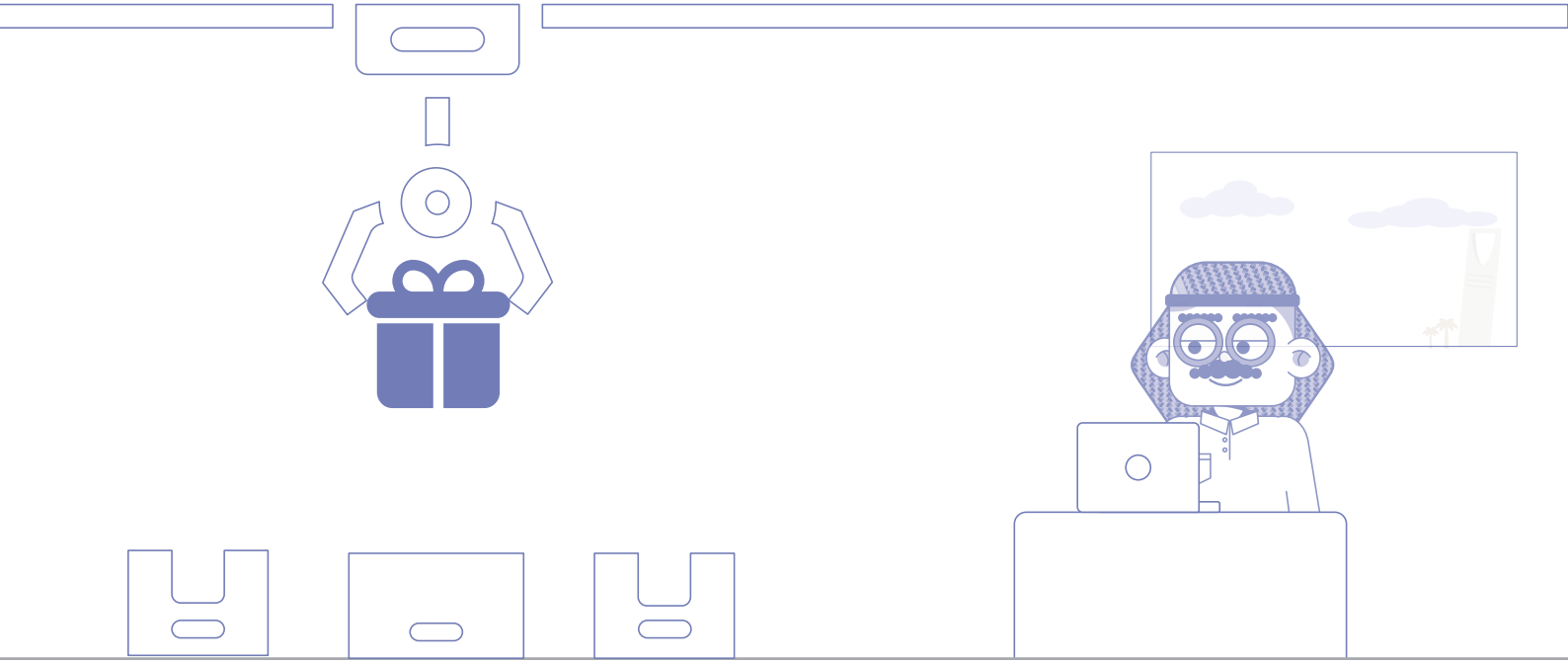
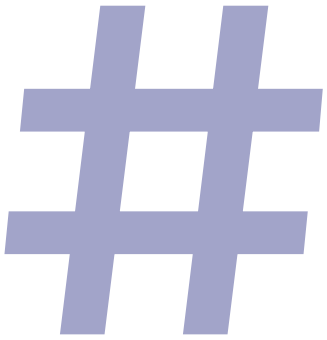
OPERATING SYSTEM

Memory Management





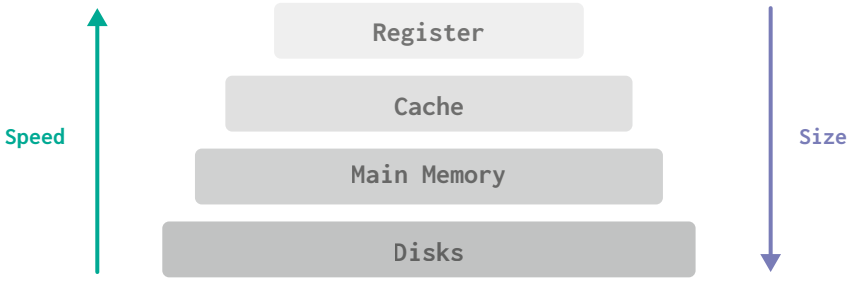
Introduction to Memory Management

Step 01 Memory Management Introduction

Computers process data, but this data doesn't appear out of nowhere; it is stored in a crucial component known as the **memory**.

