

## 4. Memory Management

CORE CURRICULUM

Assessment: Memory Management Chatbot due by Aug 4th

100% VIEWED

PROJECTS ▢

### LESSON 1

#### Introduction

[VIEW LESSON →](#)

100% VIEWED

[SHRINK CARD](#)



### LESSON 2

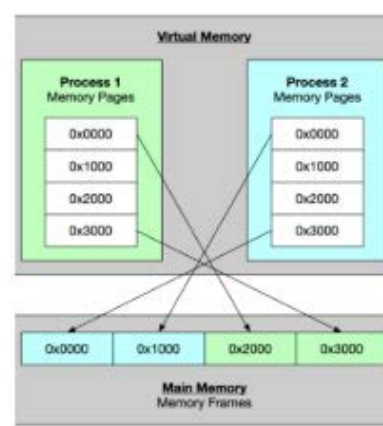
#### Overview of Memory Types

This lesson covers basic concepts such as cache, virtual memory, and the structure of memory addresses. In addition, it is demonstrated how the debugger can be used to read data from memory.

[VIEW LESSON →](#)

100% VIEWED

[SHRINK CARD](#)



### LESSON 3

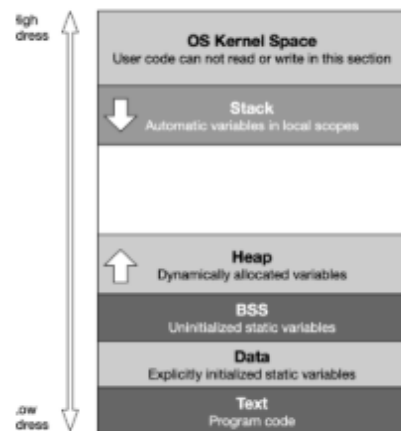
#### Variables and Memory

In this lesson the process memory model is introduced, which contains the two fundamental memory areas heap and stack, which play an important role in C++.

[VIEW LESSON →](#)

100% VIEWED

[SHRINK CARD](#)



#### LESSON 4

##### Dynamic Memory Allocation (The Heap)

This lesson introduces dynamic memory allocation on the heap. The commands malloc and free as well as new and delete are introduced for this purpose.



[VIEW LESSON →](#)

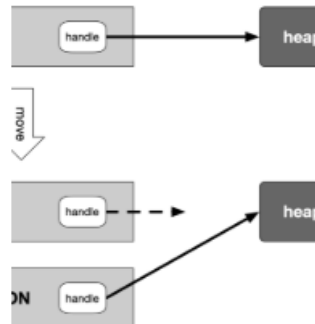
100% VIEWED

[SHRINK CARD](#)

#### LESSON 5

##### Resource Copying Policies

This section describes how to customize resource copying using the Rule of Three. Also, the Rule of Five is introduced, which helps develop a thorough memory management strategy in your code.



[VIEW LESSON →](#)

100% VIEWED

[SHRINK CARD](#)

#### LESSON 6

##### Smart Pointers

In this lesson the three types of smart pointers in C++ are presented and compared. In addition, it is shown how to transfer ownership from one program part to another using copy and move semantics.



[VIEW LESSON →](#)

100% VIEWED

[SHRINK CARD](#)

#### PROJECT

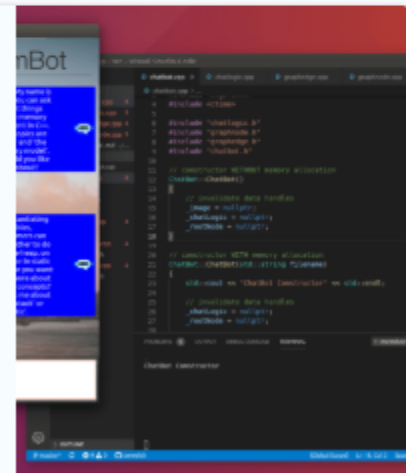
##### Memory Management Chatbot

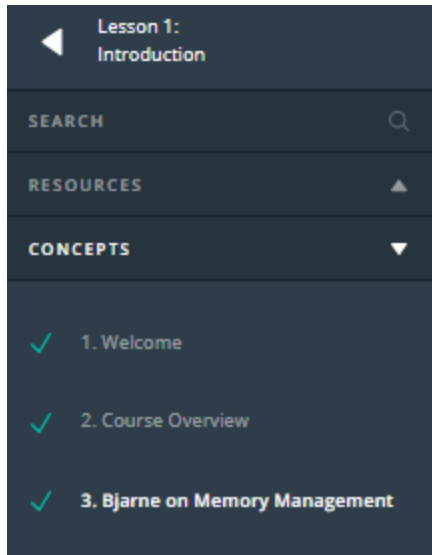
The ChatBot project creates a dialogue where users can ask questions about some aspects of memory management in C++. Your task will be to optimize the project with modern memory management in mind.

[CONTINUE →](#)

STARTED

Due in 2 months





- 1) Welcome  
<https://youtu.be/hYu3mxQzKNY>
- 2) Course Overview  
<https://youtu.be/f92guZr2jWo>

## Course Outline

### 1. Overview of Memory Types

1. Memory Addresses and Hexadecimal Numbers
2. Using the Debugger to Analyze Memory
3. Types of Computer Memory
4. Cache Memory
5. Virtual Memory

### 2. Variables and Memory

1. The Process Memory Model
2. Automatic Memory Allocation (The Stack)
3. Call-By-Value vs. Call-By-Reference

### 3. Dynamic Memory Allocation (The Heap)

1. Heap Memory
2. Using malloc and free
3. Using new and delete

#### 4. Typical Memory Management Problems

### **4.Resource Copying Policies**

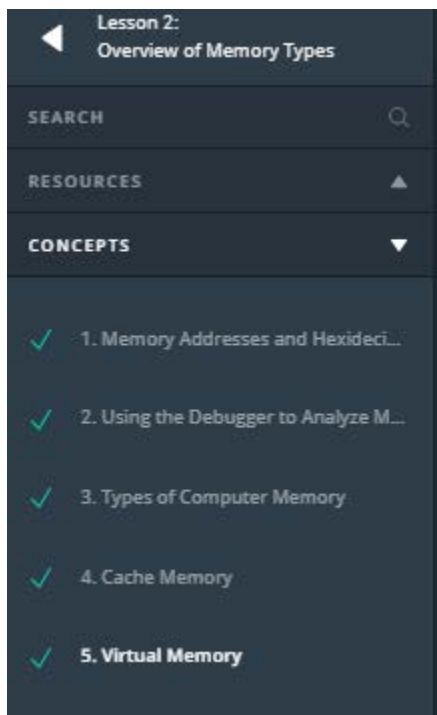
1. Copy Semantics
2. Lvalues and rvalues
3. Move Semantics

### **5. Smart Pointers**

1. Resource Acquisition Is Initialization (RAII)
2. Smart pointers
3. Transferring ownership

### **6. Project: Memory Management Chatbot**

- 3) Bjarne on Memory Management  
<https://youtu.be/Rt2p5Pgnb8g>



- 1) Memory Address and Hexadecimal Numbers  
<https://youtu.be/OAU28NfySsQ>

## Memory Addresses and Hexadecimal Numbers

Understanding the number system used by computers to store and process data is essential for effective memory management, which is why we will start with an introduction into the binary and hexadecimal number systems and the structure of memory addresses.

Early attempts to invent an electronic computing device met with disappointing results as long as engineers and computer scientists tried to use the decimal system. One of the biggest problems was the low distinctiveness of the individual symbols in the presence of **noise**. A 'symbol' in our alphabet might be a letter in the range A-Z while in our decimal system it might be a number in the range 0-9. The more symbols there are, the harder it can be to differentiate between them, especially when there is electrical interference. After many years of research, an early pioneer in computing, John Atanasoff, proposed to use a coding system that expressed numbers as sequences of only two digits: one by the presence of a charge and one by the absence of a charge. This numbering system is called Base 2 or binary and it is represented by the digits 0 and 1 (called 'bit') instead of 0-9 as with the decimal system. Differentiating between only two symbols, especially at high frequencies, was much easier and more robust than with 10 digits. In a way, the ones and zeroes of the binary system can be

compared to Morse Code, which is also a very robust way to transmit information in the presence of much interference. This was one of the primary reasons why the binary system quickly became the standard for computing.

Inside each computer, all numbers, characters, commands and every imaginable type of information are represented in binary form. Over the years, many coding schemes and techniques were invented to manipulate these 0s and 1s effectively. One of the most widely used schemes is called ASCII (*American Standard Code for Information Interchange*), which lists the binary code for a set of 127 characters. The idea was to represent each letter with a sequence of binary numbers so that storing texts on in computer memory and on hard (or floppy) disks would be possible.

The film enthusiasts among you might know the scene in the hit movie "The Martian" with Mat Daemon, in which an ASCII table plays an important role in the rescue from Mars.

The following figure shows an ASCII table, where each character (rightmost column) is associated with an 8-digit binary number:

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	\$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(	72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29	)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[	123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D	]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

The letter **u** for example can be represented by the following sequence of bits: **01010101**

QUESTION 1 OF 2

Can you figure out the binary sequence for the word "UDACITY"?

- ☐ 01001000 01100101 01101100 01101100 01101111 0100000 01010111 01101111  
01110010 01101100 01100100
- ☐ 01110100 01110010 01111001 01000001 01100111 01100001 01101001 01101110
- ☐ 01010101 01000100 01000001 01000011 01001001 01010100 01011001
- ☐ 01010101 01000100 01001111 01001110 01001101 01001101 01001101

QUESTION 1 OF 2

Can you figure out the binary sequence for the word "UDACITY"?

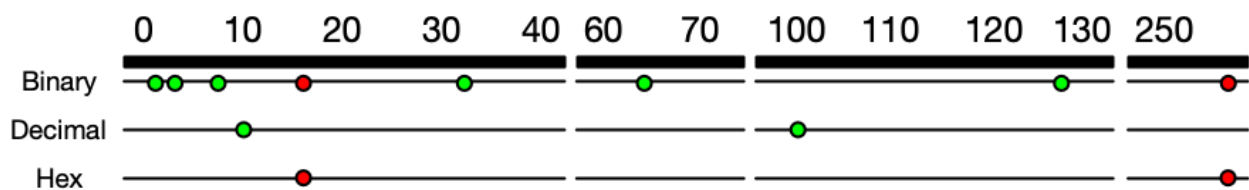
- ☐ 01001000 01100101 01101100 01101100 01101111 0100000 01010111 01101111  
01110010 01101100 01100100
- ☐ 01110100 01110010 01111001 01000001 01100111 01100001 01101001 01101110
- ☒ 01010101 01000100 01000001 01000011 01001001 01010100 01011001
- ☐ 01010101 01000100 01001111 01001110 01001101 01001101 01001101

In addition to the decimal number (column "Dec") and the binary number, the ASCII table provides a third number for each character (column "Hex"). According to the table above, the letter **z** is referenced by the decimal number **122**, by the binary number **0111 1010** and by **7A**. You have probably seen this type of notation before, which is called "*hexadecimal*". Hexadecimal (hex) numbers are used often in computer systems, e.g for displaying memory readouts - which is why we will look into this topic a little bit deeper. Instead of having a base of 2 (such as binary numbers) or a base of 10 (such as our conventional decimal numbers), hex numbers have a base of 16. The conversion between the different numbering systems is a straightforward operation and can be easily performed with any scientific calculator. More details on how to do this can e.g. be found [here](#).

There are several reasons why it is preferable to use hex numbers instead of binary numbers (which computers store at the lowest level), three of which are given below:

1. **Readability:** It is significantly easier for a human to understand hex numbers as they resemble the decimal numbers we are used to. It is simply not intuitive to look at binary numbers and decide how big they are and how they relate to another binary number.
2. **Information density:** A hex number with two digits can express any number from 0 to 255 (because  $16^2$  is 256). To do the same in the binary system, we would require 8 digits. This difference is even more pronounced as numbers get larger and thus harder to deal with.
3. **Conversion into bytes:** Bytes are units of information consisting of 8 bits. Almost all computers are byte-addressed, meaning all memory is referenced by byte, instead of by bit. Therefore, using a counting system that can easily convert into bytes is an important requirement. We will shortly see why grouping bits into a byte plays a central role in understanding how computer memory works.

The reason why early computer scientists have decided to not use decimal numbers can also be seen in the figure below. In these days (before pocket calculators were widely available), programmers had to interpret computer output in their head on a regular basis. For them, it was much easier and quicker to look at and interpret `7E` instead of `0111 1110`. Ideally, they would have used the decimal system, but the conversion between base 2 and base 10 is much harder than between base 2 and base 16. Note in the figure that the decimal system's digit transitions never match those of the binary system. With the hexadecimal system, which is based on a multiple of 2, digit transitions match up each time, thus making it much easier to convert quickly between these numbering systems.



Each dot represents an increase in the number of digits required to express a number in different number systems. For base 2, this happens at 2, 4, 8, 32, 64, 128 and 256. The red dots indicate positions where several numbering systems align. Note that there are breaks in the number line to conserve space.



QUESTION 2 OF 2

Convert the following numbers from binary into hex and vice-versa:

11101 96 1111 1111 D9

NUMBER	CONVERSION
FF	
1D	
1101 1001	
1001 0110	

QUESTION 2 OF 2

Convert the following numbers from binary into hex and vice-versa:

*Submit to check your answer choices*

NUMBER	CONVERSION
FF	1111 1111
1D	11101
1101 1001	D9
1001 0110	96

Outro

<https://youtu.be/q1C7lBXNGn0>

## 2) Using the Debugger to Analyze Memory

<https://youtu.be/-p99igKSazc>

# Using the Debugger to Analyze Memory

As you have seen in the last section, binary numbers and hex numbers can be used to represent information. A coding scheme such as an ASCII table makes it possible to convert text into binary form. In the following, we will try to look at computer memory and locate information there.

In the following example, we will use the debugger to look for a particular string in computer memory. Depending on your computer operating system and on the compiler you have installed, there might be several debugging tools available to you. In the following video, we will use the gdb debugger to locate the character sequence "UDACITY" in computer memory. The code below creates an array of characters in computer memory (on the stack, which we will learn more about shortly) and prints it to the console:

```
#include <stdio.h>

int main()
{
    char str1[] = "UDACITY";
    printf("%s", str1);

    return 0;
}
```

Let us try to locate the string in memory using gdb.

<https://youtu.be/Spj2jK1-ulE>

<https://youtu.be/Lopa5WXR1uQ>

As you have just seen in the video, the binary ASCII codes for the letters in UDACITY could be located in computer memory by using the address of the variable `str1` from the code example above. The output of gdb can also be observed in the following image:

```
(gdb) p str1
$1 = "UDACITY"
(gdb) p &str1
$2 = (char (*)[8]) 0x7ffefbfff940
(gdb) x/7tb 0x7ffefbfff940
0x7ffefbfff940: 01010101    01000100    01000001    01000011    01001001    01010100    01011001
(gdb) x/7xb 0x7ffefbfff940
0x7ffefbfff940: 0x55    0x44    0x41    0x43    0x49    0x54    0x59
```

You can clearly see that using hex numbers to display the information is a much shorter and more convenient form for a human programmer than looking at the binary numbers. Note that hex numbers are usually prepended with "0x".

Computer memory is treated as a sequence of cells. This means that we can use the starting address to retrieve the byte of information stored there. The following figure illustrates the principle:

	0x7ffeefbff940	0x7ffeefbff941	0x7ffeefbff942	
	01010101	01000100	01000001	

Computer memory represented as a sequence of data cells (e.g. 01010101) with their respective memory addresses shown on top.

Let us perform a short experiment using gdb again: By adding 1, 2, 3, ... to the address of the string variable `str1`, we can proceed to the next cell until we reach the end of the memory we want to look at.

```
(gdb) x/1xb 0x7ffeefbff940
0x7ffeefbff940: 0x55
(gdb) x/1xb 0x7ffeefbff941
0x7ffeefbff941: 0x44
(gdb) x/1xb 0x7ffeefbff942
0x7ffeefbff942: 0x41
(gdb) x/1xb 0x7ffeefbff943
0x7ffeefbff943: 0x43
(gdb) x/1xb 0x7ffeefbff944
0x7ffeefbff944: 0x49
(gdb) x/1xb 0x7ffeefbff945
0x7ffeefbff945: 0x54
(gdb) x/1xb 0x7ffeefbff946
0x7ffeefbff946: 0x59
(gdb) x/1xb 0x7ffeefbff947
0x7ffeefbff947: 0x00
```

Note that the numbers above represent the string "UDACITY" again. Also note that once we exceed the end of the string, the memory cell has the value 0x00. This means that the experiment has shown that an offset of 1 in a hexadecimal address corresponds to an offset of 8 bits (or 1 byte) in computer memory.

### Your Turn

Unfortunately, gdb will not work in Udacity Workspaces, but you can still try this exercise either in your local environment if you have g++ and gdb installed, or you can use [OnlineGDB](#) to follow along.

[https://www.onlinegdb.com/?\\_cf\\_chl\\_jschl\\_tk\\_=59ca2970e5902f13f32a7c51df0e2fd60b216ccf-1590691451-0-ASKlbu\\_UKPaAmjck08dVmJyIZGF0A8R0\\_FO\\_XoeCqt59xUQ2eDVQn\\_sADoTVuDTVmeOk\\_uIwC-\\_hWn84gLMk99LTf9HMPv0KkB9611Zo8uP0kykcPUHs31a\\_3SzkTckwt6kZW2aBvF8Pp8g56Ruh97E7CteKYyOFOOc-aFcyqBbJztxW6oJIMmsSnCV6NAvDR75nkhCTA4KBprKnfzjbKYVO38GwYbcrldZg54hD-imp11jw3jeX8AXCJ8k4L3mn0dkVTqqAu0QFowSruidBHR](https://www.onlinegdb.com/?_cf_chl_jschl_tk_=59ca2970e5902f13f32a7c51df0e2fd60b216ccf-1590691451-0-ASKlbu_UKPaAmjck08dVmJyIZGF0A8R0_FO_XoeCqt59xUQ2eDVQn_sADoTVuDTVmeOk_uIwC-_hWn84gLMk99LTf9HMPv0KkB9611Zo8uP0kykcPUHs31a_3SzkTckwt6kZW2aBvF8Pp8g56Ruh97E7CteKYyOFOOc-aFcyqBbJztxW6oJIMmsSnCV6NAvDR75nkhCTA4KBprKnfzjbKYVO38GwYbcrldZg54hD-imp11jw3jeX8AXCJ8k4L3mn0dkVTqqAu0QFowSruidBHR)

Try to locate the characters of "Udacity" using gdb in your local environment or in the online debugger.

