Lec 16 Dynamic Memory Allocation

1. 背景与问题

动态内存分配是为了解决在程序运行时数组大小无法提前确定的问题。例如：

- 学生记录数组（`student`结构体）的数量可能在学期加/退课期间动态变化。
- 静态分配（如运行时栈上的数组）无法调整大小，而动态内存分配可以在堆（heap）上灵活分配和释放内存。
- 目标：实现数组大小的动态调整（如添加100名学生或移除50名学生），并支持复杂操作（如按字母顺序排序，需结合后续课程中的链表）。

示例问题：

- 学生记录结构体定义：

```
typedef struct studentstruct {
    char *NAME;  // 动态分配的姓名字符串
    int UIN;     // 学生ID
    float GPA;   // 平均成绩
} student;
```

- 需要在运行时根据输入决定学生数量，并支持增删操作。

2. 动态内存分配的基本概念

动态内存分配通过操作系统提供的**内存分配管理器**在**堆（heap）**上分配内存，区别于：

- **静态存储**（全局数据区，固定大小，程序整个生命周期存在）。
- **自动存储**（运行时栈，函数调用时分配，函数结束时自动释放，无法调整大小）。

堆的特点：

- 程序运行时通过请求分配内存，分配后返回指针。
- 程序员负责管理内存的分配（malloc、calloc）和释放（free）。
- 内存分配是连续的，但需要注意内存泄漏和非法访问。

与栈的对比：

特性	自动分配（栈）	动态分配（堆）
分配方式	自动（函数作用域）	手动（`malloc`/`calloc`）
大小调整	不可调整	可调整（`realloc`）
内存位置	运行时栈	堆
释放方式	自动（函数结束）	手动（`free`）

3. 动态内存分配的核心函数

C语言中用于动态内存分配的三个主要函数：`malloc`、`free`、`calloc`，以及辅助函数`realloc`。

3.1 `malloc`

- **功能**：在堆上分配指定字节数的内存，返回指向该内存的指针。
- **原型**：

```
void *malloc(size_t size);
```

  - `size`：要分配的字节数。
  - 返回：`void*`类型指针，需转换为目标类型；若失败，返回`NULL`。
- **特点**：
  - 分配的内存未初始化，包含随机数据。
  - 需要检查返回指针是否为`NULL`以确保分配成功。
- **示例**：

```
char *name = (char *)malloc(22 * sizeof(char)); // 分配22字节给字符串
if (name == NULL) {
    printf("Memory allocation failed\n");
    exit(1);
}
student *student_list = (student *)malloc(22 * sizeof(student)); // 分配22个student结构体
```

3.2 `free`

- **功能**：释放由`malloc`、`calloc`或`realloc`分配的内存。
- **原型**：

```
void free(void *ptr);
```

  - `ptr`：必须是`malloc`等函数返回的指针，或`NULL`（对`NULL`无操作）。
  - 无返回值。
- **注意事项**：
  - 重复释放同一指针（double free）会导致未定义行为。
  - 释放非`malloc`返回的指针会导致错误。
  - 释放后应将指针置为`NULL`以避免野指针问题。
- **示例**：

```
free(name); // 释放name指向的内存
```

3.3 `calloc`

- **功能**：分配内存并初始化为0，适合需要清零的场景。
- **原型**：

```
void *calloc(size_t n_items, size_t item_size);
```

  - `n_items`：分配的元素数量。
  - `item_size`：每个元素的大小（字节）。
  - 返回：`void*`指针，若失败返回`NULL`。
- **特点**：
  - 与`malloc`的区别：内存初始化为0。
  - 常用于数组或结构体初始化。
- **示例**：

```
int *ptr = (int *)calloc(5, sizeof(int)); // 分配5个整数，初始化为0
if (ptr == NULL) {
    printf("Allocation Failed\n");
    exit(1);
}
for (int i = 0; i < 5; i++) {
    printf("%d ", ptr[i]); // 输出：0 0 0 0 0
}
free(ptr);
```

3.4 `realloc`

- **功能**：调整已分配内存的大小，可能移动内存块。
- **原型**：

```
void *realloc(void *ptr, size_t new_size);
```

  - `ptr`：已有内存块的指针（或`NULL`，相当于`malloc`）。
  - `new_size`：新的内存大小。
  - 返回：新内存块的指针，可能与原指针不同。
- **用途**：
  - 解决动态数组大小调整问题（如添加100名学生或减少50名学生）。
- **示例**：

```
student *new_list = (student *)realloc(student_list, new_size * sizeof(student));
if (new_list == NULL) {
    printf("Reallocation failed\n");
    exit(1);
}
student_list = new_list; // 更新指针
```