Memory comes in various types and sizes. The two main types are **Random Access Memory (RAM)**, which temporarily stores data, and **Read Only Memory (ROM),** which permanently holds data.

Additionally, the CPU has small storage areas called **registers**, which hold data needed for quick processing.



Step 02 Memory Types and Usage

01 Registers

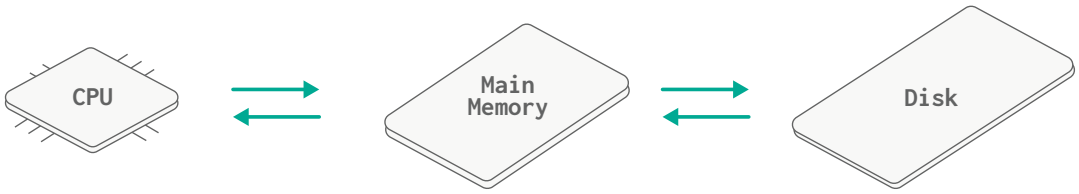
Registers are the smallest form of memory, located inside the CPU. They store data for fast access during processing.

02 Cache

Cache memory is a small, high-speed storage that makes data retrieval from RAM more efficient by holding the most frequently accessed data.

03 Random Access Memory (RAM)

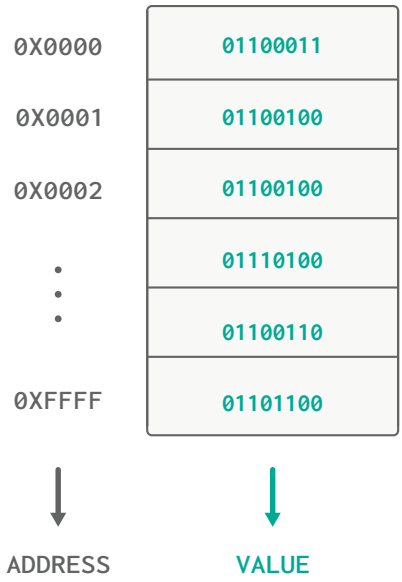
RAM is often referred to as **main memory**. It is faster than ROM, although smaller in size. The operating system uses RAM to load programs as processes for the CPU to execute. Once a process is loaded and ready, the CPU reads data which is the instructions directly from RAM to execute them. When the process finishes, the operating system frees up that memory space for other processes.



NOTE
It's important to note that RAM is volatile, meaning all data is lost when the computer is turned off or loses power.

Step 03 What Does Main Memory Look Like?

Main memory is a large array of bytes with an address that refers to each entry.



NOTE
In this chapter, we will focus on main memory management.

Step 04 Read Only Memory (ROM)

ROM is a larger memory space used to store permanent data that should not be lost when the power is turned off. It is non-volatile, which means the information remains intact.

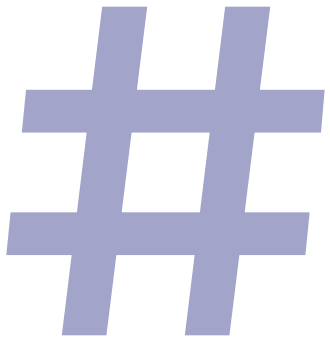
For example, if you update a file but forget to save it before a sudden power loss, that update will be lost since it was stored in volatile RAM. However, if you save your changes, they will be stored permanently in ROM, and will not get lost with the sudden power loss.

Step 05 Operating System and Memory Management

OS is responsible for managing hardware resources, which the main memory is part of. It involves the efficient allocation, utilization, and management of the computer's physical memory resources to ensure that programs and processes can run effectively and without conflicts.

OS ensures the following about memory.

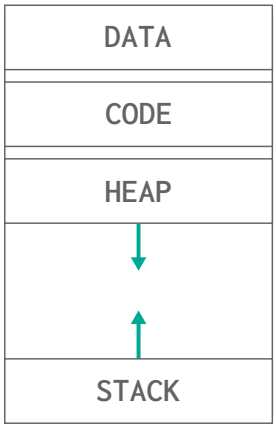
- Allocate and deallocate memory spaces as required.
- Keep track of each memory block usage and the process by which it is used.
- Determine which parts of memory blocks should move in or out of the memory.



Memory and Process

Step 01 Memory and Process

Earlier, we have introduced the concept of process. Which is a program in execution. When you run a program, the OS will map the program into memory as a process. Which consists of data, code, stack, and heap. In this lesson, we will delve into each section/segment in the process stored in memory in detail.



