Pointers, References, and Computer Memory

Explained and Illustrated Using C and C++

Written by Johnny (pse. D7EAD)

Abstract

This document, *Pointers, References, and Computer Memory*, aims to supply the reader—one who is assumed to have at least minimal experience in the C/C++ programming languages—with detailed descriptions and illustrations of both the use of and the structural components of memory in computer systems and how to manipulate them via the use of programmatic pointers and address references. To make the reader's experience more balanced, this text follows a chronological order of introduction regarding concepts of memory. For instance, this text begins with a brief yet descriptive detail of the history and evolution of memory in computing; the reader is then introduced to numerical systems—being hex and binary—memory layout, memory manipulation and addressing concepts, object files and assembly, and, ultimately, efficient use of programmatic pointers. Following the completion of this document, the reader will have successfully obtained a deeper understanding of several memory and memory-oriented programming concepts such as how memory (stack, heap, virtual address spaces, assembly segments) and values in memory are structured in computer systems, bitwise operations on numerical systems and manipulation of data stored in memory, and how to efficiently—and safely—implement and take advantage of pointers and address references using both the C and C++ programming languages.

Table of Contents

I. Introduction *History and Evolution of Computer Memory*

Consider the dedicated memory in your device, whether it be a portable laptop or a desktop—how much do you have? Four, eight, sixteen, or even 32 gibibytes of random-access memory? Is it utilizing dual channel? What about the frequencies of your cartridges? Now, after internally answering, wipe that from your memory and let us take a trip to a time when memory capacities were in the low bytes—compared to today's several GiB average.

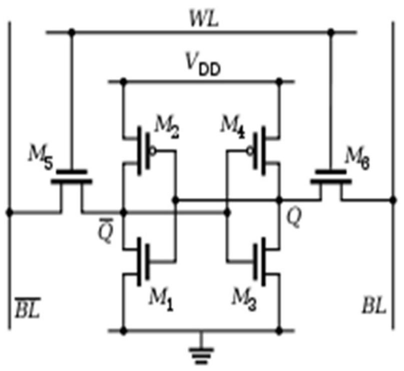
In the early '40s, the most advanced computational devices you could come across were those with only a few bytes of memory—those that could only perform simple arithmetic between numbers. As time went on towards the early '50s, memory capacities continued to grow as new methods of storing and reading memory were innovated. One such example was **delay line memory** which processed information in the form of soundwaves inside a glass tube of mercury, isolated with quartz crystals acting as the gate to reading and writing the bits. As interesting as it sounds, delay line memory was limited to an upper limit of a few thousand bits.

These archaic forms of holding information preceded that of random-access computer memory. In 1947, the first form of **random-access computer memory** was invented—the *Williams tube*. This design stored data in the form of electrically charged spots on a cathode ray tube (CRT)—the Williams tube was designated as random-access since the device's associated electron beam had the ability to read and write spots on the tube in a random order. Although the capacity of the design was a few hundred to a thousand bits, it was much faster and efficient compared to older designs like vacuum triodes.

Following several improvements after the first RAM implementation, MOS semiconductor memory appeared. Being cheaper and more efficient than its predecessors, it is not only the base for modern **dynamic** random-access memory (DRAM) implementations, but also led to the development of **synchronous** dynamic random-access memory (SDRAM) and **static** random-access memory (SRAM). In SRAM, a bit is stored using the state of a six-transistor memory cell. Although more expensive, this form of RAM is generally faster and requires less dynamic power than DRAM. In DRAM, a bit of data is stored using a transistor and capacitor pair which, together, comprise a cell. The capacitor holds a high or low charge—being zero or one—and the transistor acts as a switch that lets control circuitry on the chip read and write the capacitor's state of charge. As DRAM is the less expensive of the types, it is the most prominent form of memory in today's modern machines.

"What is a memory cell?" You may ask. Simply, a memory cell is the base building block of computer memory. Memory cells are electronic circuits that store one bit of binary information, and it must be set to store a logic 1 (high) and reset to store a logic 0 (low). Its value is cached until it is altered by the set/reset process—the value in the memory cell can be accessed by simply reading it. Below you can find a graph of an SRAM cell (6 transistors).

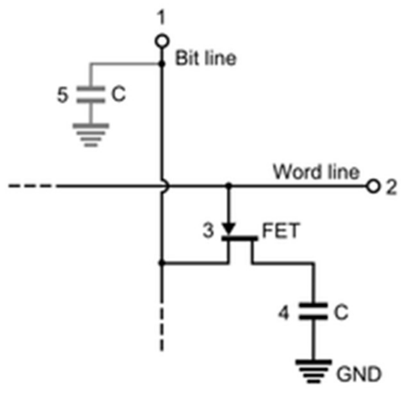
Figure 1: 6T SRAM cell ***(M_x transistors; M_1-M_4 hold charge states, M_5-M_6 delegate RW access)***



The above memory cell is a circuit of type flip-flop. SRAM requires low power when access is null but is expensive and has low storing density.

Below you can find a graph of a much simpler DRAM memory cell (1 transistor, 1 capacitor).

Figure 2: 1T1C DRAM cell (#3 transistor, #4 capacitor; #4 holds charge, #3 delegates RW access)



The above is based on a one-transistor-one-capacitor (1T1C) design. Charging and discharging this capacitor (#4) can store a high value or low value in the cell. Disadvantageously, the charge in this capacitor slowly leaks away and must be refreshed periodically via the read/write gate at transistor #3; due to this, DRAM uses more power yet achieves greater storage densities and lower unit costs compared to the former.

In the above graphs of memory cells, you may be wondering, "well, how do you read and write memory cells?" Similar to, but not so similar to, knowing the physical address to write and send a letter, memory cells incorporate multiplexing and demultiplexing circuitry to select cells. In RAM devices, there are a collection of address lines—A_0... A_n—and, for each combination of bits that may be applied per line, consequently, a set of cells are activated appropriately.

Binary

When it comes to memory—and computers in general—on the lowest level, all information is transmitted, parsed, and stored using the **binary** number system. Binary is a base-2 numerical system (using only integers 0 and 1) that can be used to represent whole numbers, decimals, and other forms of data. Due to its straightforward design and implementation in circuitry using logic gates, binary is the preferred numerical system by nearly every modern computer-based device for component communication and operation.

Binary follows a right-to-left significance order (low to high byte), meaning, as the value of the represented number increases, the binary value will expand to the left. Below you can find an example of the binary representation of **1197** and how to convert from binary to decimal.

Formula: *$B\times2^n + B\times2^n + \ldots$ (left-to-right n ending at ZERO); where B=bit, n=position*

1	0	0	1	0	1	0	1	1	0	1
1×2^{10}	0×2^9	0×2^8	1×2^7	0×2^6	1×2^5	0×2^4	1×2^3	1×2^2	0×2^1	1×2^0

= 1197

The largest value representation a binary number can hold depends on the number of bits available. For instance, in the below table, you will see how to determine the maximum representable value of a binary number.

Formula: *$T_{10} = 2^n$; where T=total possible base-10 values (including zero), n=total number of UNSIGNED bits*

0	0	-	-	-	-	-	-	$T_{10} = 2^2 = 4$: (0-3)
0	0	0	0	-	-	-	-	$T_{10} = 2^4 = 16$: (0-15)
0	0	0	0	0	0	-	-	$T_{10} = 2^6 = 64$: (0-63)
0	0	0	0	0	0	0	0	$T_{10} = 2^8 = 256$: (0-255)

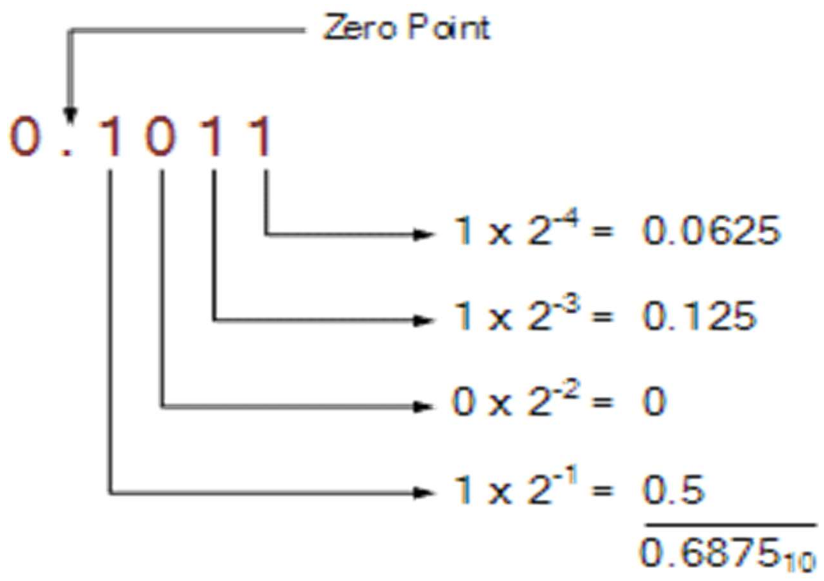
Binary is also able to represent precision numbers—such as 10.532—with the incorporation of a dot-operator (.)—or more fancily called a **radix point**— in the binary number. Below you can find a table and accompanying graph explaining binary-radix format to floating-point precision conversion.

Left Formula: $B \times 2^{n} + B \times 2^{n} + \ldots$ *(left-to-right n ending at ZERO); where B=bit, n=position*

Right Formula: $B \times 2^{-n} + B \times 2^{-n} + \ldots$ *(left-to-right n starting at -1); where B=bit, n=position*

1	1	1	1	.	1	0	1	1
1×2^{3}	1×2^{2}	1×2^{1}	1×2^{0}	.	1×2^{-1}	0×2^{-2}	1×2^{-3}	1×2^{-4}

= 15.6875



Hexadecimal

Hexadecimal is an alternate representation of binary that is simpler to interpret than binary's often long, multi-word binary numbers—a word being 2 bytes, or 16 bits. Hexadecimal, or simply hex, is a base-16 numerical system often used to more simply represent data such as memory addresses in process virtual address spaces—which we will dive into later. Hex allows the use of digits (0-9) and letters (A-F) for a total of 16 values. Each hex character is worth a total of four bits—as 4 bits has a maximum value set of 16 (2^4 | 0 to 15).

For example, below you can find a hex-binary table with corresponding values.

*Hex values are most-commonly denoted by a preceding **0x***

0xF	0x10	0xF1	0x90	0xA0
1111	00010000	11110001	10010000	10100000
15	16	241	144	160

Hex can also represent floating-point precision numbers in a way similar to that of binary's representation of such numbers. Hex achieves this by placing a radix point between characters like shown below.

0xFF.0A	0xF.3A	0x0.02	0xC3.7	0xFF.A9
11111111.0000101	1111.0011101	0.0000001	11000011.0111	11111111.10101001
255.0390625	15.2265625	0.0078125	195.4375	255.66015625

As you can see in the examples above, hexadecimal is simply an easier-to-understand representation of binary and the values it represents. In the coming entries in this text, we will expand further upon hexadecimal to illustrate how it is utilized to address locations in process memory space—and more concepts regarding hex.

Overflow

Let us assume, for a moment, that we have a 32-bit integer at our disposal and, just for simplicity, let us also assume it is **unsigned** (non-negative). As we explored earlier, the formula to determine the maximum number of values that a binary number can represent is $T_{10} = 2^n$ (including zero). A 32-bit unsigned integer has a value range of 0-4294967295 ($T_{10} = 2^{32}$; $T_{10} =$ 4294967296).

The value 4294967295 represented in binary is as follows:

Unsigned integer; 32-bits available

$$\boxed{11111111\ 11111111\ 11111111\ 11111111}$$

= 4294967295

Now, after seeing all 32 bits occupied with ones, you may be asking yourself, "what could possibly happen if we add to this already maximum binary number?" Well, if you were to do such a thing, you would be asking for a **binary overflow**. Since the value our binary number is representing is an unsigned integer—which holds only 32 bits—that means, unsurprisingly, that we can fit **ONLY** values in the unsigned 32-bit range of 0-4294967295.

However, if you were daring enough to perform an operation such as (4294967295 + 2) on the above binary value (overflowing it past its value range), it would result in the following:

Unsigned integer; 32-bits available

$$\boxed{00000000\ 00000000\ 00000000\ 00000001}$$

= 2

As expected, the entire binary number has overflowed and *reset* back to the beginning—plus what we added to it, being two.

Now, let us go a step further—how does this work with negative numbers? Similarly, however, in order to understand what comes next, we must briefly explain **signed numbers.** Signed values are values that allow the holding of both a negative and positive value with the takeaway of losing one bit for *signing*—being the most significant bit (leftmost). This effectively splits your original (unsigned) value limit of 4294967295 in half—half negative, half positive.

This means that, the formulas to find the maximum possible values (negative and positive) for a signed integer are as follows:

Formulas

$$\textbf{\textit{Positive Values}}: T_{10} = 2^{n-1}-1$$

$$\textbf{\textit{Negative Values}}: -T_{10} = -(2^{n-1})$$

where T=total possible base-10 values (including zero for positive segment), n=total number of UNSIGNED bits

With these formulas in mind, below you will find tables illustrating the maximum ranges for both positive and negative segments of 32-bit signed integers.

Positive ranges for signed 32-bit integers

Decimal	0	2147483647
Binary	**0**0000000 00000000 00000000 00000000	**0**1111111 11111111 11111111 11111111

Negative ranges for signed 32-bit integers

Decimal	-1	-2147483648
Binary	**1**1111111 11111111 11111111 11111111	**1**0000000 00000000 00000000 00000000

*Signed (reserved) bits signified by bold **1** or **0** (signs | 1:negative; 0:positive)*

Notice how, when the integer signs flip negative, the signed bit—being the left most 32nd bit—flips to one, signifying it is now negative. Nevertheless, you may be wondering, “wait, I thought a 32-bit binary number populated with all 1s was 4294967295, not -1?” This is a signed binary number; the 32nd bit is reserved! The 32-bits of 1 is -1 because of **Two’s Complement**.

Bitwise Operators

Before you can understand the concept of Two's Complement, you must first understand the purpose and function of bitwise operators—operators, like those seen in math (+, -, ×, ÷), but ones that operate on the binary representation of a number instead of the number itself.

Below you can find a table with each bitwise operator and a description of their function.

&	Compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0. (assume **only** 4 bits available)* Example: **1011(11)** **& 0101(5)** **= 0001**
|	Compares two bits and returns 1 if either or both of the bits are 1 and it gives 0 if both bits are 0. * Example: **1011(11)** **| 0101(5)** **= 1111**
^	Compares two bits and generates a result of 1 if the bits are complementary (one 1, one 0); otherwise, it returns 0. * Example: **1011(11)** **^ 0101(5)** **= 1110**
~	Inverts all bits of a given binary number (i.e., 1111 flips to 0000). * Example: **~1011(11) -----> 0100**
>>	Moves the bits to the right, discards the far-right bit, and assigns the leftmost bit a value of 0. If value being shifted is signed, the leftmost bit is filled to replace shifted bits' previous locations. * Example: (unsigned) Example: (signed) **1011(11) >> 2 -----> 0010** **1000(-8) >> 2 -----> 1110**
<<	Moves the bits to the left, discards the far-left bit, and assigns the rightmost bit a value of 0. Always fills rightmost bits with 0 to replace moved bits' previous locations. * Example: **1011(11) << 2 -----> 1100**

Two's Complement

Two's Complement is an operation on binary numbers that is used for signed binary number representations. To answer the question proposed earlier—being *why a 32-bit signed integer populated with all ones is* ***-1*** *and not* $\mathbf{2^{32}-1}$—we must learn how to convert to and from signed binary negative and positive numbers. You can do this via the help of Two's Complement.

