数据结构实验（1）

C++ 内存的学问

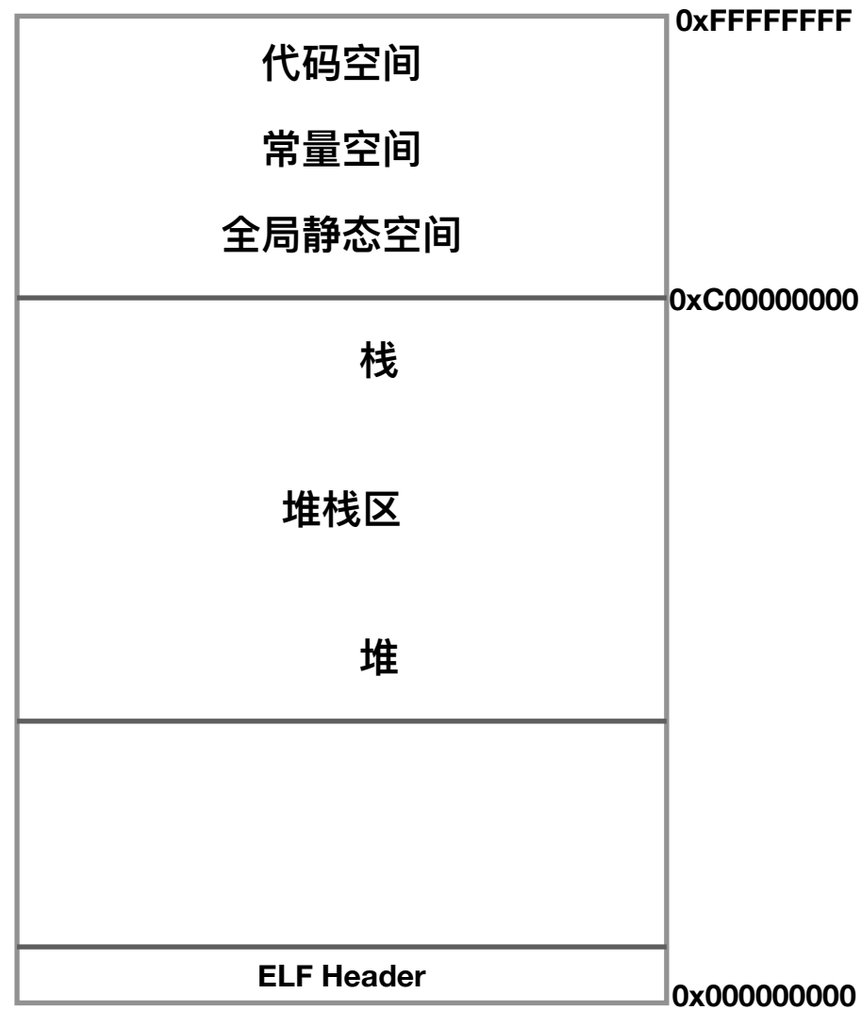
目录

前情提要

- POSIX 标准的操作系统
- 提问的技巧
- C++ 里函数重名了怎么办?

堆空间、栈空间

- 一个由 C++ 编译的程序占用的内存分为以下几个“逻辑”部分
  - 栈空间：
    - 函数参数、局部变量，由编译器自动分配释放
    - 其操作方式类似于数据结构中的栈（Stack）
  - 堆空间：
    - 变量，程序员主动管理其生命周期（分配/释放）；或程序结束时由操作系统全部释放
    - 其操作方式类似于类似于数据结构中的链表（List）
- 全局静态空间：
  - 全局变量、class/struct 静态变量，由编译器自动分配，已初始化的和未初始化的分开
- 常量空间：
  - 常量字符串，由编译器自动分配
- 代码空间：
  - 编译后的二进制代码