Step 02 Process Memory Segments

When a process is loaded into memory, it is organized into several distinct segments, each serving a specific purpose. These segments are the **code, data, heap, and stack**. Let's explore each segment in detail:

01 Code Segment

- Description

  The **code segment** (text segment) contains the compiled executable code of the program. This is the actual machine code that the CPU executes. It's typically read-only to prevent modification during execution, ensuring the integrity of the program.

- Storage

  - **Location**: The code segment is loaded into a fixed area of memory when the process starts.
  - **Access**: The CPU fetches instructions from this segment to execute them.

- Example

Consider the following simple C program:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

When compiled, the machine code corresponding to the `main` function and the `printf` call resides in the code segment.

02 Data Segment

- Description

The **data segment** contains global and static variables that are initialized by the programmer. Unlike the code segment, this area is writable, allowing for modifications during execution.

- Storage

- **Location:** It is typically located also located immediately after the code segment in memory.
- **Initialized vs. Uninitialized**: This segment can be further divided into initialized data (e.g. `int a = 10;`) and uninitialized data (e.g. `static int b;`).

- Example

In the following code,

```c
int globalVar = 5; // Initialized global variable
static int staticVar; // Uninitialized static variable

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

`globalVar` gets stored in the initialized part of the data segment, while `staticVar` is stored in the uninitialized part.

03 Heap Segment

- Description

The **heap segment** is used for dynamic memory allocation. This is where memory is allocated at runtime using functions like **malloc**() in C, or **new** operator in C++ and Java. The size of the heap can grow or shrink as needed, depending on the program's requirements.

- Storage

- **Location:** The heap grows upwards, meaning that as memory is dynamically allocated (e.g., using **malloc** in C or **new** in C++), it occupies higher memory addresses.
- **Management:** The programmer is responsible for managing memory in this segment, including freeing allocated memory to avoid memory leaks.

> NOTE
> There is an issue called **memory leak**, it happens when an allocated memory address is no longer needed but was not deallocated. It decreases the performance of the computer since there is less space to be used in the memory.

> NOTE
> Some high-level programming languages implement a garbage collector, which automatically manages memory by reclaiming unused memory spaces, thereby reducing the risk of memory leaks.

- Example

Consider the following example where memory is allocated dynamically:

```
01  #include <stdlib.h>
02  #include <stdio.h>
03
04  int main() {

// Allocating memory for an array of 10 integers

05      int *array = malloc(10 * sizeof(int));
06
07      for (int i = 0; i < 10; i++) {
08          array[i] = i * 2;

// Print array elements

09          printf(“ Array[%d]: %d \n”,i,array[i]);
10      }
11


// Freeing allocated memory

12      free(array);
13      return 0;
14  }
```

In this example, the memory for the array is allocated on the heap since we determined its size at runtime.

04 Stack Segment

Description

The **stack segment** is used for function call management, including local variables and function parameters. Each time a function is called, a new stack frame is created to hold its local data and return address.

Storage

- **Location**: The stack grows downwards from higher memory addresses towards lower ones.
- **Automatic Management**: Memory on the stack is automatically managed; it is allocated when a function is called and deallocated when the function exits.

Example

Consider a function with local variables:

```
01  #include <stdio.h>
02
03  void myFunction() {
// Local variable stored on the stack
04      int localVar = 10;
05      printf("%d\n", localVar);
06  }
07
08  int main() {
09      myFunction();
// Calling the function
10      return 0;
11  }
```

In this example, **localVar** is stored in the stack segment, and its memory is automatically freed when **myFunction** returns.

> NOTE
> **Code Segment** is specifically designed to hold executable instructions (machine code) of the program. The code segment is where all function definitions, including main, are stored in a read-only format.

> NOTE
> The **Stack** is used for managing function calls, local variables, and return addresses during program execution. It is a temporary storage area for data that needs to be accessed quickly and is automatically managed by the operating system.

Step 03 Example

This example is just to demonstrate how global and local data are stored. As we said, local variables are stored in the stack which goes from high memory addresses to low memory addresses. So, if we declare two local variables, the first one will take a high memory address and the second will take the lower one.

```
01  #include <stdio.h>
02
03
04  int main(){
05      char a = ‘a’;
06      char b = ‘b’;
07
08      printf(“the address of a is: %p \n”, &a);
09      printf(“the address of b is: %p \n”, &b);
10  }
```

OUTPUT

the address of a is: 0x16f2832bf

the address of b is: 0x16f2832be

But, global variables are stored in the data segment, which means the order will be reversed. The first variable created will take the first available free space and the second will take the next address as the following.

