C++ Tricks 2.6 I386 平臺C函數的可變參數表(Variable Arguments)

從 farseerfc.wordpress.com 導入

2.6 I386平臺C函數的可變參數表(Variable Arguments)

基於前文(2.4節)分析，我們可以不通過函數簽名，直接通過指針運算，來得到函數的參數。由於參數的壓棧和彈出操作都由主調函數進行，所以被調函數對於參數的真實數量不需要知曉。因此，函數簽名中的變量聲明不是必需的。爲了支持這種參數使用形式，C語言提供可變參數表。可變參數表的語法形式是在參數表末尾添加三個句點形成的省略號“...”：

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

省略號之前的逗號是可選的，並不影響詞法語法分析。上面的函數g可以接受2個或2個以上的參數，前兩個參數的類型固定，其後的參數類型未知，參數的個數也未知。爲了知道參數個數，我們必須通過其他方法，比如通過第一個參數傳遞：

g(3,”Hello”,2,4,5);//調用g並傳遞5個參數，其中後3個爲可變參數。

在函數的實現代碼中，可以通過2.4節敘述的，參數在棧中的排列順序，來訪問位於可變參數表的參數。比如:

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

void *pc=&c;int* pi=static_cast<int*>(pc)+1;//將pi指向首個可變參數

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

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

}

我們甚至可以讓一個函數的所有參數都是可變參數，只要有辦法獲知參數的數量即可。比如，我們約定，在傳遞給addAll的參數都是int，並且最後一個以0結束：

int addAll(...);

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

那麼addAll可以這樣實現：

int addAll(...){

int sum=0;int *p=∑ //p指向第一個局部變量

p+=3; //跳過sum，ebp，eip，現在p指向第一個參數

for(;*p;++p) //如果p不指向0就繼續循環

sum+=*p;

return sum;

}

可變參數表的最廣泛應用是C的標準庫函數中的格式化輸入輸出：printf和scanf。

void printf(char *c,...);

void scanf(char *c,...);

兩者都通過它的首個參數指出後續參數表中的參數類型和參數數量。

如果可變參數表中的參數類型不一樣，那麼操縱可變參數表就需要複雜的指針運算，並且還要時刻注意邊界對齊(align)問題，非常令人頭痛。好在C標準庫提供了用於操縱可變參數表的宏(macro)和結構(struct)，他們被定義在庫文件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)

其中結構va_list用於指示參數在棧中的位置，宏va_start接受一個va_list和函數的可變參數表之前的參數，通過第一個參數初始化va_list中的相應數據，因此要使用stdarg.h中的宏，你的可變參數表的函數必須至少有一個具名參數。va_arg返回下一個類型爲type的參數，va_end結束可變參數表的使用。還是以上文的addAll爲例，這次寫出它的使用標準宏的版本：

int addAll(int i,...)

```
{
va_list vl; //定義一個va_list結構
va_start(vl,i); //用省略號之前的參數初始化vl
if(i=0)return 0; //如果第一個參數就是0，返回
int sum=i; //將第一個參數加入sum
for(;;){
i=va_arg(vl,int); //取得下一個參數，類型是sum
if(i==0)break; //如果參數是0，跳出循環
sum+=i;
}
va_end(vl);
return sum;
}
```

可以看出，如果參數類型一致，使用標準庫要多些幾行代碼。不過如果參數類型不一致或者未知(printf的情況)，使用標準庫就要方便很多，因爲我們很難猜出編譯器處置邊界對齊(align)等彙編代碼的細節。使用標準庫的代碼是可以移植的，而使用上文所述的其它方法操縱可變參數表都是不可移植的，僅限於在I386平臺上使用。

縱使可變參數表有使用上的便利性，它的缺陷也有很多，不可移植性和平臺依賴性只是其一，最大的問題在於它的類型不安全性。使用可變參數表就意味着編譯器不對參數作任何類型檢查，這在C中算是一言難盡的歷史遺留問題，在C++中就意味着惡魔reinterpret_cast被你喚醒。C的可變參數表是C++代碼錯誤頻發的根源之一，以至於C++標準將可變參數表列爲即將被廢除的C語言遺留特性。C++語法中的許多新特性，比如重載函數、默認參數值、模板，都可以一定程度上替代可變參數表，並且比可變參數表更加安全。

可變參數表在C++中惟一值得嘉獎的貢獻，是在模板元編程(TMP)的SFINAE技術中利用可變參數表製作最差匹配重載。根據C++標準中有關函數重載決議的規則，具有可變參數表的函數總是最差匹配，編譯器在被逼無奈走頭無路時纔會選擇可變參數表。利用這一點，我們可以精心製作重載函數來提取類型信息。比如，要判斷一個通過模板傳遞來的類型是不是int：

```
long isIntImp(int);

char isIntImp(...);

template<typename T>

struct isInt

{

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

}
```

然後，在一個具有模板參數T的函數中，我們就可以寫

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

在這個(不怎麼精緻的)例子中，如果T是int，那麼isIntImp的第一個重載版本就會被選中，返回值類型就是long，這樣value就爲1。否則，編譯器只能選中第二個具有可變參數表的重載版本，返回值類型成爲char，這樣value就爲0。把它說得再明白一些，上文的代碼所表達的意思是：如果類型T是int，那它就是int，否則它就不是int，呵呵簡單吧。這種通過重載決議規則來提取類型信息的技術，在模板元編程中被稱作SFINAE，它和其它模板元編程技術被廣泛運用於STL、Boost等模板庫的開發實現之中。

值得注意的是，在上文SFINAE的運用中，isIntImp並沒有出現定義而只提供了聲明，因爲我們並沒有實際調用isIntImp函數，而只是讓它參與重載決議並用sizeof判斷其返回值類型。這是C++的一個設計準則的完美體現：不需要的東西可以不出現。由於這一準則，我們避免了在C++中調用具有可變參數表的函數這一危險舉動，而僅僅利用了可變參數表在語法分析過程中的特殊地位，這種對於危險語言特性的巧妙利用是善意而無害的。