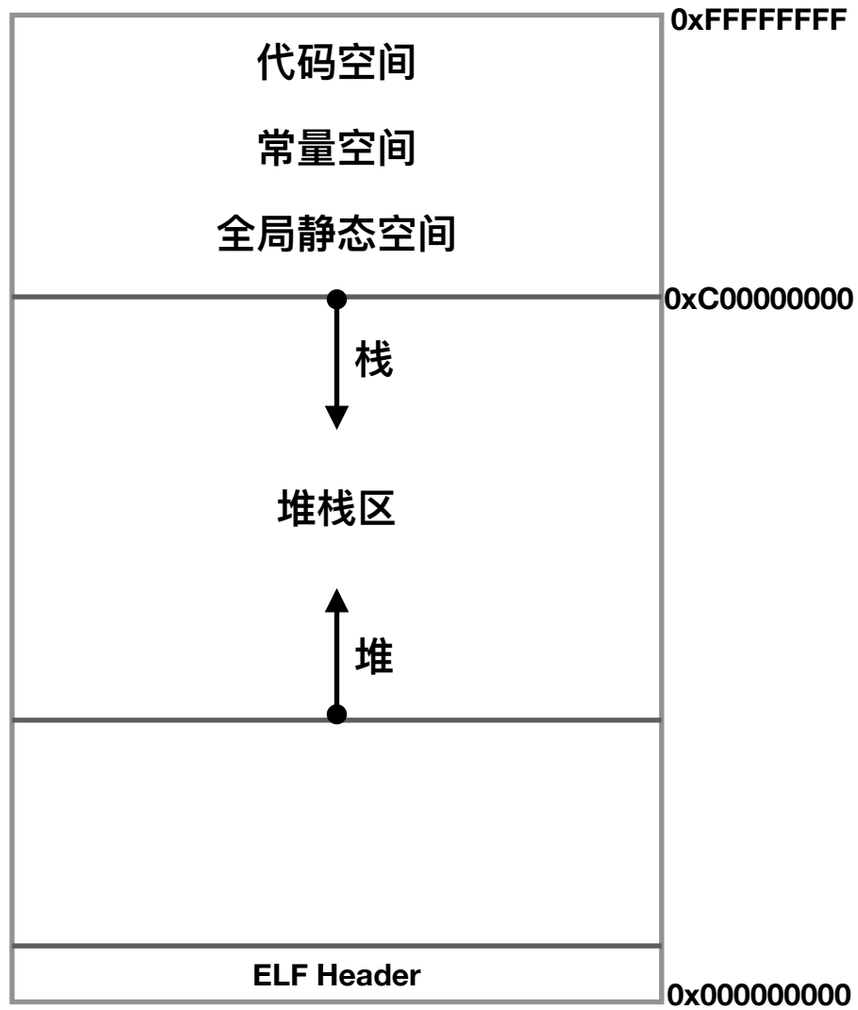


C/C++ 程序虚拟地址空间

```cpp
int k = 0;                                          //  静态空间内始化区
char *str;                                          //  静态空间内未初始化区

struct A {
    // 静态空间
    static const int OK = 0;                        //  静态空间内始化区
};

void getIntro(
        const std::string& name,                    // 栈空间
        std::string* result) {                      // 指针地址在栈空间
    std::string header = "My name is ";             // "My name is" 在常量空间, header 在栈上

    std::stringstream ss;                           // ss 在栈上, ss 内部管理的空间在堆上
    ss << header << name;

    *result = ss.str();
};

int main(int, char**) {
    std::string names[] = {"Polly", "Dummy", "Henry"};  // "Polly" 等在常量空间, names 在栈上

    char* unused = new char[10];                     // unused 在堆上分配了10个字节
    std::string* introes = new std::string[3];      // introes 在堆上, 他们管理的空间也在堆上

    for (int i = 0; i < num; ++i) {
        std::string intro;                          // intro 在栈上, intro 管理的空间在堆上
        getIntro(names[i], &intro);

        // 以下两条语句有什么区别?
        introes[i] = std::move(intro);
         // *(introes+i) = std::mvoe(intro);
    }

    // DO SOMETHIGN..

    delete[] introes;                               // 主动释放堆空间

    return 0;
}
```

堆空间、栈空间

- 分配时机与方式：
  - 栈空间：
    - 栈是一个固定的地址，向下分配：主流操作系统的栈空间一般是有限的，比如 64MB
    - 只要栈的剩余空间大于所申请空间，操作系统将全部分配给程序，否则将抛出异常
  - 堆空间：
    - 堆是一个固定的地址，向上分配：通常可以用满所有内存空间
  - 静态空间：
    - 编译器根据是否已初始化预留空间；（此处现代编译器会有不同的行为）