```
01  #include <stdio.h>
02
03      char a = ‘a’;
04      char b = ‘b’;
05
06  int main(){
07
08      printf(“the address of a is: %p \n”, &a);
09      printf(“the address of b is: %p \n”, &b);
10  }
```

OUTPUT

the address of a is: 0x100734000

the address of b is: 0x100734001

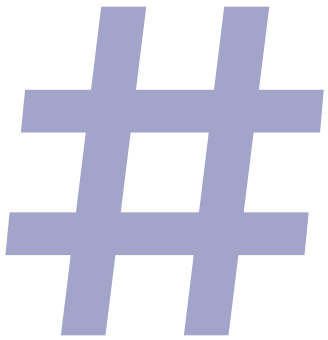
Step 04 Summary

- **Code Segment:** Contains the executable code, which is read-only during execution.
- **Data Segment:** Holds global and static variables; writable.
- **Heap Segment:** Used for dynamic memory allocation; managed by the programmer.
- **Stack Segment:** Manages function calls and local variables; automatically managed.

Understanding how these segments are stored and managed is crucial for efficient programming and memory management in any application.

01 Example

```
#include <stdio.h>

 // Data Segment (global variable)
int globalCount = 10;      // Data Segment (global variable)

void displayValues() {
 // Data Segment (static variable)
    static int staticCount = 5;
    int localCount;               // Stack (local variable)

    printf("Global Count: %d\n", globalCount);      // Code Segment
    printf("Static Count: %d\n", staticCount);      // Code Segment
    printf("Local Count: %d\n", localCount);        // Code Segment
}

int main() {
 // The function definition will be in the Code Segment but
  the execution context (variables and return address) are in the Stack segment
    displayValues();
    return 0;
 // Code Segment (return statement)
    }
```



Memory Management

Step 01 Memory Management

As described earlier, main memory is a collection of cells, each with its unique address. Memory is typically organized in such a way that each memory address corresponds to a single byte. However, data types in programming languages can occupy multiple bytes, depending on their type's size. Understanding these sizes is crucial for efficient memory management and data manipulation in programming.

> NOTE
> Computers differ in how they store and manage data, typically using a unit called a **word.** A word can consist of a single or multiple bytes, depending on the architecture of the computer. Common word sizes are 16 bits (2 bytes), 32 bits (4 bytes), or 64 bits (8 bytes)

Step 02 Data Types

Data types in programming languages specify how much space is needed to store a value.

- **Char:** 1 byte
- **Short:** 2 bytes
- **Int:** 4 bytes
- **Long:** 4 bytes or 8 bytes
- **Float:** 4 bytes
- **Double:** 8 bytes
- **Pointer:** Varies based on architecture—4 bytes on 32-bit systems and 8 bytes on 64-bit systems

> NOTE
> These sizes do not apply to all architectures.

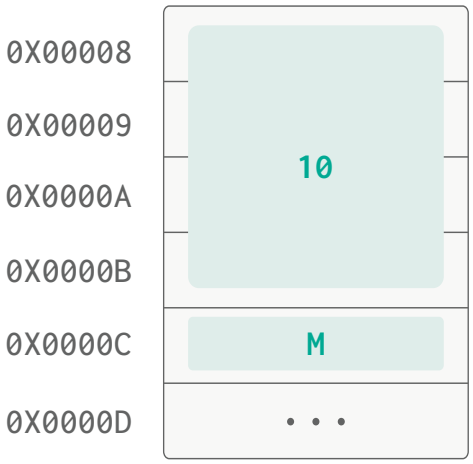
We need to understand these types to better utilize memory and understand its behavior in certain cases.

Let's take an example of variables in C:

```
01
02  int main(){
03      int num = 10;
04      char ch = 'M';
05      return 0;
06  }
```

What we did here is create two variables that are going to be stored in memory specifically in the stack segment since they are local variables for the main function.

The first variable will reserve 4 bytes to store the value **10** and the second will use 1 byte to store the value **M** in memory.



Step 03 Pointers

Pointers are variables that can store memory addresses. C provides us with the flexibility of managing pointers.

Let us take the example from earlier.

```
01  int main(){
02      int num = 10;
03      char ch = ‘M’;
04      return 0;
05  }
```

Let's create a pointer that points to the address of the variable **num** and prints its value.

```
#include <stdio.h>

int main(){
    int num = 10;
    char ch = 'M';
    int* pointer = &num;
    printf("The address of num is: %p \n", &num );
    printf("The pointer value is: %p \n",pointer);
    return 0;
}
```

OUTPUT

```
The address of num is: 0x16cea72c8
The pointer value is: 0x16cea72c8
```

The ampersand symbol means the "Address of". So, when we say `&num` we mean the address that variable num is stored in.

The `star *` on the other hand means a pointer that will store a memory address.

You can read the data in a memory address using the **dereference operator** as the following.

