Variables in Memory

The computer has finite resources for "remembering" things. So, you can't just keep asking it to remember data without at some point also telling it that it's ok to forget the data. Otherwise, at some point it will be completely full of stuff you told it to remember, and it just won't be able to remember anything new, even if most of that data isn't even accessible through variables anymore (this is called a *memory leak*). The duration of time between remembering the data (allocating space for it in memory) and telling your computer that it can forget the data (freeing/deallocating the space) is called the *lifetime* of the memory.

Date Lifetime

Three categories for the lifetime of memory in order to understand how variables are stored:

- **Global**: This memory is around for the entire lifetime of your program. Generally this is a fixed amount of space that is allocated right when the program starts, and it cannot grow.
- **Local**: The memory is allocated for you when entering a particular portion of the program, and automatically freed for you when exiting that part. This is denoted with curly braces `{}`, so it could be the scope of an...
  - entire function
  - `if` statement
  - loop
  - Anything inside curly braces `{}`

  Inside a block scope have limited lifetime and visibility. Outside that block, those variables no longer exist.
- **Dynamic**: This memory has a lifetime which is unknown at the time of allocation. The program explicitly asks for a specific size of memory, and then that memory lives until the program explicitly says the memory is no longer needed.
  - In C++, this is done via `malloc` or `new` and then the corresponding `free` or `delete` calls.

    > Many languages (like Python, but not C++) uses a "Garbage Collector" takes control of the lifetime of dynamically allocated memory. You only ever call `new` and then there is a background algorithm always running which tracks which memory is still being used and frees it for you. This avoids most causes of memory leaks, but it cost performance. C++ puts the burden on the programmer to do it manually (and correctly) to be more performant.

Static, Stack, and Heap

Stack, heap, or in static/global memory are stored in RAM (Random Access Memory).

They occupy different regions of RAM based on their storage duration and allocation mechanism.

Memory Layout Representation

```
|-----------------------------------|  <- High Memory Address (ends at 0xFFFFFFFFFFFFFFFF)
|       Stack                       |  <--- Grows Downwards
|  (Local variables, function calls)|
|-----------------------------------|
|       Unused Space                |  <--- Space available for future allocation
|-----------------------------------|
|       Heap                        |  <--- Grows Upwards
|  (Dynamically allocated memory)   |
|-----------------------------------|  <- Low Memory Address (starts at 0x0000000000000000 / null pointer)
|       Text Segment                |  <--- Program instructions / Compiled Sourcecode / Machine Code in 0 & 1
|-----------------------------------|
|       Initialized Data            |  <--- Global and static variables (initialized)
|-----------------------------------|
|       Uninitialized Data (.bss)   |  <--- Global/static variables (uninitialized)
|-----------------------------------|
```

Directionality Matters: The upward growth of the heap and downward growth of the stack is crucial for efficient memory utilization, error detection, performance, and cache optimization.

Safety Mechanism: This design allows for better control over memory usage and helps prevent memory-related errors, making it an important aspect of system and application architecture.

Memory Addresses

Within RAM there is Memory addresses, which are fundamental to how data is accessed and manipulated in programming. They allow the CPU to locate and retrieve data stored in RAM, making them essential for efficient computation and memory management.

In computer systems, a memory address is a unique identifier for a location in memory where data is stored. Whether a memory address is high or low does not affect speed, but rather designates location. Memory addresses are typically represented in hexadecimal format. Here are a few examples:

- **Hexadecimal Format**:
  - 32 bit system:
    - `0x00400000`
    - `0x7FFF1234`
  - 64 bit system:

    - `0x00007FFDCAFE1234`
- **Example Usage**:

```
#include <iostream>


int main() {
    int x = 42;  // Declare an integer variable
    std::cout << "Memory address of x: " << &x << std::endl;  // Print the memory address of x
    return 0;
}
```

```
Memory address of x: 0x7fffc3a1a4b8
```

In C++ coding, the Static, Heap, and Stack are corelated with their associated variables, which are stored through memory addresses.

Variables in Memory

variable - refers to a named memory location that can hold data that might change during program execution.

- **basic variable** - Any entity that holds a value and has a type (like `int`, `float`, `char`, etc.)

```
int number;   // Declaring an `int` variable named `number`
```

```
int number = 42;   // Declaring and initializing `number` with 42.
```

- **complex variable** - entities such as pointers, arrays, and objects of a class

When you declare a variable, the system allocates memory to store the value, and that memory is identified by the variable name. The **compiler** (GCC, Clang, etc) maps these names to actual **memory addresses** when the program is compiled and executed.

Depending on the context (stack, heap, or static), this memory may reside in different regions:

1. Static Variables

Storage Duration:

- **Static Storage Duration**: The variable is allocated when the program starts and deallocated when the program ends, persisting for the entire lifetime of the program.

Scope:

- **Global Static**: When declared outside of functions or classes, static variables have file scope (internal linkage). They are visible only in the file or translation unit they are defined in.
- **Local Static**: When declared inside a function, static variables have function scope, but their value persists across function calls.

Memory Location:

- **Data Segment** (either `.bss` for uninitialized or `.data` for initialized variables) of the program's memory.

