切换导航 Farseerfc的小窝

文

繁體

<u>简体</u>

English

日本語

- **≜** About
- <u> Links</u>
- ≜ About
- <u>Links</u>
- □ Import
- □ Life
- <u>□ Tech</u>

- <u>Import</u>
- □<u>Life</u>
- □ Tech
- Q

搜索

- 搜索
- <u>児</u>档
- 2

C++ Tricks 2.6 I386平台C函数的可变参数 表(Variable Arguments) <u></u>

2.6 I386平台C函数的可变参数表 (Variable Arguments)

基于前文(2.4节)分析,我们可以不通过函数签名,直接通过指针运算,来得到函数的参数。由于参数的压栈和弹出操作都由主调函数进行,所以被调函数对于参数的真实数量不需要知晓。因此,函数签名中的变量声明不是必需的。为了支持这种参数使用形式,C语言提供可变参数表。可变参数表的语法形式是在参数表末尾添加三个句点形成的省略号"…":

void g(int a,char* c,...);

省略号之前的逗号是可选的,并不影响词法语法分析。上面的函数g可以接受2个或2个以上的参数,前两个参数的类型固定,其后的参数类型未知,参数的个数也未知。为了知道参数个数,我们必须通过其他方法,比如通过第一个参数传递:

g(3,"Hello",2,4,5);//调用g并传递5个参数,其中后3 个为可变参数。

在函数的实现代码中,可以通过2.4节叙述的,参数 在栈中的排列顺序,来访问位于可变参数表的参 数。比如:

```
void g(int a,char* c...){
```

void *pc=&c;int* pi=static_cast<int*>(pc)+1;//将pi 指向首个可变参数

```
for(int i=0;i<a;i++)std::cout<<pi[i]<<" ";
```

std::cout<<c<<std::endl;

```
我们甚至可以让一个函数的所有参数都是可变参数,只要有办法获知参数的数量即可。比如,我们约定,在传递给addAll的参数都是int,并且最后一个以0结束:
```

int addAll(...);

}

int a=f(1,4,2,5,7,0);

```
那么addAll可以这样实现:
int addAll(...){
int sum=0;int *p=∑ //p指向第一个局部变量
p+=3; //跳过sum, ebp, eip, 现在p指向第一个参
数
for(;*p;++p) //如果p不指向0就继续循环
sum+=*p;
return sum;
}
可变参数表的最广泛应用是C的标准库函数中的格式
化输入输出:printf和scanf。
void printf(char *c,...);
void scanf(char *c,...);
两者都通过它的首个参数指出后续参数表中的参数
类型和参数数量。
```

如果可变参数表中的参数类型不一样,那么操纵可变参数表就需要复杂的指针运算,并且还要时刻注意边界对齐(align)问题,非常令人头痛。好在C标准库提供了用于操纵可变参数表的宏(macro)和结构(struct),他们被定义在库文件stdarg.h中:

typedef struct {char *p;int offset;} va_list;

#define va_start(valist,arg)

#define va_arg(valist,type)

#define va_end(valist)

其中结构va_list用于指示参数在栈中的位置,宏va_start接受一个va_list和函数的可变参数表之前的参数,通过第一个参数初始化va_list中的相应数据,因此要使用stdarg.h中的宏,你的可变参数表的函数必须至少有一个具名参数。va_arg返回下一个类型为type的参数,va_end结束可变参数表的使用。还是以上文的addAll为例,这次写出它的使用标准宏的版本:

int addAll(int i,...)

```
{
va list vl; //定义一个va list结构
va_start(vl,i); //用省略号之前的参数初始化vl
if(i=0)return 0; //如果第一个参数就是0,返回
int sum=i; //将第一个参数加入sum
for(;;){
i=va_arg(vl,int); //取得下一个参数,类型是sum
if(i==0)break; //如果参数是0,跳出循环
sum+=i;
}
va end(vl);
return sum;
}
```

可以看出,如果参数类型一致,使用标准库要多些

几行代码。不过如果参数类型不一致或者未知 (printf的情况),使用标准库就要方便很多,因为我们很难猜出编译器处置边界对齐(align)等汇编代码的细节。使用标准库的代码是可以移植的,而使用上文所述的其它方法操纵可变参数表都是不可移植的,仅限于在I386平台上使用。

纵使可变参数表有使用上的便利性,它的缺陷也有很多,不可移植性和平台依赖性只是其一,最大的问题在于它的类型不安全性。使用可变参数表就意味着编译器不对参数作任何类型检查,这在C中算是一言难尽的历史遗留问题,在C++中就意味着恶魔reinterpret_cast被你唤醒。C的可变参数表是C++代码错误频发的根源之一,以至于C++标准将可变参数表列为即将被废除的C语言遗留特性。C++语法中的许多新特性,比如重载函数、默认参数值、模板,都可以一定程度上替代可变参数表,并且比可变参数表更加安全。

可变参数表在C++中惟一值得嘉奖的贡献,是在模板 元编程(TMP)的SFINAE技术中利用可变参数表制作 最差匹配重载。根据C++标准中有关函数重载决议的 规则,具有可变参数表的函数总是最差匹配,编译 器在被逼无奈走头无路时才会选择可变参数表。利用这一点,我们可以精心制作重载函数来提取类型信息。比如,要判断一个通过模板传递来的类型是不是int:

```
long isIntImp(int);
char isIntImp(...);
template<typename T>
struct isInt
{
```

然后,在一个具有模板参数T的函数中,我们就可以 写

enum{value=sizeof(isIntImp(T()))==sizeof(long);}

if(isInt<T>::value)//...

}

在这个(不怎么精致的)例子中,如果T是int,那么 isIntImp的第一个重载版本就会被选中,返回值类 型就是long,这样value就为1。否则,编译器只能选中第二个具有可变参数表的重载版本,返回值类型成为char,这样value就为0。把它说得再明白一些,上文的代码所表达的意思是:如果类型T是int,那它就是int,否则它就不是int,呵呵简单吧。这种通过重载决议规则来提取类型信息的技术,在模板元编程中被称作SFINAE,它和其它模板元编程技术被广泛运用于STL、Boost等模板库的开发实现之中。

值得注意的是,在上文SFINAE的运用中,isIntImp并没有出现定义而只提供了声明,因为我们并没有实际调用isIntImp函数,而只是让它参与重载决议并用sizeof判断其返回值类型。这是C++的一个设计准则的完美体现:不需要的东西可以不出现。由于这一准则,我们避免了在C++中调用具有可变参数表的函数这一危险举动,而仅仅利用了可变参数表在语法分析过程中的特殊地位,这种对于危险语言特性的巧妙利用是善意而无害的。

