C 语言函数指针基础

本文写的非常详细，为初学者建立一个意识模型，来帮助他们理解函数指针的语法和基础。如果你不讨厌事无巨细，请尽情阅读吧。

函数指针虽然在语法上让人有些迷惑，但不失为一种有趣而强大的工具。本文将从 C 语言函数指针的基础开始介绍，再结合一些简单的用法和关于函数名称和地址的趣闻。在最后，本文给出一种简单的方式来看待函数指针，让你对其用法有一个更清晰的理解。

函数指针和一个简单的函数

我们从一个非常简单的”Hello World“函数入手，来见识一下怎样创建一个函数指针。

```
#include <stdio.h>
// 函数原型
void sayHello();

//函数实现
void sayHello(){
        printf("hello world\n");
}
// main 函数调用
int main()
{        sayHello();
}
```

我们定义了一个名为 sayHello 的函数，它没有返回值也不接受任何参数。当我们在 main 函数中调用它的时候，它向屏幕输出出”hello world“。非常简单。接下来，我们改写一下 main 函数，之前直接调用的 sayHello 函数，现在改用函数指针来调用它。

```
int main()
{        void (*sayHelloPtr)() = sayHello;
        (*sayHelloPtr)();
}
```

第二行 void (*sayHelloPtr)()的语法看起来有些奇怪，我们来一步一步分析。

1. 这里，关键字 void 的作用是说我们创建了一个函数指针，并让它指向了一个返回 void（也就是没有返回值）的函数。
2. 就像其他任何指针都必须有一个名称一样，这里 sayHelloPtr 被当作这个函数指针的名称。
3. 我们用*符号来表示这是一个指针，这跟声明一个指向整数或者字符的指针没有任何区别。
4. *sayHelloPtr 两端的括号是必须的，否则，上述声明变成 void *sayHelloPtr()，*会优先跟 void 结合，变成了一个返回指向 void 的指针的普通函数的声明。因此，函数指针声明的时候不要忘记加上括号，这非常关键。
5. 参数列表紧跟在指针名之后，这个例子中由于没有参数，所以是一对空括号()。

6. 将上述要点结合起来，void (*syaHelloPtr)()的意义就非常清楚了，这是一个函数指针，它指向一个不接收参数且没有返回值的函数。

在上面的第二行代码，即 void (*sayHelloPtr)() = sayHello;，我们将 sayHello 这个函数名赋给了我们新建的函数指针。关于函数名的更多细节我们会在下文中讨论，现在暂时可以将其看作一个标签，它代表函数的地址，并且可以赋值给函数指针。这就跟语句 int *x = &myint;中我们把 myint 的地址赋给一个指向整数的指针一样。只是当我们考虑函数的时候，我们不需要加上一个取地址符&。简而言之，函数名就是它的地址。接着看第三行，我们用代码’(*sayHelloPtr)();• ‘解引用并调用了函数指针。

1. 在第二行被声明之后，sayHelloPtr 作为函数指针的名称，跟其他任何指针没有差别，能够储值和赋值。
2. 我们对 sayHelloPtr 解引用的方式也与其他任何指针一样，即在指针之前使用解引用符*，也就是代码中的*sayHelloPtr。
3. 同样的，我们需要在其两端加上括号，即(*sayHelloPtr)，否则它就不被当做一个函数指针。因此，记得声明和解引用的时候都要在两端加上括号。
4. 括号操作符用于 C 语言中的函数调用，如果有参数参与，就将其放入括号中。这对于函数指针也是相似的，即代码中的(*sayHelloPtr)()。
5. 这个函数没有返回值，也就没有必要将它赋值给任何变量。单独来说，这个调用跟 sayHello()没什么两样。

接下来，我们再对函数稍加修改。你会看到函数指针奇怪的语法，以及用调用普通函数的方法来调用赋值后函数指针的现象。

```
int main() {void (*sayHelloPtr)() = sayHello;
sayHelloPtr();
}
```

