C语言指针从入门到基础详解（非常详细）

1.内存和地址

编辑我们知道电脑中的CPU在处理数据的时候需要在内存中读取数据处理后的数据也会放在内存中。把内存划分为一个个的内存单元每个单元的大小是一个字节。每个字节都有它对应的编号也就是它的地址，以便CPU可以快速的找到一个内存空间。C语言中我们把地址叫做指针，所以**内存单元的编号==地址==指针。**

2.指针变量和地址

取地址操作符（&）：

C语言中，创建变量就是在向内存申请空间，下例在创建a变量的过程中就是向内存申请了四个字节的空间，那么这四个字节都有地址，取地址操作符&就可以把a的地址（四个地址的较小的一个）取出来，这样就可以知道其他三个地址了。

```
int a=10;
printf("%p\n",&a);
```

指针变量：

我们把a的地址用&取出来了，那么我们把&a放在哪呢？放在指针变量中。指针变量是一种用来存放地址的变量，放在指针变量中的值都会理解成地址。

```
int a = 10;
int* pa = &a;
```

如何理解指针类型：

pa的类型是int*,*是说明pa是一个指针变量，int说明pa指向的是一个int类型的对象。那么如果对象是char类型的，对应的指针变量的类型就是char*。

解引用操作符（*）：

有了一个元素的地址，我们要使用他的时候就要用到解引用操作符*

```
int a = 10;
int* pa = &a;
printf("%d",*pa);
```

这里pa是一个地址，使用*pa系统就会寻找并读取这个地址中的数据。

指针变量的大小：

根据计算机的相关知识，我们可以知道32位机器上有32根地址总线，每根地址线只有两种输出0/1（低电平和高电平）那么一个地址就是32个bit位就是四个字节，所以在32环境下指针变量的大小是4个字节，同理64位环

境下就是8个字节。

注意指针变量的大小和类型是无关的，只要指针类型的变量，在相同的平台下，大小都是相同的。

3.指针变量类型的意义

指针的解引用：

int*类型指针和*char类型指针的可操作空间不一样，我们可以观察一下下面的两段代码。

```
int main()
{
    int n = 0x11223344;
    int *pi = &n;
    *pi = 0;
    return 0;
}
/************************/
int main()
{
    int n = 0x11223344;
    char *pc = (char *)&n;
    *pc = 0;
    return 0;
}
```

第一段代码会将n的四个字节全部改为0，第二段代码只是将n的第一个字节改为0。

结论：指针的类型决定了，对指针解引用的时候有多大的权限（一次能操作几个字节）。 比如： char* 的指针解引用就只能访问一个字节，而 int* 的指针的解引用就能访问四个字节。

指针+-整数：

先来看一段代码。

```
#include <stdio.h>
int main()
{
    int n = 10;
    char *pc = (char*)&n;
    int *pi = &n;
    printf("%p\n", &n);
    printf("%p\n", pc);
    printf("%p\n", pc+1);
    printf("%p\n", pi);
    printf("%p\n", pi+1);
```

```
    return 0;
}
```

运行结果：

编辑

我们可以看出， char* 类型的指针变量+1跳过1个字节， int* 类型的指针变量+1跳过了4个字节。 这就是指针变量的类型差异带来的变化。 **结论：指针的类型决定了指针向前或者向后走一步有多大（距离）。**

void* 指针：

在指针类型中有一种特殊的类型是 void* 类型的，可以理解为无具体类型的指针（或者叫泛型指 针），这种类型的指针可以用来接受任意类型地址。但是也有局限性， void* 类型的指针不能直接进行指针的+-整数和解引用的运算。 一般 void* 类型的指针是使用在函数参数的部分，用来接收不同类型数据的地址，这样的设计可以实现泛型编程的效果。使得一个函数来处理多种类型的数据

4.const修饰指针

const修饰变量：

被const修饰的变量不能被修改。

```
#include <stdio.h>
int main()
{
    int m = 0;
    m = 20;//m是可以修改的
    const int n = 0;
    n = 20;//n是不能被修改的
```

```
    return 0;
}
```

但是如果我们绕过n本身，用n的地址就能够修改n，这样做就在打破语法规则。

```
#include <stdio.h>
int main()
{
    const int n = 0;
    printf("n = %d\n", n);
    int*p = &n;
    *p = 20;
    printf("n = %d\n", n);
    return 0;
}
```

这本身是不合理的所以我们应该让n的地址也不能修该n。

const修饰指针变量：

根据下面的代码我们可以得到结论。

```
#include <stdio.h>

void test1()
{
    int n = 10;
    int m = 20;
    int *p = &n;
    *p = 20;//ok?
    p = &m; //ok?
}
void test2()
{
    int n = 10;
    int m = 20;
    const int* p = &n;
    *p = 20;//ok?
    p = &m; //ok?
}
void test3()
{
    int n = 10;
    int m = 20;
    int *const p = &n;
```

```
    *p = 20; //ok?
}
void test4()
{
    int n = 10;
    int m = 20;
    int const * const p = &n;
    *p = 20; //ok?
    p = &m; //ok?
}
int main()
{
    //测试无const修饰的情况
    test1();
    //测试const放在*的左边情况
    test2();
    //测试const放在*的右边情况
    test3();
    //测试*的左右两边都有const
    test4();
return 0;
```

