9_函数重载分析_下

重载与指针

> 重载函数的函数名看上去是一样的，但实际上每个函数的地址都是不一样的

> 当函数重载遇到函数指针会发生什么?

> 以下程序实际调用了哪个函数?

`lesson_9-1.cpp`

```cpp
#include <stdio.h>
#include <string.h>

int func(int);
int func(int, int);
int func(const char *);

typedef int(*PFUNC)(int);


int main(void)
{

    int c = 0;
    PFUNC p = func;
    c = p(1);

    printf("c = %d\n", c);


    return 0;
}

int func(int x)
{
    return x;
}

int func(int a, int b)
{
    return a + b;
}

int func(const char * s)
{
    return strlen(s);
```

```
37 }
```

输出

```
1 root@ubuntu:~/exp/DT_CPP/part01# ./a.out
2 c = 1
```

- `函数重载`遇上`函数指针`
  - 将`重载函数名`赋值给指针时
    - 根据`重载规则`挑选与`函数指针参数列表一致`的候选者
    - 候选者的函数类型与函数指针指向的函数的`函数类型`要完全一致，**包括返回值类型也要一致**

> 根据以上规则：
>
> 1. 挑出所有同名的函数，即函数名为`func`作为候选者，然后再继续筛选出与`函数指针参数列表一致`的候选者
> 2. 再根据第二条规则，筛选出来的候选者与函数指针指向的`函数类型`的返回值类型是否一致。如有一致的，那么即调用的就是该函数，否则匹配失败

`lesson_9-1-error1.cpp`

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int func(int);
5  int func(int, int);
6  int func(const char *);
7
8  typedef double(*PFUNC)(int);  // only modify this line
9
10
11 int main(void)
12 {
13
14     int c = 0;
15     PFUNC p = func;
16     c = p(1);
17
18     printf("c = %d\n", c);
19
20
21     return 0;
22 }
23
24 int func(int x)
```

```
{
    return x;
}

int func(int a, int b)
{
    return a + b;
}

int func(const char * s)
{
    return strlen(s);
}

```

输出

```
root@ubuntu:~/exp/DT_CPP/part01# g++ lesson_9-1-error1.cpp
lesson_9-1-error1.cpp: In function ‘int main()’:
lesson_9-1-error1.cpp:15:15: error: no matches converting function ‘func’ to type ‘PFUNC’ {aka ‘double(*)(int)’}
   15 |     PFUNC p = func;
      |               ^~~~
lesson_9-1-error1.cpp:6:5: note: candidates are: ‘int func(const char*)’
    6 | int func(const char *);
      |     ^~~~
lesson_9-1-error1.cpp:5:5: note:                 ‘int func(int, int)’
    5 | int func(int, int);
      |     ^~~~
lesson_9-1-error1.cpp:4:5: note:                 ‘int func(int)’
    4 | int func(int);
      |     ^~~~
lesson_9-1-error1.cpp:16:10: error: void value not ignored as it ought to be
   16 |     c = p(1);
      |         ~^~~
```

注意

- 函数重载 必然发生在同一个作用域中
- 编译器需要用 参数列表 或 函数类型 进行函数选择
  - 调用重载函数时需要关注 参数列表 来进行函数选择
  - 当函数重载遇到函数指针的时，关注的就是 函数类型
- 无法 直接通过 函数名 得到重载函数的 入口地址，不是重载函数依旧可以通过函数名得到函数入口地址

C++ 和 C相互调用

0x1 C++ 调用 C库

- C++编译器能够`兼容C语言的编译方式`
- C++编译器会`优先使用C++编译的方式`
- `extern`关键字能`强制让C++编译器进行C方式的编译`

`synax`

```
extern "C"
{
    // C style
}

extern "C" type name;
```

lesson_9-2

`add.h`

```
int add(int a, int b);
```

`add.c`

```
#include "add.h"

int add(int a, int b)
{
    return a + b;
}
```

`main1.cpp`

```
// g++ compile error, but gcc compile success
#include <stdio.h>
#include "add.h"

int main(void)
{
    printf("result = %d\n", add(1, 2));

    return 0;
}
```

进行编译

```
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# g++ main.cpp add.o
/usr/bin/ld: /tmp/ccjyfrFQ.o: in function `main':
main.cpp:(.text+0x13): undefined reference to `add(int, int)'
collect2: error: ld returned 1 exit status
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# nm add.o
0000000000000000 T add
```

> 以上提示 `add(int, int)` 未定义，使用了 `nm` 查看 `add.o` 是否包含 `add` 函数，发现是存在的。

编译失败的原因是：
`add.o` 是用 C语言编译器 将 `add.c` 源码 编译出来的目标文件， `add.o` 文件中确实存在 `add` 函数。
如果用 C++编译器编译 `add.c` 源码，编译出来的目标文件并没有 `add` 函数，而是 `_Z3addii` 。
看到这就明白了， 那么 `main.cpp` 使用的 `add` 函数，实际上是 `_Z3addii` 这个函数，所以在 链接的阶段，是不存在 `_Z3addii` 函数，那么就表明该函数是未定义的

> 为什么 使用 C++编译器编译会更改了原有的函数名?
> 这是因为在C++需要支持函数重载，所以使用符号修饰机制，即使用C++编译器 编译，原有的函数名都被修饰了。
> 由于 `add.o` 是 C编译器编译， `add` 还是 `add` ，但 `main.cpp` 是被 C++编译器编译，main 中的 `add` 已经不是原来的 `add` 。
> 所以 `main.cpp` 中的 `add` 和 `add.o` 中的符号不一致

解决符号不一致的问题，就使用 `extern` 关键字。就是让C++编译器编译 `main.cpp` 时，按照C语言编译器的规则去编译，也就是让 `add` 不受C++符号修饰机制的影响。

