

【1】引用**【2】const 引用****【3】引用传递****【4】引用作为函数返回值****【5】引用和指针的区别****【1】引用**

(1) 引用就是给一个变量**取别名**。(应用不是变量，它只是某个变量的别名而已)。

我们可以回忆下，变量的属性，变量的属性有两个，一个是**名称**，一个是**空间的属性**。

变量（名称，空间）。引用不是变量，仅仅只是变量的别名，引用**没有自己独立的空间**。

引用要和它所引用的变量共享空间。既然他们共享空间，那么对引用所做的改变就是对它所引用的变量所做的改变。

(2) 引用的一般格式：

1. **类型 &引用名=变量名**；

2. 例如，int a=1；

int &b=a; //b 是 a 的别名，因此 a 和 b 是同一个单元。

注意：引用一定要初始化，指明该引用变量是谁的别名。

(3) 在实际的应用中，引用一般用做参数传递与返回值。

接下来我们举个例子说明一下：（引用在定义的时候必须初始化）

```

#include <iostream>
using namespace std;
//引用不是变量
//引用只是变量的别名
//引用没有自己独立的空间
//引用要和所引用的变量共享空间
//对引用变量的改变实际上是对它所引用的变量的改变
//注意：引用在定义的时候要初始化
int main(void)
{
    int val=100;
    int &refval;
    //int &refval=val;
    return 0;
}

```

输出

显示输出来源(S): 生成

启动时间为 9/22/2016 10:17:18 AM。

tiatizeBuildStatus:

E在创建 "Debug\04.unsuccessfulbuild", 因为已指定 "AlwaysCreate"。

ompile:

1.cpp

Program Files\MSBuild\Microsoft.Cpp\v4.0\Platforms\Win32\Microsoft.Cpp.Win32.Targets(147,5): error MSB6006: "CL.exe" 已退出, 代码为 2。

cppnote\04\01.cpp(12): error C2530: "refval": 必须初始化引用

失败。

时间 00:00:00.50

==== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====

对引用进行赋值就是对所引用的变量所做的修改，例如：

```

#include <iostream>
using namespace std;
//引用不是变量
//引用只是变量的别名
//引用没有自己独立的空间
//引用要和所引用的变量共享空间
//对引用变量的改变实际上是对它所引用的变量的改变
//注意：引用在定义的时候要初始化
int main(void)
{
    int val=100;
    //int &refval; //Error: 引用必须初始化
    int &refval=val;
    refval=200; //将200赋值给refval这个引用，
               //那么对引用所做的修改，就是对所引用的变量所做的修改。
    cout<<"val="<<val<<endl;
    cout<<"refval="<<refval<<endl;
    return 0;
}

```

```

C:\Windows\system32\cmd.exe
val=200
refval=200
请按任意键继续. . .

```

引用一旦初始化不得重新指向其他变量。如下：

```

int main(void)
{
    int val=100;
    //int &refval; //Error: 引用必须初始化
    int &refval=val;
    refval=200; //将200赋值给refval这个引用，
               //那么对引用所做的修改，就是对所引用的变量所做的修改。
    cout<<"val="<<val<<endl;
    cout<<"refval="<<refval<<endl;
    cout<<"*****"<<endl;

    int val2=550;
    refval=val2; //这不代表将refval引用至val2变量
               //仅仅只代表将refval赋值给refval
               //这意味着改变的还是改变的refval引用的变量val
    cout<<"val="<<val<<endl;
    cout<<"refval="<<refval<<endl;
    return 0;
}