跟之前一样，我们将 sayHello 函数赋给函数指针。但是这一次，我们用调用普通函数的方法调用了它。稍后讨论函数名的时候我会解释这一现象，现在只需要知道(*syaHelloPtr)()和 syaHelloPtr()是相同的即可。

带参数的函数指针

好了，这一次我们来创建一个新的函数指针吧。它指向的函数仍然不返回任何值，但有了参数。

```
#include <stdio.h>

//函数原型
void subtractAndPrint(int x, int y);

//函数实现
void subtractAndPrint(int x, int y) {
        int z = x - y;
        printf("Simon says, the answer is: %d\n", z)
```

```
        }

//main 函数调用
int main() {
        void (*sapPtr)(int, int) = subtractAndPrint;
        (*sapPtr)(10, 2);
        sapPtr(10, 2)
        }
```

跟之前一样，代码包括函数原型，函数实现和在 main 函数中通过函数指针执行的语句。原型和实现中的特征标变了，之前的 sayHello 函数不接受任何参数，而这次的函数 subtractAndPrint 接受两个 int 作为参数。它将两个参数做一次减法，然后输出到屏幕上。

1. 在第 14 行，我们通过'(*sapPtr)(int, int)' 创建了 sapPtr 这个函数指针，与之前的区别仅仅是用(int, int)代替了原来的空括号。而这与新函数的特征标相符。
2. 在第 15 行，解引用和执行函数的方式与之前完全相同，只是在括号中加入了两个参数，变成了(10, 2)。
3. 在第 16 行，我们用调用普通函数的方法调用了函数指针。

带参数且有返回值的函数指针

这一次，我们把 subtractAndPrint 函数改成一个名为 subtract 的函数，让它把原本输出到屏幕上的结果作为返回值。

```
#include <stdio.h>

// 函数原型
int subtract(int x, int y);

// 函数实现
int subtract(int x, int y) {
        return x - y;
}

// main 函数调用
int main() {
    int (*subtractPtr)(int, int) = subtract;
      int y = (*subtractPtr)(10, 2);
    printf("Subtract gives: %d\n", y);

    int z = subtractPtr(10, 2);
    printf("Subtract gives: %d\n", z);
}
```

这与 subtractAndPrint 函数非常相似，只是 subtract 函数返回了一个整数而已，特征标也理所当然的不一样了。

1. 在第 13 行，我们通过 int (*subtractPtr)(int, int)创建了 subtractPtr 这个函数指针。与上一个例子的区别只是把 void 换成了 int 来表示返回值。而这与 subtract 函数的特征标相符。
2. 在在第 15 行，解引用和执行这个函数指针，除了将返回值赋值给了 y 以外，与调用 subtractAndPrint 没有任何区别。
3. 在第 16 行，我们向屏幕输出了返回值。
4. 18 到 19 行，我们用调用普通函数的方法调用了函数指针，并且输出了结果。

这跟之前没什么两样，我们只是加上了返回值而已。接下来我们看看另一个稍微复杂点儿的例子——把函数指针作为参数传递给另一个函数。

把函数指针作为参数来传递

我们已经了解过了函数指针声明和执行的各种情况，不论它是否带参数，或者是否有返回值。接下来我们利用一个函数指针来根据不同的输入执行不同的函数

```
#include <stdio.h>

// 函数原型
int add(int x, int y);
int subtract(int x, int y);
int domath(int (*mathop)(int, int), int x, int y);

// 加法 x+ y
int add(int x, init y) {
        return x + y;
}

// 减法 x - y
int subtract(int x, int y) {
        return x - y;
}

// 根据输入执行函数指针
int domath(int (*mathop)(int, int), int x, int y) {
        return (*mathop)(x, y);
}

// main 函数调用
int main() {

// 用加法调用 domath
int a = domath(add, 10, 2);
printf("Add gives: %d\n", a);

// 用减法调用 domath
int b = domath(subtract, 10, 2);
```

```
printf("Subtract gives: %d\n", b);
}
```