### Using GDB Locally

In order to use gdb locally, you will need to compile `main.cpp` with [debugging symbols](#).

[https://en.wikipedia.org/wiki/Debug\\_symbol](https://en.wikipedia.org/wiki/Debug_symbol)

This can be done with the `-g` option for g++:

```
g++ -g main.cpp
```

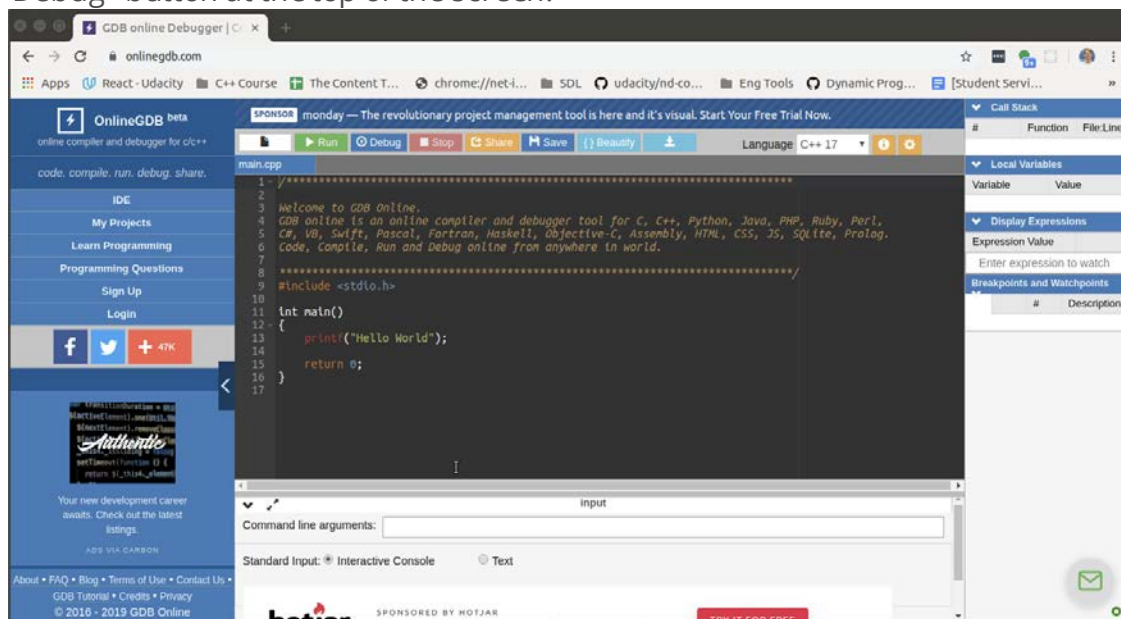
You can then run gdb on the output with:

```
gdb a.out
```

When gdb displays the line `Type <RET> for more, q to quit, c to continue without paging`, be sure to press the RETURN key to continue.

### Using GDB Online

To use the OnlineGDB application, simply paste the code into the online editor and press the "Debug" button at the top of the screen.



You can find the gdb "cheat sheet" used in the videos above [here](#).  
<https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>  
see downloaded cheat sheet

Outro

<https://youtu.be/9oESTYFVCV8>

### 3) Types of Computer Memory

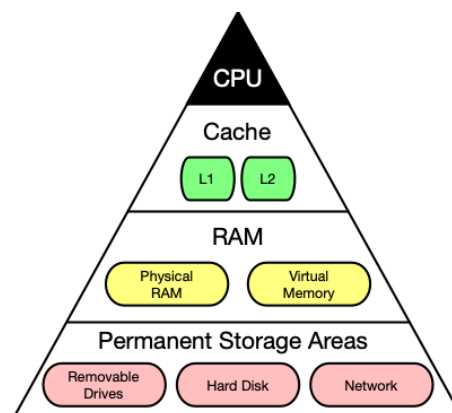
<https://youtu.be/hUKRiLAj0EM>

In a course on memory management we obviously need to take a look at the available memory types in computer systems. Below you will find a small list of some common memory types that you will surely have heard of:

- RAM / ROM
- Cache (L1, L2)
- Registers
- Virtual Memory
- Hard Disks, USB drives

Let us look into these types more deeply: When the CPU of a computer needs to access memory, it wants to do this with minimal latency. Also, as large amounts of information need to be processed, the available memory should be sufficiently large with regard to the tasks we want to accomplish.

Regrettably though, low latency and large memory are not compatible with each other (at least not at a reasonable price). In practice, the decision for low latency usually results in a reduction of the available storage capacity (and vice versa). This is the reason why a computer has multiple memory types that are arranged hierarchically. The following pyramid illustrates the principle:



Computer memory latency and size hierarchy.

As you can see, the CPU and its ultra-fast (but small) registers used for short-term data storage reside at the top of the pyramid. Below are Cache and RAM, which belong to the category of temporary memory which quickly loses its content once power is cut off. Finally, there are permanent storage devices such as the ROM, hard drives as well as removable drives such as USB sticks.

Let us take a look at a typical computer usage scenario to see how the different types of memory are used:

1. After switching on the computer, it loads data from its read-only memory (ROM) and performs a power-on self-test (POST) to ensure that all major components are working properly. Additionally, the computer memory controller checks all of the memory addresses with a simple read/write operation to ensure that memory is functioning correctly.
2. After performing the self-test, the computer loads the basic input/output system (BIOS) from ROM. The major task of the BIOS is to make the computer functional by providing basic information about such things as storage devices, boot sequence, security or auto device recognition capability.
3. The process of activating a more complex system on a simple system is called "bootstrapping": It is a solution for the chicken-egg-problem of starting a software-driven system by itself using software. During bootstrapping, the computer loads the operating system (OS) from the hard drive into random access memory (RAM). RAM is considered "random access" because any memory cell can be accessed directly by intersecting the respective row and column in the matrix-like memory layout. For performance reasons, many parts of the OS are kept in RAM as long as the computer is powered on.
4. When an application is started, it is loaded into RAM. However, several application components are only loaded into RAM on demand to preserve memory. Files that are opened during runtime are also loaded into RAM. When a file is saved, it is written to the specified storage device. After closing the application, it is deleted from RAM. This simple usage scenario shows the central importance of the RAM. Every time data is loaded or a file is opened, it is placed into this temporary storage area - but what about the other memory types above the RAM layer in the pyramid?

To maximize CPU performance, fast access to large amounts of data is critical. If the CPU cannot get the data it needs, it stops and waits for data availability. Thus, when designing new memory chips, engineers must adapt to the speed of the available CPUs. The problem they are facing is that memory which is able to keep up with modern CPUs running at several GHz is extremely expensive. To combat this, computer designers have created the memory tier system which has already been shown in the pyramid diagram above. The solution is to use expensive memory in small quantities and then back it up using larger quantities of less expensive memory.

The cheapest form of memory available today is the hard disk. It provides large quantities of inexpensive and permanent storage. The problem of a hard disk is its comparatively low speed - even though access times with modern solid state disks (SSD) have decreased significantly compared to older magnetic-disc models.

The next hierarchical level above hard disks or other external storage devices is the RAM. We will not discuss in detail how it works but only take a look at some key performance metrics of the CPU at this point, which place certain performance expectations on the RAM and its designers:

1. The **bit size** of the CPU decides how many bytes of data it can access in RAM memory at the same time. A 16-bit CPU can access 2 bytes (with each byte consisting of 8 bit) while a 64-bit CPU can access 8 bytes at a time.
2. The **processing speed** of the CPU is measured in Gigahertz or Megahertz and denotes the number of operations it can perform in one second.

From processing speed and bit size, the data rate required to keep the CPU busy can easily be computed by multiplying bit size with processing speed. With modern CPUs and ever-increasing speeds, the available RAM in the market will not be fast enough to match the CPU data rate requirements.

## Outro

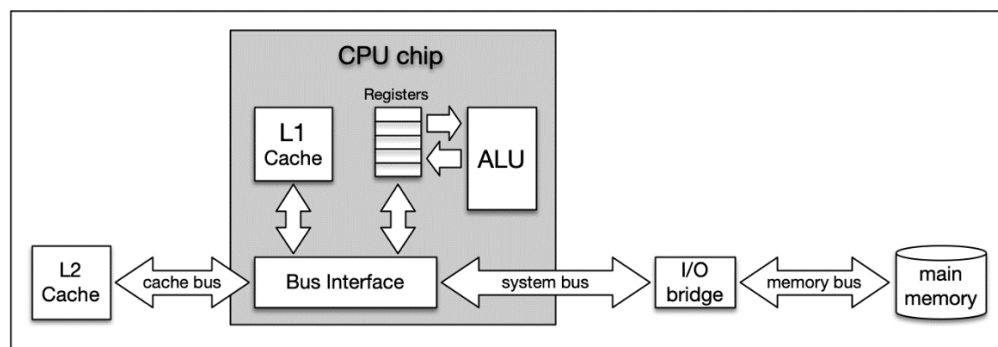
<https://youtu.be/cfnC4fBeWfU>

### 4) Cache Memory

<https://youtu.be/FNaaBipEqBw>

## Cache Levels

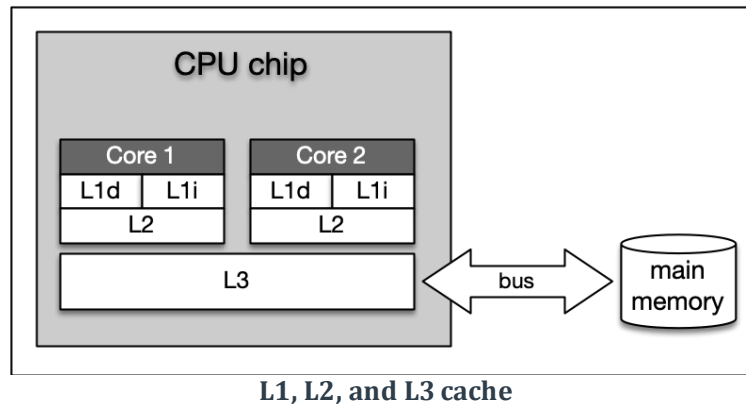
Cache memory is much faster but also significantly smaller than standard RAM. It holds the data that will (or might) be used by the CPU more often. In the memory hierarchy we have seen in the last section, the cache plays an intermediary role between fast CPU and slow RAM and hard disk. The figure below gives a rough overview of a typical system architecture:



System architecture diagram showing caches, ALU (arithmetic logic unit), main memory, and the buses connected each component.

The central CPU chip is connected to the outside world by a number of buses. There is a cache bus, which leads to a block denoted as L2 cache, and there is a system bus as well as a memory bus that leads to the computer main memory. The latter holds the comparatively large RAM while the L2 cache as well as the L1 cache are very small with the latter also being a part of the CPU itself.

The concept of L1 and L2 (and even L3) cache is further illustrated by the following figure, which shows a multi-core CPU and its interplay with L1, L2 and L3 caches:



1. **Level 1 cache** is the fastest and smallest memory type in the cache hierarchy. In most systems, the L1 cache is not very large. Mostly it is in the range of 16 to 64 kBytes, where the memory areas for instructions and data are separated from each other (L1i and L1d, where "i" stands for "instruction" and "d" stands for "data". Also see "[Harvard architecture](#)" for further reference). The importance of the L1 cache grows with increasing speed of the CPU. In the L1 cache, the most frequently required instructions and data are buffered so that as few accesses as possible to the slow main memory are required. This cache avoids delays in data transmission and helps to make optimum use of the CPU's capacity.
2. **Level 2 cache** is located close to the CPU and has a direct connection to it. The information exchange between L2 cache and CPU is managed by the L2 controller on the computer main board. The size of the L2 cache is usually at or below 2 megabytes. On modern multi-core processors, the L2 cache is often located within the CPU itself. The choice between a processor with more clock speed or a larger L2 cache can be answered as follows: With a higher clock speed, individual programs run faster, especially those with high computing requirements. As soon as several programs run simultaneously, a larger cache is advantageous. Usually normal desktop computers with a processor that has a large cache are better served than with a processor that has a high clock rate.
3. **Level 3 cache** is shared among all cores of a multicore processor. With the L3 cache, the [cache coherence](#) protocol of multicore processors can work much faster. This protocol compares the caches of all cores to maintain data consistency so that all processors have access to the same data at the same time. The L3 cache therefore has less the function of a cache, but is intended to simplify and accelerate the cache coherence protocol and the data exchange between the cores. On Mac, information about the system cache can be obtained by executing the command `sysctl -a hw` in a terminal. On Debian Linux linux, this information can be found



with `lscpu | grep cache`. On my iMac Pro (2017), this command yielded (among others) the following output:

```
hw.memsize: 34359738368
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 1048576
hw.l3cachesize: 14417920
```

- *hw.l1icachesize* is the size of the L1 instruction cache, which is at 32kB. This cache is strictly reserved for storing CPU instructions only.
- *hw.l1dcachesize* is also 32 KB and is dedicated for data as opposed to instructions.
- *hw.l2cachesize* and *hw.l3cachesize* show the size of the L2 and L3 cache, which are at 1MB and 14MB respectively.

It should be noted that the size of all caches combined is very small when compared to the size of the main memory (the RAM), which is at 32GB on my system.

Ideally, data needed by the CPU should be read from the various caches for more than 90% of all memory access operations. This way, the high latency of RAM and hard disk can be efficiently compensated.

## Temporal and Spatial Locality

The following table presents a rough overview of the latency of various memory access operations. Even though these numbers will differ significantly between systems, the order of magnitude between the different memory types is noteworthy. While L1 access operations are close to the speed of a photon traveling at light speed for a distance of 1 foot, the latency of L2 access is roughly one order of magnitude slower already while access to main memory is two orders of magnitude slower.

0.5	ns	- CPU L1 dCACHE reference
1	ns	- speed-of-light (a photon) travel a 1 ft (30.5cm) distance
5	ns	- CPU L1 iCACHE Branch mispredict
7	ns	- CPU L2 CACHE reference
71	ns	- CPU cross-QPI/NUMA best case on XEON E5-46*
100	ns	- MUTEX lock/unlock
100	ns	- own DDR MEMORY reference
135	ns	- CPU cross-QPI/NUMA best case on XEON E7-*
202	ns	- CPU cross-QPI/NUMA worst case on XEON E7-*
325	ns	- CPU cross-QPI/NUMA worst case on XEON E5-46*
10,000	ns	- Compress 1K bytes with Zippy PROCESS
20,000	ns	- Send 2K bytes over 1 Gbps NETWORK
250,000	ns	- Read 1 MB sequentially from MEMORY
500,000	ns	- Round trip within a same DataCenter
10,000,000	ns	- DISK seek
10,000,000	ns	- Read 1 MB sequentially from NETWORK
30,000,000	ns	- Read 1 MB sequentially from DISK
150,000,000	ns	- Send a NETWORK packet CA -> Netherlands
		ns
	us	
ms		

Originally from Peter Norvig: <http://norvig.com/21-days.html#answers>

In algorithm design, programmers can exploit two principles to increase runtime performance:

1. **Temporal locality** means that address ranges that are accessed are likely to be used again in the near future. In the course of time, the same memory address is accessed relatively frequently (e.g. in a loop). This property can be used at all levels of the memory hierarchy to keep memory areas accessible as quickly as possible.
2. **Spatial locality** means that after an access to an address range, the next access to an address in the immediate vicinity is highly probable (e.g. in arrays). In the course of time, memory addresses that are very close to each other are accessed again multiple times. This can be exploited by moving the adjacent address areas upwards into the next hierarchy level during a memory access.

Let us consider the following code example:

```
#include <chrono>
#include <iostream>

int main()
{
    // create array
    const int size = 4;
    static int x[size][size];

    auto t1 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            x[j][i] = i + j;
            std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
        }
    }

    // print execution time to console
    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
    std::cout << "Execution time: " << duration << " microseconds" << std::endl;

    return 0;
}
```

<https://youtu.be/Gfrfp1GtIUU>

see attached code

Cache Memory

Guide

## Cache-friendly coding

In the code sample to the right, run the code and note the results. Then please modify the code slightly by interchanging the index `i` and `j` when accessing the variable `x` and take a close look at the resulting runtime performance compared to the original version.

Page 1 of 4

main.cpp

```

1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4000;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[i][j] = i + j;
16            //std::cout << x[i][j] << ", " << i << ", " << j << " << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }

```

root@95bda9474aa: /home/
  
root@95bda9474aa: /home/workspaces

## Cache-friendly coding

In the code sample to the right, run the code and note the results. Then please modify the code slightly by interchanging the index `i` and `j` when accessing the variable `x` and take a close look at the resulting runtime performance compared to the original version.

Cache Memory

Guide

Depending on the machine used for executing the two code versions, there will be a huge difference in execution time. In order to understand why this happens, let us revisit the memory layout we investigated with the gdb debugger at the beginning of this lesson: Even though we have created a two-dimensional array, it is stored in a one-dimensional succession of memory cells. In our minds, the array will (probably) look like this:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

In memory however, it is stored as a single line as follows:

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

As can be seen, the rows of the two-dimensional matrix are copied one after the other. This format is called "row major" and is the default for both C and C++. Some other languages such as Fortran are "column major" and a memory-aware programmer should always know the memory layout of the language he or she is using.

Note that even though the row major memory layout is used in C++, this doesn't mean that all C++ libraries have the same default; for example, the

Page 2 of 4

main.cpp

```

1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4000;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[i][j] = i + j;
16            //std::cout << x[i][j] << ", " << i << ", " << j << " << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }

```

root@95bda9474aa: /home/
  
root@95bda9474aa: /home/workspaces

Depending on the machine used for executing the two code versions, there will be a huge difference in execution time. In order to understand why this happens, let us revisit the memory layout we investigated with the gdb debugger at the beginning of this lesson: Even though we have created a two-dimensional array, it is

stored in a one-dimensional succession of memory cells. In our minds, the array will (probably) look like this:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

In memory however, it is stored as a single line as follows:

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

As can be seen, the rows of the two-dimensional matrix are copied one after the other. This format is called "row major" and is the default for both C and C++. Some other languages such as Fortran are "column major" and a memory-aware programmer should always know the memory layout of the language he or she is using.