```

```

C:\Windows\system32\cmd.exe
val=200
refval=200
*****
val=550
refval=550
请按任意键继续. . .

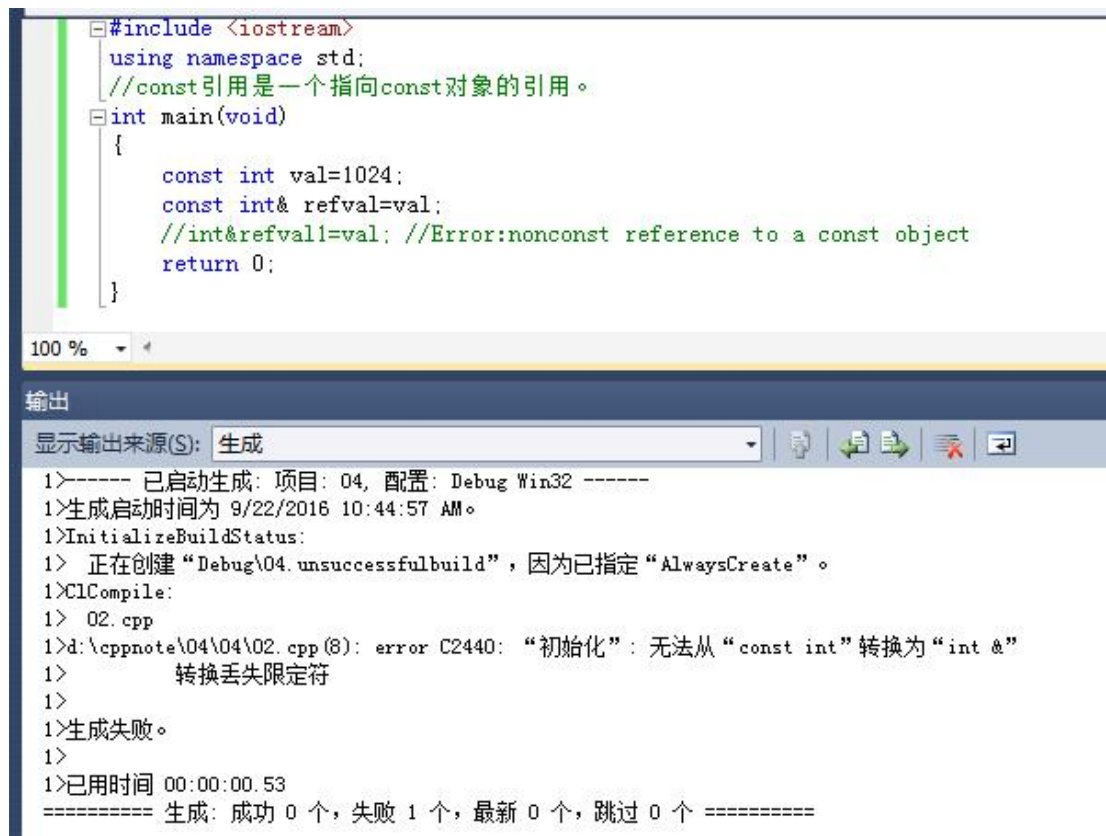
```

【2】const 引用

(1) Const 引用是指向 const 对象的引用。

```
const int ival=1024;
const int& refval=ival;//ok:both reference and object are const
int &ref2=ival;//error:nonconst reference to a const object
```

例如：



```
#include <iostream>
using namespace std;
//const引用是一个指向const对象的引用。
int main(void)
{
    const int val=1024;
    const int& refval=val;
    //int&refval=val; //Error:nonconst reference to a const object
    return 0;
}
```

输出

显示输出来源(S): 生成

```
1>----- 已启动生成: 项目: 04, 配置: Debug Win32 -----
1>生成启动时间为 9/22/2016 10:44:57 AM。
1>InitializeBuildStatus:
1> 正在创建 "Debug\04.unsuccessfulbuild", 因为已指定 "AlwaysCreate"。
1>ClCompile:
1> 02.cpp
1>d:\cppnote\04\04\02.cpp(8): error C2440: "初始化": 无法从 "const int" 转换为 "int &"
1>    转换丢失限定符
1>
1>生成失败。
1>
1>已用时间 00:00:00.53
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====
```

为什么是不允许的呢？如果是允许的会怎样呢？如果是合法的，那么意味着可以给 refval 赋值，也就是可以更改 val 这个常量。

接下来我们定义一个变量 int val2=1024，那么我们能不能定义一个 const 引用指向这个变量呢？

```
#include <iostream>
using namespace std;
//const引用是一个指向const对象的引用。
int main(void)
{
    const int val=1024;
    const int& refval=val;
    //int&refval1=val; //Error:nonconst reference to a const object
    //refval=200; //refval它是一个常量，不能赋值。
    int val2=1024;
    const int& refval2=val2;    //定义一个变量，然后用一个const引用指向它是允许的。
    cout<<"refval2="<<refval2<<endl;
    return 0;
}
```

C:\Windows\system32\cmd.exe

refval2=1024
请按任意键继续. . .