Two's Complement follows the following formulas:

Formulas

Positive to Negative: $-I_{10} = (\sim N_{10}) + 1$

Negative to Positive: $I_{10} = \sim(N_{10} - 1)$

where I_{10}=resulting new base-10 signed number, N_{10}=original signed base-10 number to flip sign on

For example, below you can find a table illustrating Two's Complement on signed 8-bit binary numbers.

All below numbers are signed—Two's Complement is a ***signed*** *operation.*

Signed Integers	Two's Complement.	Results
127 (01111111)	$(\sim 127_{10}) + 1$	-127 (10000001)
-127 (10000001)	$\sim(-127_{10} - 1)$	127 (01111111)
64 (01000000)	$(\sim 64_{10}) + 1$	-64 (11000000)

8-bit signed integers have value ranges ***(0 to 2^7-1)*** *and* ***(-1 to -2^7)***

As you can see, when signed integers become negative or become positive, Two's Complement is invoked, and the signs are flipped—causing the leftmost reserved bit to flip to **1** or **0** (1:negative, 0:positive).

Now that the concept of signed and unsigned numbers is fresh in mind, you may be wondering, how do CPUs differentiate between the two when both use Two's Complement signed representation? Consider you are given two 32-bit binary numbers…

(assume signed, value: -2^{31})

11111111 11111111 11111111 11111111

and

11111111 11111111 11111111 11111111

(assume unsigned, value: $2^{32}-1$)

…how can you—or a CPU—tell which is unsigned, and which is not without prior context…? They look exactly the same! The simple answer: it does not.

The thing about Two's Complement notation—which we went over previously—is that the CPU simply *does not care* if the given values are signed or unsigned; the CPU simply processes the given information using the given instructions and returns the result—it is up to you, as the programmer, to keep track of what your bit patterns represent.

Keep in mind though that the ignorance of the CPU for signed and unsigned values only applies to operations where it **does not matter**. For instance, operations such as equality, inequality, addition, subtraction, and multiplication can be seen as *sign-agnostic*—meaning there are no separate assembly instructions for the operations, and they operate the same on both signed and unsigned values. However, operations such as division, greater-than and less-than, greater-than or equal to, and less than or equal to are not sign-agnostic and require separate assembly instructions for their arithmetic on signed and unsigned values.

One example is x86 assembly's `div` and `idiv` instructions—unsigned, signed division.

II. Memory ***Virtual Address Spaces, Physical Memory, and Layout***

Before we discuss various concepts regarding the structure of process memory (specifically in the Windows operating system) and locations within it, we must set the foundation and explain the concepts of *virtual addresses* and *virtual address spaces*.

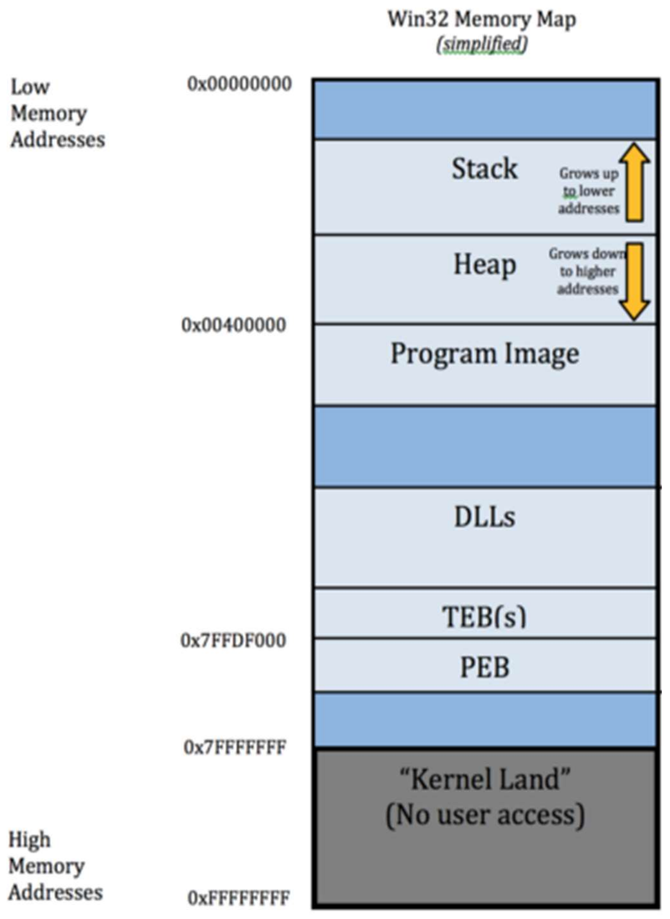
Virtual address space (VAS) is an abstract concept where each process is given a collection of virtual addresses—specific areas in memory where data is held represented by hexadecimal numbers—that map to addresses in your physical memory; this address space is private for each process and cannot be accessed by other processes unless shared. These virtual address mappings **do not** represent actual addresses of an object or value in physical memory; instead, the operating system holds a *page table* for each process. The table is responsible for translating virtual addresses within processes to addresses in physical memory. A page is a fixed-length contiguous block of virtual memory, described as a single entry in the page table.

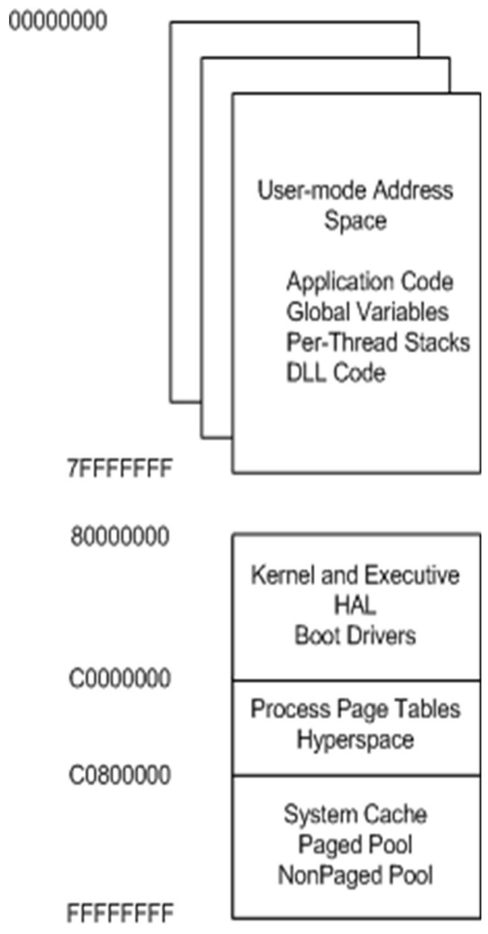
Each process' virtual address space has a maximum size depending on the architecture of your operating system. However, before we describe the memory limits of processes on 32-bit and 64-bit Windows, we must first explore *user space* and *system space*.

A processor running the Windows operating system has two different code-execution modes—*user-mode* (operates in user space) and *kernel-mode* (operates in system space). User-mode is the mode processors run code in the context of when the executing code is unprivileged in the context of the operating system (i.e., does not directly interact with system hardware, runs in the context of the user, etc.); Windows applications and user-mode drivers run in user mode, for example. Kernel-mode is switched to by processors when running privileged code that directly affects the operating system (i.e., core components, kernel-mode drivers, etc.).

Each process on Windows has its own virtual address space, as described before. However, each process' virtual address space has two partitions—one for user space memory and one for system space memory. While each process has its own, independent user space virtual address space, system space shares a **single** virtual address space.

Below you can find two simplified 32-bit memory maps of Windows processes:





As you can see, in the above examples, process memory is divided into two partitions per process. In the above images, both processes follow a 2/2 split; a 2GB address range for private user space and a 2GB address range for system shared kernel space—user space: **0x00000000 to 0x7FFFFFFF**, system space: **0x80000000 to 0xFFFFFFFF**.

We will soon dive deeper into the individual memory sections in the previous images. Nevertheless, in 64-bit Windows processes, the virtual address space structure is very similar to the previous images; however, due to the large address range that comes with 64-bit addresses (only the lower 48-bits are utilized), each partition is granted a larger address limit.

Below you can find tables describing recent Windows and Windows Server physical memory and virtual addresses space limits.

Process Memory and Virtual Address Space Limits

	32-Bit Limit	64-Bit Limit
User-mode virtual address space for each 32-bit process	2 GB Up to 3 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** and 4GT	2 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** cleared (default) 4 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** set
User-mode virtual address space for each 64-bit process	Not Applicable	**With IMAGE_FILE_LARGE_ADDRESS_AWARE** set (default): **x64: Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB **x64: Windows 8 and Windows Server 2012 or earlier** 8 TB 2 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** cleared
Kernel-mode virtual address space	2 GB From 1 GB to a maximum of 2 GB with 4GT	**Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB **Windows 8 and Windows Server 2012 or earlier** 8 TB

Physical Memory Limits (32, 64)

Windows 10		
Enterprise	4 GB	6 TB
Education	4 GB	2 TB
Pro for Workstations	4 GB	6 TB
Pro	4 GB	2 TB
Home	4 GB	128 GB
Windows Server 2016		
Datacenter	N/A	24 TB
Standard	N/A	24 TB
Windows 8		
Enterprise	4 GB	**512 GB**
Pro	4 GB	**512 GB**
Standard	4 GB	**512 GB**
Windows Server 2012		
Datacenter	N/A	4 TB
Standard	N/A	4 TB
Essentials	N/A	64 GB
Foundation	N/A	32 GB
Windows 7		
Ultimate	4 GB	192 GB
Enterprise	4 GB	192 GB
Pro	4 GB	192 GB
Hom Basic	4 GB	8 GB

In addition, below you can find a table showing the range of available addresses for both user space and system space partitions in processes:

Process Memory and Virtual Address Space Limits

Partition Split

User-mode virtual address space for each 32-bit process	2 GB: User Space: 0x00000000 – 0x7FFFFFFF 3 GB: User Space: 0x00000000 – 0xBFFFFFFF		Kernel-mode virtual address space for each 32-bit process.	2 GB: Kernel Space: 0x80000000 – 0xFFFFFFFF 1 GB: Kernel Space: 0xC0000000 – 0xFFFFFFFF
User-mode virtual address space for each 64-bit process	**Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB 0x0000000000000000 – 0x00007FFFFFFFFFFF **Windows 8 and Windows Server 2012 or earlier** 8 TB 0x0000000000000000 – 0x000007FFFFFFFFFF		Kernel-mode virtual address space for each 64-bit process.	**Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB 0xFFFF800000000000 – 0xFFFFFFFFFFFFFFFF **Windows 8 and Windows Server 2012 or earlier** 8 TB 0xFFFFF80000000000 – 0xFFFFFFFFFFFFFFFF

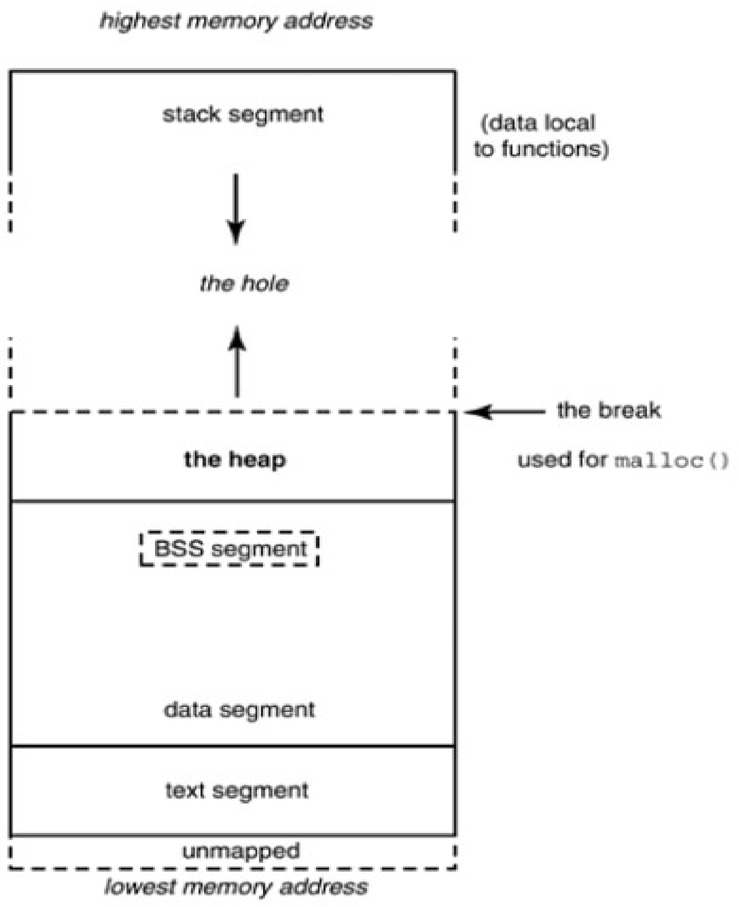
VERY IMPORTANT: do **NOT** let the 1s in bits 64-49 of the 64-bit addresses confuse you. Bits 64-49 and 64-45 are **RESERVED** in newer and older systems and set to **ZERO** in user mode and **ONE** in kernel mode address spaces. Simply ignore them and pay attention to the lower 48 or 44 bits—that is where the range is.

While this table may seem confusing at first, it is simply showing the ranges of virtual addresses for both user and system space partitions in a process depending on the architecture and version of Windows. 32-bit and 64-bit address ranges are relatively simple:

- **2GB user space partition begins:** 0x00000000; **ends:** 0x7FFFFFFF (total: 2 GB)
- **2GB system space partition begins**: 0x80000000; **ends:** 0xFFFFFFFF (total: 2 GB)
- **On systems Windows 8.1 and Windows Server 2012 R2 or later:**
  - **128 TB user space partition:**
    begins: 0x0000000000000000
    ends: 0x00007FFFFFFFFFFF (total: 128 TB)
  - **128 TB system space partition:**
    begins: 0xFFFF800000000000
    ends: 0xFFFFFFFFFFFFFFFF (total: 128 TB)
- **On systems Windows 8 and Windows Server 2012 or earlier:**
  - **8 TB user space partition:**
    begins: 0x0000000000000000
    ends: 0x000007FFFFFFFFFF (total: 8 TB)
  - **8 TB system space partition:**
    begins: 0xFFFFF80000000000
    ends: 0xFFFFFFFFFFFFFFFF (total: 8 TB)

Seen above, recent systems have an address range of 48-bits (128 TB) per VAS partition, and older ones have a 44-bit (8 TB) range per partition. Unused bits are preceded with 0(user) and 1(system).

Mapping of a process' user space virtual address space; system space above stack.



In the images you have previously seen, being rather simplified diagrams of the structure and layout of a process' virtual address space, you may have noticed sections that you are not familiar with—such as the stack, heap, system paged and non-paged pools, dynamic-link library imports, and even ones you may have not even heard of—**.data**, **.bss**, **.text**, etc.

In the following section we will explore these sections individually and their purposes within the context a process' virtual address space.

Segments – User Space

The Stack:

The stack, within a process, is an area that holds temporary variables and data—often created by functions—which are declared, stored, and initialized during runtime. Since the data the stack is meant to hold is temporary—such as function parameters, local variables, etc.—once a piece of data goes out of scope, it is erased, and its location can be replaced in the stack immediately if need be; automatic variables are also allocated on the stack. A *stack pointer* register tracks the top of the stack and is adjusted each time a new value is added to the stack—or pushed to it. The values pushed for a function call are called a *stack frame*; a stack frame contains, at minimum, a return address. Each thread of execution in a process has its own stack while only sharing one heap.

The stack segment traditionally adjoins the heap segment, and they grow towards each other; when the stack pointer met the heap pointer, free memory exhausts. With large address spaces and virtual memory techniques they tend to be grouped more freely, but they still typically grow towards each other. On an x86-architecture PC, the stack grows toward address zero, meaning recent items, deep in the call chain, are at lower addresses and closer to the heap. On other architectures, it grows the opposite direction—as can the stack.

The Heap:

Within the context of a program, *the heap* is the segment in the process' virtual address space where dynamically allocated data is held. The heap begins following the BSS segment of the virtual address space and grows towards higher addresses from there. In a given process, this segment is shared by all shared libraries and dynamically loaded modules.

