

# 结构体、共用体与C++基础1

## 结构体、共用体与C++基础1

- 1、结构体
  - 字节对齐
- 2、共用体
- 3、C++
  - 输出
  - 函数符号兼容
  - 引用
  - 字符串
    - C字符串
    - 字符串操作
  - C++ string类
  - 命名空间

## 1、结构体

结构体是C编程中一种用户自定义的数据类型，类似于Java的javaBean

```
//Student 相当于类名
//student和a 可以不定义，表示结构变量，也就Student类型的变量
struct Student
{
    char name[50];
    int age;
} student,a;
//使用typedef定义
typedef struct{
    char name[50];
    int age;
} Student;
```

当结构体需要内存过大，使用动态内存申请。结构体占用字节数和结构体内字段有关，指针占用内存就是4/8字节，因此传指针比传值效率更高。

```
struct Student *s = (Student*)malloc(sizeof(Student));
memset(s,0,sizeof(Student));
printf("%d\n", s->age);
```

## 字节对齐

内存空间按照字节划分，理论上可以从任何起始地址访问任意类型的变量。但实际中在访问特定类型变量时经常在特定的内存地址开始访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序一个接一个地存放，这就是对齐。

字节对齐的问题主要就是针对结构体。

```

struct MyStruct1
{
    short a;
    int b;
    short c;
};
struct MyStruct2
{
    short a;
    short c;
    int b;
};
//自然对齐
//1、某个变量存放的起始位置相对于结构的起始位置的偏移量是该变量字节数的整数倍；
//2、结构所占用的总字节数是结构中字节数最长的变量的字节数的整数倍。
// short = 2 补 2
// int = 4
// short = 2 补 2
sizeof(MyStruct1) = 12
// 2个short在一起组成一个 4
sizeof(MyStruct2) = 8

```

```

#pragma pack(2) //指定以2字节对齐
struct MyStruct1
{
    short a;
    int b;
    short c;
};
#pragma pack() //取消对齐
//short = 2
//int = 4
//short = 2

```

合理的利用字节可以有效地节省存储空间

不合理的则会浪费空间、降低效率甚至还会引发错误。(对于部分系统从奇地址访问int、short等数据会导致错误)

## 2、共用体

在相同的内存位置存储不同的数据类型

共用体占用的内存应足够存储共用体中最大的成员

```

//占用4字节
union Data
{
    int i;
    short j;
}
union Data data;
data.i = 1;
//i的数据损坏
data.j = 1.1f;

```

## 3、C++

### 输出

```
C使用printf向终端输出信息
C++提供了 标准输出流
#include <iostream>
using namespace std;
char *name = "David";
int time = 8;
cout << "David:" << time << "点," << "天之道不见不散"<< endl;
```

### 函数符号兼容

第一节课中说到C的大部分代码可以在C++中直接使用，但是仍然有需要注意的地方。

```
//如果需要在C++中调用C实现的库中的方法
extern "C" //指示编译器这部分代码使用C的方式进行编译而不是C++
```

众所周知,C是面向过程的语言，没有函数重载。

```
void func(int x, int y);
```

对于 `func` 函数 被C的编译器编译后在函数库中的名字可能为 `func` (无参数符号),而C++编译器则会产生类似 `funcii` 之类的名字。

```
//main.c / main.cpp
int func(int x,int y){}
int main(){return 0;}
```

```
gcc main.c -o mainc.o
gcc main.cpp -o maincpp.o

nm -A mainc.o
nm -A maincpp.o
```

main.c

```

root@iZj6c3nmbY8r3jqrqwiommZ:~/test# nm -A mainc.o
mainc.o:0000000000601030 B __bss_start
mainc.o:0000000000601030 b completed.7585
mainc.o:0000000000601020 D __data_start
mainc.o:0000000000601020 W data_start
mainc.o:0000000000400410 t deregister_tm_clones
mainc.o:0000000000400490 t __do_global_dtors_aux
mainc.o:0000000000600e18 t __do_global_dtors_aux_fini_array_entry
mainc.o:0000000000601028 D __dso_handle
mainc.o:0000000000600e28 d _DYNAMIC
mainc.o:0000000000601030 D _edata
mainc.o:0000000000601038 B _end
mainc.o:0000000000400564 T _fini
mainc.o:00000000004004b0 t frame_dummy
mainc.o:0000000000600e10 t __frame_dummy_init_array_entry
mainc.o:00000000004006c0 r __FRAME_END__
mainc.o:00000000004004d6 T func
mainc.o:0000000000601000 d _GLOBAL_OFFSET_TABLE_
mainc.o:
w __gmon_start__
mainc.o:0000000000400574 r __GNU_EH_FRAME_HDR
mainc.o:0000000000400390 T _init
mainc.o:0000000000600e18 t __init_array_end
mainc.o:0000000000600e10 t __init_array_start
mainc.o:0000000000400570 R __IO_stdin_used
mainc.o:
w __ITM_deregisterTMCloneTable
mainc.o:
w __ITM_registerTMCloneTable
mainc.o:0000000000600e20 d __JCR_END__
mainc.o:0000000000600e20 d __JCR_LIST__
mainc.o:
w __Jv_RegisterClasses
mainc.o:0000000000400560 T __libc_csu_fini
mainc.o:00000000004004f0 T __libc_csu_init
mainc.o:
U __libc_start_main@@GLIBC_2.2.5
mainc.o:00000000004004e3 T main
mainc.o:0000000000400450 t register_tm_clones
mainc.o:00000000004003e0 T _start
mainc.o:0000000000601030 D __TMC_END__
root@iZj6c3nmbY8r3jqrqwiommZ:~/test#

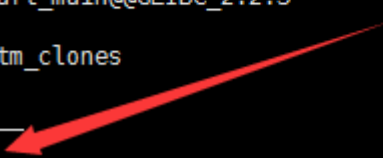
```