Note that even though the row major memory layout is used in C++, this doesn't mean that all C++ libraries have the same default; for example, the

Cache Memory

Guide

As we have created an array of integers, the difference between two adjacent memory cells will be `sizeof(int)`, which is 4 bytes. Let us verify this by changing the size of the array to 4x4 and by plotting both the address and the index numbers to the console. Be sure to revert the array access back to `x[i][j] = i + j`. You can plot by uncommenting the printout line in the inner for loop:

```
0x6021e0: i=0, j=0
0x6021e4: i=0, j=1
0x6021e8: i=0, j=2
0x6021ec: i=0, j=3

0x6021f0: i=1, j=0
0x6021f4: i=1, j=1
0x6021f8: i=1, j=2
0x6021fc: i=1, j=3

0x602200: i=2, j=0
0x602204: i=2, j=1
0x602208: i=2, j=2
0x60220c: i=2, j=3

0x602210: i=3, j=0
0x602214: i=3, j=1
0x602218: i=3, j=2
0x60221c: i=3, j=3

Execution time: 83 microseconds
```

Clearly, the difference between two inner loop cycles is at 4 as predicted.

Page 3 of 4

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4096;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[i][j] = i + j;
16            //std::cout << &x[i][j] << " "; i" << i << ", j" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

root@95bda9474aa: /home/

root@95bda9474aa: /home/workspace

As we have created an array of integers, the difference between two adjacent memory cells will be `sizeof(int)`, which is 4 bytes. Let us verify this by changing the size of the array to 4x4 and by plotting both the address and the index numbers to the console. Be sure to revert the array access back to `x[i][j] = i + j`. You can plot by uncommenting the `printout` line in the inner for loop:

```
0x6021e0: i=0, j=0
0x6021e4: i=0, j=1
0x6021e8: i=0, j=2
0x6021ec: i=0, j=3
```

```
0x6021f0: i=1, j=0
0x6021f4: i=1, j=1
0x6021f8: i=1, j=2
0x6021fc: i=1, j=3
```

```
0x602200: i=2, j=0
0x602204: i=2, j=1
0x602208: i=2, j=2
0x60220c: i=2, j=3
```

```
0x602210: i=3, j=0
0x602214: i=3, j=1
0x602218: i=3, j=2
0x60221c: i=3, j=3
```

Execution time: 83 microseconds

Clearly, the difference between two inner loop cycles is at 4 as predicted.

The screenshot displays a C++ IDE with two panes. The left pane, titled 'Guide', contains a code snippet and its output. The code snippet shows a 4x4 array access pattern where indices `i` and `j` are interchanged. The output shows the memory addresses and the corresponding `i` and `j` values for each access. The execution time is 115 microseconds. The right pane, titled 'Cache Memory', shows the source code of the program. The code defines a 4x4 array `x` and iterates over it, printing the memory address and the values of `i` and `j` for each access. The execution time is 83 microseconds.

When we interchange the indices `i` and `j` when accessing the array as

```
x[j][i] = i + j;
std::cout << &x[j][i] << " : i=" << j << ", j=" << i <<
std::endl;
```

we get the following output:

```
0x6021e0: i=0, j=0
0x6021f0: i=1, j=0
0x602200: i=2, j=0
0x602210: i=3, j=0
0x6021e4: i=0, j=1
0x6021f4: i=1, j=1
0x602204: i=2, j=1
0x602214: i=3, j=1
0x6021e8: i=0, j=2
0x6021f8: i=1, j=2
0x602208: i=2, j=2
0x602218: i=3, j=2
0x6021ec: i=0, j=3
0x6021fc: i=1, j=3
0x60220c: i=2, j=3
0x60221c: i=3, j=3
Execution time: 115 microseconds
```

As can be seen, the difference between two rows is now 0x10, which is 16 in the decimal system. This means that with each access to the matrix, four memory

Page 4 of 4

When we interchange the indices `i` and `j` when accessing the array as

```
x[j][i] = i + j;  
std::cout << &x[j][i] << ": i=" << j << ", j=" << i << std::endl;
```

we get the following output:

```
0x6021e0: i=0, j=0  
0x6021f0: i=1, j=0  
0x602200: i=2, j=0  
0x602210: i=3, j=0
```

```
0x6021e4: i=0, j=1  
0x6021f4: i=1, j=1  
0x602204: i=2, j=1  
0x602214: i=3, j=1
```

```
0x6021e8: i=0, j=2  
0x6021f8: i=1, j=2  
0x602208: i=2, j=2  
0x602218: i=3, j=2
```

```
0x6021ec: i=0, j=3  
0x6021fc: i=1, j=3  
0x60220c: i=2, j=3  
0x60221c: i=3, j=3
```

Execution time: 115 microseconds

As can be seen, the difference between two rows is now 0x10, which is 16 in the decimal system. This means that with each access to the matrix, four memory cells are skipped and the principle of spatial locality is violated. As a result, the wrong data is loaded into the L1 cache, leading to cache misses and costly reload operations - hence the significantly increased execution time between the two code samples. The difference in execution time of both code samples

Outro

<https://youtu.be/izXHpEad7hY>

## 5) Virtual Memory

<https://youtu.be/mY3j60e4ZIs>

### Problems with physical memory

Virtual memory is a very useful concept in computer architecture because it helps with making your software work well given the configuration of the respective hardware on the computer it is running on.

The idea of virtual memory stems back from a (not so long ago) time, when the random access memory (RAM) of most computers was severely limited. Programmers needed to treat memory as a precious resource and use it most efficiently. Also, they wanted to be able to run programs

even if there was not enough RAM available. At the time of writing (August 2019), the amount of RAM is no longer a large concern for most computers and programs usually have enough memory available to them. But in some cases, for example when trying to do video editing or when running multiple large programs at the same time, the RAM memory can be exhausted. In such a case, the computer can slow down drastically.

There are several other memory-related problems, that programmers need to know about:

1. **Holes in address space** : If several programs are started one after the other and then shortly afterwards some of these are terminated again, it must be ensured that the freed-up space in between the remaining programs does not remain unused. If memory becomes too fragmented, it might not be possible to allocate a large block of memory due to a large-enough free contiguous block not being available any more.
2. **Programs writing over each other** : If several programs are allowed to access the same memory address, they will overwrite each others' data at this location. In some cases, this might even lead to one program reading sensitive information (e.g. bank account info) that was written by another program. This problem is of particular concern when writing concurrent programs which run several threads at the same time.

The basic idea of virtual memory is to separate the addresses a program may use from the addresses in physical computer memory. By using a mapping function, an access to (virtual) program memory can be redirected to a real address which is guaranteed to be protected from other programs.

In the following, you will see, how virtual memory solves the problems mentioned above and you will also learn about the concepts of memory pages, frames and mapping. A sound knowledge on virtual memory will help you understand the C++ memory model, which will be introduced in the next lesson of this course.

## Quiz

On a 32-bit machine, each program has its own 32-bit address space. When a program wants to access a memory location, it must specify a 32-bit address, which directs it to the byte stored at this location. On a hardware level, this address is transported to the physical memory via a parallel bus with 32 cables, i.e. each cable can either have the information 'high voltage', and 'low voltage' (or '1' and '0').

### QUIZ QUESTION

How large is the address space on a 32-bit system? What is the upper limit for program memory in GB?

☐ 1 GB

☐ 2 GB

☒ 4 GB

☐ 8 GB

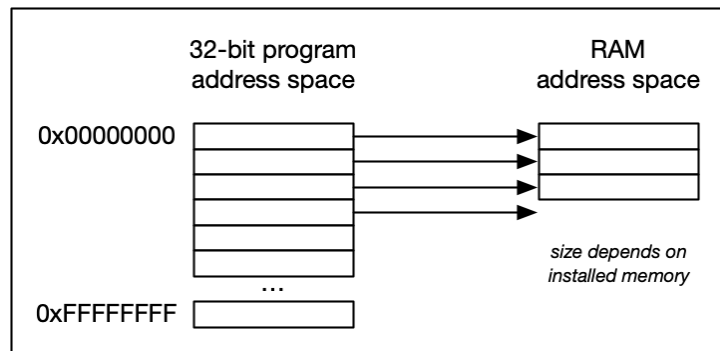
Thanks for completing that!

Correct!  $2^{32}$  bytes = 4GB; a 32-bit address space gives a program a (theoretical) total of 4 GB of memory it can address. In practice, the operating systems reserves some of this space however.

CONTINUE

## Expanding the available memory

As you have just learned in the quiz, the total amount of addressable memory is limited and depends on the architecture of the system (e.g. 32-bit). But what would happen if the available physical memory was below the upper bound imposed by the architecture? The following figure illustrates the problem for such a case:



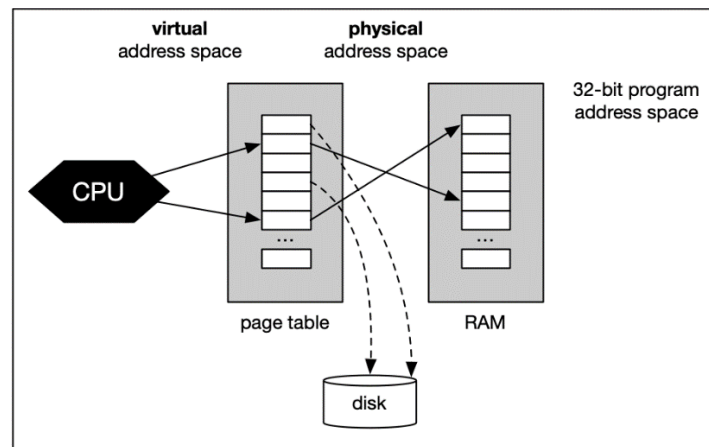
In the image above, the available physical memory is less than the upper bound provided by the 32-bit address space.

On a typical architecture such as MIPS ("Microprocessor without interlocked pipeline stages"), each program is promised to have access to an address space ranging from 0x00000000 up to 0xFFFFFFFF. If however, the available physical memory is only 1GB in size, a 1-on-1 mapping would lead to undefined behavior as soon as the 30-bit RAM address space were exceeded.

With virtual memory however, a mapping is performed between the virtual address space a program sees and the physical addresses of various storage devices such as the RAM but also the hard disk. Mapping makes it possible for the operating system to use physical memory for the parts of a process that are currently being used and back up the rest of the virtual memory to a secondary storage location such as the hard disk. With virtual memory, the size of RAM is not the limit anymore as the system hard disk can be used to store information as well.



The following figure illustrates the principle:



With virtual memory, the RAM acts as a cache for the virtual memory space which resides on secondary storage devices. On Windows systems, the file `pagefile.sys` is such a virtual memory container of varying size. To speed up your system, it makes sense to adjust the system settings in a way that this file is stored on an SSD instead of a slow magnetic hard drive, thus reducing the latency. On a Mac, swap files are stored in `/private/var/vm/`.

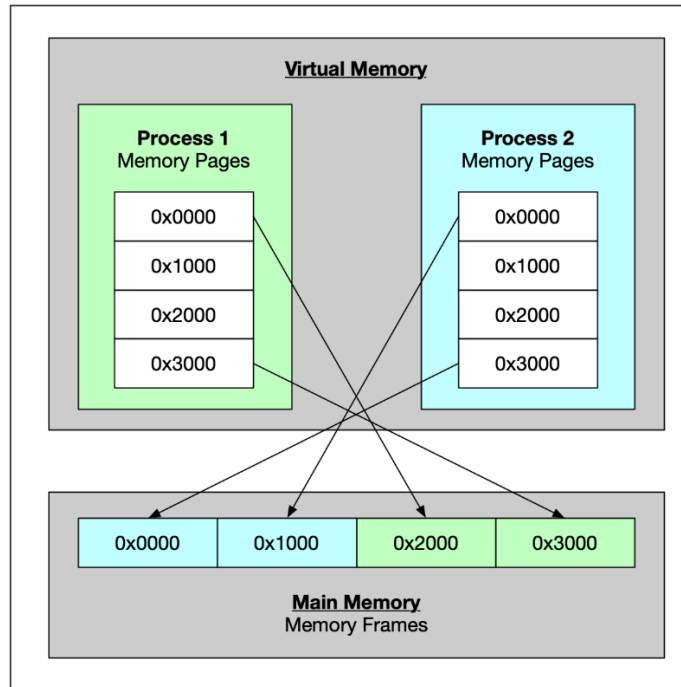
In a nutshell, virtual memory guarantees us a fixed-size address space which is largely independent of the system configuration. Also, the OS guarantees that the virtual address spaces of different programs do not interfere with each other.

The task of mapping addresses and of providing each program with its own virtual address space is performed entirely by the operating system, so from a programmer's perspective, we usually don't have to bother much about memory that is being used by other processes.

Before we take a closer look at an example though, let us define two important terms which are often used in the context of caches and virtual memory:

- A **memory page** is a number of directly successive memory locations in virtual memory defined by the computer architecture and by the operating system. The computer memory is divided into memory pages of equal size. The use of memory pages enables the operating system to perform virtual memory management. The entire working memory is divided into tiles and each address in this computer architecture is interpreted by the Memory Management Unit (MMU) as a logical address and converted into a physical address.
- A **memory frame** is mostly identical to the concept of a memory page with the key difference being its location in the physical main memory instead of the virtual memory.

The following diagram shows two running processes and a collection of memory pages and frames:

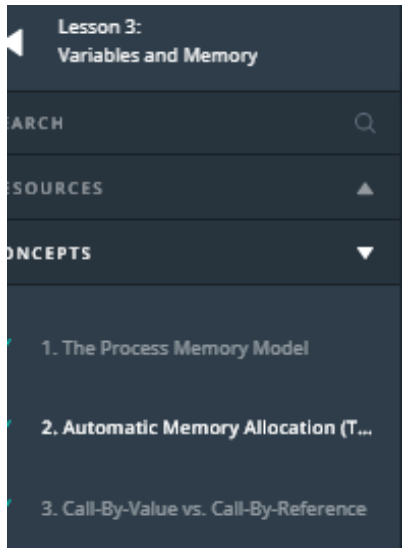


As can be seen, both processes have their own virtual memory space. Some of the pages are mapped to frames in the physical memory and some are not. If process 1 needs to use memory in the memory page that starts at address 0x1000, a page fault will occur if the required data is not there. The memory page will then be mapped to a vacant memory frame in physical memory. Also, note that the virtual memory addresses are not the same as the physical addresses. The first memory page of process 1, which starts at the virtual address 0x0000, is mapped to a memory frame that starts at the physical address 0x2000.

In summary, virtual memory management is performed by the operating system and programmers do usually not interfere with this process. The major benefit is a unique perspective on a chunk of memory for each program that is only limited in its size by the architecture of the system (32 bit, 64 bit) and by the available physical memory, including the hard disk.

## Outro

[https://youtu.be/yaDdy3j\\_IYE](https://youtu.be/yaDdy3j_IYE)

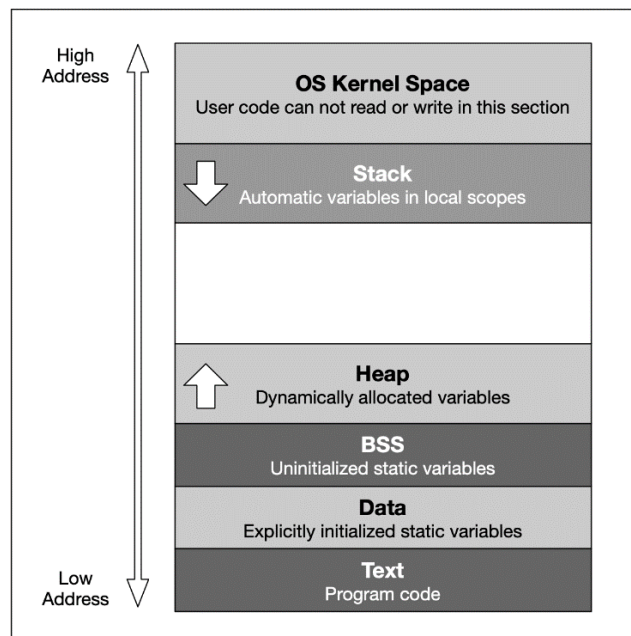


### 1) The Process Memory Model

<https://youtu.be/PlBDxf5ujyo>

## The Process Memory Model

As we have seen in the previous lesson, each program is assigned its own virtual memory by the operating system. This address space is arranged in a linear fashion with one block of data being stored at each address. It is also divided into several distinct areas as illustrated by the figure below:



The last address `0xFFFFFFFF` converts to the decimal `4.294.967.295`, which is the total amount of memory blocks that can theoretically be addressed in a 32-bit operating system - hence the well-known limit of 4GB of memory. On a 64-bit system, the available space is significantly (!) larger. Also, the addresses are stored with 8 bytes instead of 4 bytes.

From a programming perspective though, we are not able to use the entire address space. Instead, the blocks "OS Kernel Space" and "Text" are reserved for the operating system. In kernel space, only the most trusted code is executed - it is fully maintained by the operating system and serves as an interface between the user code and the system kernel. In this course, we will not be directly concerned with this part of memory. The section called 'text' holds the program code generated by the compiler and linker. As with the kernel space, we will not be using this block directly in this course. Let us now take a look at the remaining blocks, starting from the top:

1. The **stack** is a contiguous memory block with a fixed maximum size. If a program exceeds this size, it will crash. The stack is used for storing automatically allocated variables such as local variables or function parameters. If there are multiple threads in a program, then each thread has its own stack memory. New memory on the stack is allocated when the path of execution enters a scope and freed again once the scope is left. It is important to know that the stack is managed "automatically" by the compiler, which means we do not have to concern ourselves with allocation and deallocation.
2. The **heap** (also called "free store" in C++) is where data with dynamic storage lives. It is shared among multiple threads in a program, which means that memory management for the heap needs to take concurrency into account. This makes memory allocations in the heap more complicated than stack allocations. In general, managing memory on the heap is more (computationally) expensive for the operating system, which makes it slower than stack memory. Contrary to the stack, the heap is not managed automatically by the system, but by the programmer. If memory is allocated on the heap, it is the programmer's responsibility to free it again when it is no longer needed. If the programmer manages the heap poorly or not at all, there will be trouble.
3. The **BSS** (Block Started by Symbol) segment is used in many compilers and linkers for a segment that contains global and static variables that are initialized with zero values. This memory area is suitable, for example, for arrays that are not initialized with predefined values.
4. The **Data** segment serves the same purpose as the BSS segment with the major difference being that variables in the Data segment have been initialized with a value other than zero. Memory for variables in the Data segment (and in BSS) is allocated once when a program is run and persists throughout its lifetime.