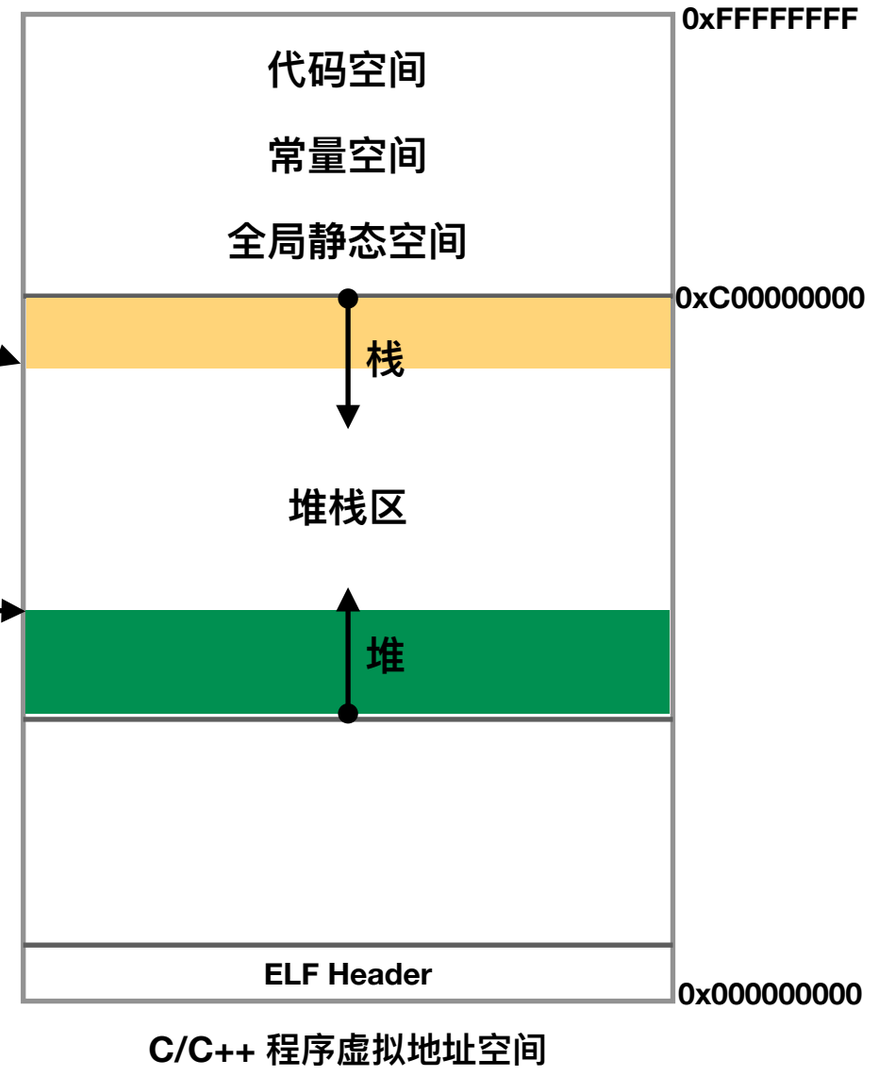
C/C++ 程序虚拟地址空间

堆空间、栈空间

- C/C++ 数组
- C++ std::array (C++11)

- C/C++ new/malloc
- C++ std::vector
- C++ std::list

- C/C++ 自定义线性表
- C/C++ 自定义链表



C/C++ 程序虚拟地址空间

堆空间、栈空间

- 使用注意
  - 栈空间是有限的，对于未知长度的空间申请，不建议在栈上申请
  - 通常使用 RAII 思想进行堆空间的管理
    - 目的：在栈空间定义变量，自动管理堆空间的内存
    - 方法：
      - 定义一个类来封装资源（内存、连接、文件操作符等）的分配与释放；
      - 类的构造函数中完成资源的分配及初始化；
      - 类的析构函数中完成资源的清理，可以保证资源的正确初始化和释放；
    - 问题：
      - **当对象需要被拷贝时，只能复制资源（深拷贝），而无法直接引用（浅拷贝）；**

```cpp
template<typename T>
struct Array {

private:
    T* elem;

public:

    Array(int n) {
        this->elem = new T[n];
    }
    ~Array() {
        delete[] this->elem;
    }

    T* get() {
        return this->elem;
    }

};

int main(int, char**) {
    // 通过 Array 对象，在堆空间创建线性表
    Array<int> arr(10);

    // 如果需要 Array<int> arr_another = arr;
    // Do something with arr_another …

    return 0;
}   // arr 会在离开作用域后自动调用析构函数释放资源
```