The Data Segment (.data):

This segment, also known as the *initialized data segment*, holds global and static variables in a program that have a pre-defined value set on them and can be modified. The values for the variables stored in the data segment are initially located in read-only memory and subsequently copied to the data segment during the startup routine of a given process.

The BSS Segment (.bss):

This segment, also known as the *uninitialized data segment*, is usually located next to the data segment. This segment contains all global and static variables defined in a program that are not explicitly initialized to a value or are set to zero in source code. On some platforms, some—or the entirety of—the BSS segment is initialized to zeroes.

The Memory Mapping Segment:

For operating systems that support memory-mapped files, the memory mapping segment can be found between both the heap and the stack. This segment is responsible for holding the contents of mapped files supplied by the system's kernel to the process; processes can request such mappings via the use of system-specific functions such as Linux's *`mmap()`* and Windows' *`CreateFileMapping()`* and *`MapViewOfFile().`* Memory mapping is a highly efficient way to perform file input/output operations, so it is often used for loading of shared dynamic libraries. Another feature of memory mapping is the creation of *anonymous memory maps* which allows program data to be mapped, rather than files.

There are other segments that can be found in processes specific to platform-dependent file formats—such as Windows' PE format—but the above is the general segments usually found in object files and virtual address spaces on systems.

System Space

As system space's virtual address space is shared among processes, think of it being comprised of and containing privileged system components and layers rather than segments—like the ones previously mentioned.

Hardware Abstraction Layer (HAL):

The Windows Hardware Abstraction Layer is a major component of the Windows kernel that offers a uniform interface between the operating system and the underlying hardware as well as collections of privileged libraries and drivers that abstract the interaction process between software and physical hardware. Simply, the job of HAL is to hide hardware differences from operating system so that code can run independent of the underlying hardware. HAL is also responsible for initializing the central-processing unit and the System Interrupt Controller during the kernel initialization process (interrupts are not yet enabled until later in the kernel initialization process); control is then returned to the kernel following the start of the interrupt controller.

The HAL is responsible for the entire kernel's construction of the basic virtual memory structure in the operating system. Once control is returned to the kernel following controller initialization, the kernel invokes the **Memory Manager Executive (MME) service** which constructs page tables and internal structures necessary to provide minimal memory services. It is responsible for the segmentation of the virtual address space into two partitions—user and system. It creates zones for the file system cache, paged and non-paged pools of memory and page table which can be accessed by both the kernel and CPU. Allocation and deallocation of memory virtually and dynamically is managed by this Executive service.

Process Page Tables:

For a quick recap of what was mentioned several pages ago, a page table is a translation table that translates a process' virtual addresses to their corresponding addresses on physical memory. This collection of process page tables is stored in the system space's virtual address space that is shared among all processes. Specifically, the CPU's memory management unit (MMU) stores a cache of recent mappings from the operating system's process page tables in a *translation lookaside buffer* (TLB). When a virtual address translation is requested, the TLB is the first searched and, if no match is located, the MMU or operating system's TLB *miss handler* will search for the mapping in the page tables. If one exists, it is written back to the TLB, and the faulting instruction is restarted—it will then be found in the TLB, and the access will pass. Although, sometimes virtual address translation can fail—causing a *page fault*.

Hyperspace:

Hyperspace is responsible for mapping process working set list entries and temporarily map physical pages for operations such as zeroing pages, nullifying page table entries, and process creation.

System File Cache:

Responsible for caching data to be read and written to and from hardware storage disks. For instance, when a write operation is requested of the operating system, the file data is written to the system cache and then to the hardware disk by the cache. Inversely, a read request would read data from the system cache. File data resting in the system cache is written to the disk at intervals dependent on the operating system—and the memory previously used by that file is freed or *flushed*.

Memory Pools:

The memory manager creates the following memory pools that the system uses to allocate memory: nonpaged pool and paged pool. Both memory pools are located in the region of the address space that is reserved for the system and mapped into the virtual address space of each process.

- ***Non-Paged Pool:**
    - Consists of virtual memory addresses that are *guaranteed* to reside in physical memory as long as the corresponding kernel objects are allocated.
- **Paged Pool:**
    - The paged pool consists of virtual memory that can be paged in and out of the system.

To improve performance, systems with a single processor have three paged pools, and multiprocessor systems have five paged pools.

*Kernel objects are kernel structures such as access token objects, file objects, thread objects, file-mapping objects, process objects, event objects, mutex objects, and wait able timer objects that are created by calling various functions.

Now that we have a fresh and relatively better understanding on the mechanics of the concept of virtual memory in computing, we are going to explore the concept and purpose of *base addresses*, *relative virtual addresses*, and *offsets*.

Base Address:

In its simplest form, a base address is exactly as it sounds—an address used as a reference point for other addresses—absolute addresses—in memory. For example, in object files, the base address is the lowest virtual address associated with the program or a section in the program (*process base address depends on file format—i.e., PE and DLL differ*). Since the base address is used as a reference point for other addresses in the address space, under the relative addressing scheme, it can be used to access other values and parts of the program's virtual memory via the use of RVAs (offsets)—more specifically, by adding them to their base address.

Relative Virtual Address:

When it comes to the concept of object files, one of the most confusing parts is the relative virtual address (RVA). In image files, a relative virtual address is the result of subtracting a base address from a virtual address—it is the distance between the base address and the address it is subtracted from. For instance, given a base address of `0x00FFA022` and a virtual address of 0x00FFF033 holding a value, the relative virtual address of the value would be 0x00005011. RVA are usually relative to a process base but can also be relative to section bases.

```
(0x00005011)RVA = (virtual address)0x00FFF033 – (base)0x00FFA022
(0x00FFF033)virtual address = (base)0x00FFA022 + (RVA)0x00005011
```

Offsets:

Offsets can be quite confusing for some depending on the context you supply them. In its simplest definition, an offset is the relative distance between a base object and an absolute object. For instance, assume you have an array with ten integers with indexes starting at base zero; to reach the tenth integer at index nine, you would have to add an offset of nine to the base index of zero. In memory, an offset is the distance from a base address to an absolute address. For instance, an absolute address of `0x00900000` has an offset—or relative address—of `0x00500000` from base address `0x00400000`. You can reach the absolute address of `0x00900000` by adding the relative address—or offset—of it to the base address.

Get RVA of absolute by subtracting the base from absolute.

```
0x00500000(RVA) = 0x00900000(absolute) – 0x00400000(base)
```

Access absolute by adding its offset RVA to base.

```
0x00900000(absolute) = 0x00400000(base) + 0x00500000(RVA)
```

As a note of caution, this section and its accompanying subsections contain memory-oriented snippets of code that chronologically increase in complexity. As does this entire text, the following area assumes the reader has, at the very least, a minimal understanding of the C programming language and its superset, C++. This basic language knowledge is necessary to understand the blocks of code that we will proceed to explore and includes, but is not limited to, familiarity with language syntax, class and struct concepts, forward-declaration, templates and their angle-bracket syntax, and, overall, all the things that uniquely make C and C++ the languages they are.

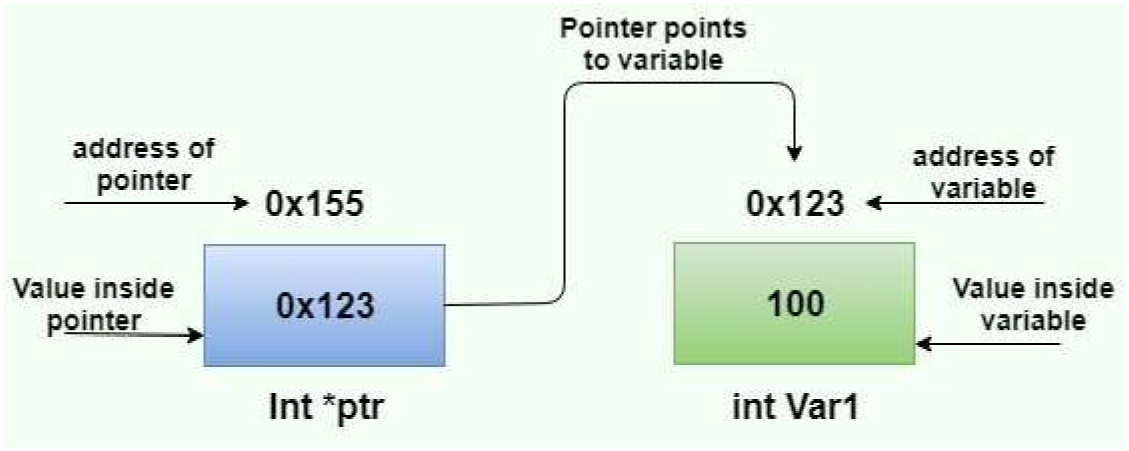
Concept

In the context of computer science, a *pointer* is an object, variable, or other datatype that **stores the address of** another object, variable, or datatype—*usually* of the same datatype as itself, with the exception of `void` pointers. Pointers store memory addresses of other datatypes in the context of virtual memory and, in some cases, memory-mapped hardware. Pointers, other than storing the location of datatypes in memory, can gain access to the value—or values—located at these memory locations through the process of dereferencing themselves.

Pointers are incredibly useful in the context of performance in computer software and embedded systems due to them being much *cheaper* on processing power by copying and dereferencing themselves, rather than copying and accessing the values held at the location pointers point.

Below you will find an illustration of a simple **pointer-to-integer**.

Illustration of simple pointer—assume truncated memory address 32-bit.



Pointers are useful for many things. Below you can find a short list of features and uses of pointers In both C and C++.

- Features:
  - Directly manipulate variables by their location in memory.
  - Save memory space and dynamic memory allocation via heap.
  - Faster execution due to direct manipulation.
  - File handling—IO.
- Uses:
  - Pass arguments by reference.
  - Allows for C forward compatibility with C++.
    - For instance, a C library using pointers for pass-by-reference is acceptable by C++, but a C++ library using references for pass-by-reference is unacceptable by C.
  - Useful for managing data structures—implementation and control.
  - Indexing in arrays.
  - Memory-mapped hardware.
  - Passing parameters to separate threads.
    - …in some threading APIs, such as Win32's `_beginthread` (specifically `void` pointers).
  - Return multiple values.
  - …and many more!

When one uses pointers in their code, they must also be aware of just how dangerous they are as well—and how they should be avoided at all times unless **entirely** necessary.

In the day of modern computing, the dangers of pointers do not seem terribly worrying when it comes to their use in software operating in a virtual address space—unaffecting other processes memory and operation. However, pointers can certainly negatively affect your software itself if used incorrectly and, in the case of systems programming, can even fault the system entirely if appropriate conditions are satisfied.

Allocation

Pointers and non-pointer datatypes handle their memory allocation differently depending on the context they are in. As we described earlier in our exploration of the segments of object files that are mapped and executed in virtual memory, data is stored in certain areas in memory depending on their context. For instance, below you can find a code block describing where certain code would be stored in memory—some may be confusing, but we will explore later.

```
FILE* initZero1 = NULL; // stored in uninitialized/initialized to zero segment(BSS)
double init1 = 12.2; // stored in initialized data segment

int main() {
        size_t (*fp1)(const char*) = strlen; // function pointer – stack alloc, points to code of strlen()

        char* dynamic1 = (char*)malloc(100); // both dynamic memory allocation
        char* dynamic2 = new char(100);      // pointers on stack, allocated data on heap
        free(dynamic1); delete dynamic2;

        int stringLength; // auto variable - allocated in stack

        static int initZero2; // stored in .bss - uninitialized

        int (*fp2)() = main; // points to .text segment – but stored on stack

        static const char* str[] = { "sample text", "arg1", "arg2", 0 }; // stored in .data - initialized

        static int init2 = 10; // stored in .data - initialized

        strcpy(dynamic, ”something”); // calling a function - uses stack
}
```

As you can see above, different values are stored in virtual memory differently depending on how they are defined. *Also*, notice how pointers not associated with function `malloc` and keyword `new` are **not associated with the heap**—but the stack.

It may be confusing, at first, why pointers that are not associated with the function `malloc` or the keyword `new` are not stored on the heap. A rule of thumb to remember when it comes to pointers is that, when used with either `malloc` or `new`, **the pointers themselves** are stored on the stack and **the data they point to** is allocated on the heap—otherwise, the pointer and the data it points to are assumed to be on the stack. The C function (and similar functions) `malloc` and C++ keyword `new` are used to dynamically allocate memory on the heap and return the new memory block's address—or base address if multiple elements allocated—to a pointer. Take, for instance, the following code.

```
int main() {
	// local, auto variable - stack
	int var = 10;

	// local, auto pointer-to-int variable - stack
	// pointer itself on stack, data it points to on stack
	// stores address of variable var
	int* pointerToInt = &var;

	// prints the value held at the address that pointerToInt points to – var’s address
	// equivalent to printing var by itself
	std::cout << *pointerToInt << std::endl;

	// local, auto pointer-to-int-array - stack
	// pointer itself on stack, data it points to on heap
	int* dynamicArray = new int[2];
	dynamicArray[0] = 4; dynamicArray[1] = 7;

	// required for pointers that use malloc, new, or new[]; cleans up allocated memory
	delete[] dynamicArray;
}
```

As you can see above, the use of dynamic memory allocation keywords and functions determines where the pointed-to data is stored—stack or heap. Variable `pointerToInt` simply points to the address of variable `var`; however, variable `dynamicArray` points to newly allocated dynamic memory on the heap that can hold two elements of type `integer`. Specifically, `dynamicArray` holds the base adress of the array of newly allocated memory—which is also the location of the first element in the array. For instance, consider the following.

```
    dynamicArray points to new memory block starting at 0x005F4AC0
index 0 holding 4 is at 0x005F4AC0; index 1 holding 7 is at 0x005F4AC4
```

Concept

In the context of both the C and C++ programming languages the ampersand, reference operator—or reference declarator (&)—has very different meanings, capabilities, and uses between the two languages.

In C, the reference operator, as we will call it, is used solely in conjunction with pointers and to obtain the address of stored elements. This is useful for changing variables and other datatypes via their direct address in memory—by, for instance, passing by value a pointer that holds the memory address of the variable you wish to alter, directly to a function. You can find an example below.

```
// takes a pointer as an argument - pointers store memory addresses
void changeValue(int* arg) {
        // dereference the address stored in the pointer - allows access to the value at the address
        // sets the value stored at the address to 5
        *arg = 5;
}

int main() {
        // variable of type int equal to 25
        int var = 25;

        // since pointers hold addresses, and the parameter is a pointer,
        // we pass in the address of variable var
        changeValue(&var);

        // ... var is now equal to 5
}
```

As you can see, the use of the reference operator in C is to be used to retrieve the memory address of a variable—often to be passed and stored in a pointer. Addresses of variables can also, unsurprisingly, be printed.

```
int main() {
        // simple variable of type int equal to 5
        int a = 5;

        // prints "value VALUE at address ADDRESS"
        printf("value %i at address %p", a, &a);
        // such as: value 5 at address 010FFB48
}
```

Pass-by-Reference (&)

In C++, while the meaning of it stays the same, the use of the reference operator is substantially expanded upon. For instance, the ability to take memory addresses as parameters (pass-by-reference) without the use of pointers, to change values stored at addresses without explicit dereferencing, and return references are some of the possibilities offered by C++'s expansion on the reference operator. Below you can find a simple example of references, comparing C++ code to its equivalent C code.

```
// C++ style pass-by-reference
void changeByValue(int& arg) {
        // dereferencing not required, handled internally “under the covers”
        arg = 25;
}

// C-style "pass-by-reference"
void changeByValue(int* arg) {
        // dereferencing required to access and set value at stored address
        *arg = 25;
}

int main() {
        // simple variable of type int equal to 3
        int a = 3;

        // C-style (int* arg)
        // pointers hold memory addresses
        // we must pass the memory address explicitly to the pointer parameter
        changeByValue(&a);

        // C++ style (int& arg)
        // reference parameters take variable names and allow change of value at address w/o dereferencing
        // references are treated as pointers internally by the compiler implementation
        changeByValue(a);

        // ... both function overloads accomplish the same thing - variable a now equals 25.
}
```

As you can see above, the use of the reference operator in C++ in lieu of pointers as parameters allows for much simpler, yet equivalent, operations on variables' values via their memory addresses.