## Memory Allocation in C++

Now that we have an understanding of the available process memory, let us take a look at memory allocation in C++.

Not every variable in a program has a permanently assigned area of memory. The term **allocate** refers to the process of assigning an area of memory to a variable to store its value. A variable is **deallocated** when the system reclaims the memory from the variable, so it no longer has an area to store its value.

Generally, three basic types of memory allocation are supported:

1. **Static memory allocation** is performed for static and global variables, which are stored in the BSS and Data segment. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. **Automatic memory allocation** is performed for function parameters as well as local variables, which are stored on the stack. Memory for these types of variables is allocated when the path of execution enters a scope and freed again once the scope is left.
3. **Dynamic memory allocation** is a possibility for programs to request memory from the operating system at runtime when needed. This is the major difference to automatic and static allocation, where the size of the variable must be known at compile time. Dynamic memory allocation is not

performed on the limited stack but on the heap and is thus (almost) only limited by the size of the address space.

From a programmer's perspective, stack and heap are the most important areas of program memory. Hence, in the following lessons, let us look at these two in turn.

Outro

<https://youtu.be/lrxMUJWfKhA>

## 2) Automatic Memory Allocation

### Properties of Stack Memory

In the available literature on C++, the terms *stack* and *heap* are used regularly, even though this is not formally correct: C++ has the *free space*, *storage classes* and the *storage duration* of objects.

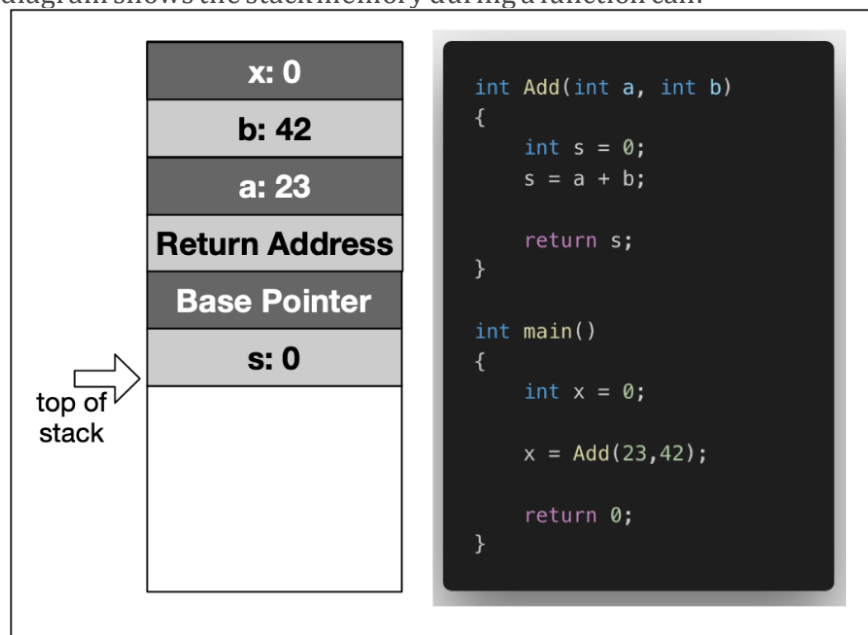
However, since stack and heap are widely used in the C++ community, we will also use it throughout this course. Should you come across the above-mentioned terms in a book or tutorial on the subject, you now know that they refer to the same concepts as stack and heap do.

As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

In the following, a short list of key properties of the stack is listed:

1. The stack is a **contiguous block of memory**. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
2. When the **maximum size of the stack** memory is exceeded, a program will crash.
3. Allocating and deallocating **memory is fast** on the stack. It only involves moving the stack pointer to a new position.

The following diagram shows the stack memory during a function call:



In the example, the variable `x` is created on the stack within the scope of `main`. Then, a stack frame which represents the function `Add` and its variables is pushed to the stack, moving the stack pointer further downwards. It can be seen that this includes the local variables `a` and `b`, as well as the return address, a base pointer and finally the return value `s`.

In the following, let us dig a little more deeply and conduct some experiments with variables on the stack.

## Stack Growth and Contraction

[https://youtu.be/W62TL4\\_NhEs](https://youtu.be/W62TL4_NhEs)

See Attached code

The screenshot shows a web application titled "Automatic Memory Allocation (The Stack)". It has a sidebar with a "Guide" tab and a main content area. The "Guide" tab is active, displaying the title "Stack Growth and Contraction". The text in the guide explains the behavior of the stack when local variables are allocated and a function is called. It mentions that within the `main` function, two local variables `i` and `j` are declared, followed by a call to `MyFunc`, where another local variable is allocated. After `MyFunc` returns, another local variable is allocated in `main`. The program generates the following output:

```
1: 0x7ffefbfff688
2: 0x7ffefbfff684
3: 0x7ffefbfff65c
4: 0x7ffefbfff680
```

Between 1 and 2, the stack address is reduced by 4 bytes, which corresponds to the allocation of memory for the `int j`.

Between 2 and 3, the address pointer is moved by 0x28. We can easily see that calling a function causes a significant amount of memory to be allocated. In addition to the local variable of `MyFunc`, the compiler needs to store additional data such as the return address.

Between 3 and 4, `MyFunc` has returned and a third local variable `k` has been allocated on the stack. The stack pointer now has moved back to a location which is 4 bytes relative to position 2. This means

The main content area displays the source code for `main.cpp`:

```
1 #include <stdio.h>
2
3 void MyFunc()
4 {
5     int k = 3;
6     printf ("3: %p \n", &k);
7 }
8
9 int main()
10 {
11     int i = 1;
12     printf ("1: %p \n", &i);
13
14     int j = 2;
15     printf ("2: %p \n", &j);
16
17     MyFunc();
18
19     int l = 4;
20     printf ("4: %p \n", &l);
21
22     return 0;
23 }
```

The terminal output at the bottom shows the program running on a system with root@2c250c63a7b2: /home/h and root@2c250c63a7b2: /home/workspacell.

## Stack Growth and Contraction

In the first experiment, we will look at the behavior of the stack when local variables are allocated and a function is called. Consider the piece of code on the right.

Within the `main` function, we see two declarations of local variables `i` and `j` followed by a call to `MyFunc`, where another local variable is allocated. After `MyFunc` returns,

another local variable is allocated in `main`. The program generates the following output:

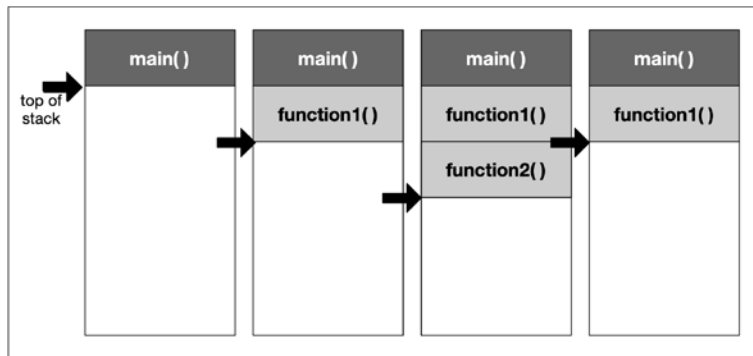
```
1: 0x7ffeefbff688
2: 0x7ffeefbff684
3: 0x7ffeefbff65c
4: 0x7ffeefbff680
```

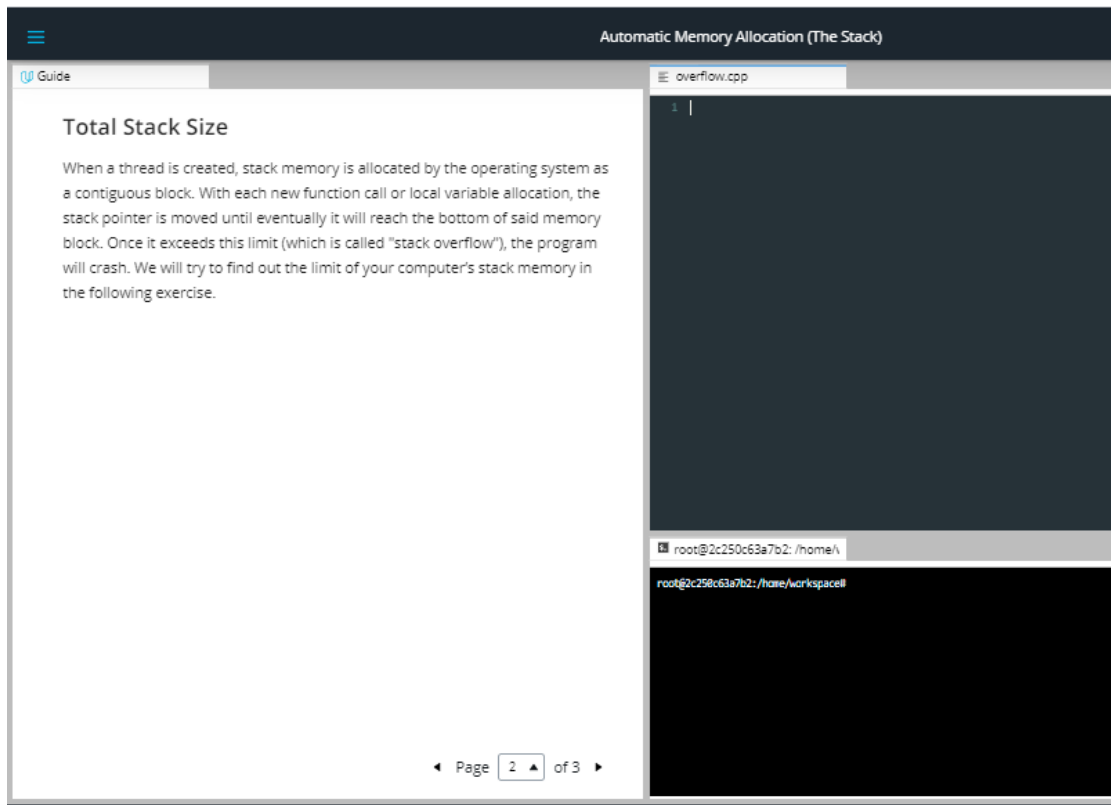
Between 1 and 2, the stack address is reduced by 4 bytes, which corresponds to the allocation of memory for the `int j`.

Between 2 and 3, the address pointer is moved by `0x28`. We can easily see that calling a function causes a significant amount of memory to be allocated. In addition to the local variable of `MyFunc`, the compiler needs to store additional data such as the return address.

Between 3 and 4, `MyFunc` has returned and a third local variable `k` has been allocated on the stack. The stack pointer now has moved back to a location which is 4 bytes relative to position 2. This means that after returning from `MyFunc`, the stack has contracted to the size it had before the function call.

The following diagram illustrates how the stack grows and contracts during program execution:





## Total Stack Size

When a thread is created, stack memory is allocated by the operating system as a contiguous block. With each new function call or local variable allocation, the stack pointer is moved until eventually it will reach the bottom of said memory block. Once it exceeds this limit (which is called "stack overflow"), the program will crash. We will try to find out the limit of your computer's stack memory in the following exercise.



Automatic Memory Allocation (The Stack)

Guide

### Exercise: Create a Stack Overflow

Your task is to create a small program that allocates so much stack memory that an overflow happens. To do this, use a function that allocates some local variable and calls itself recursively. With each new function call, the address of the local variable shall be printed to the console along with the address of a local variable in main which has been allocated before the first function call.

The output of the program should look like this:

```

...
262011: stack bottom : 0x7ffefbff688, current : 0x7ffef400704
262012: stack bottom : 0x7ffefbff688, current : 0x7ffef4006e4
262013: stack bottom : 0x7ffefbff688, current : 0x7ffef4006c4
262014: stack bottom : 0x7ffefbff688, current : 0x7ffef4006a4
262015: stack bottom : 0x7ffefbff688, current : 0x7ffef400684
262016: stack bottom : 0x7ffefbff688, current : 0x7ffef400664

```

The left-most number keeps track of the recursion depth while the difference between the stack bottom and the current position of the stack pointer lets us compute the size of the stack memory which has been used up already. On my MacBook Pro, the size of the stack memory is at 8MB. On Mac or Linux systems, stack size can be checked using the command `ulimit -s`:

```

imac-pro:~ ahaja$ ulimit -s
8192

```

overflow.cpp

```

1 |

```

root@2c250c63a7b2: /home/\

```

root@2c250c63a7b2: /home/workspace#

```

Page 3 of 3

### Exercise: Create a Stack Overflow

Your task is to create a small program that allocates so much stack memory that an overflow happens. To do this, use a function that allocates some local variable and calls itself recursively. With each new function call, the address of the local variable shall be printed to the console along with the address of a local variable in main which has been allocated before the first function call.

The output of the program should look like this:

```

...
262011: stack bottom : 0x7ffefbff688, current: 0x7ffef400704
262012: stack bottom : 0x7ffefbff688, current: 0x7ffef4006e4
262013: stack bottom : 0x7ffefbff688, current: 0x7ffef4006c4
262014: stack bottom : 0x7ffefbff688, current: 0x7ffef4006a4
262015: stack bottom : 0x7ffefbff688, current: 0x7ffef400684
262016: stack bottom : 0x7ffefbff688, current: 0x7ffef400664

```

The left-most number keeps track of the recursion depth while the difference between the stack bottom and the current position of the stack pointer lets us compute the size of the stack memory which has been used up already. On my MacBook Pro, the size of the stack memory is at 8MB. On Mac or Linux systems, stack size can be checked using the command `ulimit -s`:

```

imac-pro:~ ahaja$ ulimit -s
8192

```

On reaching the last line in the above output, the program crashed. As expected, the difference between stack bottom and current stack pointer corresponded to the maximum size of the stack:  $0x7ffef400664 - 0x7ffefbff688 = 0xffffffff800FDC = 8.384.548$  bytes

From this experiment we can draw the simple conclusion that we do not want to run out of stack memory. This can happen quickly though, even on machines with large amounts of RAM installed.

As we have seen, the size of the stack does not benefit from this at all but remains fixed at a very small size.

Outro

<https://youtu.be/gXdpjZiL7m8>

Before we take a look at the heap memory in the next lesson, let us briefly revisit the principles of call-by-value and call-by-reference with regard to stack usage.

### 3) Call-By-Value vs. Call-By-Reference

<https://youtu.be/xBO3kdZTHyc>

When passing parameters to a function in C++, there is a variety of strategies a programmer can choose from. In this section, we will take a look at these in turn from the perspective of stack usage. First, we will briefly revisit the definition of scope, as well as the strategies call-by-value and call-by-reference. Then, we will look at the amount of stack memory used by these methods. See attached code

The screenshot shows a web-based guide titled "Call-By-Value vs. Call-By-Reference". The left sidebar has a "Guide" tab. The main content area is titled "Variable Scopes in C++" and contains the following text:

The time between allocation and deallocation is called the lifetime of a variable. Using a variable after its lifetime has ended is a common programming error, against which most modern languages try to protect: Local variables are only available within their respective scope (e.g. inside a function) and are simply not available outside - so using them inappropriately will result in a compile-time error. When using pointer variables however, programmers must make sure that allocation is handled correctly and that no invalid memory addresses are accessed.

The example to the right shows a set of local (or automatic) variables, whose lifetime is bound to the function they are in.

When `MyLocalFunction` is called, the local variable `isBelowThreshold` is allocated on the stack. When the function exits, it is again deallocated.

For the allocation of local variables, the following holds:

1. Memory is allocated for local variables only after a function has been called. The parameters of a function are also local variables and they are initialized with a value copied from the caller.

At the bottom of the guide, it says "Page 1 of 10".

On the right, there is a code editor titled "scope.cpp" with the following C++ code:

```
1 bool MyLocalFunction(int myInt)
2 {
3     bool isBelowThreshold = myInt < 42 ? true : false;
4     return isBelowThreshold;
5 }
6
7 int main()
8 {
9     bool res = MyLocalFunction(23);
10    return 0;
11 }
```

Below the code editor is a terminal window showing the command prompt:

```
root@29c931534094: /home/
root@29c931534094: /home/workspace#
```

## Variable Scopes in C++

The time between allocation and deallocation is called the **lifetime** of a variable. Using a variable after its lifetime has ended is a common programming error,

against which most modern languages try to protect: Local variables are only available within their respective scope (e.g. inside a function) and are simply not available outside - so using them inappropriately will result in a compile-time error. When using pointer variables however, programmers must make sure that allocation is handled correctly and that no invalid memory addresses are accessed. The example to the right shows a set of local (or automatic) variables, whose lifetime is bound to the function they are in.

When `MyLocalFunction` is called, the local variable `isBelowThreshold` is **allocated** on the stack. When the function exits, it is again **deallocated**.

For the allocation of local variables, the following holds:

1. Memory is allocated for local variables only after a function has been called. The parameters of a function are also local variables and they are initialized with a value copied from the caller.
2. As long as the current thread of execution is within function A, memory for the local variables remains allocated. This even holds true in case another function B is called from within the current function A and the thread of execution moves into this nested function call. However, within function B, the local variables of function A are not known.
3. When the function exits, its locals are deallocated and there is now way to them afterwards - even if the address were still known (e.g. by storing it within a pointer).

Let us briefly revisit the most common ways of passing parameters to a function, which are called *pass-by-reference* and *pass-by-value*.

Quiz : How many local variables?

How many local variables are created within the scope of `MyLocalFunction`?

HIDE SOLUTION

Two variables are created, namely `myInt` and `isBelowThreshold`.

HIDE SOLUTION

Two variables are created, namely `myInt` and `isBelowThreshold`.