```

#include <stdio.h>

int main(){
    int num = 10;
    char ch = 'M';
    int* pointer = &num;
    printf("The address of num is: %p \n", &num );
    printf("The pointer value is: %p \n",pointer);
    printf("The value stored in the pointer address is: %d \n",*pointer);

    return 0;
}
```

OUTPUT

```
The address of num is: 0x16d2272c8
The pointer value is: 0x16d2272c8
The value stored in the pointer address is: 10
```

As you noticed, the value stored in the address the pointer references to is 10.

01 Increment and Decrement

You can apply arithmetic operations on pointers, let us have the same example with some changes as the following.

```
#include <stdio.h>

int main(){
    int num = 10;
    char ch = ‘M’;
    int* numPointer = &num;
    char* chPointer = &ch;
    printf(“The address of num is: %p \n”, numPointer );
    printf(“The address of ch is: %p \n”, chPointer );

    return 0;

}
```

OUTPUT

The address of num is: 0x16cf572b8

The address of ch is: 0x16cf572b7

You can do the following on the pointers

```c
#include <stdio.h>


int main(){
    int num = 10;
    char ch = 'M';
    int* numPointer = &num;
    char* chPointer = &ch;
    printf("The address of num is: %p \n", numPointer );
    printf("The address of num + 1 is: %p \n", numPointer+1 );
    printf("The address of num - 1 is: %p \n", numPointer-1 );
    printf("The address of ch is: %p \n", chPointer );
    printf("The address of ch + 1 is: %p \n", chPointer+1 );
    printf("The address of ch - 1 is: %p \n", chPointer-1 );

    return 0;
}
```

OUTPUT

```
The address of num is: 0x16b2632b8
The address of num + 1 is: 0x16b2632bc
The address of num - 1 is: 0x16b2632b4
The address of ch is: 0x16b2632b7
The address of ch + 1 is: 0x16b2632b8
The address of ch - 1 is: 0x16b2632b6
```

As you may noticed, the increment and decrement depend on the size of the data type the pointer refers to. If char, then it will increment and decrement by 1 byte. If an integer, it will use 4 bytes.

This can help us to access other data in the program without referencing them directly, let's have an example.

```c
#include <stdio.h>


int main(){
    int num = 10;
    int nextNum = 20;
    int* numPointer = &num;
    int* nextNumPointer = &nextNum;
    printf("The address of num is: %p \n", numPointer );
    printf("The address of next num is: %p \n", nextNumPointer );

    return 0;
}
```

OUTPUT

```
The address of num is: 0x16f4e32a8
The address of the next num is: 0x16f4e32a4
```

If we decrement the num address by one, we will get the nextNum address

```c
#include <stdio.h>


int main(){
    int num = 10;
    int nextNum = 20;
    int* numPointer = &num;
    int* nextNumPointer = &nextNum;
    printf("Addresses: \n");
    printf("The address of num is: %p \n", numPointer );
    printf("The address of num - 1 is: %p \n", numPointer-1 );
    printf("The address of next num is: %p \n", nextNumPointer );

    printf("Values: \n");
    printf("The value of num is: %d \n", *numPointer );
    printf("The value of num - 1 is: %d \n", *(numPointer-1) );
    printf("The value of next num is: %d \n", *nextNumPointer );


    return 0;
}
```

OUTPUT

```
Addresses:
The address of num is: 0x16f56b2a8
The address of num - 1 is: 0x16f56b2a4
The address of the next num is: 0x16f56b2a4
Values:
The value of num is: 10
The value of num - 1 is: 20
The value of the next num is: 20
```

We noticed that we accessed the next num value by only understanding how memory maps its local variables and the use of pointers.

Step 04 The Class

You may know what a class is, but we will walk through the concept from the memory perspective.

A class is a way to construct a new type, you can determine the structure of your type by combining the types defined in the language.

For example, constructing the **Person** type in C++.

```cpp
#include <iostream>
#include <string>

class Person {
public:
    int age;
    std::string name;
};

int main() {
    Person s; // Create an instance of Person
    s.name = “Sara”; // Set the name
    s.age = 17;      // Set the age

    // Output the person’s details
    std::cout << s.name << “ is “ << s.age << “ years old.” << std::endl;

    return 0;
};
```

A newly created Person will have the below layout.

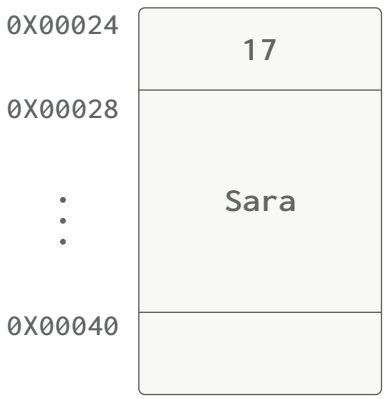
Lets print the `sizeof` each value as the following.