While still on the topic of references in C++, let us take a look at the returning of references from function calls. Below you can find a simple function signature.

```
// returns a reference to an existing variable
int& func();
```

It may seem confusing at first, but this function signature simply means *return a reference to an existing—or alive—variable*. What can we do with references? We can, as explained already, set variables' values via their addresses with them! Let us expand the function signature above and see it in action.

```
// global array of 5 integers
int globalArray[5];

// returns a reference to a chosen index in globalArray
// we can use this to change values inside given indexes in globalArray via their address
int& func(int index) {
	return globalArray[index];
}

int main() {
	// since the function returns a reference to a supplied index, it becomes an lvalue
	func(0) = 25;  // equivalent to globalArray[0] = 25
	func(1) = 50;  // equivalent to globalArray[1] = 50
	func(2) = 75;  // equivalent to globalArray[2] = 75
	func(3) = 100; // equivalent to globalArray[3] = 100
	func(4) = 125; // equivalent to globalArray[4] = 125
	// each index in globalArray is now populated
}
```

As you can see above, the returning of a reference allows variables to be manipulated via their address by "setting the return" of the function—being the value held at the returned reference location. Think of functions that return a reference as functions that return an implicit pointer to a variable's location in memory. Keep in mind that, just because references and pointers accomplish the same objective equivalently in the above example, pointers and references are NOT the same, despite being closely related.

One of the most notable examples of references not being the same as pointers is the use of `NULL` values between the two. Simply, references **cannot** be initialized as `NULL`—not as a variable nor as a parameter (*unless explicitly const*)—whereas pointers can. The reason for this is that non-const references expect a variable to **reference** while pointers simply expect a memory address to hold—memory addresses are numerical values and `NULL` is **equal to zero.**

Why would we want to initialize a pointer to `NULL`? Well, other than it being a safe practice to apply to newly created pointers, as we mentioned earlier, it is very useful for optional parameters in functions found in the C programming language—and even the C++ language, though discouraged in favor of template class `std::optional<T>.` You can find a rough example below.

```
// takes a pointer to an integer
// function behaves based on if pointer is valid or not (non-NULL or NUL)L
void foo(int* number) {
        // if (number != NULL) i.e., if (address is valid)
        if (number) {
                printf("You supplied the optional number: %i\n", *number);
        }
        // if (number == NULL) i.e., if (address is invalid)
        else {
                printf("You did not supply an optional number!\n");
        }
}

int main() {
        // local, auto variable of type int equal to 5
        int a = 5;

        // calls foo() with a valid memory address in optional parameter one
        foo(&a);
        // You supplied the optional number: 5

        // calls foo() with NULL memory address in optional parameter one
        // signifying we do NOT want to supply any actual value to this parameter
        foo(NULL);
        // You did not supply an optional number!
}
```

It is up to the definition and implementation of the appropriate function or data structure to determine if pointers as parameters will be treated as optional parameters while others may nod—simply using a pointer as a parameter does not mean it is an optional parameter, you have to actively treat it and check it as one in your code.

Concept

We have briefly explored the concept of dereferencing pointers, but we have yet to show explicit examples using this mechanic. As explained previously, dereferencing a pointer simply means, in simple words, to *peak at, or set, the value being held at the address pointed to by a pointer*—these dereferences are accomplished by appending an asterisk (*) to the front of a pointer (multiple asterisks required if multiple layers of pointers used; explained later on). Below you can find examples of a pointer being dereferenced normally and as a supplied parameter.

```
int main() {
        // auto, local variable of type int equal to 5
        int a = 5;

        // pointer-to-integer variable
        // pointers hold addresses, holds the address of variable a
        int* ptr = &a;

        // dereference the address held by variable ptr
        // allows us to peak at (i.e., print)...
        printf("%i", *ptr);

        // ...and set...
        *ptr = 25;

        // ...the value held at stored address
        // variable a now equals 25.
}
```

and

```
// takes integer as argument
// prints supplied integer
void getValue(int arg) {
        printf("%i", arg);
}

int main() {
        // auto, local variable of type int equal to 5
        int a = 5;

        // pointer-to-integer variable
        // pointers hold addresses, holds the address of variable a
        int* ptr = &a;

        // function expects a regular integer
        // we dereference the stored address to get the value it holds and pass it to function
        getValue(*ptr);
        // the above is equivalent to getValue(a)
}
```

Pass by Reference by Pointer (*)

Now, even though we have already explored the concept of taking a pointer and manipulating the value stored at its pointed address, it is necessary to hammer these concepts until it is engraved in the mind—for lack of a better expression. When I say *pass by reference by pointer*, what we really mean is *pass pointer by value and manipulate the value at its stored address*—as C is not a pass by reference language in the same sense C++ is. This, effectively, accomplishes the same thing references do in C++ but, remember, C only has pointers—not references. Below you can find an example of passing a pointer by value to a function, changing the value held at the stored address, and printing it.

```
// pointers hold addresses
// takes a pointer, dereferences and sets value at held address
void setValue(int* arg) {
        // dereference - get access to held value and set it
        *arg = 25;
}

int main() {
        // auto, local variable of type int equal to 5
        int a = 5;

        // pointer-to-int - holds addresses
        // points to address of variable a
        int* ptr = &a;

        // function expects a pointer - pointers hold addresses
        // by supplying the pointer by value, we are passing the held address
        setValue(ptr);
        // ... a now equals 25
        // above is equivalent to setValue(&a)

        printf("%i", a);
}
```

The above C code is equivalent to the below C++—as in, it completes the same objective.

```
// takes a reference
// changes the value of original passed value by reference (by address)
void setValue(int& arg) {
        arg = 25; // set by reference (by address)
}

int main() {
        int a = 5;
        // function expects a reference
        // pass value normally - pointer stuff handled under the covers
        setValue(a); // ... a now equals 25
        std::cout << a << std::endl;
}
```

As of now, we have discussed the basic mechanics of pointers and references and their uses in both languages. However, up until now, we have only been able to see pointers and references in action in conjunction with simple datatypes and arrays of those datatypes—mainly `int`. In this section, we are going to explore pointer and reference use with other basic datatypes and, ultimately, more complex datatypes. As noted in the abstract of this text, it is highly recommended to have a minimal understanding of the various datatypes in both C and C++ to better understand the upcoming concepts.

In the following sections we will be covering the following:

- Pointers to built-in datatypes (`int, char, double, float,` etc.).
- Pointers to objects (`struct, std::vector<T, Allocator>, std::string,` etc.).
- Pointers to functions (function pointers, `typedef`, etc.).
- Pointers to pointers.
- Pointers to C-strings—and their layout in memory.

Afterwards, we will be covering the concept of `void` pointers and then, finally, we will cover the specifics of dynamic heap allocation C++ keywords and C functions—`malloc(), new, new[], delete,` and `delete[]`—since we have only briefly explained their purpose thus far.

Pointers to Built-In Types

In this section, we will be covering, exploring, and illustrating the mechanics and uses of pointers with built-in types in C—as C++ has the same primitive types.

Type Integer [`int`]:

In C and C++, an integer is a primitive, signable, numeric datatype that usually holds a size of 4 bytes—or sometimes 2 bytes—though, its length depends on the machine's underlying hardware and the compiler in use. Integers can be either negative or positive through the use of signing. Below you can find a simple example illustrating pointer-to-integer dereferencing and value change by address.

```
int add(int a, int b) {
        return (a + b);
}

int main() {
        int result, a = 25, b = 50;

        // holds memory address of variable a
        int* ptr1 = &a, *ptr2 = &b;

        // supply dereferenced values of pointers
        result = add(*ptr1, *ptr2);
        // equivalent to add(a, b)

        // result equal to 75
}
```

and

```
void changeValue(int* a) {
        *a = 200;
}

int main() {
        int a = 25, b = 50;

        // holds memory address of variable a
        int* ptr1 = &a, *ptr2 = &b;

        // supply pointers by value - i.e., address held by pointers
        changeValue(ptr1); changeValue(ptr2);
        // equivalent to changeValue(&variable)

        // a now equals 200, b now equals 200
}
```

The above C example of changing value via address can be performed equally well in C++ in the following manner via the use of reference parameters.

```
void changeValue(int& a) {
	a = 200;
}

int main() {
	int a = 25, b = 50;

	// holds memory address of variable a
	int* ptr1 = &a, *ptr2 = &b;

	// supply dereferenced pointers - value at stored address for references to use
	changeValue(*ptr1); changeValue(*ptr2);
	// equivalent to changeValue(variable)

	// a now equals 200, b now equals 200
}
```

Now, we will explore pointers-to-integer-arrays. Below you can find an example of pointers being used in conjunction with arrays of integers.

```
// takes address of first element of array
// populates each index of the array via their offset--index--from the base element
void populate(int* arg) {
	arg[0] = 4; // equivalent to *(arg + index) = 4; i.e., *(0x0098C510 + 0) = 4
	arg[1] = 7; // equivalent to *(arg + index) = 7; i.e., *(0x0098C510 + 1) = 7
	arg[2] = 9; // equivalent to *(arg + index) = 9; i.e., *(0x0098C510 + 2) = 9
	arg[3] = 2; // equivalent to *(arg + index) = 2; i.e., *(0x0098C510 + 3) = 2
	arg[4] = 3; // equivalent to *(arg + index) = 3; i.e., *(0x0098C510 + 4) = 3
}

int main() {
	// dynamically allocate memory for array of five integers on the heap
	// pointer now holds address of first element in newly allocated array
	int* a = new int[5];

	// populate the array
	populate(a);

	// clear dynamically allocated memory when done
	delete[] a;
}
```

As you can see above, we create an array with a size of five elements of type `int` on the heap and have our pointer hold the address of the first element in the array. We then pass this base address, held by our pointer, to our `populate()` function. This function populates each cell of our array via their index—or offset—from the base address. Think of indexing in arrays as taking the base address, adding a desired index to it, then dereferencing and setting its value.

Type Char [`char`]**:**

In C and C++, the `char` datatype is a primitive and signable datatype that usually has a size of 1 byte and can be represented using ASCII character codes and raw characters themselves. Most often in both languages—though mainly C—you will see this datatype in its array form—holding multiple characters. As we described in the previous section, an array is simply a contiguous collection of items in memory, starting from the first item, whose memory addresses are separated by one another by the size of the underlying datatype. For instance, in an array of characters, the first item may be at location `0x0098C510` and the one next to it at location `0x0098C511`; however, an array of 4-byte integers may have the first item at location `0x0098C510` and the next item at location `0x0098C514`. See the illustration below for a character array's layout in memory.

```
char* buffer = new char[6];
buffer[0] = ‘H’;
buffer[1] = ‘e’;
buffer[2] = ‘l’;
buffer[3] = ‘l’;
buffer[4] = ‘o’;

|-----|   |-----|   |-----|   |-----|   |-----|   |------|
|  H  |   |  e  |   |  l  |   |  l  |   |  o  |   |  \0  | -----------> 0x0098C515
|-----|   |-----|   |-----|   |-----|   |-----|   |------|
   |         |         |         |         |
   |         |         |         |         -----------> 0x0098C514
   |         |         |         |
   |         |         |         -----------> 0x0098C513
   |         |         |
   |         |         -----------> 0x0098C512
   |         |
   |         -----------> 0x0098C511
   |
   -----------> 0x0098C510
```

As you can see above, our pointer points to the address of the first element of our newly created array of six elements (five for characters, one for `\0`) in the heap. The rest of the elements can be accessed by using their respective index relative to our base element's address.

Arrays themselves, whether with pointers in use or not, are always treated as a pointer to the first element of the array. Whenever we assign an array to a variable, the variable will hold the address of the first item in the array regardless of if the variable is declared as a pointer or not. For instance, see below.

```
// takes address of first element of array
// populates each index of the array
void alter(char* arg) {
        arg[0] = 'T'; // equivalent to *(arg + index) = ‘T’; i.e., *(0x0098C510 + 0) = ‘T’
        arg[1] = 'e'; // equivalent to *(arg + index) = ‘e’; i.e., *(0x0098C510 + 1) = ‘e’
        arg[2] = 's'; // equivalent to *(arg + index) = ‘s’; i.e., *(0x0098C510 + 2) = ‘s’
        arg[3] = 't'; // equivalent to *(arg + index) = ‘t’; i.e., *(0x0098C510 + 3) = ‘t’
        arg[4] = '!'; // equivalent to *(arg + index) = ‘!’; i.e., *(0x0098C510 + 4) = ‘!’
}

int main() {
        // declare regular array of type char holding 6 elements - including space for \0
        // as an array, a is equal to the address of the first item in the array
        // null-terminator added by default
        // null-terminators are used by C and C++ functions to know when C-strings end
        char a[6] = "Hello";

        // pointers store addresses
        // array variables always hold the first item's address even without use of &
        alter(a);
        // the above is equivalent to alter(&a[0])

        // a now equals "Test!"
}
```

As you can see above, even when a `char` array is declared without the specification of a pointer, it still behaves as one—as variables declared as arrays will be equal to the address of the base element of the array. Singular `char` variables are handled as **normal** variables; see below.

```
// change value of char by its passed address
void alter(char* arg) {
        *arg = 'T';
}
// alter(char arg[]) and alter(char& arg) are also acceptable

int main() {
        // regular char variable - holds value 'a', not an address
        char a = 'a';

     // alter(a);  // ... will not work--unless reference parameter--as variable a holds regular char
        alter(&a); // ... will work, as we are supplying the location of the regular char to pointer
}
```

Keep in mind, `char`’s use with pointers is **not** dissimilar to the usage of pointers with types `wchar_t` and other sizes of `char`—such as `char16_t` and `char32_t`.

Types Float and Double [`float, double`]**:**

In C and C++, both types `double` and `float` are signed, precision datatypes that allow precision values involving decimal points—for instance, 3.14159. In terms of size—without `long` specification—single-precision `float` values hold a size of 4 bytes whereas double-precision `double` values hold a size of 8 bytes. While they are similar, `float` allows for single-precision decimal representations whereas `double` allows for, unsurprisingly, double-precision decimal representations. Pointers involving `float` and `double` are not dissimilar from that of pointers involving `int`; see below for use of `float` and `double` with pointers and references.

```
// function and overload change each index value by its address
void changeValue(float* arg) {
       arg[0] = 200.2393;
       arg[1] = 2203.32;
}

void changeValue(double* arg) {
       arg[0] = 4993.22;
       arg[1] = 2112.09;
}

// function and overload change value by C++ reference
void changeValue(float& arg) {
       arg = 200.2393;
}

void changeValue(double& arg) {
       arg = 192.3223;
}

int main() {
       float a = 454.3; float fArray[2] = { 24.2, 233.2 };
       double b = -889.4; double dArray[2] = { 34.9, 799.8 };

       // supply arrays to pointer overloads - arrays hold first item address
       changeValue(fArray); changeValue(dArray);
       // supply value to reference parameter overload
       changeValue(a);  changeValue(b);

       // new values:
       // a = ~200.2393
       // b = ~192.3223
       // fArray = { ~200.2393, ~2203.32 }
       // dArray = { ~4993.22, ~2112.09 }
}
```

Pointers to Objects

In this section, we will be covering pointers and illustrate their uses in conjunction with multiple different types of objects—including C's `struct` (though, not an object) and C++'s class-object instantiations of `std::string`, `std::vector<T, Allocator>`, etc. Assuming the reader is familiar with the basics of C and its superset, C++, the concept of class instantiation (objects) and `struct` should not be foreign—as well as the understanding that `struct` and objects are **not** the same despite their often-blurred definition.