main.cpp

```

root@iZj6c3nmby8r3jqrwioMMZ:~/test# nm -A maincpp.o
maincpp.o:0000000000601030 B __bss_start
maincpp.o:0000000000601030 b completed.7585
maincpp.o:0000000000601020 D __data_start
maincpp.o:0000000000601020 W data_start
maincpp.o:0000000000400410 t deregister_tm_clones
maincpp.o:0000000000400490 t __do_global_dtors_aux
maincpp.o:0000000000600e18 t __do_global_dtors_aux_fini_array_entry
maincpp.o:0000000000601028 D __dso_handle
maincpp.o:0000000000600e28 d __DYNAMIC
maincpp.o:0000000000601030 D __edata
maincpp.o:0000000000601038 B __end
maincpp.o:0000000000400564 T __fini
maincpp.o:00000000004004b0 t frame_dummy
maincpp.o:0000000000600e10 t __frame_dummy_init_array_entry
maincpp.o:00000000004006c0 r __FRAME_END__
maincpp.o:0000000000601000 d __GLOBAL_OFFSET_TABLE__
maincpp.o:
maincpp.o:0000000000400574 r __GNU_EH_FRAME_HDR
maincpp.o:0000000000400390 T __init
maincpp.o:0000000000600e18 t __init_array_end
maincpp.o:0000000000600e10 t __init_array_start
maincpp.o:0000000000400570 R __IO_stdin_used
maincpp.o:
maincpp.o:
maincpp.o:
maincpp.o:
maincpp.o:0000000000600e20 d __JCR_END__
maincpp.o:0000000000600e20 d __JCR_LIST__
maincpp.o:
maincpp.o:
maincpp.o:0000000000400560 T __libc_csu_fini
maincpp.o:00000000004004f0 T __libc_csu_init
maincpp.o:
maincpp.o:
maincpp.o:00000000004004e3 T main
maincpp.o:0000000000400450 t register_tm_clones
maincpp.o:00000000004003e0 T __start
maincpp.o:0000000000601030 D __TMC_END__
maincpp.o:00000000004004d6 T __Z4funcii
root@iZj6c3nmby8r3jqrwioMMZ:~/test#

```



那么这样导致的问题就在于：c的.h头文件中定义了 func 函数，则.c源文件中实现这个函数符号都是 func,然后拿到C++中使用，.h文件中的对应函数符号就被编译成另一种，和库中的符号不匹配，这样就无法正确调用到库中的实现。

因此，对于C库可以：

```

#ifdef __cplusplus
extern "C"{
#endif
void func(int x,int y);
#ifdef __cplusplus
}
#endif

```

//\_\_cplusplus 是由c++编译器定义的宏，用于表示当前处于c++环境

extern 关键字 可用于变量或者函数之前，表示真实定义在其他文件，编译器遇到此关键字就会去其他模块查找

## 引用

引用是C++定义的一种新类型

```
//声明形参为引用
void change(int& i) {
    i = 10;
}
int i = 1;
change(i);
printf("%d\n",i); //i == 10
```

引用和指针是两个东西

引用：变量名是附加在内存位置中的一个标签,可以设置第二个标签

简单来说 引用变量是一个别名，表示一个变量的另一个名字

## 字符串

### C字符串

字符串实际上是使用 NULL字符 `'\0'` 终止的一维字符数组。

```
//字符数组 = 字符串
char str1[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
//自动加入\0
char str2[] = "Hello";
```

### 字符串操作

函数	描述
strcpy(s1, s2);	复制字符串 s2 到字符串 s1。
strcat(s1, s2);	连接字符串 s2 到字符串 s1 的末尾。
strlen(s1);	返回字符串 s1 的长度。
strcmp(s1, s2);	如果 s1 和 s2 相同，则返回 0；如果 s1 < s2 则返回小于0；如果 s1>s2 则返回大于0
strchr(s1, ch);	返回指向字符串 s1 中字符 ch 的第一次出现的位置的指针。
strstr(s1, s2);	返回指向字符串 s1 中字符串 s2 的第一次出现的位置的指针。

说明：strcmp:两个字符串自左向右逐个字符相比（按ASCII值大小相比较）

### C++ string类

C++ 标准库提供了 **string** 类类型，支持上述所有的操作，另外还增加了其他更多的功能。

```
#include <string>
//string 定义在 std命令空间中
using namespace std;
string str1 = "Hello";
string str2 = "World";
string str3("天之道");
string str4(str3);
```

```
// str1拼接str2 组合新的string
string str5 = str1 + str2;
// 在str1后拼接str2 str1改变
str1.append(str2);
//获得c 风格字符串
const char *s1 = str1.c_str();
//字符串长度
str1.size();
//长度是否为0
str1.empty();
.....等等
```

## 命名空间

namespace 命名空间 相当于package

```
namespace A{
    void a(){}
}

错误 : a();
// :: 域操作符
正确: A::a();

//当然也能够嵌套
namespace A {
    namespace B{
        void a() {};}
}
A::B::a();

//还能够使用using 关键字
using namespace A;
using namespace A::B;
```

当全局变量在局部函数中与其中某个变量重名，那么就可以用::来区分

```
int i;
int main(){
    int i = 10;
    printf("i : %d\n",i);
    //操作全局变量
    ::i = 11;
    printf("i : %d\n",::i);
}
```