```
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# gcc -c add.c -o add.o
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# nm add.o
0000000000000000 T add
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# g++ -c add.c -o add_cpp.o
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# nm add_cpp.o
0000000000000000 T _Z3addii
```

在 `main.cpp` 中使用 `extren` 对函数声明括起来，表示不要修饰里面的函数名，以下是修改后的源码

`main2.cpp`

```
// g++ compile success, but gcc compile error
#include <stdio.h>

extern "C"
{

#include "add.h"

}
```

```
10
11  int main(void)
12  {
13      printf("result = %d\n", add(1, 2));
14      return 0;
15  }
```

Operation

```
1   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# ls
2   add.c  add.h  CPP_Call_C_Function  main.cpp
3   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# gcc -c add.c -o add.o
4   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# ls
5   add.c  add.h  add.o  CPP_Call_C_Function  main.cpp
6   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# g++ main.cpp add.o
7   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# ls
8   add.c  add.h  add.o  a.out  CPP_Call_C_Function  main.cpp
9   root@ubuntu:~/exp/DT_CPP/part01/lesson9_2# ./a.out
10  result = 3
11  root@ubuntu:~/exp/DT_CPP/part01/lesson9_2#
```

最后以上源码是能够被C++编译器编译了，那怎么才能在C++能够编译的前提下也让C语言编译器也支持编译?
使用条件编译 `ifdef`，如下程序:

`main3.c`

```
1   // gcc and g++ compile success
2   #include <stdio.h>
3
4   #ifdef __cplusplus
5   extern "C"
6   {
7   #endif
8
9   #include "add.h"
10
11  #ifdef __cplusplus
12  }
13  #endif
14
15  int main(void)
16  {
17      printf("return = %d\n", add(1, 2));
18      return 0;
19  }
```

FQA

1. 为什么在 `add.c` 里包含 `add.h`？在 `main.cpp` 包含不就足够了吗?

注意事项

1. C++编译器 `不能以C的方式编译重载函数`，所以 `extern` 的括号内范围是不允许出现 `同名函数的`
2. `编译方式` 决定函数名被编译后的 `目标名`
    i. C++编译方式将 `函数名` 和 `参数列表` 编译成 `目标名`
    ii. C编译方式只将 `函数名` 作为 `目标名` 进行编译

小结

1. `函数重载` 是C++对C的一个重要升级
2. 函数重载通过 `函数参数列表区分不同的同名函数`
3. `extern` 关键字能够实现 `C 和 C++` 的相互调用
4. `编译方式决定` 符号表中的函数名的最终 `目标名`

0x2 C 调用 C++库

> 根据原理，C调用 C++的库，那么C++的库里面的函数名等是被g++编译器修饰后的，所以C是无法直接调用C++的库，可以通过 创建中间接口，C可以直接调用中间接口，中间接口再调用C++的库
> 然后 `extern "C"` 作用在 中间接口中。

`cppFunc.h` C++库头文件

```
int cppadd(int a, int b);
```

`cppFunc.cpp` C++库，注意是编译好的库，正常场景是没有源码的

```
int cppadd(int a, int b)
{
    return a + b;
}
```

`lib_mid.h` 中间接口头文件

```
int lib_mid(int a, int b);
```

`lib_mid.cpp` 中间接口

```
#include "cppFunc.h"

```

```
#ifdef __cplusplus
extern "C"
{
#endif

int lib_mid(int a, int b)
{

    int result = cppadd(a, b);

    return result;
}

#ifdef __cplusplus
}
#endif
```

`main.c` 主程序

```
#include <stdio.h>
#include "lib_mid.h"

int main(void)
{
    int result = lib_mid(1, 2);
    printf("result = %d\n", result);

    return 0;
}
```

```
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# ls -l
total 20
-rw-r--r-- 1 root root  48 Apr 20 08:49 cppFunc.cpp
-rw-r--r-- 1 root root  22 Apr 20 07:51 cppFunc.h
-rw-r--r-- 1 root root 175 Apr 20 08:09 lib_mid.cpp
-rw-r--r-- 1 root root  23 Apr 20 07:59 lib_mid.h
-rw-r--r-- 1 root root 144 Apr 20 08:48 main.c
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# g++ -c cppFunc.cpp
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# g++ -c lib_mid.cpp
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# ls -l
total 28
-rw-r--r-- 1 root root   48 Apr 20 08:49 cppFunc.cpp
-rw-r--r-- 1 root root   22 Apr 20 07:51 cppFunc.h
-rw-r--r-- 1 root root 1392 Apr 20 08:53 cppFunc.o  # new
-rw-r--r-- 1 root root  175 Apr 20 08:09 lib_mid.cpp
-rw-r--r-- 1 root root   23 Apr 20 07:59 lib_mid.h
-rw-r--r-- 1 root root 1576 Apr 20 08:53 lib_mid.o  # new
-rw-r--r-- 1 root root  144 Apr 20 08:48 main.c
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# gcc main.c lib_mid.o
cppFunc.o
```

```
root@ubuntu:~/exp/DT_CPP/part01/lesson9_2/C_Call_CPP_Function# ./a.out
result = 3
```

> g++ 编译 C++库，以及中间接口
> `extern "C"` 作用于中间接口。
> 然后暴露出`原有的函数名`给C编译器编译的程序进行调用
> 这就属于间接调用C++库

Refer

1. 《程序员的自我修养：链接、装载与库》： chapter3.5.3