Type Struct [`struct`]**:**

C's `struct` can be found in both C and C++ and serves its own purpose as a structure to group related data together under one name in a contiguous block of memory. However, struct is **not** an object as it doesn't support all the tenets of conceptual object-oriented programming—virtuality, member functions, inheritance, etc. Below you can find a simple `struct`—using `typedef` semantics.

```
// assigns struct of below data to names ARRAYS, *PARRAYS
typedef struct {
        // grouped data
        char   arrayOfCharacters[8];
        int    arrayOfInts[4];
        float  arrayOfFloats[4];
        double arrayOfDouble[4];
} ARRAYS, *PARRAYS;
```

As you can see above, we have defined a struct under the type names `ARRAYS` and `*PARRAYS` that stores 4 separate—but related—arrays of type `char`, `int`, `float`, and `double` with their own sizes. On the next page, we will explore examples of both pointers and references using the `struct` defined above.

Below you can find an example using the above defined struct in conjunction with pointers and references—as well as an example using arrays of `struct`.

```
void alter(PARRAYS arg) {
	// assign members in passed struct address by dereferenced stored address
	*arg = {
		.arrayOfCharacters = "Hello!!",
		.arrayOfInts = {4, 5, 6, 7},
		.arrayOfFloats = {2.1, 3.4, 2.5, 12.1},
		.arrayOfDouble = {4.3, 7.5, 3.3, 1.0}
	};
}

void alter(ARRAYS& arg) {
	// assign members in passed struct by reference
	arg = {
		.arrayOfCharacters = "Hello!!",
		.arrayOfInts = {4, 5, 6, 7},
		.arrayOfFloats = {2.1, 3.4, 2.5, 12.1},
		.arrayOfDouble = {4.3, 7.5, 3.3, 1.0}
	};
}

int main() {
	ARRAYS arr; // define variable holding struct ARRAYS

	// function expects type *PARRAYS pointers hold addresses
	// give the address of our defined struct and fill it
	alter(arr);

	PARRAYS pArr = new ARRAYS(); // also valid (in pointer overload):
	alter(pArr); delete pArr;

	// ... both struct variables are now populated with defined in alter()
}
```

and

```
// also valid: void alter(ARRAYS (&arg)[], short size) and void alter(ARRAYS arg[], short size)
void alter(PARRAYS arg, short size) {
	// assign members in each passed struct by dereferenced address
	for (short i = 0; i < size; i++) {
		arg[i] = {
			.arrayOfCharacters = "Hello!!",
			.arrayOfInts = {4, 5, 6, 7},
			.arrayOfFloats = {2.1, 3.4, 2.5, 12.1},
			.arrayOfDouble = {4.3, 7.5, 3.3, 1.0}
		};
	}
}



int main() {
	ARRAYS arr[4]; // define array of size 4 holding structs ARRAYS

	// function expects type *PARRAYS; pointers hold addresses
	// give the address of our defined struct and fill it
	alter(arr, 4);

	PARRAYS pArr = new ARRAYS[4]; // also valid (in pointer overload):
	alter(pArr, 4);  delete[] pArr;

	// ... all structs in both arrays are now populated
}
```

Type String [`std::string`]:

In C++, `string` is a class in namespace `std` that allows for objects that act as the much mor familiar container of contiguous bytes. As `string` manages its own memory, it makes it a much safer alternative to raw C-style strings like `const char*` and other raw `char` arrays.

In the following examples with C++'s `string` class, we will explore its use with pointers and references—as well as the use of the pointer-to-member arrow-operator (which can be found in pointers to `struct` and C++ objects) to access data members.

Below you can find an example of altering C++ string objects via reference and pointer function parameters.

```
// changes supplied string to supplied new string via reference semantics
void changeString(std::string& oldStr, const std::string newStr) {
        oldStr = newStr;
}

// changes supplied string pointer to supplied new string via derefencing of pointer's stored address
void changeString(std::string* oldStr, const std::string newStr) {
        *oldStr = newStr;
}

int main() {
        // basic object of class string holding bytes "Hello!"
        std::string a = "Hello!";
        // also suitable: std::string a("Hello!");

        // pointer to heap-allocated object of class string holding bytes "Hello, again!"
        std::string* b = new std::string("Hello, again!");

        // changeString reference overload - changes a to new string on left
        changeString(a, "Changed by reference!");

        // pointers hold addresses
        // changeString pointer overload - takes value of b (address of object in heap)
        // dereferences it and changes the bytes held to left string
        changeString(b, "Changed by pointer dereference!");
        // equivalent to changeString(&a, ...);

        delete b;
}
```

As you can see, we declare a local, auto object of class string and a pointer to a heap-allocated object of class string—both holding unique strings of bytes. We then provide our auto object to our reference-overloaded function and our pointer-to-object to our pointer overload. Both are then directly changed to the second function parameter in the appropriate manner.

Type String [`std::string`]:

Pointer-to-Member Arrow-Operator

If you are familiar with the concepts of classes in C++ or `struct` in C, you are certainly familiar with the dot-operator (.). The dot operator is used in both of these data structures to access stored members within their regular variable or instantiated object form—be it variables, functions, function pointers, etc. When it comes to pointers-to-objects, a separate operator is used—but not required—to access data members of a pointed-to object in an easier fashion. See below for a comparison between the dot-operator and the pointer-to-member arrow-operator using an example class.

```
// class of name basic_class holding one public member of type int named a
class basic_class {
        public:
                int a = 5;
};
```

As you can see above, we have defined an example class `basic_class`. In the usual situation (assuming non-static), we would access this member like so…

```
int main() {
        basic_class object;
        int b = object.a;

        // variable b now equals the value of class basic_class member a (basic_class::a)
}
```

However, how would we accomplish what we just did when `object` is a pointer to a heap-allocated object of `basic_class`? We would use the pointer-to-member arrow-operator like below.

```
int main() {
        basic_class* object = new basic_class();

        // arrow-operator is used to access members of pointed-to object in simpler manner than equivalent
        int b = object->a; // equivalent: int b = (*object).a – i.e., deref object address, access member

        // variable b now equals the value of class basic_class member a (basic_class::a)
}
```

The use case of the pointer-to-member arrow-operator is universal among all objects in C++. For instance, below you can find an example of a function resizing the bytes in an object of class `string` via the class member function `resize()` with the use of a pointer-to-string parameter.

```
void changeStringSize(std::string* str) {
        // resize string of bytes stored at address provided via object method resize()
        // the below is equivalent to (*str).resize(2) - dereference of the object address
        // and access of its resize() method
        str->resize(2);
}

int main() {
        std::string a = "Hello!";
        std::string* b = new std::string("Hello, again!");

        changeStringSize(&a);
        changeStringSize(b);

        // both strings have been resized to 2 bytes
        // both strings now hold bytes "He"

        delete b;
}
```

As you can see above, the arrow-operator is a simpler equivalent to dereferencing an object to get access to it and then using the dot-operator on the—now accessible—object to access its class members. Keep in mind that, since references are treated as pointers internally, they do not require the use of the arrow-operator themselves; however, members of classes (or `struct`) that are declared as pointers can require the use of it if necessary. See below.

```
typedef struct {
        int value;
} basic_struct, *pbasic_struct;

// class of name basic_class holding one heap-allocated pointer-to-struct named p
class basic_class {
        public:
                pbasic_struct p = new basic_struct();
                ~basic_class() { delete p; } // destructor to clear heap-allocated memory of p
};

void setValue(int& arg) { arg = 25; } // sets supplied integer's value by its address reference

int main() {
        basic_class object; // auto object of class basic_class
        setValue(object.p->value); // pass member of dereferenced class member struct p
        // value of member ‘value’ of basic_class member struct ‘p’ is now equal to 25
}
```

Type Vector [`std::vector<T, Allocator>`]:

In C++, the use of angled-brackets—other than for comparison and bitwise shifts—is for templates. Templates allow functions and classes to operate with generic datatypes rather than having to write overloads for each datatype that can be used in a function or class. For example, see below a simple template class and member function.

```
// class containing one template member function of template T
class basic_class {
        public:
                // takes argument of user-supplied type T and returns supplied argument of type T
                template <typename T>
                T returnValue(T value) {
                        return value;
                }
};

int main() {
        basic_class object;
        std::cout << object.returnValue<int>(54) << std::endl;               // prints integer 54
        std::cout << object.returnValue<bool>(true) << std::endl;            // prints bool true (1)
        std::cout << object.returnValue<char>('c') << std::endl;             // prints char 'c'
        std::cout << object.returnValue<std::string>("Hello!") << std::endl; // prints string "Hello!"
}
```

As you can see above, the use of templates in C++ allows us to use different user-defined types in functions and classes without the need to have an overload for each individual type we expect to use with the functions or class member functions.

The class `vector<T, Allocator>` of namespace `std` in C++ is a template class, allowing us to store dynamically sized arrays of a user-defined type—which is ***much*** safer than raw C-style arrays. For example, see below.

```
int main() {
        // defined and initialized vector of strings - string objects
        std::vector<std::string> names = { "Tommy", "Jack", "Andrew", "Xavier" };

        // we can also add new names when we find it necessary via the class member function push_back()
        names.push_back("Alex"); // adds string "Alex" to end of 'names'
        // we can also remove items via member function pop_back()
        names.pop_back(); // removes item at the back of the vector - in this case, "Alex"

        // ‘names’ now holds strings { “Tommy”, "Jack", "Andrew", "Xavier" }
}
```

As you can see above, we create an object of template class `vector<T, Allocator>` that holds strings; we then add and remove an element.

Since vectors are automatically sized and managed arrays of data defined by user-supplied template types, we can supply whichever datatype comes to mind to a vector—pointers-to-datatypes, other objects, other vectors holding other objects, and so on. Below you can find an example changing the values inside a vector by pointer as well as their address references.

```
// more C++-style function to change values held inside vector by passed vector object reference
void changeValuesInVector(std::vector<int>& vector) {
        // C++ range-based for-statement
        // "for each element in vector 'vector', assign a reference of element to 'i'"
        for (auto &i : vector) {
                // change element by its reference
                i = 5;
        }
}
// more C-style function to change values held inside vector by passed vector object address
// since std::vector is a C++ class, it can only be made as C-style as possible
void changeValuesInVector(std::vector<int>* vector) {
        for (int i = 0; i < vector->size(); i++) {
                (*vector)[i] = 5;
        }
}

int main() {
        // vector of integers
        std::vector<int> ints = { 4, 2, 6, 1, 2, 3 };

        // pointer overload; pointers hold addresses
        // pass the address of our vector; each item changed by their reference
        changeValuesInVector(&ints);

        // reference overload
        // take vector as a reference and change its held values by their references
        changeValuesInVector(ints);

        // the vector is filled with the integer '5' in each case
}
```

As you can see above, since `std::vector` is a C++ class, it can only be made as C-style as possible. In the first function overload, we take a reference to a `std::vector<int>` object, we use a C++-style range-based for statement to assign a reference to each element, and then we change the value of each said referenced element. In the second function overload, we take a pointer to an address of an object of type `std::vector<int>`, we use a traditional for statement to loop through each index of the dereferenced vector until we reach the last index, we then change the value held at each index in the vector—as vectors can be treated as arrays if needed, and arrays are treated as pointers to their elements.

If a vector is filled with pointers-to-integers `std::vector<int*>`—or another pointer-to-datatype—we can pass the vector by value and change its held members by address since the vector holds their addresses anyway. For instance, see the example below.

```
// C-style function to change values at addresses held in passed by value vector object
void changeValuesInVector(std::vector<int*> vector) {
        for (int i = 0; i < vector.size(); i++) {
                // vector[i] returns an int*, so we must dereference it to set its value
                *vector[i] = 5;
        }
}

int main() {
        // vector of heap-allocated pointers-to-integers
        // as we know by now, C++ keyword ‘new’ returns an address to newly heap-allocated data
        std::vector<int*> ints = { new int(3), new int(7), new int(8), new int(9), new int(1) };

        // pass vector by value as we want to change the values at addresses it holds; not vector itself
        changeValuesInVector(ints);

        // the vector is filled with the integer '5' in each case

        // clear memory for each heap-allocated int in vector
        for (auto i : ints) {
                delete i;
        }
}
```

As you can see above, when a container contains a pointer—or collection of pointers—to another piece of data in memory, that container need not be passed by reference or address since it contains the needed addresses to the respective data within it; hence, we can pass the container—in this case, `std::vector<int*>`—by value and edit its copied members by their address since they are pointers to addresses.

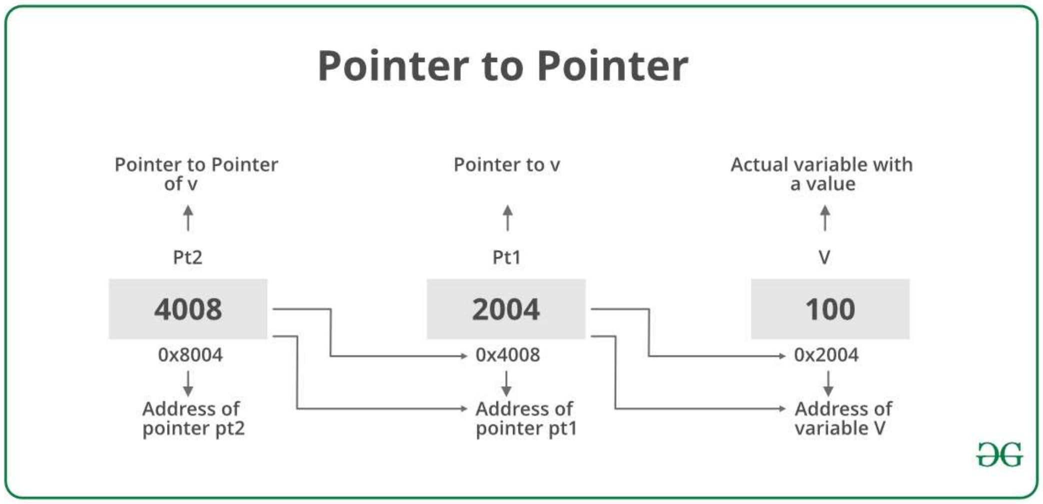
However, something like the above with the use of C++ references—i.e., `std::vector<int&>` (with appropriate code changes)—would not work without further modifications—such as using `std::reference_wrapper` instead—as this is due to the nature of references in C++. References in C++ are non-assignable datatypes and can only be initialized at declaration and cannot reference something else later; the component type of containers, such as `std::vector<T, Allocator>`, must be assignable—`const` types are also illegal.

Pointers to Pointers

In C and C++, pointers-to-pointers are exactly what they sound like—pointers that hold the address of another pointer (which assumably holds another address). It may sound confusing at first, but pointers-to-pointers can be used to take pointers by their address in function calls and other areas in your source code—though rare, and even discouraged in C++. For instance, with the help of a pointer-to-pointer (or reference to a pointer), we can have a pointer-to-pointer as a function parameter, give the address of a pointer that points to another address, and then we can change the address that the passed pointer points to; to simplify, a pointer-to-pointer allows us to *modify a pointer by its address.* For an example, see below.

```
// function that takes a pointer-to-pointer and changes the address it points to
// pointers-to-pointers hold addresses of pointers; NOT the address a pointer holds
// pointers-to-pointers hold the address of the pointer variable itself
void modifyPtr(int** ptr) {
        delete* ptr; // delete old memory before reassignment

        // we dereference the passed pointer's address.
        // pointers hold addresses so, when we dereference the pointer's address,
        // we can change the address the passed pointer holds.
        // we make passed pointer hold a new address via its address
        *ptr = new int(50); // assign new memory address to passed pointer by address
}

// function that takes a reference to a pointer
// changes the address that the passed pointers point to
void modifyPtr(int*& ptr) {
        delete ptr; // delete old memory before reassignment

        // no need to dereference; we have a reference to the pointer
        ptr = new int(50); // assign new held memory address to passed pointer by reference
}

int main() {
        // pointer to address of heap-allocated integer value 4
        int* intptr = new int(4);

        // pass the address of our pointer; NOT the address it holds
        // we pass the address OF the above pointer that points to new heap memory
        modifyPtr(&intptr);

        // pass our pointer by reference
        // we pass the regular version of our pointer to our reference overload
        modifyPtr(intptr);

        // both above examples will reassign the stored address of intptr to a new heap-allocated address
        // ... and delete the old heap-allocated memory

        delete intptr; // finally delete heap-allocated memory
}
```

For an illustration of how pointers to pointers operate and are formatted, you can find a graphic below.



