

11_25---函数重载

MFC应用程序就是带界面的 有客户端的 qq 微信什么的 但这种技术相对落后

3. C++输入&输出

vs.c文件就调用c编译器 .cpp文件就调用cpp编译器,自动识别

控制台应用程序就是黑框框

说明:

1. 使用cout标准输出(控制台)和cin标准输入(键盘)时, 必须包含< iostream >头文件以及std标准命名空间。注意: 早期标准库将所有功能在全局域中实现, 声明在.h后缀的头文件中, 使用时只需包含对应头文件 即可, 后来将其实现在std命名空间下, 为了和C头文件区分, 也为了正确使用命名空间, 规定C++头文件不带.h; 旧编译器(vc 6.0)中还支持格式, 后续编译器已不支持, 因此推荐使用+std的方式。vc6.0不支持命名空间 好多c的语法不支持 它也是微软出的编译器 只是比较老了 vs2013 19的爷爷辈
2. Devc++ 不需要安装 很小, 但是调试和做一些大型项目很不方便
3. windows平台下最好的编译器就是vs系列 (推荐2010以上) linux是gcc g++
4. 使用C++输入输出更方便, 不需增加数据格式控制, 比如: 整形--%d, 字符--%c

```
1  #include <iostream> //某些平台下面间接包含了printf头文件
2  using namespace std; //日常练习为了方便就全部展开c++库的命名空间
3  int main()
4  {
5      cout << "hello world" << endl;
6      cout << "hello world" << '\n';
7      cout << "hello world\n";
8      printf("hello world\n");
9      int a = 0;
10     double d = 3.14;
11     //机制和scanf一样 都会清空缓冲区
12     cout << a << ' ' << d << endl;
13     //<<流插入运算符
14     cin >> a >> d;
15     //<<流提取运算符
16     printf("%d %lf\n", a, d); //控制输出宽度还是建议printf 或者查文档也可以
17     cout << a << ' ' << d << endl;
18 }
```

c语言和cpp的输入 cin和scanf多项值都是以空格或者换行间隔数据的, cin scanf不读取空格, 认为空格是数据间隔 忽略掉

c语言printf scanf可以和c++cout cin混着用 推荐都可以用, 哪个方便就用哪个

cout比printf慢就相当于说前置++比后置++快 对于早期的计算机或许有这样的差异 对于现在来说已经没有什么差异, cpu发展非常迅速。

摩尔定律:

- 1、集成电路芯片上所集成的电路的数目, 每隔18个月就翻一番;
- 2、微处理器的性能每隔18个月提高一倍, 而价格下降一半;

3、用一美元所能买到的计算机性能，每隔18个月翻两番

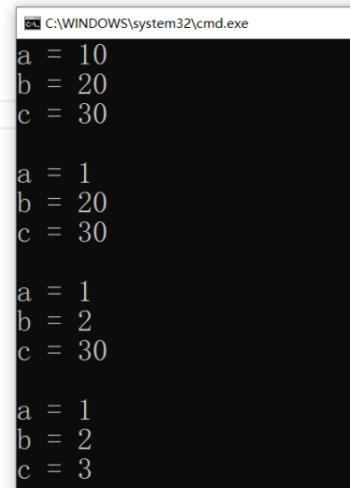
4. 缺省参数

缺省参数是声明或定义函数时为函数的参数指定一个默认值。在调用该函数时，如果没有指定实参则采用该默认值，否则使用指定的实参。

```
1 //缺省参数
2 void Func(int a = 0)
3 {
4     cout << a << endl;
5 }
6 int main()
7 {
8     Func(1); // 1是实参 a为形参传参时，使用指定的实参
9     Func(); // 没有传参时，使用参数的默认值 0为实参，a 为形参 0作为实参传给形参a
10 }
```