堆空间、栈空间

- 使用注意
  - 作用与区别：
    - 构造函数
    - 拷贝构造函数
    - 移动构造函数
    - 重载operator=
- 不要迷恋、执着于 零拷贝

```cpp
struct UserInfo {
    std::string user_id;
    cv::Mat frame;

    UserInfo() {
    }

    // 拷贝构造
    UserInfo(const UserInfo& u)
            : user_id(u.user_id),
            frame(u.frame) {
    }

    // 重载 =
    UserInfo& operator=(const UserInfo& u) {
        if (this == &u) {
            return *this;
        }
        user_id = u.user_id;
        frame = u.frame;
        return *this;
    }

    // 移动构造
    UserInfo(UserInfo&& u)
            : user_id(std::move(user_id)),
            frame(std::move(u.frame)) {
    }

    ~UserInfo() {
    }
};
```

头文件里声明的是什么

- 回忆上一次课的编译过程：
  - C++的编译模式是分别编译。编译期间每个 cpp 并不需要知道其它 cpp 存在，只有到链接阶段，才会将汇编期间形成的每个obj链接成一个 exe 或 out 文件，这个过程带来两个问题：
  - 编译期间怎么知道其它 cpp/lib 提供了什么方法?
  - 运行期间，怎么知道变量、方法（函数）的实际代码地址?

头文件里声明的是什么

a.cpp:

```
1 int a = 123;
2
```

b.cpp:

```
1 #include <cstdio>
2
3 extern int a;
4
5 int main(int, char**) {
6     printf("I know a = %d\n", a);
7 }
```

正确的编译、链接、运行

```
z@Yuhans-MacBook-Pro-2 ~ % g++ -c a.cpp -o a.o
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % g++ -c b.cpp -o b.o
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % g++ a.o b.o -o test
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % ./test
I know a = 123
```

错误的编译、链接

```
z@Yuhans-MacBook-Pro-2 ~ % g++ b.o -o test
Undefined symbols for architecture x86_64:
  "_a", referenced from:
      _main in b.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

头文件里声明的是什么

a.cpp:

```cpp
int a = 123;

int plus1(int x) {
    return x + 1;
}
```

a.h:

```cpp
#pragma once

int plus1(int);

```

b.cpp:

```cpp
#include <cstdio>

#include "a.h"

extern int a;

int main(int, char**) {
    a = plus1(a);
    printf("I know a = %d\n", a);
}
```

正确的编译、链接、运行

```
z@Yuhans-MacBook-Pro-2 ~ % g++ -c a.cpp -o a.o
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % g++ -c b.cpp -o b.o
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % g++ a.o b.o -o test
z@Yuhans-MacBook-Pro-2 ~ %
z@Yuhans-MacBook-Pro-2 ~ % ./test
I know a = 124
```

错误的编译、链接

```
z@Yuhans-MacBook-Pro-2 ~ % g++ b.o -o test
Undefined symbols for architecture x86_64:
  "plus1(int)", referenced from:
      _main in b.o
  "_a", referenced from:
      _main in b.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code
```

头文件里声明的是什么

- 头文件的作用：
  - https://www.cprogramming.com/declare_vs_define.html
- 什么情况下头文件里可以写定义：
  - const 变量
  - inline 内联方法
    - 内联的作用与成本：https://docs.microsoft.com/zh-cn/cpp/cpp/inline-functions-cpp?view=msvc-160
  - class 类方法
  - ~~template 模版方法~~

头文件里声明的是什么

- 探寻本质：
  - 动态语言（如 Python 里），对象的生命周围由 GC 管理，完全创建在堆上；一个函数可以赋值给一个变量；一个 Class 可以在运行期间给它增加一个方法（修改 meta）…
- C++ 里，Function / Struct / Class 只是一个面向程序员的概念
  - 面向过程编程 / 面向对象编程
  - 在编译好的代码中，Struct / Class 只是一小段连续的内存，访问 Struct 成员或调用函数只是是用内存地址+成员变量/函数 的偏移来完成的，这意味着编译好的代码中完全没有 Struct / Class 的概念
    - 通过函数指针、手工管理符号表、vtable 等手段，可以在 C 里实现面向对象编程

头文件里声明的是什么

- 探寻本质：
  - 那么，最有意思的事情来了：
- 推荐一个毕业设计的题目：实现一种脚本语言的虚拟机（Lua、Erlang..）
  - 实现符号表，Object，GC，各种容器…
  - 在内存中翻译各种代码，循环、switch …
  - 实现多线程

Modern C++

智能指针

- C 里，我们通常建议程序员：
  - 注意 谁申请、谁释放 的原则
  - 使用 pclint 在编译期检查内存使用
  - 使用 valgrind 在运行期检查内存使用
- C++ 里，我们建议程序员使用 RAII 管理所有动态资源
- C++11 开始，提供了两种 RAII 的标准实现：
  - std::unique_ptr<T>
  - std::shared_ptr<T>

Modern C++

智能指针

- std::unique_ptr<T>
  - 唯一的所有者
  - 显示减少不必要的内存拷贝

Modern C++

智能指针