```
#include <iostream>
#include <string>

class Person {
public:
    int age;
    std::string name;
};

int main() {
    Person s; // Create an instance of Person
    s.name = “Sara”; // Set the name
    s.age = 17;      // Set the age

    std::cout << sizeof s.name << “\n”;
    std::cout << sizeof s.age << “\n”;

    // Output the person’s details
    std::cout << s.name << “ is “ << s.age << “ years old.” << std::endl;

    return 0;
};
```

OUTPUT

```
24
4
Sara is 17 years old.
```

This means the string takes 24 bytes from the memory to store its value while int takes 4 bytes.



NOTE
We assumed that each cell contains a single byte. But, some devices store word size which can be 2 bytes or more.

Step 05 The Struct

C language does not support OOP natively. But it has a concept it uses to construct user-defined types such as classes which is the **struct.**

A struct is a user-defined data type in C (also supported in C++) that allows you to group related variables of different types into a single unit. Structs are useful for organizing data into meaningful collections, making it easier to manage and manipulate complex data structures.

```c
#include <stdio.h>

struct student{
    int id;
    int age;
    char grade;
};


int main(){
    struct student sami;
    sami.id = 102;
    sami.age = 17;
    sami.grade = 'A';
    printf(" Sami's id is %d \n", sami.id);
    printf(" Sami's age is %d \n", sami.age);
    printf(" Sami's grade is %c \n", sami.grade);
    return 0;
}
```

In the above example, we have created a new struct to construct the Student type.

OUTPUT

Sami's id is 102

Sami's age is 17

Sami's grade is A

You can also use the pointer concept to create a function that takes a pointer to a student and print its information as the following.

```c
#include <stdio.h>

struct student{
    int id;
    int age;
    char grade;

};

void printInfo(struct student* s){
    printf(" The student id is %d \n", s->id);
// -> is a shorthand for (*s).id, which mean dereference and read the value of id
    printf(" The student age is %d \n", s->age);
    printf(" The student grade is %c \n", s->grade);
}


int main(){
    struct student sami;
    sami.id = 102;
    sami.age = 17;
    sami.grade = 'A';
    printInfo(&sami);

    return 0;
}
```

OUTPUT

The student id is 102

The student age is 17

The student grade is A

properties addresses as well.

```c
#include <stdio.h>

struct student{
    int id;
    int age;
    char grade;

};

void printInfo(struct student* s){
    printf(" The student id is %d \n", s->id);
    printf(" The student age is %d \n", s->age);
    printf(" The student grade is %c \n", s->grade);
}


int main(){
    struct student sami;
    sami.id = 102;
    sami.age = 17;
    sami.grade = 'A';

    printf("Address of sami %p \n", &sami);
    printf("Address of id %p \n", &sami.id);
    printf("Address of age %p \n", &sami.age);
    printf("Address of grade %p \n", &sami.grade);

    return 0;
}
```

OUTPUT

Address of sami 0x16d9972a0

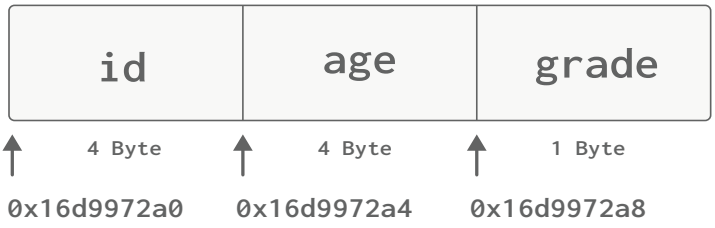
Address of id 0x16d9972a0

Address of age 0x16d9972a4

Address of grade 0x16d9972a8

As it appears from the above example, `&sami` is the same as `&sami.id`. But how did the function `printInfo()` understands where to read the data of age and grade if it only has the address of id?

What happened exactly is that the pointer refers to the first address that stores object sami in. And based on the data type (struct) it will be offset by the size of the data in each section.



01 Struct with String

The string data type is not defined in C natively, so we use a character array to represent the string data type.

> **NOTE**
> You can use the concept of struct to build your own string data type.

- Example

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

#define NAME_LENGTH 20
struct Person{
    int age;
    char name[NAME_LENGTH];
};


int main(){
    struct Person sara;
    sara.age = 17;
    strncpy(sara.name, “Sara”, NAME_LENGTH - 1);
    sara.name[NAME_LENGTH-1] = ‘\0’;
// the \0 character specifies the end of string
    printf(“ %s is %d years old”, sara.name, sara.age);
    return 0;
}
```

> **NOTE**
> In C, you cannot directly assign a string literal to a character array after its declaration because arrays in C are not assignable. Instead, you must use a function like strcpy or strncpy to copy the string into the array.

> **NOTE**
> The \0 character is known as the null character in C and C++. It serves as a string terminator, indicating the end of a string in memory.