Call-By-Value vs. Call-By-Reference

Guide

## Passing Variables by Value

When calling a function as in the previous code example, its parameters (in this case `myInt`) are used to create local copies of the information provided by the caller. The caller is not sharing the parameter with the function but instead a proprietary copy is created using the assignment operator `=` (more about that later). When passing parameters in such a way, it is ensured that changes made to the local copy will not affect the original on the caller side. The upside to this is that inner workings of the function and the data owned by the caller are kept neatly separate.

However, there are two major downsides to this:

1. Passing parameters by value means that a copy is created, which is an expensive operation that might consume large amounts of memory, depending on the data that is being transferred. Later in this course we will encounter "move semantics", which is an effective way to compensate for this downside.
2. Passing by value also means that the created copy can not be used as a back channel for communicating with the caller, for example by directly writing the desired information into the variable.

Consider the example on the right in the `pass_by_value.cpp` file. In main, the integer `val` is initialized with 0. When passing it to the function `AddTwo`, a local copy of `val` is created, which

pass\_by\_value.cpp

```
1 #include <iostream>
2
3 void AddTwo(int val)
4 {
5     val += 2;
6 }
7
8 int main()
9 {
10     int val = 0;
11     AddTwo(val);
12     val += 2;
13     std::cout << "val = " << val << std::endl;
14
15     return 0;
16 }
```

root@29c931534094: /home/

root@29c931534094: /home/workspace#

Page 2 of 10

## Passing Variables by Value

When calling a function as in the previous code example, its parameters (in this case `myInt`) are used to create local copies of the information provided by the caller. The caller is not sharing the parameter with the function but instead a proprietary copy is created using the assignment operator `=` (more about that later). When passing parameters in such a way, it is ensured that changes made to the local copy will not affect the original on the caller side. The upside to this is that inner workings of the function and the data owned by the caller are kept neatly separate. However, there are two major downsides to this:

1. Passing parameters by value means that a copy is created, which is an expensive operation that might consume large amounts of memory, depending on the data that is being transferred. Later in this course we will encounter "move semantics", which is an effective way to compensate for this downside.

2. Passing by value also means that the created copy can not be used as a back channel for communicating with the caller, for example by directly writing the desired information into the variable.

Consider the example on the right in the `pass_by_value.cpp` file. In `main`, the integer `val` is initialized with 0. When passing it to the function `AddTwo`, a local copy of `val` is created, which only exists within the scope of `AddTwo`, so the add-operation has no effect on `val` on the caller side. So when `main` returns, `val` has a value of 2 instead of 4.

The screenshot shows a code editor with two panes. The left pane, titled 'Guide', contains text explaining how to create a backchannel to the caller side by passing a pointer variable. The right pane, titled 'pass\_by\_pointer.cpp', contains the following C++ code:

```
1 #include <iostream>
2
3 void AddThree(int *val)
4 {
5     *val += 3;
6 }
7
8 int main()
9 {
10     int val = 0;
11     AddThree(&val);
12     val += 2;
13
14     std::cout << "val = " << val << std::endl;
15
16     return 0;
17 }
```

Below the code editor, a terminal window shows the command prompt: `root@29c931534094:/home/workspace#`. At the bottom of the left pane, it says 'Page 3 of 10'.

However, with a slight modification, we can easily create a backchannel to the caller side. Consider the code on the right.

In this case, when passing the parameter to the function `AddThree`, we are creating a local copy as well but note that we are now passing a pointer variable. This means that a copy of the memory address of `val` is created, which we can then use to directly modify its content by using the dereference operator `*`.

Call-By-Value vs. Call-By-Reference

Guide

## Passing Variables by Reference

The second major way of passing parameters to a function is by reference. With this way, the function receives a reference to the parameter, rather than a copy of its value. As with the example of `AddThree` above, the function can now modify the argument such that the changes also happen on the caller side. In addition to the possibility to directly exchange information between function and caller, passing variables by reference is also faster as no information needs to be copied, as well as more memory-efficient.

A major disadvantage is that the caller does not always know what will happen to the data it passes to a function (especially when the function code can not be modified easily). Thus, in some cases, special steps must be taken to protect ones data from inappropriate modification.

Let us now look at an example of passing a variable by reference, shown in the code on the right.

pass\_by\_reference.cpp

```
1 #include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 int main()
9 {
10     int val = 0;
11     AddFour(val);
12     val += 2;
13
14     std::cout << "val = " << val << std::endl;
15
16     return 0;
17 }
```

root@29c931534094: /home/h

root@29c931534094: /home/workspace#

◀ Page 4 of 10 ▶

## Passing Variables by Reference

The second major way of passing parameters to a function is by reference. With this way, the function receives a reference to the parameter, rather than a copy of its value. As with the example of `AddThree` above, the function can now modify the argument such that the changes also happen on the caller side. In addition to the possibility to directly exchange information between function and caller, passing variables by reference is also faster as no information needs to be copied, as well as more memory-efficient.

A major disadvantage is that the caller does not always know what will happen to the data it passes to a function (especially when the function code can not be modified easily). Thus, in some cases, special steps must be taken to protect ones data from inappropriate modification.

Let us now look at an example of passing a variable by reference, shown in the code on the right.

Guide

To pass a variable by reference, we simply declare the function parameters as references using `&` rather than as normal variables. When the function is called, `val` will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

Page 5 of 10

Call-By-Value vs. Call-By-Reference

pass\_by\_reference.cpp

```
1 |include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 int main()
9 {
10     int val = 0;
11     AddFour(val);
12     val += 2;
13
14     std::cout << "val = " << val << std::endl;
15
16     return 0;
17 }
```

root@29c931534094: /home/

root@29c931534094: /home/workspace#

To pass a variable by reference, we simply declare the function parameters as references using `&` rather than as normal variables. When the function is called, `val` will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

Guide

### Quiz : Modifying several parameters

An additional advantage of passing variables by reference is the possibility to modify several variables. When using the function return value for such a purpose, returning several variables is usually very cumbersome.

Your task here is to create a function `AddFive` that modifies the `int` input variable by adding 5 and modifies the `bool` input variable to be `true`. In the code to the right you will find the function call in `main()`.

HIDE SOLUTION

```
void AddFive(int &val, bool &success)
{
    val += 5;
    success = true;
}
```

◀ Page 6 of 10 ▶

quiz.cpp

```
1 #include <iostream>
2
3 int main()
4 {
5     int val = 0;
6     bool success = false;
7     AddFive(val, success);
8     val += 2;
9
10    std::cout << "val = " << val << ", success = " << success << std::endl;
11
12    return 0;
13 }
```

root@29c931534094: /home/

root@29c931534094: /home/workspace#

### Quiz : Modifying several parameters

An additional advantage of passing variables by reference is the possibility to modify several variables. When using the function return value for such a purpose, returning several variables is usually very cumbersome.

Your task here is to create a function `AddFive` that modifies the `int` input variable by adding 5 and modifies the `bool` input variable to be `true`. In the code to the right you will find the function call in `main()`.

HIDE SOLUTION

```
void AddFive(int &val, bool &success)
{
    val += 5;
    success = true;
}
```



Guide

## Pointers vs. References

As we have seen in the examples above, the use of pointers and references to directly manipulate function arguments in a memory-effective way is very similar. Let us compare the two methods in the code on the right.

As can be seen, pointer and reference are both implemented by using a memory address. In the case of `AddFour` the caller does not even realize that `val` might be modified while with `AddSix`, the reference to `val` has to be explicitly written by using `&`.

If passing by value needs to be avoided, both pointers and references are a way to achieve this. The following selection of properties contrasts the two methods so it will be easier to decide which to use from the perspective of the use-case at hand:

- Pointers can be declared without initialization. This means we can pass an uninitialized pointer to a function who then internally performs the initialization for us.
- Pointers can be reassigned to another memory block on the heap.
- References are usually easier to use (depending on the expertise level of the programmer). Sometimes however, if a third-party function is used without properly looking at the parameter definition, it might go unnoticed that a value has been modified.

◀ Page 7 of 10 ▶

pointers\_v\_references.cpp

```
1 #include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 void AddSix(int *val)
9 {
10    *val += 6;
11 }
12
13 int main()
14 {
15     int val = 0;
16     AddFour(val);
17     AddSix(&val);
18
19     std::cout << "val = " << val << std::endl;
20
21     return 0;
22 }
```

root@29c931534094: /home/

root@29c931534094: /home/workspace#

## Pointers vs. References

As we have seen in the examples above, the use of pointers and references to directly manipulate function arguments in a memory-effective way is very similar. Let us compare the two methods in the code on the right.

As can be seen, pointer and reference are both implemented by using a memory address. In the case of `AddFour` the caller does not even realize that `val` might be modified while with `AddSix`, the reference to `val` has to be explicitly written by using `&`. If passing by value needs to be avoided, both pointers and references are a way to achieve this. The following selection of properties contrasts the two methods so it will be easier to decide which to use from the perspective of the use-case at hand:

- Pointers can be declared without initialization. This means we can pass an uninitialized pointer to a function who then internally performs the initialization for us.
- Pointers can be reassigned to another memory block on the heap.
- References are usually easier to use (depending on the expertise level of the programmer). Sometimes however, if a third-party function is used without properly looking at the parameter definition, it might go unnoticed that a value has been modified.



Now, we will compare the three strategies we have seen so far with regard to stack memory usage. Consider the code on the right.

After creating a local variable `i` in `main` to give us the address of the stack bottom, we are passing `i` by-value to our first function. Inside `CallByValue`, the memory address of a local variable `j` is printed to the console, which serves as a marker for the stack pointer. With the second function call in `main`, we are passing a reference to `i` to `CallByPointer`. Lastly, the function `CallByReference` is called in `main`, which again takes the integer `i` as an argument. However, from looking at `main` alone, we can not tell whether `i` will be passed by value or by reference.

On my machine, when compiled with `g++` (Apple clang version 11.0.0), the program produces the following output:

```
stack bottom: 0x7ffefbfff698
call-by-value: 0x7ffefbfff678
call-by-pointer: 0x7ffefbfff674
call-by-reference: 0x7ffefbfff674
```

Depending on your system, the compiler you use and the compiler optimization techniques, you may not always see this result. In some cases

Let us take a look at the respective differences to the stack bottom in turn:

1. `CallByValue` requires 32 bytes of memory. As discussed before, this is reserved for e.g. the function return address and for the local variables within the function (including the copy of `i`).
2. `CallByPointer` on the other hand requires - perhaps surprisingly - 36 bytes of memory. Let us complete the examination before going into more details on this result.
3. `CallByReference` finally has the same memory requirements as `CallByPointer`.

### Quiz: Why does CallByValue require more memory?

In this section, we have argued at length that passing a parameter by reference avoids a costly copy and should - in many situations - be preferred over passing a parameter by value. Yet, in the experiment above, we have witnessed the exact opposite.

Can you explain why?

**HIDE SOLUTION**

Let us take a look at the size of the various parameter types using the `sizeof` command:

```
printf("size of int: %lu\n", sizeof(int));
printf("size of *int: %lu\n", sizeof(int *));
```

The output here is

```
size of int: 4
size of *int: 8
```

Obviously, the size of the pointer variable is larger than the actual data type. As my machine has a 64 bit architecture, an address

```
1 #include <stdio.h>
2
3 void CallByValue(int i)
4 {
5     int j = 1;
6     printf ("call-by-value: %p\n",&j);
7 }
8
9 void CallByPointer(int *i)
10 {
11     int j = 1;
12     printf ("call-by-pointer: %p\n",&j);
13 }
14
15 void CallByReference(int &i)
16 {
17     int j = 1;
18     printf ("call-by-reference: %p\n",&j);
19 }
20
21 int main()
22 {
23     int i = 0;
24     printf ("stack bottom: %p\n",&i);
25
26     CallByValue(i);
27
28     CallByPointer(&i);
```

root@29c931534094: /home/

root@29c931534094: /home/workspace#

Page 10 of 10

## Quiz: Why does CallByValue require more memory?

In this section, we have argued at length that passing a parameter by reference avoids a costly copy and should - in many situations - be preferred over passing a parameter by value. Yet, in the experiment above, we have witnessed the exact opposite.

Can you explain why?

HIDE SOLUTION

Let us take a look at the size of the various parameter types using the `sizeof` command:

```
printf("size of int: %lu\n", sizeof(int));  
printf("size of *int: %lu\n", sizeof(int*));
```

The output here is

```
size of int: 4  
size of *int: 8
```

Obviously, the size of the pointer variable is larger than the actual data type. As my machine has a 64 bit architecture, an address requires 8 byte.

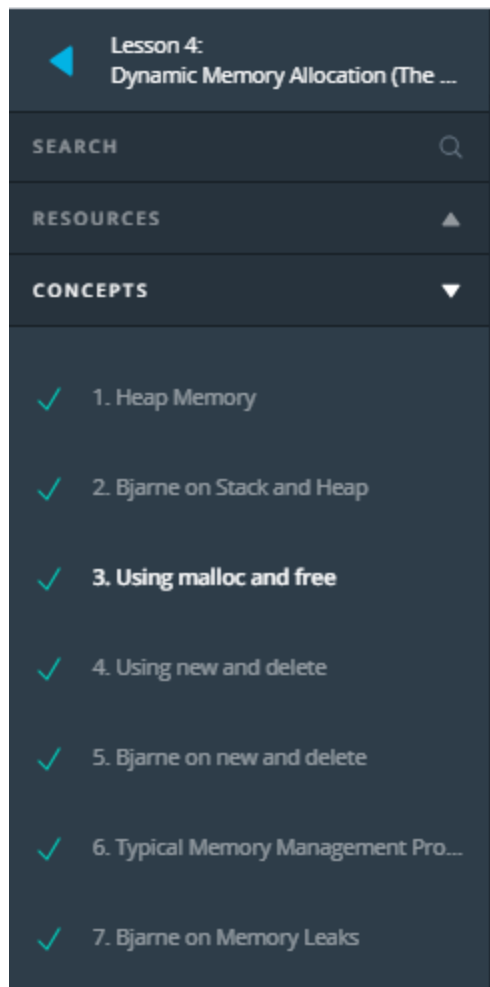
As an experiment, you could use the `-m32` compiler flag to build a 32 bit version of the program. This yields the following output:

```
size of int: 4  
size of *int: 4
```

In order to benefit from call-by-reference, the size of the data type passed to the function has to surpass the size of the pointer on the respective architecture (i.e. 32 bit or 64 bit).

Outro

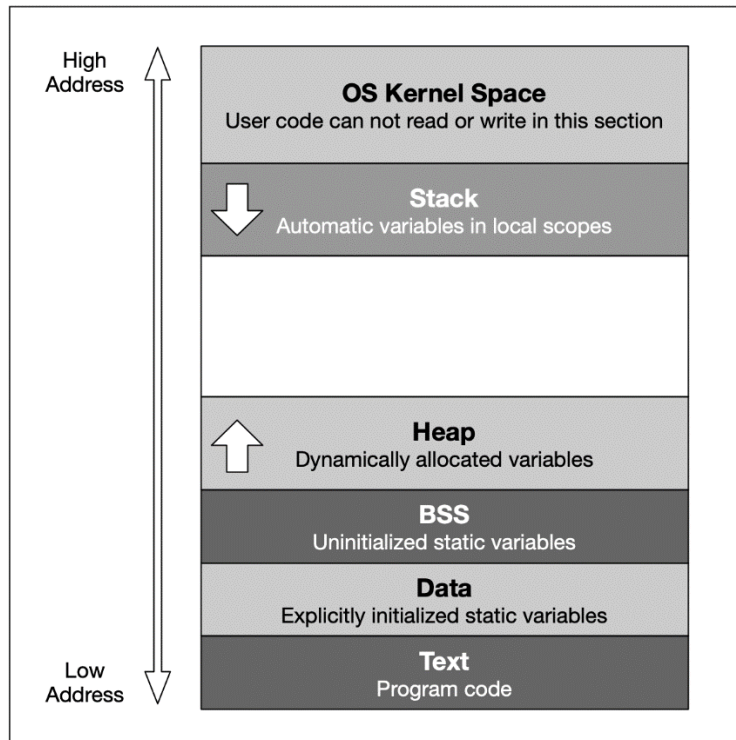
<https://youtu.be/S4NSWgZyvT4>



## 1) Heap Memory

<https://youtu.be/t8i6V0l-Awg>

Heap memory, also known as dynamic memory, is an important resource available to programs (and programmers) to store data. The following diagram again shows the layout of virtual memory with the heap being right above the BSS and Data segment.



As mentioned earlier, the heap memory grows upwards while the stack grows in the opposite direction. We have seen in the last lesson that the automatic stack memory shrinks and grows with each function call and local variable. As soon as the scope of a variable is left, it is automatically deallocated and the stack pointer is shifted upwards accordingly.

Heap memory is different in many ways: The programmer can request the allocation of memory by issuing a command such as `malloc` or `new` (more on that shortly). This block of memory will remain allocated until the programmer explicitly issues a command such as `free` or `delete`. The huge advantage of heap memory is the high degree of control a programmer can exert, albeit at the price of greater responsibility since memory on the heap must be actively managed.

Let us take a look at some properties of heap memory:

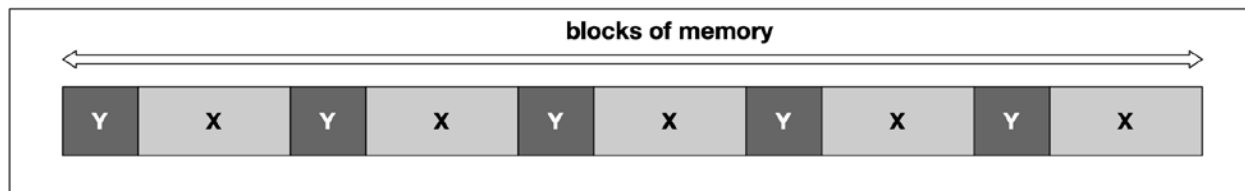
1. As opposed to local variables on the stack, memory can now be allocated in an arbitrary scope (e.g. inside a function) without it being deleted when the scope is left. Thus, as long as the address to an allocated block of memory is returned by a function, the caller can freely use it.
2. Local variables on the stack are allocated at compile-time. Thus, the size of e.g. a string variable might not be appropriate as the length of the string will not be known until the program is executed and the user inputs it. With local variables, a solution would be to allocate a long-enough array of and hope that the actual length does not exceed the buffer size. With dynamically allocated heap memory, variables are allocated at run-time. This means that the size of the above-mentioned string variable can be tailored to the actual length of the user input.
3. Heap memory is only constrained by the size of the address space and by the available memory. With modern 64 bit operating systems and large RAM memory and hard disks the

programmer commands a vast amount of memory. However, if the programmer forgets to deallocate a block of heap memory, it will remain unused until the program is terminated. This is called a "memory leak".

