完美世界

**1.大端/小端问题(按字节编址)**

大端和小端是指数据在内存中的存储模式，它由 CPU 决定：

1) 大端模式（Big-endian）是指将数据的低位（比如 1234 中的 34 就是低位）放在内存的高地址上，而数据的高位（比如 1234 中的 12 就是高位）放在内存的低地址上。这种存储模式有点儿类似于把数据当作字符串顺序处理，地址由小到大增加，而数据从高位往低位存放。

2) 小端模式（Little-endian）是指将数据的低位放在内存的低地址上，而数据的高位放在内存的高地址上。这种存储模式将地址的高低和数据的大小结合起来，高地址存放数值较大的部分，低地址存放数值较小的部分，这和我们的思维习惯是一致，比较容易理解。

为什么有大小端模式之分?

计算机中的数据是以字节（Byte）为单位存储的，每个字节都有不同的地址。现代 CPU 的位数（可以理解为一次能处理的数据的位数）都超过了 8 位（一个字节），PC机、服务器的 CPU 基本都是 64 位的，嵌入式系统或单片机系统仍然在使用 32 位和 16 位的 CPU。

对于一次能处理多个字节的CPU，必然存在着如何安排多个字节的问题，也就是大端和小端模式。以 int 类型的 0x12345678 为例，它占用 4 个字节，如果是小端模式（Little-endian），那么在内存中的分布情况为（假设从地址 0x 4000 开始存放）：

内存地址 0x4000 0x4001 0x4002 0x4003

存放内容 0x78 0x56 0x34 0x12

如果是大端模式（Big-endian），那么分布情况正好相反：

内存地址 0x4000 0x4001 0x4002 0x4003

存放内容 0x12 0x34 0x56 0x78

我们的 PC 机上使用的是 X86 结构的 CPU，它是小端模式；51 单片机是大端模式；很多 ARM、DSP 也是小端模式（部分 ARM 处理器还可以由硬件来选择是大端模式还是小端模式）。

借助共用体，我们可以检测 CPU 是大端模式还是小端模式，请看代码：

#include <stdio.h>

int main()

{

union{

int n;

char ch;

} data;

if(data.ch == 1){

printf("Little-endian\n");

}else{

printf("Big-endian\n");

}

return 0;

}

在PC机上的运行结果：

Little-endian

共用体的各个成员是共用一段内存的。1 是数据的低位，如果 1 被存储在 data 的低字节，就是小端模式，这个时候 data.ch 的值也是 1。如果 1 被存储在 data 的高字节，就是大端模式，这个时候 data.ch 的值就是 0。

**2.如何避免头文件循环引用**

循环引用的错误代码：

**A.h**

#include "B.h"

class A

{

B b;

};

**B.h**

#include "A.h"

class B

{

A a;

};

解决方法：

前向声明解决互相引用造成循环依赖

**A.h**

#pragma once

#include "B.h"

class A

{

public:

A();

~A();

private:

B\* b;

};

**B.h**

#pragma once

//前向声明取代互相引用头文件

class A;

class B

{

public:

B();

~B();

private:

A\* a;

};

注意：只能用于定义指针、引用、以及用于函数形参的指针和引用。而不能定义对象(因为此时编译器只知道这是个类，还不知道这个类的大小有多大)，也不能访问类的对象，任何形式的访问都不允许。

使用前向声明取代引用的这个类，在以下情况会报错：

**A.h**

#pragma once

#include "B.h"

class A

{

public:

A();

~A();

int a\_;

private:

B\* Cls\_B\_;

};

**B.h**

#pragma once

class A;

class B

{

public:

B();

~B();

void BCallA(A\* a);

private:

A\* Cls\_A\_;

};

**B.cpp**

#include "B.h"

//#include "A.h” 反注释解决找不到A定义的报错

B::B(void){

};

B::~B(void){

};

void B::BCallA(A\* a){

int tmp = a->a\_;

return ;

}

原因也比较简单，是因为B类由于前向声明没有#include “A.h”，导致找不到A类的定义，解决方案就是在B.cpp中#include "A.h"一下即可。

**3.O(n)解决:找出[1,100]数组中的和为100的连续序列(双指针)**

答：给定两个指针，分别为左指针和右指针，首先左指针不动，右指针不断移动

三种情况:1)若和小于100，则右指针向右移动 2)若和等于100，得到一个答案，左指针向右移动，右指针向右移动 3)和大于100，左指针向右移动

**4.字节对齐问题：**

一个struct有4字节int 1字节char 2字节short 32位机器，这个struct实际占用的内存为？

答：应为8字节，字节对齐，简单来说就是一个变量若小于对齐单位，则该变量不能跨过对其单位。若变量大于对齐单位，则该变量应占用若干完整的对齐单位，最后一个对齐单位可占用部分。