Initialization:

- **Global Static Variables**: Automatically initialized to zero (or `nullptr` for pointers) if not explicitly initialized.
- **Local Static Variables**: Initialized the first time the code execution passes through their declaration (e.g., in a function). If not explicitly initialized, they default to zero.

When to Use:

- **Need for a variable to persist across function calls** or to exist throughout the lifetime of the program.
- **Global configuration data** or constants that should be shared across the entire program but should not be modified frequently.
- To **preserve the state of a local variable** in a function between calls (like a counter)

Pros:

- No need for manual memory management.
- Efficient for persisting values across function calls or through the program's lifetime.

Cons:

- Overuse of global static variables can lead to **tight coupling** and **difficult debugging**.
- Can increase **memory usage** if used excessively, as memory is allocated for the entire program duration.

```
#include <iostream>

static int globalStatic = 42; // Static global variable (file scope)

void function() {
```

```
    static int localStatic = 10; // Static local variable (function scope)
    localStatic++;
    std::cout << "Local Static: " << localStatic << std::endl;
}

int main() {
    function(); // Prints 11
    function(); // Prints 12
    return 0;
}
```

2. Heap (Free Store) Variables

Storage Duration:

- **Dynamic Storage Duration**: Variables on the heap are dynamically allocated using `new` (or `malloc` in C) and persist until they are explicitly deallocated using `delete` (or `free`).

Scope:

- The **scope** of a pointer to a heap-allocated variable depends on where the pointer is declared, but the memory allocated on the heap persists independently of the pointer's scope until manually freed.

Memory Location:

- **Heap** is managed dynamically.

Initialization:

- **Non-initialized heap variables** are left with **indeterminate values** unless you explicitly initialize them.
- **Heap-allocated objects** are initialized through constructors if they are class objects.

When to Use:

- When you **don't know the size of the data at compile time** (e.g., user input, file data, dynamically sized arrays).
- When **objects need to persist beyond the scope of the function** that created them.

- For **large objects** that might exceed the limited size of the stack (e.g., large arrays, buffers, or structures).
- When you need to allocate memory for **complex data structures** (like linked lists, trees, or graphs) that can grow or shrink dynamically.

Pros:

- Provides flexibility to allocate memory at runtime based on dynamic needs.
- Can allocate larger chunks of memory that may not fit in the stack.

Cons:

- **Manual memory management** is required (you must use `new` to allocate and `delete` to free).
- Potential for **memory leaks** if memory is not properly deallocated.
- Slower than stack memory due to the overhead of dynamic memory management (allocation and deallocation).

```
#include <iostream>

int main() {
    int* heapVar = new int(5); // Dynamically allocated on the heap
    std::cout << "Heap Variable: " << *heapVar << std::endl;

    delete heapVar; // Must be manually deallocated to avoid memory leak
    return 0;
}
```

3. Stack Variables (Automatic Storage Duration)

Stack will pretty much never exhaust it unless

- you define absurdly large objects on the stack (e.g. an array of milions of objects)
- you recurse too deeply (usually as a result of a bug of infinite recursion or unnecessarily large stack frame size)

Unique to stack is the **stack frame**, which is a structured block of memory on the stack that is created when a function is called.

It holds:

- Local variables

    - Function parameters
    - Return address (where to return after the function call)
    - Saved registers and other context information

- **Lifetime**: The stack frame is automatically created when the function is invoked and destroyed when the function exits.
- **No "Heap Frames" or "Static Frames"**: Heap memory is more flexible and does not use frames. Instead, it uses blocks of memory allocated as needed, allowing for dynamic data structures and sizes.
- Example: https://caseymuratori.com/blog_0015

Storage Duration:

- **Automatic Storage Duration**: Variables declared on the stack (inside a function or block scope) are automatically allocated when their block of code is entered and deallocated when the block is exited.

Scope:

- Stack variables have **local scope**, which means they are visible only inside the function or block in which they are declared.

Memory Location:

- **Stack**, which is a part of memory where automatic variables are stored.

Initialization:

- **Uninitialized primitive types** (like `int`, `char`, etc.) in C++ have **indeterminate values**.
- Local variables of class types are initialized through constructors.
- Always good practice to initialize stack variables to avoid undefined behavior.

When to Use:

- For **simple, small variables** that are **local** to a function or block.
- When the **size of the variable is known at compile time**.
- When you need **automatic memory management**—stack variables are automatically deallocated when they go out of scope, making them safer and faster.
- When **short-lived** variables are sufficient (i.e., variables that don’t need to persist beyond their block or function scope).

Pros:

- **Automatic memory management**: No need to explicitly free memory; variables are automatically destroyed when they go out of scope.

- **Fast allocation and deallocation** compared to heap allocation.
- Safer in terms of preventing memory leaks, as memory management is handled by the program.

Cons:

- **Limited by stack size**: Large variables can cause stack overflow, especially with recursion or large data structures.
- Variables are destroyed as soon as they go out of scope, so they can't persist beyond their function or block.

```
#include <iostream>

void function() {
    int stackVar = 10;  // Allocated on the stack
    std::cout << "Stack Variable: " << stackVar << std::endl;
} // 'stackVar' is destroyed here when function exits

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

REFERENCES

Stack Vs Heap: Key Difference Between Stack & Heap Memory | Simplilearn

Revision #23
Created 24 September 2024 21:51:45 by victor
Updated 13 October 2024 21:47:18 by victor