传参方式取决于平台，叫做调用惯例

```
56 // 全缺省：所有参数都给了缺省值
57
58 void Func(int a = 10, int b = 20, int c = 30)
59 {
60     cout << "a = " << a << endl;
61     cout << "b = " << b << endl;
62     cout << "c = " << c << endl << endl;
63 }
64
65 int main()
66 {
67     Func(); // 传参是从左向右传递
68     Func(1); // 1传给a
69     Func(1, 2); // 1传给a, 2传给b
70     Func(1, 2, 3);
71
72
73     return 0;
74 }
```



```
C:\WINDOWS\system32\cmd.exe
a = 10
b = 20
c = 30

a = 1
b = 20
c = 30

a = 1
b = 2
c = 30

a = 1
b = 2
c = 3
```

缺省参数不支持只缺少中间参数和不连续缺省，c++11加了包装器，有些也支持只传递中间的参数，但过于复杂。

语言之间都在互相学习

```

75 // 半缺省 -- 缺省部分参数 -- 必须从右往左缺省, 必须连续缺省
76 void Func(int a, int b = 20, int c = 30)
77 //void Func(int a, int b, int c = 30)
78 {
79     cout << "a = " << a << endl;
80     cout << "b = " << b << endl;
81     cout << "c = " << c << endl << endl;
82 }
83
84 // 调用方式至少传递没有缺省的参数的个数 就是a没有缺省, 至少传递一个参数
85 int main()
86 {
87     Func(1);
88     Func(1, 2);
89     Func(1, 2, 3);
90
91     return 0;
92 }

```

```

C:\WINDOWS\system32\cmd.exe
a = 1
b = 20
c = 30

a = 1
b = 2
c = 30

a = 1
b = 2
c = 3

请按任意键继续. . .

```

直播延迟意味着丢包严重, 网络知识。

```

int main()
{
    struct Stack st;
    //StackInit(&st); // 不知道栈最多存多少数据, 就用缺省值初始化
    StackInit(&st, 100); // 知道栈最多存100数据, 显示传值。这样可以减少增容次数, 提高效率
}

```

```

94 struct Stack
95 {
96     int* a;
97     int top;
98     int capacity;
99 };
100
101 void StackInit(struct Stack* ps, int capacity = 4)
102 {
103     ps->a = (int*)malloc(sizeof(int)*capacity);
104     //
105     ps->top = 0;
106     ps->capacity = capacity;
107 }
108
109 void StackPush(struct Stack* ps, int x)
110 {
111     // ...

```

增容有消耗

c++很多语法都在弥补c语言的不足, c++入门的一些零碎语法为了后面的类和对象打基础。

c++是面向对象的语言中大哥级别的, java避开了它的很多坑, python更简单, 三天就能差不多学会语法, 框架需要学习时间长。

vs建源文件.cpp的时候不要点成.h, 会出现很奇怪的错误。

2. 缺省参数不能在函数声明和定义中同时出现

要么在声明，要么在定义。推荐写在声明

```
//a.h
void Func(int a = 10);
```

```
// a.cpp
void Func(int a = 20)
{}
```

缺省值必须是常量或者全局变量

C语言不支持（编译器不支持）

// 注意：如果生命与定义位置同时出现，恰巧两个位置提供的值不同，那编译器就无法确定到底该用那个缺省值。

1.声明和定义一般指的是函数。

2.全局变量不能定义在.h文件中，.h文件会被包含的.cpp文件包含多次，出现全局变量重定义。

5. 函数重载

自然语言中，一个词可以有多重含义，人们可以通过上下文来判断该词真实的含义，即该词被重载了。比如：以前有一个笑话，国有两个体育项目大家根本不用看，也不用担心。一个是乒乓球，一个是男足。前者是“谁也赢不了！”，后者是“谁也赢不了！”

5.1函数重载概念

5.1 函数重载概念

函数重载：是函数的一种特殊情况，C++允许在同一作用域中声明几个功能类似的同名函数，这些同名函数的形参列表（参数个数 或 类型 或 顺序）不同，常用来处理实现功能类似数据类型不同的问题

// 返回值不同，不能构成重载 -- 调用的时候不能区分

```
int f(double d)
{
}

void f(double d)
{
}
```

f(1.1);