也就是说 const 引用可以指向普通变量（非 const 对象）。当然它是不能更改的，不能够给常量赋值。

(全局范围) main(void)

```
#include <iostream>
using namespace std;
//const引用是一个指向const对象的引用。
int main(void)
{
    const int val=1024;
    const int& refval=val;
    //int&refval1=val; //Error:nonconst reference to a const object
    //refval=200; //refval它是一个常量，不能赋值。
    int val2=1024;
    const int& refval2=val2;    //定义一个变量，然后用一个const引
    cout<<"refval2="<<refval2<<endl;
    refval2=100;
    return 0;
}
```

100 %

输出

显示输出来源(S): 生成

1>----- 已启动生成: 项目: 04, 配置: Debug Win32 -----
1>生成启动时间为 9/22/2016 10:58:30 AM。
1>InitializeBuildStatus:
1> 正在对“Debug\04.unsuccessfulbuild”执行 Touch 任务。
1>ClCompile:
1> 02.cpp
1>d:\cppnote\04\04\02.cpp (13): error C3892: “refval2”: 不能给常量赋值
1>

接下来我们再看一个例子（double 类型的变量，对应 const int 类型的引用）：

```
//refval2=100; //Error: 常量不能赋值
//*****
double val3=3.14;
const int& refval3=val3;
return 0;
}
```

100 %

输出

显示输出来源(S): 生成

1>----- 已启动生成: 项目: 04, 配置: Debug Win32 -----
1>生成启动时间为 9/22/2016 11:02:50 AM。
1>InitializeBuildStatus:
1> 正在对“Debug\04.unsuccessfulbuild”执行 Touch 任务。
1>ClCompile:
1> 02.cpp
1>.\cppnote\04\04\02.cpp (17): warning C4244: “初始化”: 从“double”转换到“const int”, 可能丢失数据

可以看到它是允许的，但是可能会丢失数据。既然他说可能丢失了数据，那么我们可以看一下 refval3 这个 const 引用的值：

```
//*****
int val2=1024;
const int& refval2=val2; //定义一个变量，然后用一个const引用指向它是允许的。
cout<<"refval2="<<refval2<<endl; //const reference to non const object is permitted
//refval2=100; //Error: 常量不能赋值
//*****
double val3=3.14;
const int& refval3=val3;
cout<<"refval3="<<refval3<<endl;
return 0;
}
```

100 %

输出

显示

C:\Windows\system32\cmd.exe

refval2=1024
refval3=3
请按任意键继续. . .

可以看到 3.14 只输出了 3。而前面提到了引用和被引用的变量的值是共享空间的，而他们的值确是不一样的，这是是否有矛盾呢？实际上是没有矛盾的。这里它等价于两条语句。

```
//*****
double val3=3.14;
const int& refval3=val3; //int temp=val3;
//const int& refval3=temp;
```

这里我们还可以看一下，如果 refval3 不是 const 的引用。


```
cout<<"refval3="<<refval3<<endl;

//*****
int& refval4=val3;
return 0;
}
```

100 %

输出

显示输出来源(S): 生成

```
1>----- 已启动生成: 项目: 04, 配置: Debug Win32 -----
1>生成启动时间为 9/22/2016 11:14:21 AM。
1>InitializeBuildStatus:
1> 正在创建“Debug\04.unsuccessfulbuild”，因为已指定“AlwaysCreate”。
1>ClCompile:
1> 02.cpp
1>d:\cppnote\04\04\02.cpp (17): warning C4244: “初始化”: 从“double”转换到“const int”，可能丢失数据
1>d:\cppnote\04\04\02.cpp (22): error C2440: “初始化”: 无法从“double”转换为“int &”
1>
```

