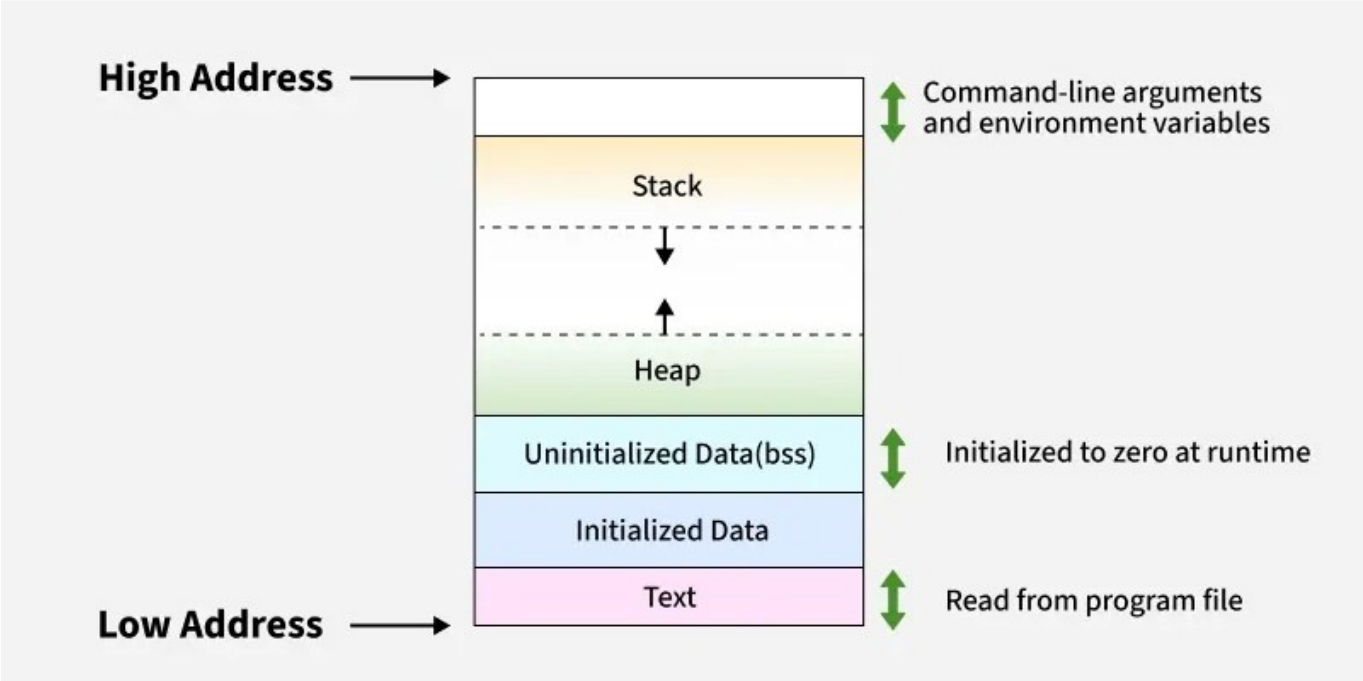
C++程式設計：記憶體簡介(以 c 語言示範)

目錄

1. Memory 架構

C 程式的記憶體被組織成特定的區域 (段)，如下圖所示，每個區域 (段) 在程式執行時都有不同的用途。



2. Text Segment

Text Segment 是用來儲存程式的可執行程式碼的記憶體區域。它包含了經過編譯後的機器碼，也就是你在程式中寫的函式、流程控制語句等，最終轉換成 CPU 能執行的指令。這個區段通常是唯讀的（Read-Only），放在記憶體的較低位址區域，以避免在程式執行期間被不小心修改，保障系統安全性與穩定性。

在 Linux 等作業系統中，這個區段的保護是由硬體（如記憶體管理單元 MMU）與作業系統共同實作的，確保程式碼執行時不會因錯誤或惡意程式修改而出錯。

```c
#include <stdio.h>

void say_hello() { // Text Segment
    printf("Hello, world!\n"); // Text Segment
}

int main() { // Text Segment
    say_hello();
    return 0;
}
```

3. Data Segment

Data Segment 主要儲存已由程式設計師初始化的全域變數（global variables）與靜態變數（static variables）。這些變數在程式執行期間會一直存在於記憶體中，不會因為函式結束而消失。

```c
int a = 10;       // 已初始化的全域變數，放在 Data Segment
static int b = 20; // 已初始化的靜態變數，也放在 Data Segment
```