具体举例如下：

//32位系统

#include<stdio.h>

struct

{

int i;

char c1;

char c2;

}x1;

struct{

char c1;

int i;

char c2;

}x2;

struct{

char c1;

char c2;

int i;

}x3;

int main()

{

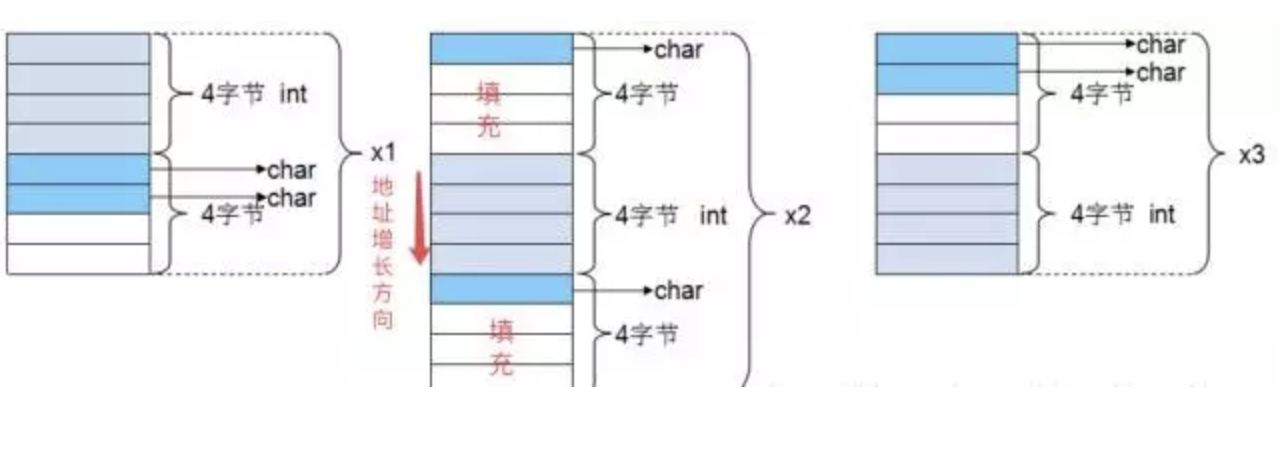
printf("%d\n",sizeof(x1)); // 输出8

printf("%d\n",sizeof(x2)); // 输出12

printf("%d\n",sizeof(x3)); // 输出8

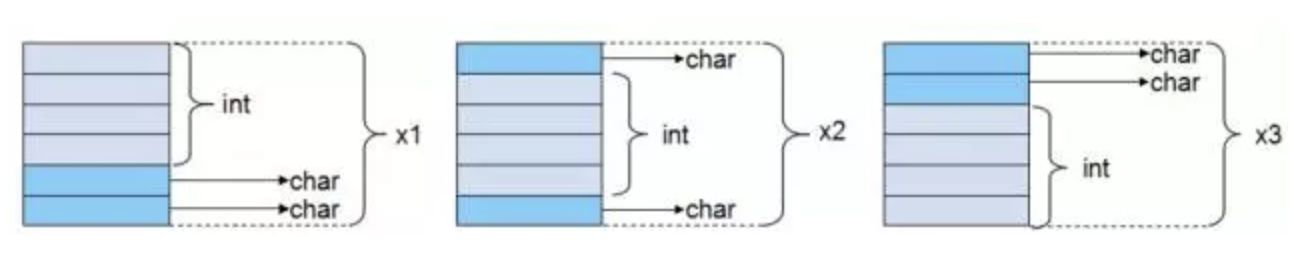
return 0;

}

以上测试都是在Linux环境下进行的，linux下默认#pragma pack(4)，且结构体中最长的数据类型为4个字节，所以有效对齐单位为4字节。上面例子三个结构体的内存布局如下：

**#pragma pack(n)**

不同平台上编译器的 pragma pack 默认值不同。而我们可以通过预编译命令#pragma pack(n), n= 1,2,4,8,16来改变对齐系数。

例如，对于上个例子的三个结构体，如果前面加上#pragma pack(1)，那么此时有效对齐值为1字节，此时根据对齐规则，不难看出成员是连续存放的，三个结构体的大小都是6字节。

如果前面加上#pragma pack(2)，有效对齐值为2字节，此时根据对齐规则，三个结构体的大小应为6,8,6。内存分布图如下：