实际上是不允许通过的，提示：“无法从“double”转换为“int &””。也就是说他不会产生一个临时变量来引用的。不会等价于前面两条语句。

这里我们对 const 引用做一些总结：（1）const 引用不能够重新赋值。

【3】引用传递

（1） 引用传递方式在函数定义时在参数前面加上运用运算符“&”。

例如：**swap(int &a,int &b);**

谈到参数传递，我们知道在 C 语言中有两种参数传递方式。

一种是**值传递**：形参不能更改实参。

一种是**指针传递**：形参能够更改实参。

那么我们今天讲到的引用传递也能改变这一点，形参值的改变也能改变实参。

（2） 按值传递方式容易理解，但是形参值的改变不能对实参产生影响。

（3） 地址传递方式（也就是指针传递方式）通过形参的改变使相应的实参改变，但是程序容易出错并且难以读懂。

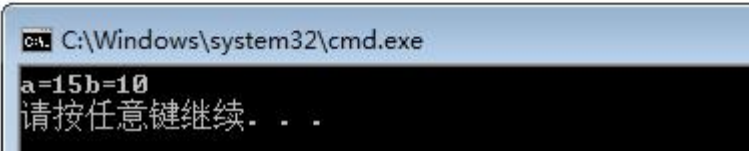
（4） 引用作为参数对形参的任何操作都能改变相应的实参的数据，又使函数调用显得方便和自然。

这里举一个非常简单的例子，两数交换的例子：

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main(){
    int a, b;
    a = 10;
    b = 20;
    swap(a, b);
    cout<<"a="<<a<<"b="<<b<<endl;
    return 0;
}
void swap(int &x, int &y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

上面这个例子我们知道，如果 swap()函数是按照值传递的方式，那么形参不能影响实参 a 和 b 的值，那么最后输出 a,b 的值并没有做交换操作。我们把这个例子先演示一下：

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main(void)
{
    int a, b;
    a=10, b=15;
    swap(a, b);
    cout<<"a="<<a<<"b="<<b<<endl;
    return 0;
}
void swap(int &x, int &y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
```



具体的，我们可以看到 swap(int &x, int &y); 形参 x,y，那么它们在调用的时候初始化，相当于

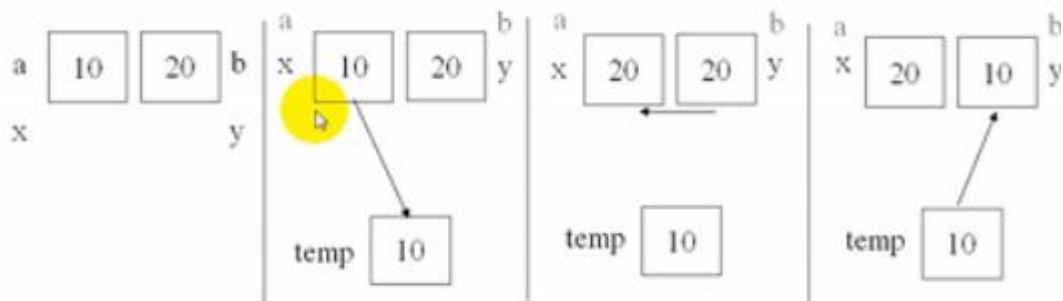
```
int &x=a;    //x 和 a 共享一个地址空间
int &y=b;    //y 和 b 共享一个地址空间
```

也就是说对 x 做的改变就是对 a 做的改变。

对 y 做的改变就是对 b 做的改变。因而引用能够更改实参。

前面我们已经说过，应用通常是在定义的时候就初始化了，现在的情况是在函数引用的时候初始化。引用作为函数参数的时候，引用在函数调用的时候初始化。

□ 程序运行过程中参数值的变化图示



注意：引用作参数时，函数的实参与形参在内存中共用存储单元，因此形参的变化会使实参同时变化。

以上就是引用做为参数传递的例子，另外一个例子就是引用作为返回值，

【4】引用作为函数返回值

- (1) 引用的另一个作用就是用于返回引用的函数。
- (2) 函数返回的引用的一个主要的目的就是函数放在赋值运算符的左边。

注意：不能返回对局部变量的引用。

```
//这个例子说明引用作为函数返回值
#include <iostream>
using namespace std;
int a[]={0,1,2,3,4};
int& index(int i)
{
    return a[i];
}
int main(void)
{
    index(3)=100; //引用作为函数返回值，使得函数可以放在赋值运算符的左边
    cout<<"a[3]="<<a[3]<<endl;
    return 0;
}
```

C:\Windows\system32\cmd.exe

a[3]=100
请按任意键继续. . .

我们发现 a(3)的值被更改为 100 了，这是为什么呢？这是因为 index(3)一个引用，对这个引用重新进行初始化为 100，那么引用的是哪个变量呢？引用的是 a[3]，那么这个应用什么时候初始化呢？在函数返回的时候初始化。


