**C语言**

.h文件:头文件，只有函数声明

.c文件:C语言的源码文件

.hpp文件:头文件和C++的源码文件，不止有函数声明，还有函数实现

.cpp文件:C++的源码文件

**基本数据类型**

在不同的系统上，这些类型占据的字节长度是不同的：

在32 位的系统上

short 占据的内存大小是2 个byte；

int占据的内存大小是4 个byte；

long占据的内存大小是4 个byte；

float占据的内存大小是4 个byte；

double占据的内存大小是8 个byte；

char占据的内存大小是1 个byte。

**常量**

**1、#define定义的宏常量**

编译时替换

#include <iostream>

using namespace std;

#define ALIGN\_SIZE 8

int \_tmain(int argc, \_TCHAR\* argv[])

{

cout << ALIGN\_SIZE << endl;

ALIGN\_SIZE = 10;

getchar();

return 0;

}

**2、const修饰的变量**

即不可修改

#include <iostream>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 10;

const int b = 20;

a = 11;

b = 22;

getchar();

return 0;

}

**字符串**

**1、如何定义**

char \*str1 = "hi";

char str2[] = "hi";

char strArr1[] = { 'h', 'i' };

char strArr2[] = { 'h', 'i', '0' };

char strArr3[] = { 'h', 'i', '\0' };

**2、字符串以【\0】为结束符**

#include <iostream>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[])

{

char \*str1 = "hi";

char str2[] = "hi";

char strArr1[] = { 'h', 'i' };

char strArr2[] = { 'h', 'i', '0' };

char strArr3[] = { 'h', 'i', '\0' };

cout << str1 << endl;

cout << strlen(str1) << endl;

cout << str2 << endl;

cout << strlen(str2) << endl;

cout << strArr1 << endl;

cout << strArr2 << endl;

cout << strArr3 << endl;

getchar();

return 0;

}

**3、相关函数**

|  |  |
| --- | --- |
| 序号 | 函数 & 目的 |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。 |
| 5 | **strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。 |
| 6 | **strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。 |

**指针**



**1、指针基础知识**

\* 定义指针的符号

& 取地址符

-> 指针访问数据（结构体）

. 指针访问数据（结构体）

指针变量存储的是内存地址

比如我们写的这段代码：int a = 10;

|  |  |
| --- | --- |
| 0x01 | 10 |
| 0x02 |  |
| 0x03 |  |

指针大小，指针的长度只跟操作系统位数有关和数据类型无关，固定长度。

32bit机，指针占4B；64bit机，指针占8B。

指针互转

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 0x12345678;

char \*p = (char \*)&a;

long double \*ldp = (long double \*)&a;

printf("%X\n", \*p);// 少读取,输出0x87,因为操作系统是小端存储，所以是从右往左存储数据

printf("%X\n", \*ldp);// 内存往后多读取12字节，这12字节数据是其他占用内存的数据,long double占16字节，int占4字节

// 接收用户的输入

getchar();

return 0;

}

通过指针修改数据

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 0x12345678;

char \*p = (char \*)&a;

\*p = 0x11;

printf("%X\n", a);

// 接收用户的输入

getchar();

return 0;

}

空指针、空类型指针、野指针

1、空指针：无法访问

int \*p = NULL;

2、空类型指针void \*p：可以指向任意数据类型，但是无法访问。

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 10;

void \*p = (int \*)a;

printf("%d", \*p);

// 接收用户的输入

getchar();

return 0;

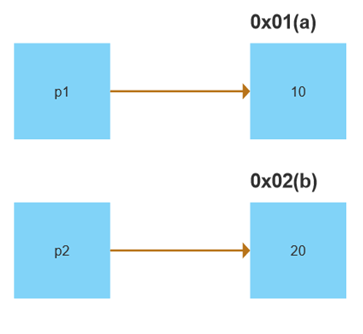
}

3、野指针：指针指向的地址是非法地址；指针指向的内存区域已经被释放了（free）

int \*a = 0x00000000;// 指针指向的地址是非法地址

ini \*a = calloc();free(a) // 指针指向的内存区域已经被释放了

**2、指针与const**



const在指针前面就是常量指针，const在常量前面就是指针常量。

const修饰指针，指针指向可以该，const修饰常量，指针值可以该。也就是const修饰什么，什么可以改，其余不能该。

1. 指针常量：const修饰常量（指针指向不可以改，指针指向的值可以改）

int \* const p2 = &a;

1. 常量指针：const修饰指针（指针指向可以改，指针指向的值不可以改，p1能指向0x02，但0x01的值10不能改变）

const int \*p1 = &a;

1. 修饰指针 + 修饰常量（指针、指针指向的值都不能修改）

看一个综合案例：

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 10;

int b = 20;

const int \*p1 = &a;// 常量指针

p1 = &b;// 正常，指针指向可以改

\*p1 = 30;// 报错，指针指向的值不可以改

//=====

int \* const p2 = &a;// 指针常量

\*p2 = 40;// 正常，指针指向的值可以改

p2 = &b;// 报错，指针指向不可以改

//=====

const int \* const p3 = &a;// 修饰指针+修饰常量

\*p3 = 60;

p3 = &b;