As you can see above, we have a pointer-to-pointer of name `pt2` with address `0x00008004`. Pointer-to-pointer `pt2` holds the address of pointer-to-int `pt1`—`0x00004008`. `pt1` holds the address of integer variable `v`—`0x00002004`—with value 100. In the above scenario, we can do something like the following.

```
int main() {
        int v = 100;

        // pointer-to-int; holds (points) to address of var 'v'
        int* pt1 = &v;

        // pointer-to-pointer; hold      s (points) to address of pointer pt1
        int** pt2 = &pt1;

        printf("%p\n", &pt2);  // prints address of pointer-to-pointer pt2 (i.e., 0x00008004)
        printf("%p\n", pt2);   // prints address of pt1                     (i.e., 0x00004008)
        printf("%p\n", &pt1);  // prints address of pt1                     (i.e., 0x00004008)
        printf("%p\n", pt1);   // prints address of variable 'v'            (i.e., 0x00002004)
        printf("%p\n", &v);    // prints address of variable 'v'            (i.e., 0x00002004)

        printf("%i\n", **pt2); // prints value of variable 'v'              (100)
        printf("%p\n", *pt2);  // prints address of variable 'v'            (i.e., 0x00002004)
        printf("%i\n", *pt1);  // prints value of variable 'v'              (100)
        printf("%i\n", v);     // prints value of variable 'v'              (100)
}
```

Keep in mind that, despite being available to C++, use of raw pointers-to-pointers is usually discouraged in favor of C++ references-to-pointers—unless dealing with legacy APIs.

Pointers to Functions

In C and C++, pointers-to-functions can be used for myriad purposes—callback function parameters, dynamically loading symbols from shared libraries, reducing code redundancy, etc. While function pointers are incredibly powerful, they can sometimes be very difficult to read; as a result, the C++ polymorphic, function wrapping, template class `std::function` (although it is more expensive than a regular function pointer) can be used instead for an easier-to-read syntax—among many other highly useful things that come with the abstractable class. For instance, see below for a function pointer and roughly equivalent `std::function` object—as well as an illustration on the syntax of raw function pointers.

```
Function pointers abide by the following syntax: (^optional)
type (^macro *name)(arguments)
 ^       ^     ^        ^
 |       |     |        |
 |       |     |        ------> the arguments of the function that will be stored
 |       |     |                in this function pointer
 |       |     |
 |       |     ------> desired name of the function pointer
 |       |
 |       ------> optional macro (such as Win32's WINAPI calling convention)
 |
 ------> the return type of the function that will be
                     stored in this function pointer.
```

```
int returnInteger(int arg) {
        return arg;
}

int main() {
        // raw function pointer of same signature that holds address of function returnInteger
        int (*functionPtrName)(int) = returnInteger;

        // C++-style std::function object roughly equivalent to the above
        std::function<int(int)> fp(returnInteger);

        // both functionally equivalent - integer value 5 is returned
        functionPtrName(5); fp(5);
}
```

On the next page, we will explore the use of function pointers for callback functions and reducing code redundancy using both raw function pointers and C++-style `std::function` objects.

Let us assume for a moment that, in your source code, you are performing a switch statement on an integer and calling a function based on the value of said integer; see below pseudocode block.

```
switch (integer_variable) {
	case 0: func1(); break;
	case 1: func2(); break;
	case 2: func3(); break;
	case 3: func4(); break;
	case 4: func5(); break;
}
```

With the help of function pointers, we can easily create a more easy-to-read version of the above. See below.

```
void func1() { std::cout << "func1() called!" << std::endl; }
void func2() { std::cout << "func2() called!" << std::endl; }
void func3() { std::cout << "func3() called!" << std::endl; }
void func4() { std::cout << "func4() called!" << std::endl; }
void func5() { std::cout << "func5() called!" << std::endl; }

int main() {
	int integer_variable = 3;

	// array of raw function pointers
	void (*fpArr[])() = { func1, func2, func3, func4, func5 };

	// array of std::function objects conforming to proper signature
	std::function<void()> a[] = { func1, func2, func3, func4, func5 };

	// both equivalent to switch statement previously shown - though much more unsafe
	// calls function in appropriate index based on previously shown switch value
	fpArr[integer_variable]();
	a[integer_variable]();
}
```

As you can see above, with the use of both raw function pointers and C++-style `std::function` objects—as well as a little creativity—we can take a long and jumbled switch statements and condense them to a short array-oriented function call based on a switch-style index value. Be aware that, due to the nature of raw C-style arrays, the above rough example is vulnerable to buffer overflow and should only be used as an example of one of the possibilities of function pointers.

Another use of raw function pointers and `std::function` objects is for callback functions. Let us assume, for a moment, that you were using a C library for networking. Using this library, you want to have a function be called whenever a new packet is received by the interface you are listening on. The library allows for a function to take a packet `struct` as a parameter to do whatever it pleases with. See a rough example of what one may do below.

```
// callback function called when number is even
void even(int num) { std::cout << num << " is even!" << std::endl; }

// callback function called when number is odd
void odd(int num) { std::cout << num << " is odd!" << std::endl; }

// function to count to an upper limit and call a provided callback function depending on
// whether the counted-to number is even/odd
void count(int limit, std::function<void(int)> evenCallback, std::function<void(int)> oddCallback) {
        for (int i = 1; i <= limit; i++) {
                (i % 2 == 0) ? evenCallback(i) : oddCallback(i);
        }
}

// overload with raw function pointers
// functionally equivalent
void count(int limit, void(*evenCallback)(int), void(*oddCallback)(int)) {
        for (int i = 1; i <= limit; i++) {
                (i % 2 == 0) ? evenCallback(i) : oddCallback(i);
        }
}

int main() {
        // count to 100 from 1, calls function 'even(int)' when even, 'odd(int)' when odd
        count(100, even, odd);
}
```

As you can see above, we have declared functions even() and odd()—that will be provided as callback functions to count()—when appropriate conditions are met. The useful thing with callback functions is that *the programmer* gets to choose what is done with the data a function provides to them when called. In the above example, we have defined our callback functions to simply print the number and its accompanying odd or evenness; whenever a number is counted to in `count()`, one of the supplied callbacks are called.

A real example of callback functions can be found in networking libraries. For instance, assume you are using a library to listen on a network interface. For each packet received, one may have a callback called each time a packet is received and have the packet passed to it.

Function pointers can also be used for dynamically loading function symbols from dynamic-link libraries—or shared objects on Linux machines—with runtime dynamic linking. In the case of Windows, which this text largely focuses on, the use of the Win32 API functions `LoadLibrary()` and `GetProcAddress()` (or appropriate alternatives) can be used to retrieve the address of an exported function from a DLL and store it in an appropriate function pointer for later calling. Below you can find an example—using the Windows C API—of dynamically exporting the address for the `User32.dll` function `MessageBoxA()`, storing it in an appropriate function pointer, calling it, and then clearing all allocated resources, if necessary.

```
typedef int(WINAPI *fpMessageBoxA)(HWND, LPCSTR, LPCSTR, UINT);
         ^     ^          ^        ^
         |     |          |        |
         |     |          |        ------> the arguments of the function that will be stored
         |     |          |                in this function pointer
         |     |          |
         |     |          ------> desired name of the function pointer
         |     |
         |     ------> optional macro (such as Win32's WINAPI calling convention)
         |
         ------> the return type of the function that will be
                             stored in this function pointer.

// function pointer that matches below function signature:
//    int (WINAPI) MessageBoxA(HWND, LPCSTR, LPCSTR, UINT)
// function pointer given datatype name fpMessageBoxA via typedef semantics
typedef int(WINAPI* fpMessageBoxA)(HWND, LPCSTR, LPCSTR, UINT);

int main(void) {
        HINSTANCE libInstance; // variable for handle to library; acquired from LoadLibrary
        fpMessageBoxA showMessage; // function pointer; stores casted address from GetProcAddress

        // get a Win32 handle/instance of library ‘User32.dll’
        libInstance = LoadLibrary(TEXT("User32.dll"));

        // if handle to library is valid, proceed to attempt to get address of desired function
        if (libInstance) {
                // get address of ‘MessageBoxA()' from instance of library 'User32.dll' via GetProcAddress
                // addresses must be same type that they are assigned to; cast address to fpMessageBoxA
                showMessage = (fpMessageBoxA)GetProcAddress(libInstance, "MessageBoxA");

                // if acquired function address is valid (non-NULL), call function at address
                if (showMessage) {
                        showMessage(NULL, "Message box shown!", "Message box title", MB_OK);
                }
                else {
                        printf("Failed to get fp to MessageBoxA\n");
                }
                // free DLL module acquired via LoadLibrary
                FreeLibrary(libInstance);
        }
}
```

As you saw previously, we created a function pointer matching the function signature of `MessageBoxA()` under the type name `fpMessageBoxA`. We create a function pointer with this newly defined type called `showMessage`. Using an `HINSTANCE` variable that we have also created, we load the dynamic library `User32.dll` and store a handle to it in this `HINSTANCE` variable, `libInstance`. If `libInstance` is non-NULL, we attempt to retrieve the address of function `MessageBoxA()` in `User32.dll` using `GetProcAddress()` and `libInstance`, cast it to the same type, and assign it to `showMessage`. If the address stored in `showMessage` is non-NULL and valid, we call it with the desired parameters. If all is completed successfully, we then free the handle to our library, `libInstance`, via `FreeLibrary().`

Function pointers also support the passing of *non-capturing lambda expressions*. For instance, see below an example using a template function that accepts a generic function and passes generic `rvalue` parameters to said function.

```
// function that calls a supplied template function with parameter-packed arguments
template <typename T, typename... Args> // T: function signature, Args: parameter-packed arguments
void callFunction(T function, Args... args) {
        std::cout << function(args...); // print return value of called function
}

int main() {
        // call with appropriate template type, non-capturing lambda, and parameters for lambda
        callFunction<int(int, int)>([](int a, int b) -> int { return a + b; }, 3, 5);
}
```

As you can see above, we define a function that takes a generic function as its first parameter—that the user defines via template parameter—as well as a parameter-packed template for arguments as its second parameter; this function then calls the supplied function with the supplied arguments. We call the function by indicating the type of the lambda we will pass (returning `int`, taking two `int` parameters), passing our custom lambda to it, and then providing the arguments we wish to have passed to our lambda.

Calling Conventions

In the previous example we explored, you may have noticed the keyword—or macro—`WINAPI` in our function pointer `typedef`. The `WINAPI` macro—found between return type and function name (or type name)—is better referred to as a *calling convention*. When it comes to Windows, the available Microsoft compilers support the use of various calling conventions. Calling conventions, in their simplest form, are schemes that determine how parameters are received by subroutines from their callers and how are result is returned.

See below for a table on the support calling conventions for Microsoft's Visual C/C++ compiler.

Keyword	Stack Cleanup	Parameter Passing
__cdecl	Caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	N/A	Load parameters onto CLR expression stack in order (left to right).
__stdcall	Callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	Callee	Stored in registers, then pushed on stack
__thiscall	Callee	Pushed on stack; **this** pointer stored in ECX
__vectorcall	Callee	Pushed on stack; **this** pointer stored in ECX

On the next few pages, we will explore the definitions and differences between each calling convention outlined in the above table.

Calling Convention [`__cdecl`]:

The `__cdecl` calling convention, in C and C++ programs, is the default calling convention. This calling convention, compared to another convention we will cover later, `__stdcall`, results in larger executables due to the inclusion of stack-cleaning code being required in function calls. The below table describes the implementation of the Microsoft-specific `__cdecl` calling convention.

Element	Implementation
Argument-passing order	Right to left.
Stack-maintenance responsibility	Calling function pops the arguments from the stack.
Name-decoration convention	Underscore character (_) is prefixed to names, except when __cdecl functions that use C linkage are exported.
Case-translation convention	No case translation performed.

As you can see above, in the case of the `__cdecl` calling convention, function arguments are pushed onto the stack in the order of right-to-left, arguments are popped from the stack by the calling function, function names are decorated with the use of an underscore (_) being prefixed to their name—with the exception of exported `__cdecl` functions using C linkage—and no case-translation convention is performed.

Calling Convention [`__clrcall`]:

The Microsoft-specific calling convention `__clrcall` is used to specify that a function can only be called from managed code—high-level code that is written in a high-level language that runs atop `.NET`. This calling convention cannot be used for functions that will be invoked from native, unmanaged code (i.e., C/C++).

Entry points are separate and compiler-generated functions. If a function has entry points of both native and managed code, one of them will be the actual function with the function implementation. The other function will be a separate one that calls into the actual function and lets the *common language runtime* (CRT) perform `PInvoke`. When marking a function as `__clrcall`, you indicate the function implementation must be *Microsoft-Intermediate Language* and that the native entry function is not generated. When taking the address of a native function, the compiler uses the native entry point if `__clrcall` is not specified. The `__clrcall` calling convention indicates that the function is managed and requires no need to transition from managed-to-native; in that case the compiler uses the managed entry.

Calling Convention [`__stdcall`]:

The `__stdcall` calling convention is used when a programmer wishes to call Windows C API functions—the `WINAPI` macro previously seen expands to `__stdcall`. See the table below for implementation information regarding the `__stdcall` calling convention.

Element	Implementation
Argument-passing order	Right to left.
Argument-passing convention	By value—unless a pointer or reference type is passed.
Stack-maintenance responsibility	Called function pops its own arguments from the stack.
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at sign (@) followed by the number of bytes (in decimal) in the argument list. Therefore, the function declared as `int func( int a, double b )` is decorated as follows: `_func@12`
Case-translation convention	None

On ARM and x86-64 architecture processors, `__stdcall` is both accepted and ignored by the compiler; by convention, arguments are passed in registers when possible, and subsequent arguments are passed on the stack. In the case of non-static class functions, if the function is not defined inline, the calling convention modifier does not have to be specified on the out-of-line definition. That is, for non-static member methods, the calling convention specified during declaration is inherited at definition.

Calling Convention [`__fastcall`]:

The `__fastcall` calling convention **only** applies to the x86 architecture and specifies that arguments passed to functions are to be forwarded into registers—when possible. See the table below for implementation information regarding the `__fastcall` calling convention.

Element	Implementation
Argument-passing order	The first two DWORD or smaller arguments that are found in the argument list from left to right are passed in ECX and EDX registers; all other arguments are passed on the stack from right to left.
Stack-maintenance responsibility	Called function pops the arguments from the stack.
Name-decoration convention	At sign (@) is prefixed to names; an at sign followed by the number of bytes (in decimal) in the parameter list is suffixed to names.
Case-translation convention	No case translation performed.

The `__fastcall` keyword is accepted and ignored by compilers that target ARM and x86-64 architectures; on an x86-64 chip, by convention, the first four arguments are passed in registers—when possible—while additional arguments are passed on the stack. On an ARM chip, up to four integer arguments and eight floating-point arguments may be passed in registers and—similar to the x86-64 architecture—additional arguments are passed on the stack.