Step 06 Arrays

Arrays are collections of data in one container, in C arrays can be defined as the following.

```c
#include <stdio.h>

int main(){
    int numbers[] = {1,2,3};
    printf("%p \n", numbers);
    printf("%d \n", numbers[0]);
    printf("%d \n", numbers[1]);
    printf("%d \n", numbers[2]);

    return 0;
}
```

OUTPUT

```
0x16bc5f2a8
1
2
3
```

As you noticed, when printing `numbers` array we see an address. It is the address of the start of the array. Let us print other elements' addresses.

```c
#include <stdio.h>

int main(){
    int numbers[] = {1,2,3};
    printf("%p \n", numbers);
    printf("%p \n", &numbers[0]);
    printf("%p \n", &numbers[1]);
    printf("%p \n", &numbers[2]);

    return 0;
}
```

OUTPUT

```
0x16d7b32a8
0x16d7b32a8
0x16d7b32ac
0x16d7b32b0
```

The address of `numbers[0]` is **0x16d7b32a8**, and if we offset by one (size of an int) it will give us

the address of `numbers[1]` which is **0x16d7b32ac**.

We conclude that in order to get the next element, we will offset it by the size of the data type in the array.

> NOTE
> The array in this example is stored in the stack since it is declared inside a function.

> NOTE
> Arrays, regardless of where they are allocated (stack or heap), will have their elements in contiguous memory. Thus, the memory addresses for the elements of the array are sequentially higher.

Step 07 malloc & free

We previously introduced the concept of system calls (syscalls), which are services that can be requested from the operating system to perform specific actions. One of these services related to memory management is the allocation and deallocation of memory spaces as needed. To facilitate this, programming languages such as C provide wrapper functions that are responsible for making these calls according to the correct protocols. In C, to allocate memory space, we use **malloc**, and to deallocate the memory that has been reserved, we use **free.**

Using **malloc**, we can create a dynamic array that is stored in the heap, even if the allocation occurs within a function. This is because **malloc** is called at runtime and requests memory space from the operating system, allowing for flexible sizing. Unlike stack-allocated arrays, which are automatically deallocated when the function exits, heap-allocated arrays persist until they are explicitly freed using **free**. This difference in lifetime gives developers more control over memory management but also requires careful handling to avoid memory leaks.

Let's have an example, of creating a dynamic array that will be stored in the heap.

```c
#include <stdio.h>
#include <stdlib.h> //used to call malloc and free

int main() {
    // Number of elements
    int n = 5;

    // Allocate memory for an array of integers
    int* array = malloc(n * sizeof(int));

    // Check if memory allocation was successful
    if (array == NULL) {
        puts("Memory allocation failed\n");
        return 1; // Exit with error
    }

    // Initialize the array
    for (int i = 0; i < n; i++) {
        array[i] = i + 1; // Assigning values to the array
    }

    // Print the values in the array
    printf("Array elements:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Deallocate the memory
    free(array);
    array = NULL; // Set pointer to NULL after freeing

    return 0; // Successful exit
}
```

OUTPUT

Array elements:

1 2 3 4 5

Let us print the value of an element address before and after the free function.

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

int main() {
    int n = 5;

    int* array = malloc(n * sizeof(int));

    if (array == NULL) {
        puts("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        array[i] = i + 1;
    }

    printf("Array elements:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    int* ptr = &array[1];  // create a pointer to the second element
    printf("%d \n", *ptr); // print the value before freeing the memory space

    free(array);
    printf("%d \n", *ptr); // print the value after freeing the memory space

    array = NULL;