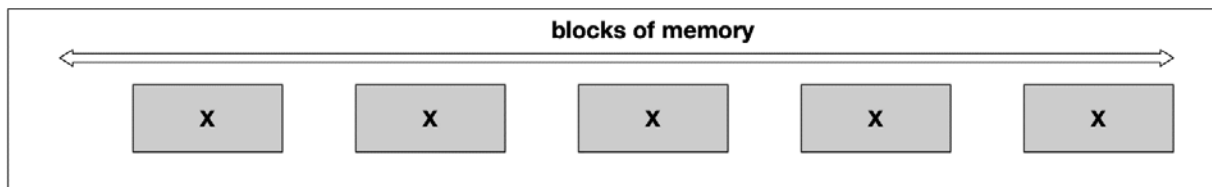
4. Unlike the stack, the heap is shared among multiple threads, which means that memory management for the heap needs to take concurrency into account as several threads might compete for the same memory resource.
5. When memory is allocated or deallocated on the stack, the stack pointer is simply shifted upwards or downwards. Due to the sequential structure of stack memory management, stack memory can be managed (by the operating system) easily and securely. With heap memory, allocation and deallocation can occur arbitrarily, depending on the lifetime of the variables. This can result in fragmented memory over time, which is much more difficult and expensive to manage.

## Memory Fragmentation

Let us construct a theoretic example of how memory on the heap can become fragmented: Suppose we are interleaving the allocation of two data types **X** and **Y** in the following fashion: First, we allocate a block of memory for a variable of type **X**, then another block for **Y** and so on in a repeated manner until some upper bound is reached. At the end of this operation, the heap might look like the following:



At some point, we might then decide to deallocate all variables of type **Y**, leading to empty spaces in between the remaining variables of type **X**. In between two blocks of type "X", no memory for an additional "X" could now be squeezed in this example.



A classic symptom of memory fragmentation is that you try to allocate a large block and you can't, even though you appear to have enough memory free. On systems with virtual memory however, this is less of a problem, because large allocations only need to be contiguous in virtual address space, not in physical address space.

When memory is heavily fragmented however, memory allocations will likely take longer because the memory allocator has to do more work to find a suitable space for the new object.

Until now, our examples have been only theoretical. It is time to gain some practical experience in the next section using `malloc` and `free` as C-style methods for dynamic memory management.

## Outro

<https://youtu.be/0k2CqV063zw>


### 2) Bjarne on Stack and Heap


<https://youtu.be/wsdf7Dz4ykk>

### 3) Using malloc and free

<https://youtu.be/nawlbRI6xzo>

So far we only considered primitive data types, whose storage space requirement was already fixed at compile time and could be scheduled with the building of the program executable. However, it is not always possible to plan the memory requirements exactly in advance, and it is inefficient to reserve the maximum memory space each time just to be on the safe side. C and C++ offer the option to reserve memory areas during the program execution, i.e. at runtime. It is important that the reserved memory areas are released again at the "appropriate point" to avoid memory leaks. It is one of the major challenges in memory management to always locate this "appropriate point" though.

 Using malloc and free SEND FEEDBACK

 Guide

### Allocating Dynamic Memory

To allocate dynamic memory on the heap means to make a contiguous memory area accessible to the program at runtime and to mark this memory as occupied so that no one else can write there by mistake.

To reserve memory on the heap, one of the two functions `malloc` (stands for *Memory Allocation*) or `calloc` (stands for *Cleared Memory Allocation*) is used. The header file `stdlib.h` or `malloc.h` must be included to use the functions.



Here is the syntax of `malloc` and `calloc` in C/C++:

```
pointer_name = (cast-type*) malloc(size);
pointer_name = (cast-type*) calloc(num_elems, size_elem);
```

`malloc` is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form.

`calloc` is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

Both functions return a pointer of type `void` which can be cast into a pointer of any form. If the space for the allocation is insufficient, a NULL pointer is returned.

Page 1 of 8



## Allocating Dynamic Memory

To allocate dynamic memory on the heap means to make a contiguous memory area accessible to the program at runtime and to mark this memory as occupied so that no one else can write there by mistake.

To reserve memory on the heap, one of the two functions `malloc` (stands for *Memory Allocation*) or `calloc` (stands for *Cleared Memory Allocation*) is used. The header file `stdlib.h` or `malloc.h` must be included to use the functions.

Here is the syntax of `malloc` and `calloc` in C/C++:

```
pointer_name = (cast-type*) malloc(size);  
pointer_name = (cast-type*) calloc(num_elems, size_elem);
```

`malloc` is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form.

`calloc` is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

Both functions return a pointer of type `void` which can be cast into a pointer of any form. If the space for the allocation is insufficient, a `NULL` pointer is returned.

### Using malloc and free

Guide

In the code example on the right, a block of memory the size of an integer is allocated using `malloc`.

The `sizeof` command is a convenient way of specifying the amount of memory (in bytes) needed to store a certain data type. For an `int`, `sizeof` returns 4. However, when compiling this code, the following warning is generated on my machine:

```
warning: ISO C++ does not allow indirection on operand of type 'void *' [-Wvoid-ptr-dereference]  
printf("address=%p, value=%d", p, *p);
```

In the virtual workspace, when compiling with `g++`, an error is thrown instead of a warning.

The problem with `void` pointers is that there is no way of knowing the offset to the end of the allocated memory block. For an `int`, this would be 4 bytes but for a `double`, the offset would be 8 bytes. So in order to retrieve the entire block of memory that has been reserved, we need to know the data type and the way to achieve this with `malloc` is by casting the return pointer:

```
int *p = (int*)malloc(sizeof(int));
```

This code now produces the following output without compiler

malloc\_example.cpp

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     void *p = malloc(sizeof(int));  
7     printf("address=%p, value=%d\n", p, *p);  
8  
9     return 0;  
10 }
```

root@4e1b694fa58f: /home/v

root@4e1b694fa58f: /home/workspace#

Page 2 of 8

In the code example on the right, a block of memory the size of an integer is allocated using malloc. The sizeof command is a convenient way of specifying the amount of memory (in bytes) needed to store a certain data type. For an int, sizeof returns 4. However, when compiling this code, the following warning is generated on my machine:

warning: ISO C++ does not allow indirection on operand of type 'void \*' [-Wvoid-ptr-dereference]

```
printf("address=%p, value=%d", p, *p);
```

In the virtual workspace, when compiling with g++, an error is thrown instead of a warning.

The problem with void pointers is that there is no way of knowing the offset to the end of the allocated memory block. For an int, this would be 4 bytes but for a double, the offset would be 8 bytes. So in order to retrieve the entire block of memory that has been reserved, we need to know the data type and the way to achieve this with malloc is by casting the return pointer:

```
int *p = (int*)malloc(sizeof(int));
```

This code now produces the following output without compiler warnings: address=0x1003001f0, value=0

Obviously, the memory has been initialized with 0 in this case. However, you should not rely on pre-initialization as this depends on the data type as well as on the compiler you are using.

At compile time, only the space for the pointer is reserved (on the stack). When the pointer is initialized, a block of memory of sizeof(int) bytes is allocated (on the heap) at program runtime. The pointer on the stack then points to this memory location on the heap.

The screenshot shows a web-based coding guide interface. On the left, a 'Quiz' section asks the user to 'Modify the example in a way that memory for 3 integers is reserved.' Below the question is a 'HIDE SOLUTION' button and a code block showing the solution: 

```
// reserve memory for several integers
int *p2 = (int*)malloc(3*sizeof(int));
printf("address=%p, value=%d\n", p2, *p2);
```

 On the right, a code editor titled 'malloc\_example.cpp' shows the original code: 

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     void *p = malloc(sizeof(int));
7     printf("address=%p, value=%d\n", p, *p);
8
9     return 0;
10 }
```

 At the bottom, a terminal window shows the command prompt 'root@4e1b694fa58f: /home/workspace#'. A page indicator at the bottom center shows 'Page 3 of 8'.

## Quiz

Modify the example in a way that memory for 3 integers is reserved.

HIDE SOLUTION

```
// reserve memory for several integers
int *p2 = (int*)malloc(3*sizeof(int));
printf("address=%p, value=%d\n", p2, *p2);
```

The screenshot shows a web-based IDE interface. On the left, a 'Guide' tab is active, displaying a page titled 'Memory for Arrays and Structs'. The page explains that arrays and pointers are processed identically internally and provides an example of allocating memory for an array of integers using `malloc`. It also introduces structures and shows how to allocate memory for an array of structures using `calloc`. On the right, a code editor tab titled 'malloc\_example.cpp' contains a C++ program that includes `<stdio.h>` and `<stdlib.h>`, and uses `malloc` to allocate memory for a single integer, printing its address and value. Below the code editor, a terminal window shows the command prompt `root@4e1b694fa58f: /home/workspace#`. The IDE interface includes a navigation bar at the top with a menu icon and the title 'Using malloc and free'.

Using malloc and free

Guide

### Memory for Arrays and Structs

Since arrays and pointers are displayed and processed identically internally, individual blocks of data can also be accessed using array syntax:

```
int *p = (int*)malloc(3*sizeof(int));
p[0] = 1; p[1] = 2; p[2] = 3;
printf("address=%p, second value=%d\n", p, p[1]);
```

Until now, we have only allocated memory for a C/C++ data primitive (i.e. `int`). However, we can also define a proprietary structure which consists of several primitive data types and use `malloc` or `calloc` in the same manner as before:

```
struct MyStruct {
    int i;
    double d;
    char a[5];
};

MyStruct *p = (MyStruct*)calloc(4, sizeof(MyStruct));
p[0].i = 1; p[0].d = 3.14159; p[0].a[0] = 'a';
```

After defining the struct `MyStruct` which contains a number of data primitives, a block of memory four times the size of

Page 4 of 8

malloc\_example.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     void *p = malloc(sizeof(int));
7     printf("address=%p, value=%d\n", p, *p);
8
9     return 0;
10 }
```

root@4e1b694fa58f: /home/v

root@4e1b694fa58f: /home/workspace#

## Memory for Arrays and Structs

Since arrays and pointers are displayed and processed identically internally, individual blocks of data can also be accessed using array syntax:

```
int *p = (int*)malloc(3*sizeof(int));
p[0] = 1; p[1] = 2; p[2] = 3;
```

```
printf("address=%p, second value=%d\n", p, p[1]);
```

Until now, we have only allocated memory for a C/C++ data primitive (i.e. `int`). However, we can also define a proprietary structure which consists of several primitive data types and use `malloc` or `calloc` in the same manner as before:

```
struct MyStruct {  
    int i;  
    double d;  
    char a[5];  
};
```

```
MyStruct *p = (MyStruct*)calloc(4, sizeof(MyStruct));  
p[0].i = 1; p[0].d = 3.14159; p[0].a[0] = 'a';
```


After defining the struct `MyStruct` which contains a number of data primitives, a block of memory four times the size of `MyStruct` is created using the `calloc` command. As can be seen, the various data elements can be accessed very conveniently.

Using malloc and free

SEND FEEDBACK

Guide

### Changing the Size of Memory Blocks



◀ Page 5 of 8 ▶

[https://video.udacity-data.com/topher/2019/September/5d855a6d\\_nd213-c03-l03-02.2-using-malloc-and-free-sc/nd213-c03-l03-02.2-using-malloc-and-free-sc\\_720p.mp4](https://video.udacity-data.com/topher/2019/September/5d855a6d_nd213-c03-l03-02.2-using-malloc-and-free-sc/nd213-c03-l03-02.2-using-malloc-and-free-sc_720p.mp4)

Using malloc and free

Guide

The size of the memory area reserved with `malloc` or `calloc` can be increased or decreased with the `realloc` function.

```
pointer_name = (cast-type*) realloc( (cast-type*)old_memblock, new_size );
```

To do this, the function must be given a pointer to the previous memory area and the new size in bytes. Depending on the compiler, the reserved memory area is either (a) expanded or reduced internally (if there is still enough free heap after the previously reserved memory area) or (b) a new memory area is reserved in the desired size and the old memory area is released afterwards.

The data from the old memory area is retained, i.e. if the new memory area is larger, the data will be available within new memory area as well. If the new memory area is smaller, the data from the old area will be available only up until the site of the new area - the rest is lost.

In the example on the right, a block of memory of initially 8 bytes (two integers) is resized to 16 bytes (four integers) using `realloc`.

Note that `realloc` has been used to increase the memory size and then decrease it immediately after assigning the values 3 and 4 to

realloc\_example.cpp

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     // reserve memory for two integers
7     int *p = (int*)malloc(2*sizeof(int));
8     p[0] = 1; p[1] = 2;
9
10    // resize memory to hold four integers
11    p = (int*)realloc(p,4*sizeof(int));
12    p[2] = 3; p[3] = 4;
13
14    // resize memory again to hold two integers
15    p = (int*)realloc(p,2*sizeof(int));
16
17    printf("address=%p, value=%d\n", p+0, *(p+0)); // valid
18    printf("address=%p, value=%d\n", p+1, *(p+1)); // valid
19
20    printf("address=%p, value=%d\n", p+2, *(p+2)); // INVALID
21    printf("address=%p, value=%d\n", p+3, *(p+3)); // INVALID
22
23    return 0;
24 }

```

root@4e1b694fa58f: /home/v

root@4e1b694fa58f: /home/workspace#

Page 6 of 8

The size of the memory area reserved with `malloc` or `calloc` can be increased or decreased with the `realloc` function.

```
pointer_name = (cast-type*) realloc( (cast-type*)old_memblock, new_size );
```

To do this, the function must be given a pointer to the previous memory area and the new size in bytes. Depending on the compiler, the reserved memory area is either (a) expanded or reduced internally (if there is still enough free heap after the previously reserved memory area) or (b) a new memory area is reserved in the desired size and the old memory area is released afterwards.

The data from the old memory area is retained, i.e. if the new memory area is larger, the data will be available within new memory area as well. If the new memory area is smaller, the data from the old area will be available only up until the site of the new area - the rest is lost.

In the example on the right, a block of memory of initially 8 bytes (two integers) is resized to 16 bytes (four integers) using `realloc`.

Note that `realloc` has been used to increase the memory size and then decrease it immediately after assigning the values 3 and 4 to the new blocks. The output looks like the following:

address=0x100300060, value=1  
address=0x100300064, value=2  
address=0x100300068, value=3  
address=0x10030006c, value=4

Interestingly, the pointers `p+2` and `p+3` can still access the memory location they point to. Also, the original data (numbers 3 and 4) is still there. So `realloc` will not erase memory but merely mark it as "available" for future allocations. It should be noted however that accessing a memory location *after* such an operation must be avoided as it could cause a segmentation fault. We will encounter segmentation faults soon when we discuss "dangling pointers" in one of the next lessons.

Using malloc and free

Guide

## Freeing up Memory

If memory has been reserved, it should also be released as soon as it is no longer needed. If memory is reserved regularly without releasing it again, the memory capacity may be exhausted at some point. If the RAM memory is completely used up, the data is swapped out to the hard disk, which slows down the computer significantly.

The `free` function releases the reserved memory area so that it can be used again or made available to other programs. To do this, the pointer pointing to the memory area to be freed is specified as a parameter for the function. In the `free_example.cpp`, a memory area is reserved and immediately released again.

free\_example.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     void *p = malloc(100);
7     free(p);
8
9     return 0;
10 }
```

root@4e1b694fa58f: /home/v

root@4e1b694fa58f: /home/workspace#

◀ Page 7 of 8 ▶

## Freeing up Memory

If memory has been reserved, it should also be released as soon as it is no longer needed. If memory is reserved regularly without releasing it again, the memory capacity may be exhausted at some point. If the RAM memory is completely used up, the data is swapped out to the hard disk, which slows down the computer significantly.

The `free` function releases the reserved memory area so that it can be used again or made available to other programs. To do this, the pointer pointing to the memory area to be freed is specified as a parameter for the function. In the `free_example.cpp`, a memory area is reserved and immediately released again.

The screenshot displays a web-based IDE interface. On the left, a 'Guide' tab is active, showing text about dynamic memory management. It lists two points: 1. `free` can only free memory that was reserved with `malloc` or `calloc`. 2. `free` can only release memory that has not been released before. Releasing the same block of memory twice will result in an error. Below this, it explains that in the example on the right, a pointer `p` is copied into a new variable `p2`, which is then passed to `free` after the original pointer has been released. A terminal output shows a 'malloc: \*\*\* error for object 0x1003001f0: pointer being freed was not allocated.' message. Further down, it states that in the workspace, a 'double free or corruption' error will be seen. The pointer `p2` is noted as invalid after `free(p)` is called. At the bottom of the guide, a page indicator shows 'Page 8 of 8'. On the right, a code editor shows the contents of `free_example.cpp`, which includes `<stdio.h>` and `<stdlib.h>`, and contains a `main` function that allocates memory with `malloc(100)`, immediately frees it with `free(p)`, and returns 0. Below the code editor, a terminal window shows the command prompt `root@4e1b694fa58f:/home/workspace#`.

Using malloc and free

Guide

Some things should be considered with dynamic memory management, whose neglect in some cases might result in unpredictable program behavior or a system crash - in some cases unfortunately without error messages from the compiler or the operating system:

1. `free` can only free memory that was reserved with `malloc` or `calloc`.
2. `free` can only release memory that has not been released before. Releasing the same block of memory twice will result in an error.

In the example on the right, a pointer `p` is copied into a new variable `p2`, which is then passed to `free` AFTER the original pointer has been already released.