函数重载的判断条件只有函数参数个数/类型/顺序不同这三个条件

```
// 1、参数类型不同
int Add(int left, int right)
{
    cout << "int Add(int left, int right)" << endl;

    return left + right;
}

double Add(double left, double right)
{
    cout << "double Add(double left, double right)" << endl;

    return left + right;
}
```

```
// 3、参数顺序不同
void f(int a, char b)
{
    cout << "f(int a,char b)" << endl;
}
```

```
// 2、参数个数不同
void f()
{
    cout << "f()" << endl;
}

void f(int a)
{
    cout << "f(int a)" << endl;
}
```

```
int main()
{
    Add(10, 20);
    Add(10.1, 20.2);

    f();
    f(10);
}
```

不传参数存在二义性，不知道该调用哪个函数。

```
.91 // 1、缺省值不同，不能构成重载
.92 void f(int a)
.93 {
.94     cout << "f()" << endl;
.95 }
.96
.97 void f(int a = 0)
.98 {
.99     cout << "f(int a)" << endl;
100 }
101
```

已启动生成: 项目: 11-25, 配置: Debug Win32

错误 C2084: 函数“void f(int)”已有主体
d:\课堂代码\就业课102期\11-25\test.cpp(192): 参见“f”的前一个定义

```
// 2、构成重载，但是使用时会有问题：f(); // 调用存在歧义
void f()
{
    cout << "f()" << endl;
}

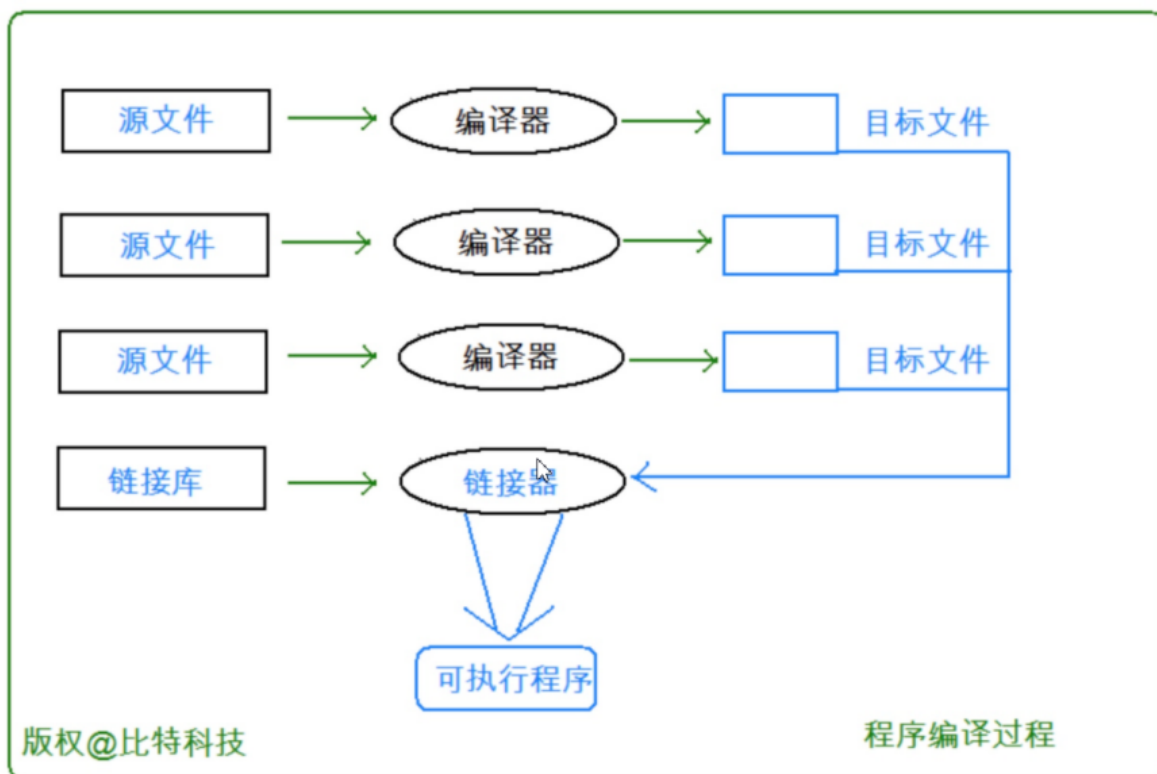
void f(int a = 0)
{
    cout << "f(int a)" << endl;
}
```

```
int main()
{
    // f(); // 调用存在歧义
    f(1);
}
```