    return 0;
}
```

OUTPUT

```
1 2 3 4 5
2
0
```

Before freeing the memory space, it used to store the number 2 as a value.

> **NOTE**
> Wrapper functions are functions that serve as intermediaries between a client (such as a program or module) and a more complex underlying system or function. They are used to simplify the interface and improve usability.

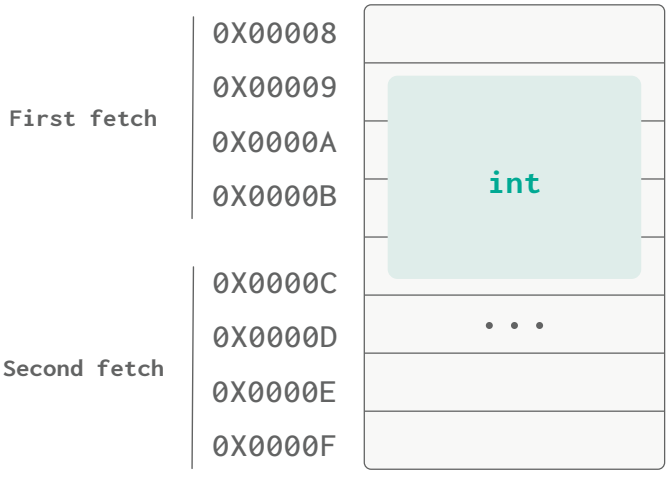
> **NOTE**
> In languages like C, functions such as **malloc** and **free** act as wrappers that interact with the operating system to manage memory.

Step 08 Alignment and Padding

Memory alignment and padding are essential concepts in computer architecture that affect data storage and access efficiency. Alignment refers to arranging data at memory addresses that are multiples of a specific size, typically dictated by the data type and architecture's word size, which enhances CPU retrieval speed. Padding involves adding extra bytes to data structures to ensure proper alignment of their members, potentially increasing memory usage. Understanding these concepts is crucial for developers aiming to optimize performance in systems programming and embedded systems.

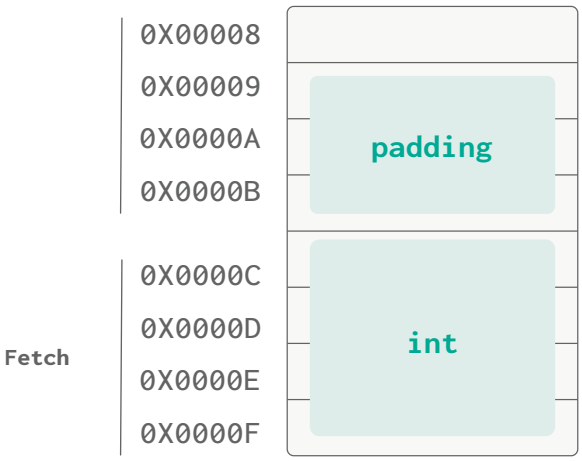
> **NOTE**
> Word size is the size of bytes the computer uses for processing data.

In a 4 byte word size (32-bit) architecture, the CPU typically fetches memory in chunks that are multiples of 4 bytes. Look at the following image.



Integers typically need 4 bytes to store their data. If the CPU reads by a multiple of 4 bytes, then it will read the integer data in two cycles, the first 4 bytes and then the second 4 bytes to complete reading the value.

To enhance the CPU performance, padding by 3 bytes will be added to align data properly.



By padding, the int value will be stored at address **0x0000C** which needs one fetch cycle to read and write.

Let's apply the concept in C. When creating a struct, we need to mind how we order variable definitions to ensure the correct alignment of data in memory.

```c
#include <stdio.h>

struct AlignmentExample {
    char a;      // 1 byte
    int b;       // 4 bytes
    char c;      // 1 byte
};

int main() {
    struct AlignmentExample example;
    printf("Size of the struct: %lu bytes\n", sizeof(example));
    printf("Address of a: %p\n", &example.a);
    printf("Address of b: %p\n", &example.b);
    printf("Address of c: %p\n", &example.c);

    return 0;
}
```

What is the size of the struct example? It should be 6 bytes since we have an int which is 4 bytes long, and 2 characters each take 1 byte.

OUTPUT

Size of the struct: 12 bytes

Address of a: 0x16ef132b0

Address of b: 0x16ef132b4

Address of c: 0x16ef132b8

Why is the output 12 bytes? Since int b is not aligned correctly, which means it is not in an address multiple of 4 like 0, 4, 8, 16 ..., there will be padding to align it properly on a multiple of 4 address as the following.

a	padding	b	c	padding
1	3 bytes	4	1	3 bytes
0x16ef132b0	—	0x16ef132b4	0x16ef132b8	—

As you noticed, we took more space than we needed. It might not be an issue right now, but assume we have an array of the above struct? This will lead to increased memory usage and potential inefficiencies.

We need to know how to align our data to ensure optimal solutions. Here is where alignment comes in to help. If we reordered the variable creations in the struct by letting the int take the first multiple of 4 address as the following.

```c
#include <stdio.h>

struct AlignmentExample {
    int b;       // 4 bytes
    char a;      // 1 byte
    char c;      // 1 byte
};

int main() {
    struct AlignmentExample example;
    printf("Size of the struct: %lu bytes\n", sizeof(example));
    printf("Address of a: %p\n", &example.a);
    printf("Address of b: %p\n", &example.b);
    printf("Address of c: %p\n", &example.c);

    return 0;
}
```

We will get to enhance the size needed as the output shows.

OUTPUT

Size of the struct: 8 bytes

Address of a: 0x16d7a32b8

Address of b: 0x16d7a32b4

Address of c: 0x16d7a32b9

b	a	c	padding
4 byte	1 byte	1 byte	2 byte
0x16b6c32b4	0x16d7a32b8	0x16d7a32b9	—

The amount of trailing padding is determined by the largest alignment requirement of any member within the struct. For example, if the largest member is of type int (which typically requires 4-byte alignment), padding will be added at the end of the struct to ensure that the total size is a multiple of 4 bytes. This way, when an array of such structs is created, each element will also be aligned correctly.