```

//这个例子说明引用作为函数返回值
#include <iostream>
using namespace std;
int a[]={0,1,2,3,4};
int& index(int i)
{
    return a[i];
}
int main(void)
{
    index(3)=100; //引用作为函数返回值，使得函数可以放在赋值运算符的左边
                //函数返回引用，引用在函数返回的时候初始化。
                //也就是说index(3)这个引用在函数返回的时候初始化，
                //也就是说引用index(3)被初始化为a[3]
                //那么对index(3)这个引用所做的改变（赋值100），也就是对a[3]赋值100。

    cout<<"a[3]="<<a[3]<<endl;
    return 0;
}

```

另外还有一个点需要注意：**就是不能返回局部变量的引用**。为什么呢？因为局部变量在返回的时候，已经销毁。也就是说无效的引用，它所指向的变量已经不存在了，所以是无效的引用，而我们现在返回的是全局变量。接下来我们就举一个返回局部变量的例子：

```

//这个例子说明引用作为函数返回值(返回局部变量的例子，错误的例子，无法返回局部变量)
#include <iostream>
using namespace std;
int& add(int a,int b)
{
    int sum;
    sum=a+b;
    return sum;
}
int main(void)
{
    int n=add(3,4);
    cout<<"n="<<n<<endl;
    return 0;
}

```

输出

显示输出来源(S): 生成

1> 生成启动时间: 9/22/2016 2:17:03 PM

1> InitializeBuildStatus:

1> 正在创建 "Debug\04.unsuccessfulbuild", 因为已指定 "AlwaysCreate"。

1> ClCompile:

1> 05.cpp

1> d:\cppnote\04\04\05.cpp(9): warning C4172: 返回局部变量或临时变量的地址

只是有一个警告，返回的值是7，是我们所期望的值。

```

//这个例子说明引用作为函数返回值(返回局部变量的例子，错误的例子，无法返回局部变量)
#include <iostream>
using namespace std;
int& add(int a,int b)
{
    int sum;
    sum=a+b;
    return sum;
}
int main(void)
{
    int n=add(3,4);
    cout<<"n="<<n<<endl;
    return 0;
}

```

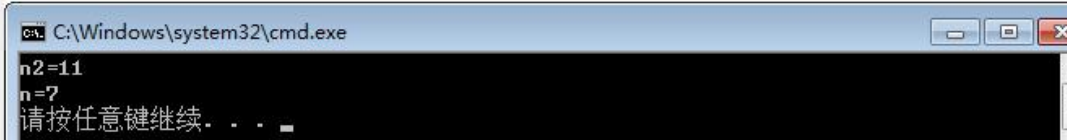
C:\Windows\system32\cmd.exe

n=7

请按任意键继续. . .

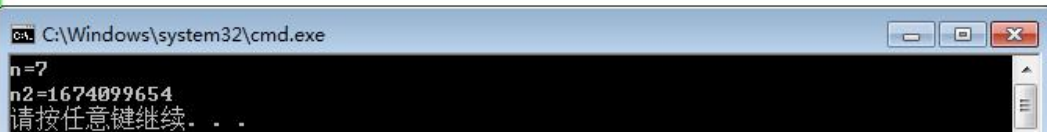
接下来我们在修改一下；

```
//这个例子说明引用作为函数返回值(返回局部变量的例子，错误的例子，无法返回局部局部变量)
#include <iostream>
using namespace std;
int& add(int a,int b)
{
    int sum;
    sum=a+b;
    return sum;
}
int main(void)
{
    int n=add(3,4);
    int& n2=add(5,6);
    //cout<<"n="<<n<<endl;
    cout<<"n2="<<n2<<endl;
    cout<<"n="<<n<<endl;
    return 0;
}
```



我们可以看到得到需要的值 $n2=11$;但是实际上真相被隐藏了，将输出的两行调换一下，在输出，可以得到：