5.2 名字修饰(name Mangling)

为什么C++支持函数重载，而C语言不支持函数重载呢？

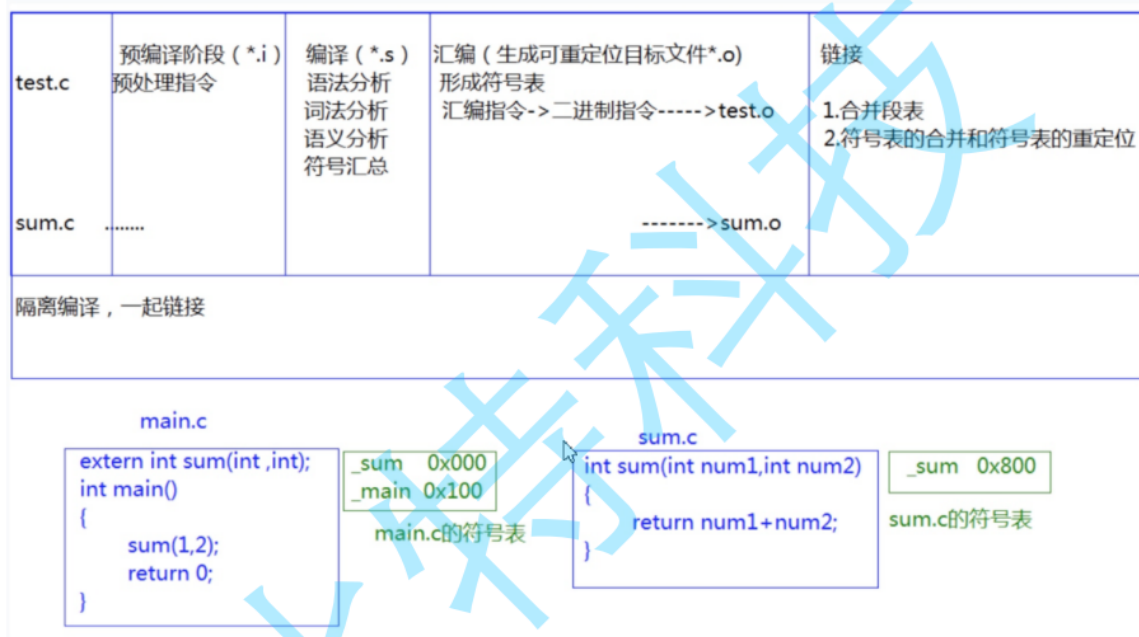
在C/C++中，一个程序要运行起来，需要经历以下几个阶段：预处理、编译、汇编、链接。



除非需要做编译器，深入学习编译链接

深入学习汇编 做驱动开发，硬件和软件系统的驱动

汇编是一门指令级别的语言代码，方便我们观察中间过程细节。



c语言不支持函数重载

```
1: func.h
1 #include<stdio.h>
2
3 void f();
4 void f(int a);
5
```

```
1: func.c
1 #include"func.h"
2
3 void f()
4 {
5     printf("f()\n");
6 }
7
8 void f(int a)
9 {
10    printf("f(int a)");
11 }
```

```
1: test.c
1 #include"func.h"
2
3 int main()
4 {
5     f();
6     return 0;
7 }
```

验证，c语言不支持函数重载

```
[xjh@VM-0-3-centos 11-25]$ gcc func.c test.c
func.c: In function 'f':
func.c:4:1: error: number of arguments doesn't match prototype
{
^
In file included from func.c:1:0:
func.h:4:6: error: prototype declaration
void f(int a);
^
func.c: At top level:
func.c:8:6: error: redefinition of 'f'
void f(int a)
^
func.c:3:6: note: previous definition of 'f' was here
void f()
```