Calling Convention [`__thiscall`]:

The `__thiscall` calling convention is an x86, C++-only convention that is used by default in the signature of class member functions that do not incorporate variable arguments—`varargs`. While impossible for functions that use variable arguments, under `__thiscall`, the callee cleans the stack. Arguments for functions using this calling convention are pushed onto the stack from right to left while the C++ self-referential class pointer, `this`, is passed via the register `ECX` and not on the stack.

On machines operating atop architectures ARM, ARM64, and x86-64 `__thiscall` is both accepted and ignored by the compiler because they use a register-based calling convention. Because this calling convention applies to classes and their member functions—which are strictly a C++ feature—it does not have a C decoration scheme.

Calling Convention [__vectorcall]:

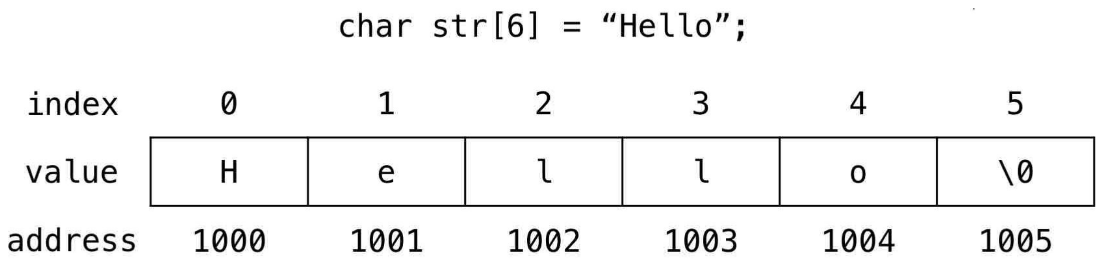
The `__vectorcall` calling convention specifies that arguments passed to functions are to be forwarded into registers—when possible. This convention uses more registers for arguments than its `__fastcall` counterpart or the default x86-64 convention use of registers. This calling convention is only supported in native code on x86 and x86-64 processors that include support for SSE2+—it is ignored by the compiler on ARM systems. `__vectorcall` should be used to speed functions that pass several SIMD vector or floating-point arguments and perform operations that take advantage of the arguments being loaded into the registers. See the table below for implementation information regarding the `__vectorcall` calling convention.

Element	Implementation
C name-decoration convention	Function names are suffixed with two "at" signs (@@) followed by the number of bytes (in decimal) in the parameter list.
Case-translation convention	No case translation is performed.

On the x86-64 architecture, this convention extends the standard number of registers to take advantage of additional registers—arguments are mapped to registers based on their position while HVA arguments are allocated to unused vector registers. On the x86 architecture, this convention follows the `__fastcall` calling convention for 32-bit integer arguments and takes advantage of the `SSE` vector registers for vector type and HVA arguments. The first two integer arguments, from left to right, are place into `ECX` and `EDX`—where a hidden `this` pointer is treated as the first integer. The first six vector arguments are passed by value to `SSE` registers 0-5 left to right while floating-point and `__m128` are passed in `XMM` registers and `__m256` types are passed in `YMM` registers.

Character Pointers and Arrays

During our exploration of pointers to built-in types, we explored the layout of pointers-to-character-arrays in memory; we also explored the concept that pointers and arrays are *very* closely related. For instance, a variable declared as an array will always hold the value of the base address of elements—being the first item in the array. See the image below for the layout of a character array in memory.



```
char str[6] = “Hello”;
```

index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

In the above image, you can see that a character array of size 6 is given the value “Hello,” including space for the necessary null-terminating character, The array variable will hold the base address—being the first element in the array—and all other indexes can be access via an offset from the base index.

However, some people may be confused by the difference between character arrays and pointers to character arrays. See the below quote from the C standard.

> “Except when it is the operand of the `sizeof` operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type ‘‘array of *type*’’ is converted to an expression with type ‘‘pointer to *type*’’ that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.”

To put it as simply as possible, ***arrays are pointers***—pointers to the first element of the array. Functions that accept a pointer as a parameter will accept an array of the same type—and can even index the passed array as it could any other pointer-to-array.

You may be looking for certain clarification though. What about `const char*` (pointer-to-const-character-array) and `char*` (pointer-to-character-array)? Well, to put it simply, there isn't much of a difference other than being constant or non-constant—a pointer-to-const-character-array is exactly that; see below.

```
int main() {
        // mostly equivalent to: const char a[] = "Hello!";
        const char* a = "Hello!";

        // the above is a pointer that points to a constant array of literal data elsewhere in memory
        // the pointer itself is not const and can be reassigned like below; the data is const itself
        // and cannot be changed
        a = "Hello, again!";

        // we have just reassigned pointer 'a' to point to the base address of a new const array of char

        // while they hold the base item's address, the reason char arrays or pointers-to-char-arrays
        // can be printed in their entirety is solely due to how functions like printf() and std::cout
        // handle them and their terminating null value
}
```

As you can see above, when we declare a pointer to a string-character literal (i.e., a collection of characters enclosed in quotes) array, the pointer **will** hold the address of the first element in the array—in the above case, 'H.' Since the data pointed to by the pointer is constant, it cannot be changed itself; however, the pointer itself is not constant and can be changed to point to another address—being a new string-character literal array, like "Hello, again!" in the above case. For instance, see the example on the next page for the difference between a pointer-to-constant-character-array and constant-pointer-to-constant-character-array.

`const char*` is the preferred, and often seen, C-style string in both C and C++. It is a reassignable pointer to a string-literal (constant array of characters) elsewhere in memory. See below for the difference between a pointer-to-constant-character-array and constant-pointer-to-constant-character-array.

```
int main() {
        // pointer to a constant array of characters elsewhere in memory
        // specifically, 'a' points to the address of the first element in
        // (const char[7])"Hello!"
        const char* a = "Hello!";

        // constant pointer to a constant array of characters elsewhere in memory
        // specifically, 'b' constantly points to the address of the first element in
        // (const char[7])"Hello!"
        const char* const b = "Hello!";

        // pointer 'a' is reassignable to another address, as it is non-const
        a = "Good morning!";
        // pointer 'a' now points to the first element of array (const char[14])"Good morning!"

        // constant pointer 'b' is not reassignable; the below is a compile-error
        b = "Goodnight!"
}
```

Throughout our exploration of the different pointers-to-types, we have encountered several keywords and functions along the way—these keywords and functions being C's `malloc()` and `free()`, and C++'s `new`, `delete`, and `delete[]`. As far sa you have come, it is safe to assume that the meaning and use of these keywords and functions has been realized along the way; however, not all have been mentioned and, as previously stated, the hammering of these concepts into the head of thea reader is paramount to their understanding.

In the following pages, we will explore the functions and keywords mentioned above and their use in dynamic memory allocation in greater detail than we have been able to thus far.

`new`, `placement new`, and `malloc()`:

In the case of dynamic memory allocation in C and C++, the C function `malloc()` (and variations of it) and C++ keyword `new` are used—`placement new` does not allocate memory, it simply *uses* the preordained memory block provided. Both C and C++'s dynamic memory allocating methods should never, **ever**, be used together; to remain on the safe side, if you are using C, use C's memory allocation methods—if you are using C++, use C++'s memory allocation methods, never mix-and-match as it is not only bad design, but reckless, and can lead to unnecessary, and major, problems.

Function [`malloc(size_t size)`]:

`malloc()`, with the function signature above, is used in C to allocate—and return a void pointer to—the requested memory block, in bytes, requested by the function. As this function returns a `void` pointer, it is necessary to cast `malloc()` to match the type of pointer the memory is being assigned to. For instance, see below.

```
int main() {
	// assign pointer to base address of newly allocated block memory
	// the below call to malloc() allocates a memory block to fit 4 integers
	// return value of malloc() must be casted to corresponding type
	int* ptr = (int*)malloc(4 * sizeof(int));

	// check if newly assigned memory block is valid--if malloc() succeeded
	if (ptr) {
		// fill new block of memory with integers via index
		ptr[0] = 56;
		ptr[1] = 25;
		ptr[2] = 22;
		ptr[3] = 66;

		// free dynamically-allocated memory acquired from malloc()
		free(ptr);
	}
}
```

As you saw in the previous example, we assigned the base address of the memory block—of which we requested the size be 4 integers—to our pointer. We then checked if the address our pointer now pointed to was valid (non-NULL) and, if it was, we proceeded to fill our allocated space of integers and then free the dynamically allocated memory block our pointer pointed to. Let us explore another example of malloc, however, this time, with `struct`.

```
typedef struct {
	int int_buffer[4];
	char char_buffer[5];
} TESTSTRUCT, *PTESTSTRUCT;

int main() {
	// assign pointer to base address of newly allocated block memory
	// the below call to malloc() allocates a memory block to fit 2 TESTSTRUCT structures
	// return value of malloc() must be casted to corresponding type
	PTESTSTRUCT pStruct = (PTESTSTRUCT)malloc(2 * sizeof(TESTSTRUCT));

	// check if newly assigned memory block is valid--if malloc() succeeded
	if (pStruct) {
		// fill new block of memory with brace initialized TESTSTRUCT structures via index
		pStruct[0] = { {1, 2, 3, 4}, {'a', 'b', 'c', 'd'} };
		pStruct[1] = { {4, 3, 2, 1}, {'d', 'c', 'b', 'a'} };

		// free dynamically allocated memory acquired from malloc()
		free(pStruct);
	}
}
```

As you can see above, we assigned the base address of the memory block—of which we requested the size be the amount needed to fit 2 structures of type `TESTSTRUCT`—to our pointer. We then checked if the address our pointer now pointed to was valid (non-NULL) and, if it was, we proceeded to fill our allocated space of structures with brace initialized corresponding structures and then free the dynamically allocated memory block our pointer pointed to.

As you can also notice above—and in all previous sections using `malloc()`—the use of the C function `free()` is required on any pointer that acquires its pointed-to memory block via `malloc()` in order to free said block.

Keyword [`new`]:

In C++, the `new` keyword—and it's array form factor `new[]`—is the means by which dynamic memory is allocated on the heap—while also much simpler and more straightforward than C's `malloc()`. Specifically, C++'s `new` keyword—parentheses syntax— is used to heap-allocate a block of memory for exactly one item—following the formula *1 × sizeof(type)*. For instance, see below.

```
int main() {
	// allocate a block in memory for the integer value '3'
	// the allocated block is equal to size 1*sizeof(int) or 1*4
	// enough room for one integer value
	int* a = new int(3);

	delete a;
}
```

As you can see above, when using keyword `new` in conjunction with parentheses, the allocated memory block is equal to the size of the requested type—enough space to fit exactly one of those items of that type. In the case of `new[]`, this syntax is used to allocate a memory block that is able to fit an array of items of type—it will allocate a memory block large enough to fit the number of items specified in between the square brackets. For instance, see below.

```
int main() {
	// allocates a large enough block of memory on the heap to fit an array of five integers
	// new[] allocates a block of memory equal to 'item*sizeof(type) for item in items'
	int* array = new int[5];

	delete[] array;
}
```

As you can see, use of the array syntax of keyword `new` allocates a memory block on the heap large enough to fit each item in the number of items placed in between square brackets. The size, in bytes, of the memory block is equal to *items × sizeof(type)* where *items* is the number is the requested number of elements in between the square brackets. Both form factors of keyword `new` require the use of their own definition of `delete`—to free heap allocated memory.

Keyword [new] Placement Syntax:

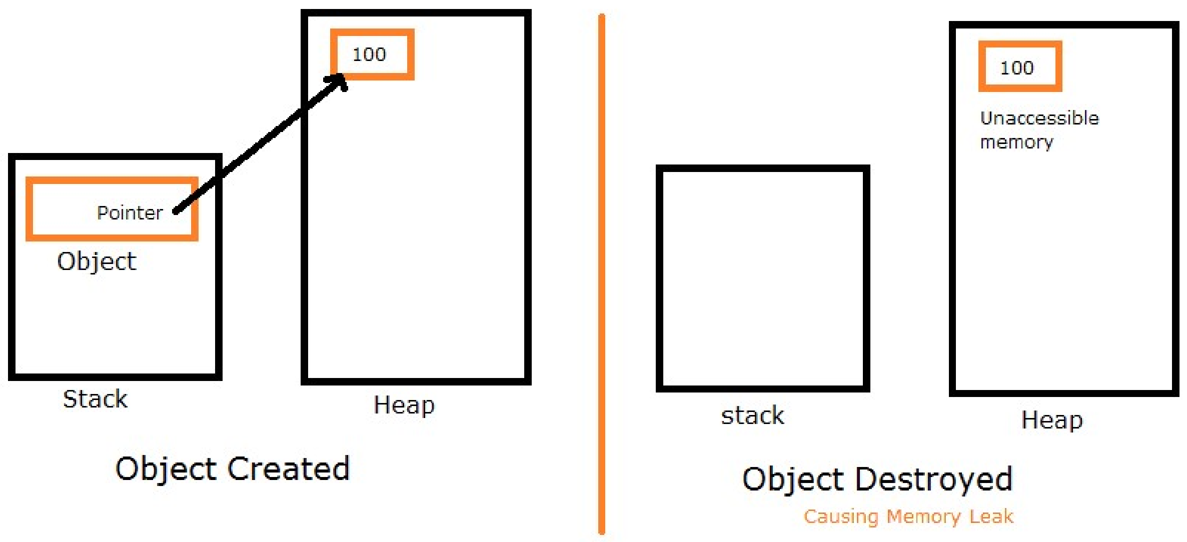
In C++, the use of keyword `new` and its placement syntax—`placement new`—allows a programmer, rather than allocating a new block of memory on the heap, to use a predefined block of memory—on the heap or stack. Since the reuse of preexisting memory is faster than creating and assigning to newly created memory, we can take advantage of `placement new` to create more efficient applications that reuse blocks of memory rather than freeing dynamically allocated blocks and creating new ones. See below for a simple example of `placement new`.

```
typedef struct {
        int iArray[8];
        char cArray[9];
} TESTSTRUCT, *PTESTSTRUCT;

int main() {
        // regular array of unsigned characters with size, in bytes, of TESTSTRUCT
        unsigned char buffer[sizeof(TESTSTRUCT)];

        // placement new - ptr is instructed to use the pre-defined memory above
        PTESTSTRUCT ptr = new (buffer) TESTSTRUCT;

        // allocate structure
        // this structure lies in the memory defined by buffer[]
        *ptr = {
                {1, 2, 3, 4, 5, 6, 7, 8},
                {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}
        };

        // ptr points to the base address of buffer and will use it
        // for construction of the above structure
}
```

As you can see above, we declare a basic character array with a size, in bytes, of `TESTSTRUCT`. All values defined using pointer `ptr` will use the memory of variable `buffer`, as instructed by the `placement new` statement. Keep in mind that the necessity of the use of keywords `delete` and `delete[]` depend on the whether the address passed to `placement new` is heap allocated via `new` or not—it is not necessary to delete a basic `placement new` pointer, but only the address passed to `placement new` if that address is **heap allocated**. Objects that are defined as `placement new` pointers should have their destructors manually called if its side effects is depended by the program.

`free()`, `delete`, and `delete[]`:

As you have likely learned along the way, heap allocated memory must be freed once it is no longer needed—failure to do so will lead to memory leaks and can cause eventual problems in your application. The standard way to free heap allocated memory blocks in C is via `free()` and, in C++, by the use of `delete` for non-arrays and `delete[]` for array allocations. Failure to delete heap allocated memory results in inaccessible memory which can lead to memory issues and crashes. See below.