```
free(41143,0x1000a55c0) malloc: *** error for object
0x1003001f0: pointer being freed was not allocated.
```

In the workspace, you will see this error:

```
*** Error in './a.out': double free or corruption
(fasttop): 0x0000000000755010 ***
```

The pointer `p2` in the example is invalid as soon as `free(p)` is called. It still holds the address to the memory location which has been freed, but may not access it anymore. Such a pointer is

◀ Page 8 of 8 ▶

free\_example.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     void *p = malloc(100);
7     free(p);
8
9     return 0;
10 }
```

root@4e1b694fa58f:/home/v

root@4e1b694fa58f:/home/workspace#

Some things should be considered with dynamic memory management, whose neglect in some cases might result in unpredictable program behavior or a system

crash - in some cases unfortunately without error messages from the compiler or the operating system:

1. `free` can only free memory that was reserved with `malloc` or `calloc`.
2. `free` can only release memory that has not been released before. Releasing the same block of memory twice will result in an error.

In the example on the right, a pointer `p` is copied into a new variable `p2`, which is then passed to `free` AFTER the original pointer has been already released.

`free(41143,0x1000a55c0) malloc: *** error for object 0x1003001f0: pointer being freed was not allocated.`

In the workspace, you will see this error:

```
*** Error in './a.out': double free or corruption (fasttop): 0x0000000000755010 ***
```

The pointer `p2` in the example is invalid as soon as `free(p)` is called. It still holds the address to the memory location which has been freed, but may not access it anymore. Such a pointer is called a "dangling pointer".

3. Memory allocated with `malloc` or `calloc` is not subject to the familiar rules of variables in their respective scopes. This means that they exist independently of block limits until they are released again or the program is terminated. However, the pointers which refer to such heap-allocated memory are created on the stack and thus only exist within a limited scope. As soon as the scope is left, the pointer variable will be lost - but not the heap memory it refers to.



## Quiz : Dynamic Memory Management with `malloc`, `calloc`, `resize` and `free`

Question 1: Match the code snippets to the respective comments

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // (X)
    int *m = (int*)malloc(sizeof(int));
    m = NULL;

    // (Y)
    int *n = (int*)malloc(sizeof(int));
    free(n);
    *n = 23;

    // (Z)
    char *o;
    *o = 'a';

    return 0;
}
```

Comments:

1. uses a dangling pointer
2. uses an uninitialized pointer
3. generates a memory leak

### QUESTION 1 OF 2

In the code above, there are three snippets marked with X, Y, and Z. Below the code there are three comments. Match the code snippets to the respective comments. Which pairing of comments and code snippets is the correct one?

- ☐ X-1, Y-3, Z-2
- ☐ X-2, Y-1, Z-3
- ☐ X-3, Y-2, Z-1
- ☐ X-3, Y-1, Z-2

QUESTION 1 OF 2

In the code above, there are three snippets marked with X, Y, and Z. Below the code there are three comments. Match the code snippets to the respective comments. Which pairing of comments and code snippets is the correct one?

☐ X-1, Y-3, Z-2

☐ X-2, Y-1, Z-3

☐ X-3, Y-2, Z-1

☒ X-3, Y-1, Z-2

Question 2 : Problems with pointers

```
int *f1(void)
{
    int x = 10;
    return (&x);
}

int *f2(void)
{
    int *px;
    *px = 10;
    return px;
}

int *f3(void)
{
    int *px;
    px = (int *)malloc(sizeof(int));
    *px = 10;
    return px;
}
```

QUESTION 2 OF 2

Which of the functions above is likely to cause pointer-related problems?

- ☐ only `f3`
- ☐ `f1` and `f3`
- ☐ `f1` and `f2`
- ☐ `f1`, `f2`, and `f3`

QUESTION 2 OF 2

Which of the functions above is likely to cause pointer-related problems?

- ☐ only `f3`
- ☐ `f1` and `f3`
- ☒ `f1` and `f2`
- ☐ `f1`, `f2`, and `f3`

Thanks for completing that!

Great work! In `f1`, the pointer variable `x` is a local variable to `f1`, and `f1` returns the pointer to that variable. `x` can disappear after `f1()` is returned if `x` exists on the stack. So `&x` can become invalid.

In `f2`, the pointer variable `px` is assigned a value without allocating its space.

`f3` works fine. Memory is allocated to the pointer variable `px` using `malloc()`.

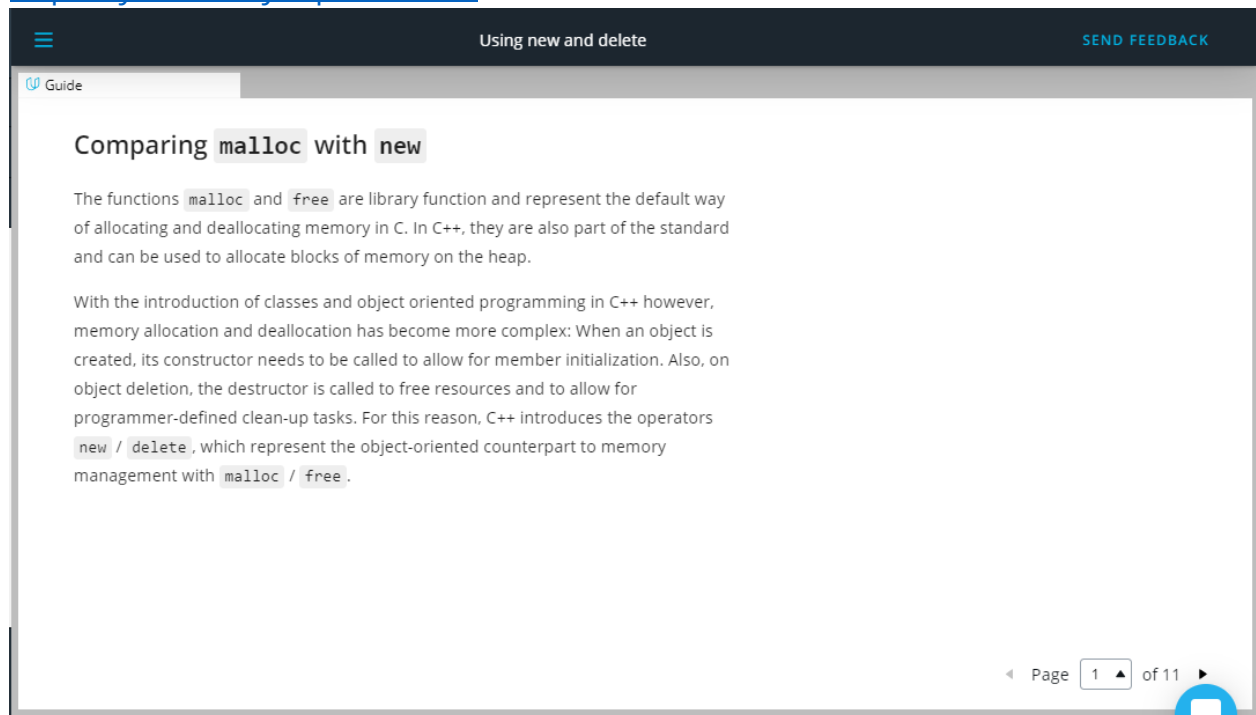
So, `px` exists on the heap, its existence will remain in memory even after the return of `f3()` as it is on the heap.

Outro

<https://youtu.be/X8OcmAwAYp8>

#### 4) Using new and delete

<https://youtu.be/y6xpWulUsEM>



## Comparing `malloc` with `new`

The functions `malloc` and `free` are library function and represent the default way of allocating and deallocating memory in C. In C++, they are also part of the standard and can be used to allocate blocks of memory on the heap.

With the introduction of classes and object oriented programming in C++ however, memory allocation and deallocation has become more complex: When an object is created, its constructor needs to be called to allow for member initialization. Also, on object deletion, the destructor is called to free resources and to allow for programmer-defined clean-up tasks. For this reason, C++ introduces the operators `new` / `delete`, which represent the object-oriented counterpart to memory management with `malloc` / `free`.

### Using new and delete

[SEND FEEDBACK](#)

#### Guide

If we were to create a C++ object with `malloc`, the constructor and destructor of such an object would not be called. Consider the class on the right. The constructor allocates memory for the private element `_number` (yes, we could have simply used `int` instead of `int*`, but that's for educational purposes only), and the destructor releases memory again. The setter method `setNumber` finally assigns a value to `_number` under the assumption that memory has been allocated previously.

In main, we will allocate memory for an instance of `MyClass` using both `malloc` / `free` and `new` / `delete`.

With `malloc`, the program crashes on calling the method `setNumber`, as no memory has been allocated for `_number` - because the constructor

#### malloc\_v\_new.cpp

```
1 #include <stdlib.h>
2 #include <iostream>
3
4 class MyClass
5 {
6 private:
7     int *_number;
8
9 public:
10    MyClass()
11    {
12        std::cout << "Allocate memory\n";
13        _number = (int *)malloc(sizeof(int));
14    }
15    ~MyClass()
16    {
17        std::cout << "Delete memory\n";
18        free(_number);
19    }
20    void setNumber(int number)
21    {
```

```
root@554e3bae6b8a: /home/
root@554e3bae6b8a: /home/workspace#
```

◀ Page 2 of 11 ▶

If we were to create a C++ object with `malloc`, the constructor and destructor of such an object would not be called. Consider the class on the right. The constructor allocates memory for the private element `_number` (yes, we could have simply used `int` instead of `int*`, but that's for educational purposes only), and the destructor releases memory again. The setter method `setNumber` finally assigns a value to `_number` under the assumption that memory has been allocated previously.

In main, we will allocate memory for an instance of `MyClass` using both `malloc` / `free` and `new` / `delete`.

With `malloc`, the program crashes on calling the method `setNumber`, as no memory has been allocated for `_number` - because the constructor has not been called. Hence, an `EXC_BAD_ACCESS` error occurs, when trying to access the memory location to which `_number` is pointing. With `_new`, the output looks like the following:

```
Allocate memory
Number: 42
Delete memory
```

Using new and delete

Guide

Before we go into further details of `new` / `delete`, let us briefly summarize the major differences between `malloc` / `free` and `new` / `delete`:

1. **Constructors / Destructors** Unlike `malloc( sizeof(MyClass) )`, the call `new MyClass()` calls the constructor. Similarly, `delete` calls the destructor.
2. **Type safety** `malloc` returns a void pointer, which needs to be cast into the appropriate data type it points to. This is not type safe, as you can freely vary the pointer type without any warnings or errors from the compiler as in the following small example:

```
MyObject *p =
(MyObject*)malloc(sizeof(int));
```

In C++, the call `MyObject *p = new MyObject()` returns the correct type automatically - it is thus

malloc\_v\_new.cpp

```

1 #include <stdlib.h>
2 #include <iostream>
3
4 class MyClass
5 {
6 private:
7     int *_number;
8
9 public:
10    MyClass()
11    {
12        std::cout << "Allocate memory\n";
13        _number = (int *)malloc(sizeof(int));
14    }
15    ~MyClass()
16    {
17        std::cout << "Delete memory\n";
18        free(_number);
19    }
20    void setNumber(int number)
21    {

```

root@554e3bae6b8a: /home/

root@554e3bae6b8a: /home/workspace#

◀

Page 3 of 11 ▶

Before we go into further details of `new/delete`, let us briefly summarize the major differences between `malloc/free` and `new/delete`:

1. **Constructors / Destructors** Unlike `malloc( sizeof(MyClass) )`, the call `new MyClass()` calls the constructor. Similarly, `delete` calls the destructor.
2. **Type safety** `malloc` returns a void pointer, which needs to be cast into the appropriate data type it points to. This is not type safe, as you can freely vary the pointer type without any warnings or errors from the compiler as in the following small example: `MyObject *p = (MyObject*)malloc(sizeof(int));`

In C++, the call `MyObject *p = new MyObject()` returns the correct type automatically - it is thus type-safe.

3. **Operator Overloading** As `malloc` and `free` are functions defined in a library, their behavior can not be changed easily. The `new` and `delete` operators however can be overloaded by a class in order to include optional proprietary behavior. We will look at an example of overloading `new` further down in this section.

Using new and delete

Guide

## Creating and Deleting Objects

As with `malloc` and `free`, a call to `new` always has to be followed by a call to `delete` to ensure that memory is properly deallocated. If the programmer forgets to call `delete` on the object (which happens quite often, even with experienced programmers), the object resides in memory until the program terminates at some point in the future causing a *memory leak*.

Let us revisit a part of the code example to the right:

```
myClass = new MyClass();
myClass->setNumber(42); // works as expected
delete myClass;
```

malloc\_v\_new.cpp

```
1 #include <stdlib.h>
2 #include <iostream>
3
4 class MyClass
5 {
6 private:
7     int *_number;
8
9 public:
10     MyClass()
11     {
12         std::cout << "Allocate memory\n";
13         _number = (int *)malloc(sizeof(int));
14     }
15     ~MyClass()
16     {
17         std::cout << "Delete memory\n";
18         free(_number);
19     }
20     void setNumber(int number)
21     {
22         // ...
23     }
24 }
```

root@554e3bae6b8a: /home/

root@554e3bae6b8a: /home/workspace#

◀ Page 4 of 11 ▶

## Creating and Deleting Objects

As with `malloc` and `free`, a call to `new` always has to be followed by a call to `delete` to ensure that memory is properly deallocated. If the programmer forgets to call `delete` on the object (which happens quite often, even with experienced programmers), the object resides in memory until the program terminates at some point in the future causing a *memory leak*.

Let us revisit a part of the code example to the right:

```
myClass = new MyClass();
myClass->setNumber(42); // works as expected
delete myClass;
```

The call to `new` has the following consequences:

1. Memory is allocated to hold a new object of type `MyClass`
2. A new object of type `MyClass` is constructed within the allocated memory by calling the constructor of `MyClass`

The call to `delete` causes the following:

1. The object of type MyClass is destroyed by calling its destructor
2. The memory which the object was placed in is deallocated

The screenshot shows a web interface with a dark header bar containing a menu icon and the title "Using new and delete". Below the header, there are two main panels. The left panel, titled "Guide", contains the text "Optimizing Performance with placement new". The right panel, titled "malloc\_v\_new.cpp", displays C++ code for a class MyClass that uses malloc and free for memory management. At the bottom of the right panel, there is a terminal window showing the command prompt "root@554e3bae6b8a:/home/workspace#".

**Using new and delete**

**Guide**

### Optimizing Performance with placement new

In some cases, it makes sense to separate memory allocation from object construction. Consider a case where we need to reconstruct an object several times. If we were to use the standard `new / delete` construct, memory would be allocated and freed unnecessarily as only the content of the memory block changes but not its size. By separating allocation from construction, we can get a significant performance increase.

C++ allows us to do this by using a construct called *placement new*: With `placement new`, we can pass a preallocated memory and construct an object at that memory location. Consider the

Page 5 of 11

```
1 #include <stdlib.h>
2 #include <iostream>
3
4 class MyClass
5 {
6 private:
7     int *_number;
8
9 public:
10    MyClass()
11    {
12        std::cout << "Allocate memory\n";
13        _number = (int *)malloc(sizeof(int));
14    }
15    ~MyClass()
16    {
17        std::cout << "Delete memory\n";
18        free(_number);
19    }
20    void setNumber(int number)
21    {
```

root@554e3bae6b8a:/home/

root@554e3bae6b8a:/home/workspace#

## Optimizing Performance with `placement new`

In some cases, it makes sense to separate memory allocation from object construction. Consider a case where we need to reconstruct an object several times. If we were to use the standard `new/delete` construct, memory would be allocated and freed unnecessarily as only the content of the memory block changes but not its size. By separating allocation from construction, we can get a significant performance increase.

C++ allows us to do this by using a construct called *placement new*: With `placement new`, we can pass a preallocated memory and construct an object at that memory location. Consider the following code:

```
void *memory = malloc(sizeof(MyClass));
MyClass *object = new (memory) MyClass;
```



The syntax `new (memory)` is denoted as *placement new*. The difference to the "conventional" `new` we have been using so far is that that no memory is allocated. The call constructs an object and places it in the assigned memory location. There is however, no delete equivalent to placement new, so we have to call the destructor explicitly in this case instead of using `delete` as we would have done with a regular call to `new`:

```
object->~MyClass();  
free(memory);
```

**Important:** Note that this should never be done outside of placement new.

In the next section, we will look at how to overload the `new` operator and show the performance difference between placement new and `new`

The screenshot shows a web application titled "Using new and delete" with a "SEND FEEDBACK" button. The left sidebar contains a "Guide" section titled "Overloading new and delete". The main content area displays the C++ code for `overloading.cpp`.

**Overloading new and delete**

One of the major advantages of `new` / `delete` over `free` / `malloc` is the possibility of overloading. While both `malloc` and `free` are function calls and thus can not be changed easily, `new` and `delete` are operators and can thus be overloaded to integrate customized functionality, if needed.

The syntax for overloading the `new` operator looks as follows:

```
void* operator new(size_t size);
```

The operator receives a parameter `size` of type `size_t`, which specifies the number of bytes of memory to be allocated. The return type of the overloaded `new` is a void pointer, which

```
1 #include <iostream>  
2 #include <stdlib.h>  
3  
4 class MyClass  
5 {  
6     int _mymember;  
7  
8 public:  
9     MyClass()  
10    {  
11        std::cout << "Constructor is called\n";  
12    }  
13  
14    ~MyClass()  
15    {  
16        std::cout << "Destructor is called\n";  
17    }  
18  
19    void *operator new(size_t size)  
20    {  
21        std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
```

## Overloading new and delete

One of the major advantages of `new`/`delete` over `free`/`malloc` is the possibility of overloading. While both `malloc` and `free` are function calls and thus can not be changed easily, `new` and `delete` are operators and can thus be overloaded to integrate customized functionality, if needed.

The syntax for **overloading the new operator** looks as follows:

```
void* operator new(size_t size);
```

The operator receives a parameter `size` of type `size_t`, which specifies the number of bytes of memory to be allocated. The return type of the overloaded `new` is a void pointer, which references the beginning of the block of allocated memory.

The syntax for **overloading the delete operator** looks as follows:

```
void operator delete(void*);
```

The operator takes a pointer to the object which is to be deleted. As opposed to `new`, the operator `delete` does not have a return value.

Let us consider the example on the right.

Using new and delete

SEND FEEDBACK

Guide

00:00 / 03:26

1x CC

◀ Page 7 of 11 ▶

[https://video.udacity-data.com/topher/2019/September/5d855a4f\\_nd213-c03-l03-03.2-using-new-and-delete-sc/nd213-c03-l03-03.2-using-new-and-delete-sc\\_720p.mp4](https://video.udacity-data.com/topher/2019/September/5d855a4f_nd213-c03-l03-03.2-using-new-and-delete-sc/nd213-c03-l03-03.2-using-new-and-delete-sc_720p.mp4)

Using new and delete

SEND FEEDBACK

Guide

In the code to the right, both the `new` and the `delete` operator are overloaded. In `new`, the size of the class object in bytes is printed to the console. Also, a block of memory of that size is allocated on the heap and the pointer to this block is returned. In `delete`, the block of memory is freed again. The console output of this example looks as follows:

```
new: Allocating 4 bytes of memory
Constructor is called
Destructor is called
delete: Memory is freed again
```

As can be seen from the order of text output, memory is instantiated in `new` before the constructor is called, while the order is reversed for the destructor and the call to `delete`.

overloading.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 class MyClass
5 {
6     int _mymember;
7
8 public:
9     MyClass()
10    {
11        std::cout << "Constructor is called\n";
12    }
13
14    ~MyClass()
15    {
16        std::cout << "Destructor is called\n";
17    }
18
19    void *operator new(size_t size)
20    {
21        std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22    }
23
24    void operator delete(void *p)
25    {
26        std::cout << "delete: Freeing memory" << std::endl;
27    }
28
29    int main()
30    {
31        MyClass *p = new MyClass();
32        delete p;
33    }
34 }
```

root@554e3bae6b8a: /home/

root@554e3bae6b8a: /home/workspace#

In the code to the right, both the `new` and the `delete` operator are overloaded. In `new`, the size of the class object in bytes is printed to the console. Also, a block of memory of that size is allocated on the heap and the pointer to this block is returned. In `delete`, the block of memory is freed again. The console output of this example looks as follows:

```
new: Allocating 4 bytes of memory
Constructor is called
Destructor is called
delete: Memory is freed again
```

As can be seen from the order of text output, memory is instantiated in `new` before the constructor is called, while the order is reversed for the destructor and the call to `delete`.