// 接收用户的输入

getchar();

return 0;

}

**3、指针与函数**

函数指针与指针函数、指针作为函数参数

**4、指针与数组**

数组的本质及利用指针访问数组

int \_tmain(int argc, \_TCHAR\* argv[])

{

char name[] = "xiaobi";

printf("%c \n",name[1]); // 输出i

printf("%c \n",\*(name+1)); // 输出i，name是字符内存的首地址,先往后移一位，在输出数据

printf("%c \n",\*name+1); // 输出y，name是字符内存的首地址,先输出name[0] = x，再x的ascii加1,也就是y

// 接收用户的输入

getchar();

return 0;

}

**5、指针与结构体**

**函数**

**1、函数的定义与声明**

函数声明：相当于JAVA中的接口/抽象方法。函数声明的时候，形参可以写，可以不写。

函数定义：相当于JAVA中的实现。函数定义的时候，形参可以写，可以不写。

// 函数的声明

void test(int, int);

void test2(int a, int);

// 函数的定义

void test(int, int){}

void test2(int a, int b){}

为什么要有函数声明？

test函数调用test2函数，如果test上面没有test2函数声明，则会发生运行时异常，如果test上面有test2函数定义，则正常运行。发生运行时异常，是因为C语言是面向过程的，从上至下顺序的执行代码，所以此时需要函数声明。JAVA可以这样，是因为JAVA是动态解析代码。

**2、默认参数**

思考：Java支持默认参数吗？为什么

Java不支持默认参数，这是Java的特性

案例一，上代码

正确

void add(int a, int b = 10)

{

}

案例二，上代码

错误原因：如果某个位置有了默认值，这个位置之后的所有值都得有默认参数

void add(int a, int b = 10, int c)

{

}

案例三，上代码

错误原因：声明与实现只能一个地方有，不然运行时不知道取哪个值

void add(int a, int b = 20);

void add(int a, int b = 10)

{

cout << a + b << endl;

}

**3、占位参数**

add(int a) a是形参，add(15) 15是实参

格式：形参中数据类型后面不写变量名

函数声明时，形参可使用占位，也可以不使用

int add(int a, int);

函数定义时，形参可使用占位，也可以不使用。

int add(int a, int)

{

return 10;

}

使用占位的情况下，实参就无法用了（真的无法用吗？），通过内联汇编依然可以拿到占位参数的值，利用了压栈原理拿到实参数据。

int add(int a, int)

{

int b;

\_\_asm {

mov eax, [ebp + 0xc];

mov [b], eax;

}

return 10;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

add(10, 20);

// 接收用户的输入

getchar();

return 0;

}

占位参数可以使用默认值

int add(int a, int = 20);

**4、函数重载（与Java对比着学习）**

1、需要满足条件

1、函数名相同

2、参数类型、个数、顺序不同

3、注意：与返回值无关

void add()

{

printf("add() \n");

}

int add()

{

printf("add() \n");

}

void add(int a)

{

printf("add(int a) \n");

}

2、函数重载与函数默认参数

void add(int a)

{

printf("add(int &a) \n");

}

void add(int a, int b=30)

{

printf("add(int a, int b=30) \n");

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a = 10;

add(a); // 错误

add(a, 30); // 正确

// 接收用户的输入

getchar();

return 0;

}

3、函数重载与引用

案例一

void add(int a)

{

printf("add(int a) \n");

}

void add(int &a)

{

printf("add(int &a) \n");

}

案例二

void add(int &a)

{

printf("add(int &a) \n");

}

void add(const int &a)

{

printf("add(const int &a) \n");

}

add(10)调用下面这个

**5、函数指针与指针函数**

1、函数指针：一个函数总是占用一段连续的内存区域，函数名有时会被转换为该函数所在内存区域的首地址。我们可以把函数的这个首地址（或称入口地址）赋予一个指针变量(这个指针变量就叫作函数指针变量，简称函数指针, 指向函数的指针变量，可作为参数、返回值，也可作为数据类型使用)，使指针变量指向函数所在的内存区域，然后通过指针变量就可以找到并调用该函数。

定义格式：函数返回值类型 (\* 指针变量名)(函数参数列表);

void (\* p\_fun)(int, int);

2、指针函数：返回值为指针的函数

定义格式：函数返回值类型 \*函数名(函数参数列表);

void fun(int, int);普通函数，返回值是一个 int 类型，是一个数值。

void \*fun(int, int);指针函数，函数名前面多了一个\*号，其返回值是一个 int 类型的指针，是一个地址。

**结构体、联合体、枚举**

1、结构体

声明1

struct Teacher

{

int age;

char isCool;

};

声明1+定义变量

struct Teacher

{

int age;

char isCool;

}teacher;

声明2+定义变量

typedef struct

{

int age;

char isCool;

}Node;