通过gcc验证c语言语法，

```
[xjh@VM-0-3-centos 11-25]$ gcc func.c test.c
[xjh@VM-0-3-centos 11-25]$ ls
a.out func.c func.h test.c
[xjh@VM-0-3-centos 11-25]$ ./a.out
f()
```

C++兼容c，这个程序也可以用g++去编译，就支持重载

vs是根据文件后缀是去调用对应编译器。c就是c编译器 .cpp就是C++编译器
Linux 不用文件后缀区分，gcc编译就是c，g++就是cpp

```
[xjh@VM-0-3-centos 11-25]$ g++ func.c test.c
[xjh@VM-0-3-centos 11-25]$ ls
a.out func.c func.h test.c
[xjh@VM-0-3-centos 11-25]$ ./a.out
f()
f(int a)
```

C++支持重载

调用一个函数最重要的是拿到函数的地址 call后面跟着函数地址，然后执行串指令建立栈帧

为什么C语言不支持函数重载，而C++支持函数重载？C++是如何支持的。

```
1: func.h
1 #include<stdio.h>
2
3 void f();
4 void f(int a);
5
```

```
1: func.c
1 #include"func.h"
2
3 void f()
4 {
5     printf("f()\n");
6 }
7
8 void f(int a)
9 {
10    printf("f(int a)");
11 }
```

```
1: test.c
1 #include"func.h"
2
3 int main()
4 {
5     f();
6     return 0;
7 }
```

回顾一下编译器编译这个过程

func.h func.c test.c

// 1、预处理 -> 头文件展开、宏替换、条件编译、去掉注释

func.i test.i

// 2、编译 -> 检查语法，生成汇编代码

func.s test.s

// 3、汇编 -> 汇编代码转换成二进制机器码

func.o test.o

// 4、链接

a.out



```
1 #include<stdio.h>
2
3 void f();
4 void f(int a);
5
6 int main()
7 {
8     f();
9     f(1);
10    return 0;
11 }
```

C语言不支持函数重载，因为编译的时候，两个重载函数，函数名相同，在func.o符号表中存在歧义和冲突，其次链接的时候也存在歧义和冲突，因为他们都是直接使用函数名去标识和查找，而重载函数，函数名相同

函数只包含了func.h，func.h只有函数的声明吗，没有定义，找不到函数的地址，声明是承诺一件事，定义才是实现这件事。

call执行函数指令，f11进入函数先跳转jmp到函数地址 严格来说call后面的地址是jmp指令的地址，不是函数地址


```
test.c
1 #include "func.h"
2
3 int main()
4 {
5     f();
6     f(1);
7     return 0;
8 }

func.h
1 #include <stdio.h>
2
3 void f();
4 void f(int a);
```

C语言函数处理就是用函数名，按名称来看好像是一样，所以c不支持重载 链接错误是函数名

Test.obj : error LNK2019: 无法解析的外部符号 _StackInit, 该符号在函数 _main 中被引用

c++对 链接错误报错

外部符号 "void __cdecl StackInit(struct Stack *,int)" (?StackInit@@YAXPAUStack@@H@Z), 该符号在函数

c++把参数的规则带进函数命名中

```

0054601C rep stos dword ptr es:[edi]
f();
0054601E call f (05414B5h)
f(1);
00546023 push 1
00546025 call f (05414BAh)
0054602A add esp,4

```

```

void f()
{
005457C0 push ebp
005457C1 mov ebp,esp
005457C3 sub esp,0C0h
005457C9 push ebx
005457CA push esi
005457CB push edi
005457CC lea edi,[ebp-0C
005457D2 mov ecx,30h
005457D7 mov eax,0CCCCCCC
005457DC rep stos dword ptr e
cout << "f()" << endl;
esi,esp
}

```

1、如果在当前文件有函数的定义，那么编译时就填上地址了

1、如果在当前文件只有函数的声明，那么定义就在其他xxx.cpp中，编译时没有地址，只能链接的时候去其他xxx.o符号表中根据函数修饰名字去找，这就是链接的重要工作

func.s test.s

func.o test.o

```

1 #include<stdio.h>
2
3 void f();
4 void f(int a);
5
6 int main()
7 {
8     f();
9     f(1);
10    return 0;
11 }