```cpp
// std::unique_ptr 和 std::shared_ptr 在 memory.h 中定义
#include <memory>

int main(int, char**) {
    {
        // 定义了一个 unique_ptr, 指向 int 对象, 但这个时候它指向 NULL
        std::unique_ptr<int> ptr;

        // 有两种方式可以申请一个新的堆空间, 并让 ptr 指向新对象
        ptr.reset(new int);
        ptr = std::unique_ptr<int>(new int);
    }   // 在作用域离开时, ptr 会自动释放
    return 0;
}
```

Modern C++

智能指针

```cpp
#include <functional>
// std::unique_ptr 和 std::shared_ptr 在 memory.h 中定义
#include <memory>

int main(int, char**) {
    {
        // 定义了一个 unique_ptr, 指向 int 对象, 但这个时候它指向 NULL
        std::unique_ptr<int> ptr;

        // 有两种方式可以申请一个新的堆空间, 并让 ptr 指向新对象
        ptr.reset(new int);
        ptr = std::unique_ptr<int>(new int);
    }   // 在作用域离开时, ptr 会自动释放

    {
        // 定义了一个 unique_ptr, 指向 int 的 **数组**
        std::unique_ptr<int[]> ptr;
        ptr.reset(new int[100]);
    }   // 在作用域离开时, ptr 会自动释放

    {
        // 手动指定 unique_ptr 的析构函数
        std::unique_ptr<int[], std::function<void(int *)> > ptr(
                new int[10],
                [&](int *p) { delete[] p; });
    }

    return 0;
}
```

Modern C++

智能指针

```cpp
// std::unique_ptr 和 std::shared_ptr 在 memory.h 中定义
#include <memory>

int main(int, char**) {
    {
        std::unique_ptr<int> ptr;
        std::unique_ptr<int> ptr2(new int(123));

        ptr = ptr2;                     // ERROR
        ptr.reset(ptr2);                // ERROR

        ptr = std::move(ptr);           // OK
        ptr.reset(ptr2.release());      // OK
    }

    return 0;
}
```

Modern C++

智能指针

- std::unique_ptr<T> 作为函数参数：
  - 传递引用

```
// calling with dosomething(x);
void dosomething(const std::unique_ptr<X>& x_ptr);
```

  - 传递真实指针

```
// calling with dosomething(x.get())
void dosomething(const X* x_ptr);
```

- std::unique_ptr<T> 作为函数返回值：
  - 配合 std::move() 返回

Modern C++

智能指针

- std::shared_ptr<T>
  - 引用计数的必要性
  - 通常与异步编程、Lambda 语法结合使用

```
// 上传新的人脸
http_svr.onPut(
        "/api/user",
        [face_db](const FaceRecogSvr::HttpConnPtr &con) {
    FRS_TRACE("PUT /api/user");

    const FaceRecogSvr::HttpRequest& req = con.getRequest();

    FaceRecogSvr::AjaxRequest req_ajax(req);
    FRS_TRACE("AJAX req pasrsed");

    FaceRecogSvr::HttpResponse resp;
    resp.setStatus(200, "OK");

    auto user_id_it = req.args.find("user_id");
    if (user_id_it == req.args.end()) {
        resp.setStatus(400, "Url param [user_id] does not exists.");
    } else if (!req_ajax.content.isMember("image")) {
        resp.setStatus(400, "AJAX Param [image] does not exists.");
    } else {
        const std::string& user_id = user_id_it->second;
        const std::string& image_b64 =
                req_ajax.content.get("image", Json::Value("")).asString

        int res = face_db->faceCreate(image_b64, user_id, NULL);

        if (0 != res) {
            resp.setStatus(500, "Failed to create face.");
        } else {
            FRS_INFO(
                    "AJAX-API: Created user. [user_id='%s']",
                    user_id.c_str());
        }
    }

    con.sendResponse(resp);
});
```