```
//这个例子说明引用作为函数返回值(返回局部变量的例子，错误的例子，无法返回局部局部变量)
#include <iostream>
using namespace std;
int& add(int a,int b)
{
    int sum;
    sum=a+b;
    return sum;
}
int main(void)
{
    int n=add(3,4);
    int& n2=add(5,6);
    cout<<"n="<<n<<endl;
    cout<<"n2="<<n2<<endl;
    return 0;
}
```



我们就发现了一个问题，就是 $n2$ 为一个不确定的数。那么我们来分析一下这个例子，我们可以看到不管这个两个变量的输出如何都不会影响到 n 这个变量的值，但是会影响到 $n2$ 这个引用的值，那么为什么不会影响到 n 这个变量的值呢？原因在于 n 是一个变量，在这个函数返回的时候已经成功的将这个函数的值赋值给了这个变量，所以说这个变量的值是实实在在的，为什么 $n2$ 就不行了呢？因为 $n2$ 它不是一个变量，它没有自己的地址空间，

```

//这个例子说明引用作为函数返回值(返回局部变量的例子，错误的例子，无法返回局部变量)
#include <iostream>
using namespace std;
int& add(int a,int b)
{
    int sum;
    sum=a+b;
    return sum;
}
int main(void)
{
    int n=add(3,4);
    int& n2=add(5,6); //n2是引用，没有自己独立的地址空间
                      //所以n2依赖于它所引用的变量。
                      //如果n2所引用的变量，生命期已经结束了，也就是说n2是一个无效的引用。
                      //那么n2的值将是不确定的。

    cout<<"n="<<n<<endl;
    cout<<"n2="<<n2<<endl;
    return 0;
}

```

那么为什么，我们在先输出 n2 的时候，输出结果是正确的呢？直接输出为什么是正常的呢？那是因为这个局部变量虽然已经销毁了，但是内容还在。里面的内容并没有被其他的变量所覆盖。但是不直接输出，这个局部变量的内容就会被覆盖，所以此时在输出，输出变量的值是不确定的。总而言之呢？**返回一个引用的时候不能返回一个局部变量**。不能返回对局部的引用。这是引用作为返回值的情况。

【5】引用和指针的区别

- (1) 引用访问一个变量是**直接访问**，而指针是**间接访问**(指针里面保存的是变量的地址)。要得到这个变量的地址，才能访问这个变量，所以说是间接访问。
- (2) 引用是变量的别名，不是一个变量，本身不单独分配自己的内存空间，而**指针有自己的内存空间**。(对于 32 位系统来说，指针总是 4 个字节)。指针本身也是一个变量，它有自己的地址空间的。指针变量的空间保存的是这个**指针所指向的变量的地址**。指针本身有自己的地址空间，它本身还是一个变量。
- (3) 引用已经初始化，不能再应用其他的变量，而指针可以。指针初始化之后，还可以指向别的变量。只要它不是一个用 const 修饰的指针。
- (4) C++推荐大家尽量使用引用，不得以才使用指针。实际上最主要的原因在于，

(1) **值传递**：在进行**值传递**的时候，如果我们不用引用的话，实参要初始化形参。这时候就会分配一个形参变量出来，这时候就会涉及到实参拷贝到形参，如果这个对象时是一个类的话，还需要拷贝构造函数，这一点我们在后面会讲解。

(2) **引用传递**：如果是**引用传递**的话，实参初始化形参的时候，不分配空间，因为引用和它引用的变量共享空间，所以说不需要分配空间，也就不需要调用拷贝构造函数，因而他的效率比值传递要高。

(3) **指针传递**：指针传递的本质还是值传递，为什么这个说呢？如果我们要修改指针的地址，这时候单纯用指针传递也是不行的，它仅仅只能修改指针所指向的内容，而不能修改指针的地址，也就是说我们修改指针变量本身的话，是不能够用指针传递的，要用指针的指针 (**).才可以，或者我们用指针引用也是可以的，*&。对指针的引用，也就是说指针传递本身也是一种值传递。实参初始化形参的时候，也需要分配指针变量的地址，对 32 为系统来说是 4 个字节。它也是要分配空间的。如果是 64 位系统就是 8 个字节的空间。