```

g++的修饰函数名规则

_Z + 函数名长度 + 函数名 + 参数首字母

符号表

```

000000000040055d <_Z1fv>:
000000000040056d <_Z1fi>:

```

```

000000000040055d <_Z1fv>:
40055d: 55          push %rbp
40055e: 48 89 e5    mov %rsp,%rbp
400561: bf 30 06 40 mov $0x400630,%edi
400566: e8 e5 fe ff callq 400450 <puts@plt>
40056b: 5d          pop %rbp
40056c: c3          retq

000000000040056d <_Z1fi>:
40056d: 55          push %rbp
40056e: 48 89 e5    mov %rsp,%rbp
400571: 48 83 ec 10 sub $0x10,%rsp
400575: 89 7d fc    mov %edi,-0x4(%rbp)
400578: bf 34 06 40 mov $0x400634,%edi
40057d: e8 ce fe ff callq 400450 <puts@plt>
400582: c9          leaveq %rbp
400583: c3          retq

```

```

<main>:
55          push %rbp
48 89 e5    mov %rsp,%rbp
e8 d0 ff ff callq 40055d <_Z1fv>
bf 01 00 00 mov $0x1,%edi
e8 d6 ff ff callq 40056d <_Z1fi>
b8 00 00 00 mov $0x0,%eax
5d          pop %rbp
c3          retq
66 90      xchg %ax,%ax

```

链接时使用修饰后函数名进行查找

C++的目标文件符号表中不是直接用函数名来标识和查找函数。

- 1、函数名修饰规则，但是这个修饰规则，不同的编译器下面不同。
- 2、有了函数名修饰规则，只要参数不同，func.o 符号表里面重载的函数就不存在二义性和冲突了
- 3、链接的时候，test.o的main的函数里面去调用两个重载的函数，查找地址时，也是明确。

```

test.c
1 #include "func.h"
2
3 int main()
4 {
5     f();
6     f(1);
7     return 0;
8 }

```

程序执行 先进来call主函数，main函数在call两个重载函数 编译的时候只有声明，没有定义，编译时call的地址是问号 链接的时候把地址填入

void无参数 一个int 参数

填地址就是链接的一个步骤，链接先把改找到的地址找到填上，最后在合并。

链接是一个大步骤，分为很多小步骤

再熟悉一下修饰规则

```

0000000000400584 <_Z4funciPi>:
400584: 55          push %rbp
400585: 48 89 e5    mov %rsp,%rbp
400588: 89 7d fc    mov %edi,-0x4(%rbp)
40058b: 48 89 75 f0 mov %rsi,-0x10(%rbp)
40058f: 5d          pop %rbp
400590: c3          retq

```

```

void func(int a, int* p)
{
}

```

g++的修饰函数名规则

_Z + 函数名长度 + 函数名 + 参数首字母

这章节比较难的一部分，只有把这部分搞清楚了，才能把方方面面搞清楚。c++要学到一定深度。

```
00000000040052d <f>:
40052d: 55          push    %rbp
40052e: 48 89 e5    mov     %rsp,%rbp
400531: 48 83 ec 10  sub    $0x10,%rsp
400535: 89 7d fc    mov     %edi,-0x4(%rbp)
400538: bf 00 06 40 00 mov     $0x400600,%edi
40053d: e8 ce fe ff ff callq   400410 <puts@plt>
400542: c9         leaveq  %rbp
400543: c3         retq

000000000400544 <func>:
400544: 55          push    %rbp
400545: 48 89 e5    mov     %rsp,%rbp
400548: 89 7d fc    mov     %edi,-0x4(%rbp)
40054b: 48 89 75 f0  mov     %rsi,-0x10(%rbp)
40054f: 5d         pop     %rbp
400550: c3         retq
```

```
/*
void f()
{
    printf("f()\n");
}

void f(int a)
{
    printf("f(int a)\n");
}

void func(int a, int* p)
{
}
```