使用

使用结构体定义变量时，前面的struct可写可不写

struct Teacher

{

int age;

char isCool;

}teacher,\*p\_teacher;

// 结构体使用

int \_tmain(int argc, \_TCHAR\* argv[])

{

struct Teacher t1;

Teacher t2;

t1.age = 11; // 方式一

t2.age = 22; // 方式二

teacher.age = 33; // 方式三

p\_teacher->age = 44; // 方式四，如果是结构体形式使用.访问；如果是结构体指针使用->访问成员变量

(\*p\_teacher).age = 55; // 方式五，指针前加\*号就是解引用

// 接收用户的输入

getchar();

return 0;

}

定义+赋初值

struct Teacher

{

int age;

char isCool;

}teacher;

int \_tmain(int argc, \_TCHAR\* argv[])

{

Teacher t1 = {11, 1};

// 接收用户的输入

getchar();

return 0;

}

声明+定义变量

struct Teacher

{

int age;

char isCool;

}teacher;

int \_tmain(int argc, \_TCHAR\* argv[])

{

teacher.age = 10;

printf("%d\n", teacher.age);

// 接收用户的输入

getchar();

return 0;

}

结构体数组

struct Teacher

{

int age; // 4

char isCool; // 1

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

Teacher teacher[2];

teacher[0].age = 10;

teacher[0].isCool = 1;

(\*teacher).age = 20;

teacher->age = 30;

\*(teacher+1).age = 10;

int arr[] = {0x11223344,0xaabbccdd};

int a = \*arr;// 11223344

int b = \*arr + 1;// 11223345

int c = \*(arr+1);// AABBCCDD

int \*p\_int = arr;

char \*p\_char = (char \*)arr;

p\_int;// 输出内存地址

p\_int + 1;// 输出内存地址+4

p\_char;// 输出内存地址

p\_char + 1;// 输出内存地址 + 1

// 接收用户的输入

getchar();

return 0;

}

结构体指针

访问结构体成员时用->，而不是.

// 32bit，4字节对齐

struct Teacher

{

int age; // 4

char isCool; // 1

};

void print1(Teacher t)

{

printf("%d\n", t.age);

}

void print2(Teacher \*t)

{

printf("%d\n", t->age);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

struct Teacher teacher;

teacher.age = 10;

teacher.isCool = 1;

// 为什么差8个字节，因为结构体长度是8

// printf("%X, %X\n", &teacher, &teacher + 1);

// printf("%d\n", \*(char \*)(&teacher + 4));

printf("%d\n", \*(char \*)((char \*)&teacher + 4));

print1(teacher);

print2(&teacher);

// 接收用户的输入

getchar();

return 0;

}

结构体嵌套结构体

结构体做函数参数

值传递、引用传递

struct Teacher

{

int age;

char isCool;

}teacher;

void print\_1(Teacher t)

{

printf("%d\n", t.age);

}

void print\_2(Teacher \*t)

{

printf("%d\n", t->age);

}

void print\_3(Teacher &t)

{

printf("%d\n", t.age);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

Teacher t1 = {11, 1};

print\_1(t1);

print\_2(&t1);

print\_3(t1);

// 接收用户的输入

getchar();

return 0;

}

加const修饰（重要！！读源码经常看到这个语法）

目的是为了防止外部拿到结构体变量的引用修改结构体成员的值

struct Teacher

{

int age;

char isCool;

}teacher;

void print(Teacher \*t)

{

t->age = 20;

printf("%d\n", t->age);

}

void print\_1(const Teacher \*t)

{

t->age = 20;

printf("%d\n", t->age);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

Teacher t1 = {11, 1};

print(&t1);

printf("%d\n", t1.age);

// 接收用户的输入

getchar();

return 0;

}

2、联合体（union）

允许在相同的内存位置存储不同的数据类型

但是任何时候只能有一个成员带有值

联合体的大小由字节数最多的成员决定

3、枚举（enum）

跟Java中的差不多

**引用**

可理解为：给变量起别名

其本质其实就是指针常量：int \* const a = &b;

语法：

int a = 10;

int &b = a;

注意：

1、引用必须初始化

2、引用初始化后无法修改

引用做函数参数

struct Teacher

{

int age;

char isCool;

}teacher;

void print(Teacher &t)

{

t.age = 30;

printf("%d\n", t.age);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

Teacher t1 = {11, 1};

print(t1);

printf("%d\n", t1.age);

// 接收用户的输入

getchar();

return 0;

}

引用做函数返回值

int & add()

{

int a = 10;

return a;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int &a = add();

printf("%d\n", a);

printf("%d\n", a);

printf("%d\n", a);

// 接收用户的输入

getchar();

return 0;

}

如果函数的返回值是引用，这个函数可以作为左值使用

int & add()

{

static int a = 10;

return a;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int &a = add();

printf("%d\n", a);

printf("%d\n", a);

add() = 100;

printf("%d\n", a);

// 接收用户的输入

getchar();

return 0;

}

常量引用

const int &a = b;