As you can see above memory that is heap allocated but **not** deleted remains allocated and inaccessible. See below for a examples of clearing memory using all methods—for illustration purposes, **never** mix-and-match in your **own** code.

```
int main() {
	int* a = (int*)malloc(4 * sizeof(int)); // C malloc, allocates memory for four integers
	int* b = new int(65); // C++ new() alloc, allocates memory for one integer '65'
	int* c = new int[4]; // C++ new[] alloc, allocates memory for an array of four integers

	free(a);    // C free of heap memory, frees memory at held address of 'p'
	delete b;   // C++ delete of heap memory, frees memory at held address of 'b'
	delete[] c; // C++ array delete[] of heap memory, frees memory at held address of 'c'
}
```

While void pointers may sound confusing at first, they are incomplete datatypes that can hold the address of any other type and be typecasted to any other type—as they are incomplete datatypes, void pointers cannot be dereferenced. In C, void pointers are largely used to implement generic functions; likewise, void pointers in C++ can be made completely obsolete with the use of carefully crafted templates.

For examples of functions that make use of void pointers, you can look to C's malloc() and C++'s operator new variants. These functions and operators return void pointers to the newly allocated blocks of heap memory due to their operation with unformatted memory and leave the formatting—typecasting—to the programmer. This is why you must explicitly set the type you wish to cast the memory to, like so:

```
int* a = (int*)malloc(4 * sizeof(int));
int* c = new int[4];
```

As you can see above, C's malloc() makes use of an explicit C-style cast to convert the returned void pointer to a pointer-to-int, whereas C++'s new[] makes use of a syntactical cast directly following the new keyword—this is to ensure the returned pointer **matches** the type it is being assigned to, as required.

There is *no way* to tell the type that a void pointer is pointing to as this information is non-existent in the finalized code unless a means of finding out is in the source code itself—such as there being a guarantee on what the void pointer points to, or the programmer uses bits to identify types pointed to by such a pointer. In general, void pointers have no consistent way of extrapolating the pointed-to type and functions that use them must be constructed carefully.

Another example of a function that uses `void` pointers in its arguments can be C threading functions—specifically, Win32's `_beginthreadex()`. See below for the signature of `_beginthreadex()`.

```
uintptr_t _beginthreadex(
        void* security,
        unsigned stack_size,
        unsigned(__stdcall* start_address)(void*),
        void* arglist,
        unsigned initflag,
        unsigned* thrdaddr
);
```

The above thread-starting function is described as follows by Microsoft as…

> "The `_beginthread` function creates a thread that begins execution of a routine at `start_address`. The routine at `start_address` must use the `__cdecl` (for native code) or `__clrcall` (for managed code) calling convention and should have no return value. […] The `_beginthreadex` function gives you more control over how the thread is created than `_beginthread` does. "

You can find an example of a program counting to a user-defined number (which is casted to, then from, a `void` pointer) on another thread and waiting on the main thread for it to finish on the following page. We are using Win32's `_beginthreadex()` due to its support for the function `WaitForSingleObject()` which allows us to pause the main thread until the detached thread is complete. You can find the signature of `WaitForSingleObject()` below.

```
DWORD WaitForSingleObject(
        HANDLE hHandle, // handle to thread
        DWORD  dwMilliseconds // time to wait
);
```

Below you can find an example of a program counting to a user-defined number (which is casted to, then from, a `void` pointer) on another thread and waiting on the main thread for it to finish on the following page.

```
unsigned __stdcall countTo(void* limit) {
        // convert void to pointer-to-int and dereference for value – assign to ‘lim’
        int lim = *(int*)limit;

        printf("Counting to %i on thread %i.\n", lim, GetCurrentThreadId());
        for (int i = 1; i < lim; i++) {
                printf("%i\n", i);
        }

        return 0;
}

int main() {
        // user-defined count limit
        int limit = 10;

        // handle of detached thread - used for waiting
        HANDLE complete;

        // start counting to limit on another thread
        complete = (HANDLE)_beginthreadex(NULL, 0, countTo, &limit, NULL, NULL);

        // check if handle to created thread is valid - thread successfully created
        if (complete != INVALID_HANDLE_VALUE) {
                printf("Waiting on thread %i for detached thread to complete.\n", GetCurrentThreadId());
                WaitForSingleObject(complete, INFINITE); // wait until thread signals it is done
                printf("countTo finished, control returned to thread %i.\n", GetCurrentThreadId());
        }
}
```

```
        Example output:
Waiting on thread 9432 for detached thread to complete.
Counting to 10 on thread 9636.
1
2
3
4
5
6
7
8
9
countTo finished, control returned to thread 9432.
```

As you can see above, Windows C API function `_beginthreadex()` uses a `void` pointer to pass generic type-agnostic parameters to user-supplied functions. The passed function is the one responsible for handling the typecasting to the correct type needed for correct usage in the function.

III. Memory in Programming ***Smart Pointers***

Smart pointers are a collection of template classes in the C++ language that encapsulate heap allocated pointers returned via the new operator within one of three smart pointer template classes that allow for heap allocated memory to be automatically *garbage collected* by the class destructor. This effectively eliminates the responsibility of the programmer having to do manual memory cleanup as the smart pointers clean up the memory provided to the by themselves once they go out of scope via their class destructor. Before we illustrate the use of smart pointers, we must first go over the three types of smart pointer template classes.

Smart Pointer [`std::unique_ptr`]:

According to Microsoft, a `std::unique_ptr` is…

> "[`std::unique_ptr` is a smart pointer that] allows exactly one owner of the underlying pointer. Use as the default choice for POCO unless you know for certain that you require a `shared_ptr`. Can be moved to a new owner, but not copied or shared. Replaces `auto_ptr`, which is deprecated. Compare to `boost::scoped_ptr`. `unique_ptr` is small and efficient; the size is one pointer, and it supports `rvalue` references for fast insertion and retrieval from C++ Standard Library collections."

Smart Pointer [`std::shared_ptr`]:

According to Microsoft, a `std::shared_ptr` is…

> "[`std::shared_ptr` is a] reference-counted smart pointer. Use when you want to assign one raw pointer to multiple owners, for example, when you return a copy of a pointer from a container but want to keep the original. The raw pointer is not deleted until all `shared_ptr` owners have gone out of scope or have otherwise given up ownership. The size is two pointers; one for the object and one for the shared control block that contains the reference count."

Smart Pointer [`std::weak_ptr`]:

According to Microsoft, a `std::weak_ptr` is…

> "[`std::weak_ptr` is a] special-case smart pointer for use in conjunction with `shared_ptr`. A `weak_ptr` provides access to an object that is owned by one or more `shared_ptr` instances, but does not participate in reference counting. Use when you want to observe an object, but do not require it to remain alive. Required in some cases to break circular references between `shared_ptr` instances."

Now, with the concept of all three types of smart pointers fresh in mind, we will explore illustrative source code of each. Below you can find a simple example using `std::unique_ptr`.

```
int main() {
	// std::make_unique returns a std::unique_ptr object holding a heap allocated
	// type with given value - assign to ptr1
	std::unique_ptr<int> ptr1 = std::make_unique<int>(50);
	// also acceptable: std::unique_ptr<int> ptr(new int(50)); (dangerous)

	// the above can be operated on as if it were a regular pointer-to-int,
	// as well as take advantage of the class methods offered by std::unique_ptr
	std::cout << *ptr1 << std::endl; // prints 50
	std::cout << ptr1 /* or ptr1.get() */ << std::endl; // prints address of stored heap-alloc value

	// the value and accompanying pointer stored in the unique_ptr can be reset and changed like so
	ptr1.reset(new int(44));

	std::cout << *ptr1 << std::endl; // prints 44

	// reassign--or "move"--address of heap allocated data to new unique_ptr
	// ptr1 no longer holds moved address after this call
	std::unique_ptr<int> ptr2 = std::move(ptr1);
	std::cout << *ptr2 << std::endl; // prints 44
	std::cout << ptr2 /* or ptr2.get() */ << std::endl; // prints address of stored heap-alloc value

	// no need to call delete or delete[]
	// all handled by the class destructor and whatnot
}
```

As you can see above, `std::unique_ptr` can largely be operated on as if it were a normal raw pointer—with the advantage of `std::unique_ptr` class methods. As `std::unique_ptr` is only allowed one owner, nothing else can own its underlying pointer (doing so would be *undefined behavior* in accordance with the standard)—but it can be moved to another `std::unique_ptr` via assigning `std::move()` to the new `std::unique_ptr`. After all is done with the object and it goes out of scope, the heap allocated data is cleared and freed by the `std::unique_ptr` internally—so there is no need to call manual delete operators.

`std::unique_ptr` is a non-copiable type and should be passed using move-semantics; by a passed return value (such as from std::move), by non-`const lvalue` reference, by `const` lvalue reference, or by `rvalue` reference (&&).

Below you will find an example use of `std::shared_ptr`—a special type of smart pointer that allows for multiple references, each tracked by an internal pointer to a reference counter in the class. When this internal counter eventually reaches zero, the dynamically allocated data this smart pointer holds is deallocated without insight of the programmer. This type of smart pointer is useful when you want to **share** your dynamically allocated data around—the same way you would with raw (or *dumb*, in this case) pointers and references.

```
// change value of pointer held by sharing ownership to another shared_ptr
// when a shared_ptr is passed to arg, both shared_ptr reference counts are incremented
//
// NOTE: this function can be done more efficiently and SHOULD in your
// own source code. Prefer ‘std::shared_ptr<std::string>& arg’ over value.
// std::shared_ptr uses an atomic reference counter which is VERY heavy
// to copy compared to basic incrementation--so, prefer using lvalue
// references or rvalue move semantics whenever possible
void change(std::shared_ptr<std::string> arg) {
        // print reference count: 2 -> two shared_ptr's hold the same dynamic data
        std::cout << arg.use_count() << std::endl;

        // change pointed-to data held by both shared_ptr objects
        *arg = "Hello!";
}

int main() {
        // dynamically allocate a new std::string "Hey!" within smart pointer 'a'
        std::shared_ptr<std::string> a = std::make_shared<std::string>("Hey!");

        // dereference held address, print its data "Hey!"
        std::cout << *a << std::endl;

        // pass smart pointer 'a' by value, creating a copy and
        change(a);

        // reference count now 1 -> smart pointer copy in above function out of scope, destroyed
        std::cout << a.use_count() << std::endl;

        // dereference held address, print its data "Hello!"
        std::cout << *a << std::endl;
}
```

`std::shared_ptr` also introduces a problem that the programmer should be very weary of. While smart pointers are designed to take the manual memory deallocation out of the hands of the programmer—it does not mean it *always* will if used improperly. One such case of improper use with smart pointers—specifically `std::shared_ptr`—is *circular references*. You can find an example of a circular reference on the following page.

A circular reference, or a cy*clical reference*, is a series of references where each references the next and the last references the first—which causes a referential loop. In the case of `std::shared_ptr` this will cause problems with the smart pointer being able to deallocate data automatically—usually not being able to at all. See an example of a cyclical reference below.

```
class example {
        public:
                std::shared_ptr<example> ptr;
                ~example() { std::cout << "~example\n"; }
};

int main() {
        std::shared_ptr<example> example1 = std::make_shared<example>();
        std::shared_ptr<example> example2 = std::make_shared<example>();

        example1->ptr = example2;
        example2->ptr = example1;
}
```

When `example1` goes out of scope, its destructor can't clean the held dynamic memory; there is still one smart pointer pointing at `example1`, which is `example2->ptr`. Similarly, when `example2` goes out of scope, its destructor can't cleanup the held dynamic memory: a reference to `example2` still lives through `example1->ptr`. Without the ability to actually free the memory, the program will exit without freeing the dynamically allocated memory—which causes a memory leak. In the above code, the destructor of the class is never called—signifying the dynamically allocated data is never freed.

Thanks to modern operating systems, the above example with a memory leak is not a massive issue as, on exit, the system will handle the leak itself; however, in the case of large software, regardless of the underlying operating system, a memory leak occurring in the middle of the software can quickly lead to memory issues and crashes.

On the following page, we will explore the mechanics of `std::weak_ptr` and how it can be used to solve problems that raw, dumb pointers cannot.

A `std::weak_ptr` can be defined as a `std::shared_ptr` that does not partake in incrementing the reference count—it holds a *non-owning reference* to something that is managed by another `std::shared_ptr`. With `std::weak_ptr`, you can only create one out of a `std::shared_ptr` or another `std::weak_ptr`. See below for an example declaration of both.

```
int main() {
	std::shared_ptr<std::string> shared = std::make_shared<std::string>("Hello, world!");
	std::weak_ptr<std::string> weak1(shared);
	std::weak_ptr<std::string> weak2(weak1);
}
```

Below you can see an example of `std::weak_ptr` in action with illustrative comments.

```
int main() {
	// shared_ptr holding dynamic string pointing to value "200"
	std::shared_ptr<int> shared = std::make_shared<int>(200);

	// weak_ptr holding shared_ptr 'shared' -> does not influence refcount
	std::weak_ptr<int> weak1(shared);

	// weak_ptr holding weak_ptr 'weak1' -> does not influence refcount
	std::weak_ptr<int> weak2(weak1);

	// weak_ptr can be seen as an "inspector" of the shared_ptr it holds
	std::cout << shared.use_count() << std::endl; // refcount: 1
	std::cout << weak1.use_count() << std::endl;  // refcount: 1
	std::cout << weak2.use_count() << std::endl;  // refcount: 1

	// as an inspector by default, if you want to work on the object it inspects
	// it must be converted from weak_ptr to a shared_ptr
	std::shared_ptr<int> shared_original = weak1.lock();

	std::cout << *shared_original << std::endl;   // print held value
	std::cout << shared.use_count() << std::endl; // refcount: 1 -> two hold same dynamic data
}
```

`std::weak_ptr` can be used to solve the aforementioned—and dangerous—circular reference issues—among more, such as with reference validity via `weak_ptr::expired`. Let us return to our previous example, but with the addition of `std::weak_ptr`.

```
class example {
	public:
		// holds weak reference to assigned data - not taking part in refcount
		std::weak_ptr<example> ptr;
		~example() { std::cout << "~example\n"; }
};

int main() {
	std::shared_ptr<example> example1 = std::make_shared<example>();
	std::shared_ptr<example> example2 = std::make_shared<example>();

	example1->ptr = example2;
	example2->ptr = example1;

	// destructors are called appropriately as weak_ptr avoids circular reference
}
```

Smart pointers allow you to specify a custom *deleter*—a function—or `operator()` overload—that deletes the stored dynamically allocated data in a manner defined by the programmer. This is useful for custom containers and in scenarios where the programmer wishes to provide their own memory management routines to the smart pointers for said containers. See below for examples.

```
struct deleter {
	void operator()(int* i) {
		delete i;
		std::cout << "Deleted!" << std::endl;
	}
};

void deleterFunc(int* i) { delete i; std::cout << "Deleted!" << std::endl; }

int main() {
	std::unique_ptr<int, deleter> uptr(new int(100)); // unique_ptr deleter via operator()
	std::shared_ptr<int> sptr1(new int(100), deleterFunc); // shared_ptr deleter via function ptr
	std::shared_ptr<int> sptr2(new int(100), [](int* i) {
		delete i;
		std::cout << "Deleted!" << std::endl;
	}); // shared_ptr deleter via lambda

	// once out of scope, custom deleter is called
	// output:
	//   Deleted!
	//   Deleted!
	//   Deleted!
}
```

Custom deleters are also useful for prior versions of the C++ standard—prior to C++17—that did not differentiate between calls to `delete` or `delete[]` for `std::shared_ptr` holding pointers-to-arrays or regular pointers. For instance, see below—**assume pre-C++17**.

```
int main() {
	// shared_ptr holding dynamically allocated array of 10 integers
	// prior to C++17, a custom deleter for type[] is required to
	// effectively free memory based on array
	//
	// prior to C++17, delete is called by default (not delete[])
	// on arrays by shared_ptr--issues arise!
	std::shared_ptr<int[]> sptr2(new int[10], [](int* i) {
		delete[] i;
		std::cout << "Deleted!" << std::endl;
	});

	// the above effectively clears the memory allocated by the array,
	// whereas...
	std::shared_ptr<int[]> sptr2(new int[10]);
	// ...would not
	// prior to C++17, the above calls delete (not delete[])
	// which is an issue
}
```