结论：const修饰指针变量的时候 • const如果放在*的左边，修饰的是指针指向的内容，保证指针指向的内容不能通过指针来改变。 但是指针变量本身的内容可变。 • const如果放在*的右边，修饰的是指针变量本身，保证了指针变量的内容不能修改，但是指针指 向的内容，可以通过指针改变。

5. 指针运算

指针+-整数：

我们这里直接以例子来说明

```
#include <stdio.h>

int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *p = &arr[0];
    int i = 0;
    int sz = sizeof(arr)/sizeof(arr[0]);
    for(i=0; i<sz; i++)
    {
        printf("%d ", *(p+i));//p+i 这里就是指针+整数
    }
    return 0;
}
```

指针-指针：

指针-指针的结果是两个指针之间的元素个数。可以借这个属性来模拟实现strlen：

```
size_t my_strlen(char* s)//指针-指针
{
    char* tmp = s;
    while (*(++s))
        ;
    return s - tmp;
}
```

模拟实现C语言库函数（strlen，strcpy，strcat）-CSDN博客三种方法。
https://blog.csdn.net/2301_80194476/article/details/136588134?spm=1001.2014.3001.5502

指针的关系计算：

直接上例子：

```
#include <stdio.h>
int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *p = &arr[0];
    int i = 0;
    int sz = sizeof(arr)/sizeof(arr[0]);
    while(p<arr+sz) //指针的大小比较
    {
        printf("%d ", *p);
        p++;
    }
    return 0;
}
```

6.野指针

野指针成因：

未初始化的指针

```c
#include <stdio.h>
int main()
{
    int *p;//局部变量指针未初始化，默认为随机值
    *p = 20;
    return 0;
}
```

指针越界访问

```c
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    *p = &arr[0];
    int i = 0;
    for(i=0; i<=11; i++)
    {
        //当指针指向的范围超出数组arr的范围时，p就是野指针
        *(p++) = i;
    }
    return 0;
}
```

指针指向的空间释放

```c
#include <stdio.h>
int* test()
{
    int n = 100;
    return &n;
}
int main()
{
    int*p = test();
    printf("%d\n", *p);
    return 0;
}
```

如何避免野指针

指针初始化

如果明确知道指针指向哪里就直接赋值地址，如果不知道指针应该指向哪里，可以给指针赋值NULL。NULL 是C语言中定义的一个标识符常量，值是0，也是地址，这个地址是无法使用的，读写该地址会报错。

```
#include <stdio.h>
int main()//初始化
{
    int num = 10;
    int*p1 = &num;
    int*p2 = NULL;
    return 0;
}
```

小心指针越界

指针变量不再使用时，及时置NULL，指针使用之前检查有效性

避免返回局部变量的地址

7.assert断言

[C语言指针部分易错-CSDN博客文章浏览阅读824次，点赞22次，收藏24次。关于assert宏，它是一个断言，在写代码的过程中，如果要使用assert，就要包括头文件。否则就终止程序，这在代码的调试中很常用，我们通过定义NDEBUG宏来进行assert的禁用。sizeof(i)的值为4，而i为-1，此时如果以为选择B就万事大吉了的话，那也太小看此题了。B：int (*ptr)10]这是一个数组指针，代码的意思是将整个的数组地址（&arr）放进这个数组指针里面，没有问题。D：前面说了&arr表示整个数组的地址，把整个数组的地址放进一个指针里面是不可行的。 https://blog.csdn.net/2301_80194476/article/details/136235453?spm=1001.2014.3001.5502

8.传值调用和传址调用

学习指针的目的是使用指针解决问题，那什么问题，非指针不可呢？ 题目：写一个函数，交换两个整型变量的值。

```
#include <stdio.h>
void Swap1(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int main()
{
```

```
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
    printf("交换前: a=%d b=%d\n", a, b);
    Swap1(a, b);
    printf("交换后: a=%d b=%d\n", a, b);
    return 0;
}
```

但是这样写的结果是错误的，其实a和b并没有交换，这里只将a，b的值传给了x，y是传值调用。

结论：实参传递给形参的时候，形参会单独创建一份临时空间来接收实参，对形参的修改不影响实 参。所以Swap是失败的了。

想要真正交换，就要进行传址调用。

```
#include <stdio.h>
Swap2(int*px, int*py)
{
    int tmp = 0;
    tmp = *px;
    *px = *py;
    *py = tmp;
}
int main()
{
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
    printf("交换前: a=%d b=%d\n", a, b);
    Swap1(&a, &b);
    printf("交换后: a=%d b=%d\n", a, b);
    return 0;
}
```

传址调用，可以让函数和主调函数之间建立真正的联系，在函数内部可以修改主调函数中的变量；所以未来函数中只是需要主调函数中的变量值来实现计算，就可以采用传值调用。如果函数内部要修改主调函数中的变量的值，就需要传址调用。 **本期博客到这里就结束了，如果有什么错误，欢迎指出，如果对你有帮助，请点个赞，谢谢！**