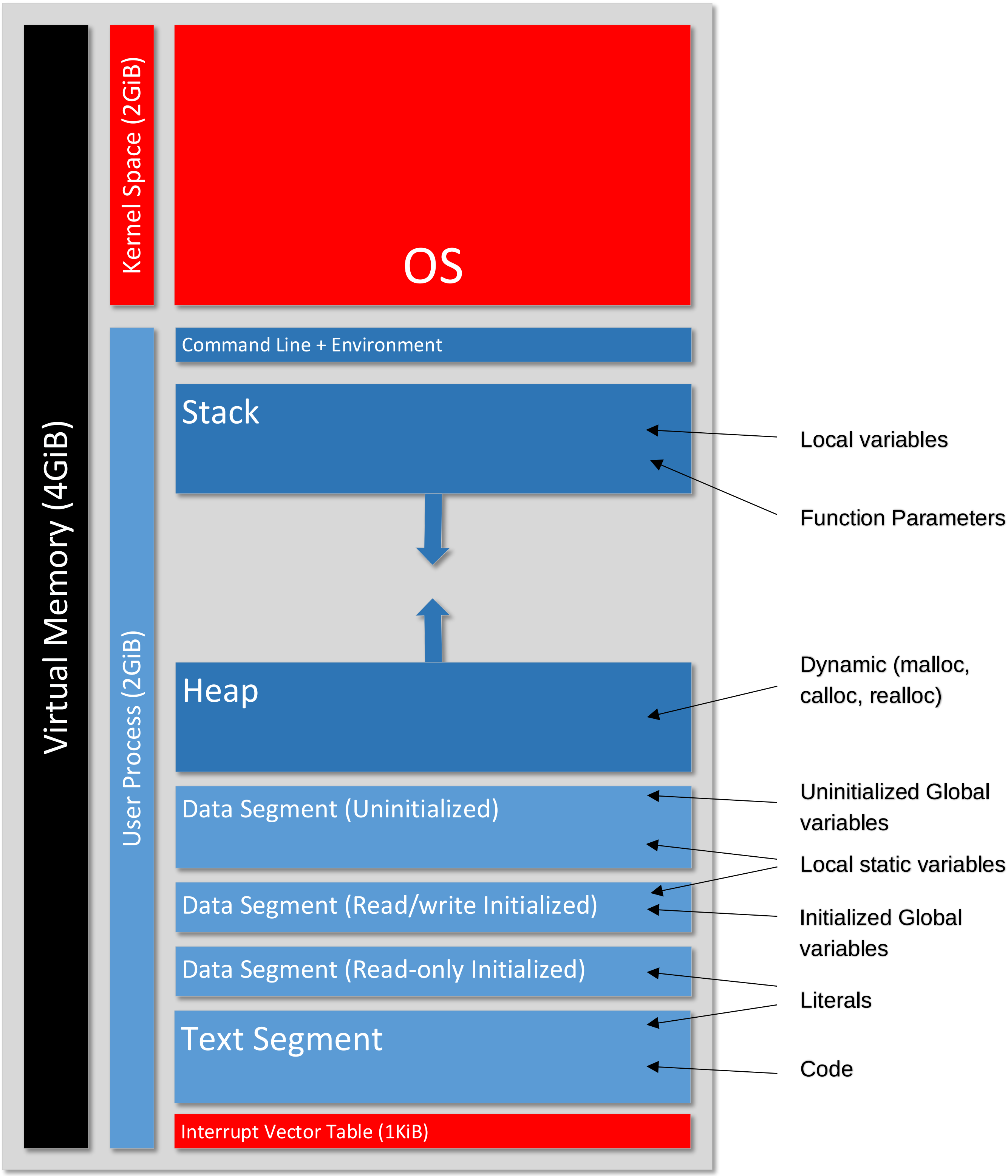
Static class attributes [C++, Java]



Static class attributes override the class's normal allocation location and follow the rules of global variable allocations.

```
class Foo {
  int a;
  static int b;
};

int main() {

  Foo bar;
}
```

Appendix A: C Memory Layout



Terminology

POD	Plain-old-data. A variable that can be copied by duplicating the binary representation of the variable. No additional process is required to copy the value.
process	A running program.
virtual memory	A logical memory system that maps virtual memory locations to physical memory locations. This permits more than one process to operate in memory at the same time, without the process knowing of the other processes existence.

References

- C dynamic memory allocation – https://en.wikipedia.org/wiki/C_dynamic_memory_allocation