The screenshot shows a web application interface. On the left, there is a 'Quiz' section with a question: 'What will happen to the console output of the overloaded new operator when the data type of `_mymember` is changed from int to double?'. Three multiple-choice options are listed: 1. Nothing, 2. There will be a memory leak as not enough memory is allocated on the heap, and 3. The output will show a changed size in bytes (8 instead of 4). A 'HIDE SOLUTION' button is present, followed by the text 'The correct answer is 3.' At the bottom of the quiz section, it says 'Page 9 of 11'. On the right, there is a code editor titled 'overloading.cpp' showing C++ code for a class `MyClass` with a constructor, destructor, and an overloaded `new` operator. The `new` operator prints the size of the object and allocates memory. Below the code editor is a terminal window showing the command prompt 'root@554e3bae6b8a: /home/workspace# '.

Quiz :

What will happen to the console output of the overloaded new operator when the data type of `_mymember` is changed from int to double?

1. Nothing.
2. There will be a memory leak as not enough memory is allocated on the heap.
3. The output will show a changed size in bytes (8 instead of 4).

HIDE SOLUTION

The correct answer is 3.

Guide

## Overloading new[] and delete[]

In addition to the `new` and `delete` operators we have seen so far, we can use the following code to create an array of objects:

```
void* operator new[](size_t size);
void operator delete[](void*);
```

Let us consider the example on the right, which has been slightly modified to allocate an array of objects instead of a single one.

◀ Page 10 of 11 ▶

overloading\_array.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 class MyClass
5 {
6     int _mymember;
7
8 public:
9     MyClass()
10    {
11        std::cout << "Constructor is called\n";
12    }
13
14    ~MyClass()
15    {
16        std::cout << "Destructor is called\n";
17    }
18
19    void *operator new[](size_t size)
20    {
21        std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22    }
23 }
```

root@554e3bae6b8a: /home/

root@554e3bae6b8a: /home/workspace#

## Overloading new[] and delete[]

In addition to the `new` and `delete` operators we have seen so far, we can use the following code to create an array of objects:

```
void* operator new[](size_t size);
```

```
void operator delete[](void*);
```

Let us consider the example on the right, which has been slightly modified to allocate an array of objects instead of a single one.

Guide

In main, we are now creating an array of three objects of `MyClass`. Also, the overloaded `new` and `delete` operators have been changed to accept arrays. Let us take a look at the console output:

```
new: Allocating 20 bytes of memory
Constructor is called
Constructor is called
Constructor is called
Destructor is called
Destructor is called
Destructor is called
delete: Memory is freed again
```

Interestingly, the memory requirement is larger than expected: With `new`, the block size was 4 bytes, which is exactly the space required for a single integer. Thus, with three integers, it should now be 12 bytes instead of 20 bytes. The reason

◀ Page 11 of 11 ▶

overloading\_array.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 class MyClass
5 {
6     int _mymember;
7
8 public:
9     MyClass()
10    {
11        std::cout << "Constructor is called\n";
12    }
13
14    ~MyClass()
15    {
16        std::cout << "Destructor is called\n";
17    }
18
19    void *operator new[](size_t size)
20    {
21        std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22    }
23 }
```

root@554e3bae6b8a: /home/

root@554e3bae6b8a: /home/workspace#

In main, we are now creating an array of three objects of MyClass. Also, the overloaded `new` and `delete` operators have been changed to accept arrays. Let us take a look at the console output:

```
new: Allocating 20 bytes of memory
Constructor is called
Constructor is called
Constructor is called
Destructor is called
Destructor is called
Destructor is called
delete: Memory is freed again
```

Interestingly, the memory requirement is larger than expected: With `new`, the block size was 4 bytes, which is exactly the space required for a single integer. Thus, with three integers, it should now be 12 bytes instead of 20 bytes. The reason for this is the memory allocation overhead that the compiler needs to keep track of the allocated blocks of memory - which in itself consumes memory. If we change the above call to e.g. `new MyClass[100]()`, we will see that the overhead of 8 bytes does not change:

```
new: Allocating 408 bytes of memory
Constructor is called
...
Destructor is called
delete: Memory is freed again
```

## Reasons for overloading `new` and `delete`

Now that we have seen how to overload the `new` and `delete` operators, let us summarize the major scenarios where it makes sense to do this:

1. The overloaded `new` operator function allows to **add additional parameters**. Therefore, a class can have multiple overloaded `new` operator functions. This gives the programmer more flexibility in customizing the memory allocation for objects.
2. Overloaded the `new` and `delete` operators provides an easy way to **integrate a mechanism similar to garbage collection** capabilities (such as in Java), as we will shortly see later in this course.
3. By adding **exception handling capabilities** into `new` and `delete`, the code can be made more robust.
4. It is very easy to add customized behavior, such as overwriting deallocated memory with zeros in order to increase the security of critical application data.

## Outro

[https://youtu.be/mT6Mn754d\\_o](https://youtu.be/mT6Mn754d_o)

- 5) Bjarne on new and delete  
<https://youtu.be/DN2qXt27Hxc>
- 6) Typical Memory Management Problems  
<https://youtu.be/FfyFNqAFIIY>

## Overview of memory management problems

One of the primary advantages of C++ is the flexibility and control of resources such as memory it gives to the programmer. This advantage is further amplified by a significant increase in the performance of C++ programs compared to other languages such as Python or Java.

However, these advantages come at a price as they demand a high level of experience from the programmer. As Bjarne Stroustrup put it so elegantly:

*"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".*

In this chapter, we will look at a collection of typical errors in memory management that you need to watch out for.

1. **Memory Leaks** Memory leaks occur when data is allocated on the heap at runtime, but not properly deallocated. A program that forgets to clear a memory block is said to have a memory leak - this may be a serious problem or not, depending on the circumstances and on the nature of the program. For a program that runs, computes something, and quits immediately, memory leaks are usually not a big concern. Memory leaks are mostly problematic for programs that run for a long time and/or use large data structures. In such a case, memory leaks can gradually fill the heap until allocation requests can no longer be properly met and the program stops responding or crashes completely. We will look at an example further down in this section.
2. **Buffer Overruns** Buffer overruns occur when memory outside the allocated limits is overwritten and thus corrupted. One of the resulting problems is that this effect may not become immediately visible. When a problem finally does occur, cause and effect are often hard to discern. It is also sometimes possible to inject malicious code into programs in this way, but this shall not be discussed here.  
In this example, the allocated stack memory is too small to hold the entire string, which results in a segmentation fault:

```
char str[5];  
strcpy(str, "BufferOverrun");  
printf("%s", str);
```

3. **Uninitialized Memory** Depending on the C++ compiler, data structures are sometimes initialized (most often to zero) and sometimes not. So when allocating memory

on the heap without proper initialization, it might sometimes contain garbage that can cause problems.

Generally, a variable will be automatically initialized in these cases:

- it is a class instance where the default constructor initializes all primitive types
- array initializer syntax is used, such as `int a[10] = {}`
- it is a global or extern variable
- it is defined `static`

The behavior of the following code is potentially undefined:

```
int a;  
int b=a*42;  
printf("%d",b);
```

4. **Incorrect pairing of allocation and deallocation** Freeing a block of memory more than once will cause a program to crash. This can happen when a block of memory is freed that has never been allocated or has been freed before. Such behavior could also occur when improper pairings of allocation and deallocation are used such as using `malloc()` with `delete` or `new` with `free()`.

In this first example, the wrong `new` and `delete` are paired

```
double *pDb1=new double[5];  
delete pDb1;
```

In this second example, the pairing is correct but a double deletion is performed:

```
char *pChr=new char[5];  
delete[] pChr;  
delete[] pChr;
```

5. **Invalid memory access** This error occurs then trying to access a block of heap memory that has not yet or has already been deallocated.

In this example, the heap memory has already been deallocated at the time when `strcpy()` tries to access it:

```
char *pStr=new char[25];  
delete[] pStr;  
strcpy(pStr, "Invalid Access");
```

## Debugging memory leaks with Valgrind

[https://youtu.be/y0m\\_6V8fvzA](https://youtu.be/y0m_6V8fvzA)

Even experienced developers sometimes make mistakes that cannot be discovered at first glance. Instead of spending a lot of time searching, it makes sense for C and C++ programmers to use helper tools to perform automatic analyses of their code.

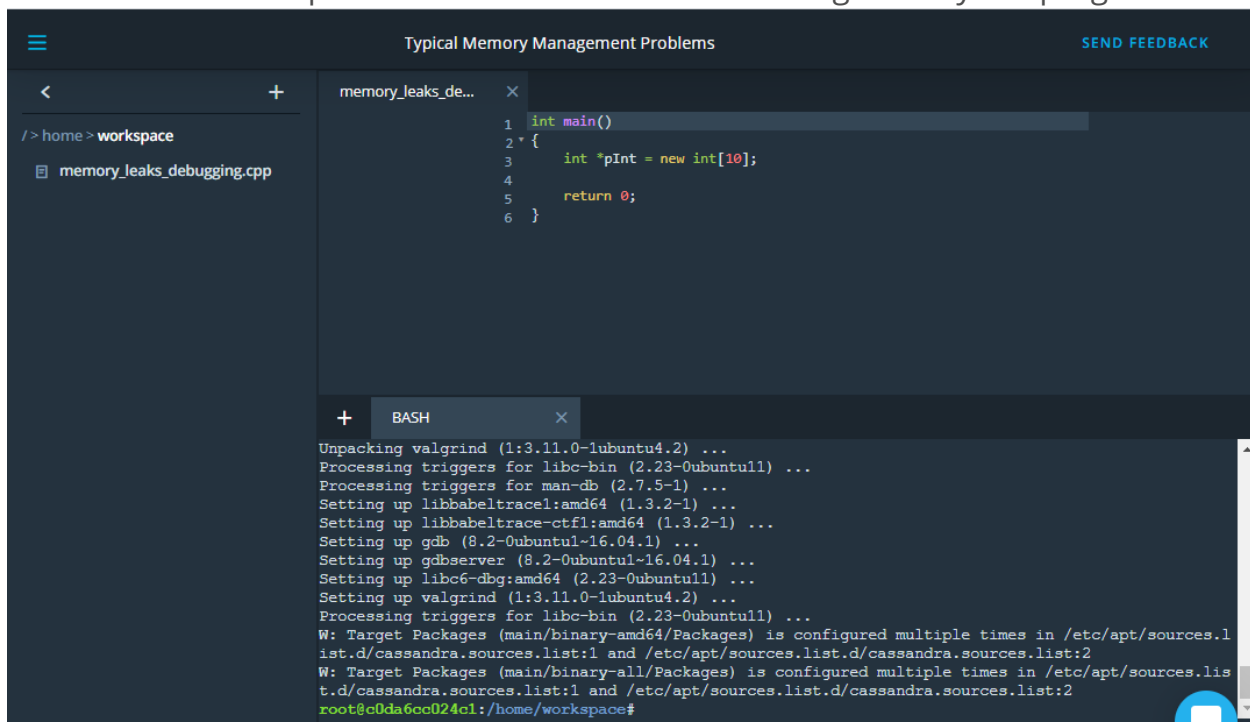
In this section, we will look at *Valgrind*, a free software for Linux and Mac that is able to automatically detect memory. Windows programmers can for example use the Visual Studio debugger and C Run-time Library (CRT) to detect and identify memory leaks. More information on how to do this can be found here: [Find memory leaks with the CRT Library - Visual Studio | Microsoft Docs](#)

With recent versions of MacOS, occasional difficulties have been reported with installing Valgrind. A working version for MacOS Mojave can be downloaded from GitHub via Homebrew: [GitHub - sowson/valgrind:Experimental Version of Valgrind for macOS 10.14.6 Mojave](https://github.com/sowson/valgrind:ExperimentalVersionofValgrindformacOS10.14.6Mojave)

Valgrind is a framework that facilitates the development of tools for the dynamic analysis of programs. Dynamic analysis examines the behavior of a program at runtime, in contrast to static analysis, which often checks programs for various criteria or potential errors at the source code level before, during, or after translation. More information on Valgrind can be found here: [Valgrind:About](#)

The Memcheck tool within Valgrind can be used to detect typical errors in programs written in C or C++ that occur in connection with memory management. It is probably the best-known tool in the Valgrind suite, and the name Valgrind is often used as a synonym for Memcheck.

The following code generates a memory leak as the integer array has been allocated on the heap but the deallocation has been forgotten by the programmer:



The screenshot shows a code editor with a dark theme. The top bar says 'Typical Memory Management Problems' and 'SEND FEEDBACK'. The editor has two panes. The left pane shows a file explorer with a folder icon and the file 'memory\_leaks\_debugging.cpp'. The right pane shows the code for 'memory\_leaks\_de...'. The code is as follows:

```
1 int main()
2 {
3     int *pInt = new int[10];
4     return 0;
5 }
6 }
```

Below the code editor is a terminal window titled 'BASH'. It shows the output of installing Valgrind on Ubuntu 16.04.1. The output includes:

```
Unpacking valgrind (1:3.11.0-1ubuntu4.2) ...
Processing triggers for libc-bin (2.23-0ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up libbabeltrace1:amd64 (1.3.2-1) ...
Setting up libbabeltrace-ctf1:amd64 (1.3.2-1) ...
Setting up gdb (8.2-0ubuntu1~16.04.1) ...
Setting up gdbserver (8.2-0ubuntu1~16.04.1) ...
Setting up libc6-dbg:amd64 (2.23-0ubuntu1) ...
Setting up valgrind (1:3.11.0-1ubuntu4.2) ...
Processing triggers for libc-bin (2.23-0ubuntu1) ...
W: Target Packages (main/binary-amd64/Packages) is configured multiple times in /etc/apt/sources.list.d/cassandra.sources.list:1 and /etc/apt/sources.list.d/cassandra.sources.list:2
W: Target Packages (main/binary-all/Packages) is configured multiple times in /etc/apt/sources.list.d/cassandra.sources.list:1 and /etc/apt/sources.list.d/cassandra.sources.list:2
root@c0da6cc024c1:/home/workspace#
```

The array of integers on the heap to which `pInt` is pointing has a size of `10 * sizeof(int)`, which is 40 bytes. Let us now use Valgrind to search for this leak. After compiling the `memory_leaks_debugging.cpp` code file on the right to `a.out`, the terminal can be used to start Valgrind with the following command:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --log-file=/home/
/workspace/valgrind-out.txt /home/workspace/a.out
```

Let us look at the call parameters one by one:



- **--leak-check**: Controls the search for memory leaks when the client program finishes. If set to `summary`, it says how many leaks occurred. If set to `full`, each individual leak will be shown in detail.
  - **--show-leak-kinds**: controls the set of leak kinds to show when `--leak-check=full` is specified. Options are `definite`, `indirect`, `possible`, `reachable`, `all` and `none`
  - **--track-origins**: can be used to see where uninitialised values come from.
- You can read the file into the terminal with:

```
cat valgrind-out.txt
```

<https://youtu.be/K2PQTpldolw>

In the following, a (small) excerpt of the `valgrind-out.txt` log file is given:

```
==952== 40 bytes in 1 blocks are definitely lost in loss record 18 of 45
...
==952==    by 0x10019A377: operator new(unsigned long) (in /usr/lib/libc++abi.dylib)
...
==952==    by 0x100000F8A: main (memory_leaks_debugging.cpp:12)
...
==952== LEAK SUMMARY:
==952==    definitely lost: 40 bytes in 1 blocks
==952==    indirectly lost: 0 bytes in 0 blocks
==952==    possibly lost: 72 bytes in 3 blocks
==952==    still reachable: 200 bytes in 6 blocks
==952==    suppressed: 18,876 bytes in 160 blocks
```

As expected, the memory leak caused by the omitted deletion of the array of 10 integers in the code sample above shows up in the leak summary. Additionally, the exact position where the leak occurs in the code (line 12) can also be seen together with the responsible call with caused the leak.

This short introduction into memory leak search is only an example of how powerful analysis tools such as Valgrind can be used to detect memory-related problems in your code.

## Outro

<https://youtu.be/lhFaPswyBII>

### 7) Bjarne on Memory Leaks

<https://youtu.be/lwZa3Z5rvtg>