4. 未初始化資料區段（BSS, Block Started by Symbol）

BSS 區段用來存放未經程式設計師初始化的全域變數與靜態變數。但這些變數並沒有被賦值。會由作業系統自動初始化為 0（或等價的空值）。

```c
int a;        // 全域變數，沒有初始化 → 放在 BSS 區段
static int b; // 靜態變數，沒有初始化 → 放在 BSS 區段
```

5. 堆區（Heap Segment）

Heap Segment 主要用來動態配置記憶體。動態記憶體的特點是大小在編譯時無法確定，而是在程式執行過程中決定。

Heap Segment 從 BSS 區段的結尾開始，向高位址方向成長。成長空間大小不固定，可在程式執行時透過動態記憶體管理函式進行分配或釋放。

5.1 常用的函式：

- malloc()：配置一塊指定大小的記憶體。
- realloc()：調整已配置記憶體的大小。

- free()：釋放記憶體。

這些函式在底層可能會使用系統呼叫 brk() 與 sbrk() 來調整堆區大小。

```
int *ptr = (int*) malloc(sizeof(int) * 10); // 動態配置一塊可以存放 10 個 int 的記憶體空間
```

6. Stack Segment

Stack Segment 專門用來儲存區域變數（local variables）與函式呼叫相關資訊的區域。它的運作方式就像「堆疊」（Stack）資料結構——後進先出（LIFO, Last In First Out）。Stack Segment 從高位址向低位址成長（與 Heap 相反）。

用途：

- 存放區域變數（local variables）
- 存放函式參數（function parameters）
- 存放返回位址（return address）

建立與釋放

- 當呼叫一個函式時，系統會建立一個「堆疊框架（stack frame）」，用來保存該函式的區域變數、參數與返回位址。
- 當函式結束時，對應的堆疊框架會被自動移除，所佔用的記憶體也會釋放。

```
void function() {
    int local_var = 10;  // 存放在堆疊區
}

// local_var 是一個區域變數，因此會被儲存在該函式的堆疊框架中。
// 當 function() 執行完畢後，local_var 對應的記憶體空間會被自動釋放。
```

7. Dynamic Memory Allocation in C

7.1 malloc()：Memory Allocation

功能：從 Heap 區域配置一塊指定大小的記憶體，內容不初始化。

size：要分配的記憶體大小（單位：bytes）

傳回值：void pointer（需要轉型）

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

int main() {
    // 宣告 5 個 int 空間
    int *arr = (int*) malloc(5 * sizeof(int));

    arr[0] = 10; // 可以直接存取
    printf("%d\n", arr[0]); // 10
    free(arr); // 記得釋放記憶體
    return 0;
```

```
}
```

7.2 calloc()：Contiguous Allocation

功能：從 Heap 區域配置一塊記憶體並全部初始化為 0。

size：要分配的記憶體大小（單位：bytes）

傳回值：void pointer（需要轉型）

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

int main() {
    // 分配 5 個 int 空間且每個都初始化為 0
    int *arr = (int*) calloc(5, sizeof(int));

    printf("%d\n", arr[0]); // 0
    free(arr); // 記得釋放記憶體
    return 0;
}
```

7.3 free() ：Memory Deallocation

功能：釋放之前用 malloc() 或 calloc() 分配的記憶體。

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

int main() {
    int *arr = (int*) malloc(5 * sizeof(int));
    if (arr == NULL) return 1;
    free(arr);   // 釋放記憶體
    return 0;
}
```

7.4 realloc()：Resize Allocated Memory

調整之前用 malloc()、calloc() 分配的記憶體大小。

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

int main() {
    int *arr = (int*) malloc(3 * sizeof(int)); // 原本分配 3 個
```

```
    if (arr == NULL) return 1;

    arr = (int*) realloc(arr, 5 * sizeof(int)); // 重新分配成 5 個
    if (arr == NULL) return 1;

    arr[3] = 40;
    arr[4] = 50;
    printf("%d %d\n", arr[3], arr[4]); // 40 50

    free(arr);
    return 0;
}
```

8. Memory Leak

記憶體洩漏（Memory Leak）發生在：

- 程式使用 malloc()、calloc()、或 realloc() 分配記憶體後，
- 忘記用 free() 釋放，導致記憶體空間無法回收，造成浪費。

```
// memory leak
void f() {
    int* ptr = (int*)malloc(sizeof(int));
    return; // 忘記釋放 ptr
}
```