我们来一步一步分析。

1. 我们有两个特征标相同的函数，add 和 subtract，它们都返回一个整数并接受两个整数作为参数。
2. 在第六行，我们定义了函数 int domath(int (*mathop)(int, int), int x, int y)。它第一个参数 int (*mathop)(int, int)是一个函数指针，指向返回一个整数并接受两个整数作为参数的函数。这就是我们之前见过的语法，没有任何不同。它的后两个整数参数则作为简单的输入。因此，这是一个接受一个函数指针和两个整数作为参数的函数。
3. 19 到 21 行，domath 函数将自己的后两个整数参数传递给函数指针并调用它。当然，也可以像这么调用。mathop(x, y);
4. 27 到 31 行出现了我们没见过的代码。我们用函数名作为参数调用了 domath 函数。就像我之前说过的，函数名是函数的地址，而且能代替函数指针使用。

main 函数调用了两次 domath 函数，一次用了 add，一次用了 subtract，并输出了这两次结果。

函数名和地址

既然有约在先，那我们就讨论一下函数名和地址作为结尾吧。一个函数名（或称标签），被转换成了一个指针本身。这表明在函数指针被要求当作输入的地方，就能够使用函数名。这也导致了一些看起来很糟糕的代码却能够正确的运行。瞧瞧下面这个例子。

```
#include <stdio.h>

// 函数原型
void add(char *name, int x, int y);

// 加法 x + y
void add(char *name, int x, int y) {
        printf("%s gives: %d\n", name, x + y);
}

// main 函数调用
int main() {

        // 一些糟糕的函数指针赋值
        void (*add1Ptr)(char*, int, int) = add;
        void (*add2Ptr)(char*, int, int) = *add;
        void (*add3Ptr)(char*, int, int) = &add;
        void (*add4Ptr)(char*, int, int) = **add;
        void (*add5Ptr)(char*, int, int) = ***add;

        // 仍然能够正常运行
        (*add1Ptr)("add1Ptr", 10, 2);
        (*add2Ptr)("add2Ptr", 10, 2);
```

```
        (*add3Ptr)("add3Ptr", 10, 2);
        (*add4Ptr)("add4Ptr", 10, 2);
        (*add5Ptr)("add5Ptr", 10, 2);

        // 当然，这也能运行
        add1Ptr("add1PtrFunc", 10, 2);
        add2Ptr("add2PtrFunc", 10, 2);
        add3Ptr("add3PtrFunc", 10, 2);
        add4Ptr("add4PtrFunc", 10, 2);
        add5Ptr("add5PtrFunc", 10, 2);
}
```

这是一个简单的例子。运行这段代码，你会看到每个函数指针都会执行，只是会收到一些关于字符转换的警告。但是，这些函数指针都能正常工作。

1. 在第 15 行，add 作为函数名，返回这个函数的地址，它被隐式的转换为一个函数指针。我之前提到过，在函数指针被要求当作输入的地方，就能够使用函数名。
2. 在第 16 行，解引用符作用于 add 之前，即*add，在返回在这个地址的函数。之后跟函数名一样，它被隐式的转换为一个函数指针。
3. 在第 17 行，取地址符作用于 add 之前，即&add，返回这个函数的地址，之后又得到一个函数指针。
4. 18 到 19 行，add 不断地解引用自身，不断返回函数名，并被转换为函数指针。到最后，它们的结果都和函数名没有区别。

显然，这段代码不是优秀的实例代码。我们从中收获到了如下知识：其一，函数名会被隐式的转换为函数指针，就像作为参数传递的时候，数组名被隐式的转换为指针一样。在函数指针被要求当作输入的任何地方，都能够使用函数名。其二，解引用符*和取地址符&用在函数名之前基本上都是多余的。

总结

本文帮助大家认清了函数指针以及它的用途。只要你掌握了函数指针，它就是 C 语言中一个强大的工具。也许会在以后的文章中讲述更多函数指针的细节用法，包括回调和 C 语言中基本的面向对象等等。