4. 动态内存分配的常见问题与解决方案

4.1 内存泄漏（Memory Leak）

- **问题**：分配的内存未被释放，导致程序占用内存不断增加。
- **示例**：

```
int *ptr1 = (int *)malloc(sizeof(int));
*ptr1 = 10;
int *ptr2 = (int *)malloc(sizeof(int));
*ptr2 = 5;
ptr1 = ptr2; // ptr1原指向的内存失去引用，无法释放
free(ptr2);  // 只释放了ptr2指向的内存
```

  - 结果：`ptr1`原指向的内存未释放，造成内存泄漏。

- **解决方案**：
  - 释放所有分配的内存：

```
free(ptr1); // 先释放ptr1
ptr1 = ptr2;
free(ptr2); // 再释放ptr2
```

4.2 双重释放（Double Free）

- **问题**：同一内存块被多次释放，导致未定义行为。
- **示例**：

```
int *ptr1 = (int *)malloc(sizeof(int));
int *ptr2 = (int *)malloc(sizeof(int));
ptr1 = ptr2;
free(ptr2);
free(ptr1); // 错误：ptr1和ptr2指向同一块内存
```

  - 结果：运行时错误。

- **解决方案**：
  - 确保每个内存块只释放一次。
  - 释放后将指针置为`NULL`。

4.3 分配失败

- **问题**：内存不足时，`malloc`或`calloc`返回`NULL`。
- 解决方案：
  - 始终检查返回值：

```
int *ptr = (int *)malloc(sizeof(int));
if (ptr == NULL) {
    printf("Error - malloc failure\n");
    return -1;
}
```

5. 动态内存分配的实际应用

5.1 动态创建学生记录数组

- **代码示例**：

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

typedef struct studentstruct {
    char *NAME;
    int UIN;
    float GPA;
} student;

void print_data(student *list, int n) {
    for (int i = 0; i < n; i++) {
        printf("Student %d: Name=%s, UIN=%d, GPA=%.2f\n",
               i, list[i].NAME, list[i].UIN, list[i].GPA);
    }
}

int main() {
    int no_of_student;
    printf("Please enter no_of_students: ");
    scanf("%d", &no_of_student);

    // 动态分配学生数组
    student *student_list = (student *)malloc(no_of_student * sizeof(student));
    if (student_list == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // 为每个学生的NAME分配内存
    for (int i = 0; i < no_of_student; i++) {
```

```
        student_list[i].NAME = (char *)malloc(20 * sizeof(char));
        if (student_list[i].NAME == NULL) {
            printf("Memory allocation for name failed\n");
            return 1;
        }
        strcpy(student_list[i].NAME, "To be set");
        student_list[i].UIN = -1;
        student_list[i].GPA = 0.0;
    }

    // 打印数据
    print_data(student_list, no_of_student);

    // 释放内存
    for (int i = 0; i < no_of_student; i++) {
        free(student_list[i].NAME);
    }
    free(student_list);

    return 0;
}
```

- 改进点：
  - 原代码使用栈上数组`char name[no_of_student][20]`，不灵活，且未释放内存。
  - 修正版使用`malloc`为每个`NAME`动态分配内存，并确保释放。

5.2 动态调整数组大小

- 问题：
  - 如何添加100名学生?
  - 如何移除50名学生?
- **解决方案**：
  - 使用`realloc`调整数组大小：

```
// 添加100名学生
int new_size = no_of_student + 100;
student *new_list = (student *)realloc(student_list, new_size * sizeof(student));
if (new_list == NULL) {
    printf("Reallocation failed\n");
    return 1;
}
student_list = new_list;
no_of_student = new_size;
// 初始化新添加的学生记录
for (int i = no_of_student - 100; i < no_of_student; i++) {
    student_list[i].NAME = (char *)malloc(20 * sizeof(char));
    strcpy(student_list[i].NAME, "To be set");
    student_list[i].UIN = -1;
```

```
    student_list[i].GPA = 0.0;
}
```

7. 总结与注意事项

- 动态内存分配的核心：
  - 使用`malloc`/`calloc`分配，`free`释放，`realloc`调整大小。
  - 堆上内存由程序员手动管理，灵活但易出错。
- 关键注意事项：
  - 始终检查分配是否成功（`NULL`）。
  - 避免内存泄漏（未释放）和双重释放。
  - 动态分配的内存需要手动初始化（`malloc`不初始化，`calloc`初始化为0）。
  - 对于复杂操作（如排序、频繁增删），考虑使用链表而非数组。
- 后续学习：
  - 链表（Linked Lists）：解决数组动态调整的低效问题，支持按序插入等操作。