引用：对象的另一个名字。只有在C++中才有引用，C语言中没有。

例：程序quote\_test1

int a = 0;

int& a\_reference = a; // 引用必须初始化

// 引用一旦初始化，只要该引用存在，就保持绑定到初始化时指定的对象，不能将引用维系到另一个对象上。

// 引用是对象的另一个名字，作用在引用上的所有操作都是作用在该引用绑定的对象上

a = 5;

std::cout << a << std::endl; // 5

std::cout << a\_reference << std::endl; // 5

a\_reference = 7;

std::cout << a << std::endl; // 7

std::cout << a\_reference << std::endl; // 7

std::cout << &a << std::endl; // 0x001DF938

std::cout << &a\_reference << std::endl; // 0x001DF938

输出的时候可以看出：对象的地址与引用的地址完全相同。

相对引用而言，指针是一个变量，可以改变其指向的对象，而且指针不一定需要初始化。

const引用：指向const对象的引用，类比指向常量的指针，不同的是引用指向的对象无法改变。

例：程序const\_quote\_test1

const int a = 10;

const int& quote\_a = a; // ok

const int& quote\_d = 15; // ok，可以绑定到右值

// int& quote\_b = a; // error 不能将非const引用绑定的const对象上

// int& quote\_c = 15; // error

int n = 5;

const int& quote\_n = n; // ok，可以绑定到非const对象

n = 6; // 可以改变const引用所绑定的对象

std::cout << "n = " << n << std::endl; // n = 6

std::cout << "quote\_n = " << quote\_n << std::endl; // quote\_n = 6

// quote\_n = 89; // error,不能给常量赋值。这一点和指向常量的指针类似，const int\* p = &a，\*p = b是非法操作

可以看出：const引用可以绑定类型相同的const对象和非const对象，而非const引用只能绑定类型相同的非const对象。

钱能主编的C++程序设计教程关于引用的描述有部分错误（P189），引用的引用是存在的，数组的引用也是存在的：

例：程序quote\_test2

int a = 5;

int& b = a;

int& c = b; // 引用的引用

std::cout << "a = " << a << " b = " << b << " c = " << c << std::endl;

引用的指针不存在：

// int\* &p = &b; // error，引用的指针不存在，b是一个引用

指针变量的引用是存在的，指针也是一个变量

int\* q = &b; // ok

int\* p = &a;

int\*& quote\_p = p; // 指针的引用

std::cout << &a << std::endl;

std::cout << q << std::endl;

std::cout << p << std::endl;

std::cout << quote\_p << std::endl;

输出的地址是完全相同的。

int a = 5;

int\* p = &a;

int\*& quote\_p = p;

std::cout << &a << std::endl; // 0x0025F744

std::cout << p << std::endl; // 0x0025F744

std::cout << quote\_p << std::endl; // 0x0025F744

int d = 10;

quote\_p = &d; // 修改了指针的引用，指针本身也会随之被修改。

std::cout << &a << std::endl; // 0x0025F744

std::cout << p << std::endl; // 0x0025F6FC

std::cout << quote\_p << std::endl; // 0x0025F6FC

\*quote\_p = 15; // 通过指针的引用来修改指针所指向的对象

std::cout << a << std::endl; // 5

std::cout << d << std::endl; // 15

可以看出，quote\_p就是指针p的一个别名。

数组的引用：

int array[10] = {0};

int (&quote\_array)[10] = array; // ok，数组的引用

for (int i = 0; i != 10; ++i)

{

quote\_array[i] = i + 1;

}

for (int j = 0; j != 10; ++j)

{

std::cout << array[j] << std::endl; // 输出为1,2,3,4,5,6,7,8,9,10

}

引用最重要的作用是：函数传递参数，每次调用函数时，都会重新创建该函数所有的形参，此时所传递的实参将会初始化对应的形参。

函数的形参分为两种：非引用形参和引用形参。

1. 非引用形参：

普通的非引用类型的形参通过复制对应的实参实现初始化。当用实参副本初始化形参时，函数并没有访问调用所传递的实参本身，因此不会修改实参的值。形参为指针时，一样是通过实参副本初始化形参，对于指针来说，实参副本与实参都是同一个地址，所有修改实参副本指向的值会修改实参指向的值，但修改实参副本本身，即修改指针，不会改变实参指针。

例：程序quote\_test3

void func1(int\* p);

int main(int argc, char\* argv[])

{

int\* p = NULL;

int a = 3;

p = &a;

std::cout << p << std::endl; // 0x001FF8AC

func1(p);

std::cout << p << std::endl; // 0x001FF8AC

std::cout << a << std::endl; // a = 5

return 0;

}

void func1(int\* p)

{

\*p = 5;

p = NULL;

}

可以看出：指针作为形参时，在func1函数中改变指针本身，在main函数中，指针p并未改变。

1. 引用形参

引用形参直接关联其所绑定的对象，而并非这些对象的副本。每次调用函数时，引用形参被创建并与相应实参关联

void swap( int &vi, int& vj );

int i = 1;

int j = 2;

swap( i, j );

调用函数swap()时，形参vi为i的引用，形参vj为j的引用。

用引用形参时，修改引用等于修改实参本身，但切记，修改引用不能改变引用的绑定关系，引用一旦被初始化，就一直绑定一个对象，不会改变。

例：程序quote\_test3

void func2(int& n);

// void func2(int b); // error，不能这样进行函数重载。

在C语言的学习中，发现在C/C++中，如果数组作为形参，数组形参的长度实际并不起作用，原因是：C/C++在编译过程中，把数组形参看成了指针，这是非引用形参的传递，初始化时，形参会初始化其相应的实参的副本，而实参是指向数组第一个元素的指针，形参复制的是这个指针的值，所以定义的数组形参的长度没有任何作用。

当数组形参声明为数组的引用时，编译器不会将数组实参转换为指针，而是传递数组的引用本身。这时，编译器会检查数组实参的大小与形参是否匹配。

例：程序quote\_test4

void PrintValue(int (&array)[10]);

int main(int argc, char\* argv[])

{

int array1[10];

int array2[5];

for (int i = 0; i != 10; ++i)

{

array1[i] = i + 1;

}

PrintValue(array1);

// PrintValue(array2); // error，长度不匹配

}

void PrintValue(int (&array)[10])

{

for (int i = 0; i != 10; ++i)

{

std::cout << array[i] << std::endl;

}

}

指针的引用：

例：程序quote\_test5

// v1为pi指针的引用，v2为pj指针的引用，修改引用等于修改实参本身。

void PtrSwap( int\*& v1, int\*& v2)

{

int\* tmp = v2; // 中间变量为一个指针

v2 = v1;

v1 = tmp;

}

int main( int argc, char \*argv[] )

{

int i = 10;

int j = 20;

int\* pi = &i;

int\* pj = &j;

std::cout << pi << std::endl; // 0x0025FE68

std::cout << pj << std::endl; // 0x0025FE5C

// pi与pj本身发生了交换

PtrSwap( pi, pj );

std::cout << pi << std::endl; // 0x0025FE5C

std::cout << pj << std::endl; // 0x0025FE68

std::cout << \*pi << std::endl; // 输出为：20

std::cout << \*pj << std::endl; // 输出为：10

return 0;

}

引用作为返回值：

函数的返回值可以分为返回非引用类型和返回引用类型。

1. 返回非引用类型

函数在返回非引用类型时，C++会创建一个临时对象（返回值的副本），然后将返回值赋给临时对象，在函数调用的地方，用临时对象来赋值。在绝大多数情况下，返回非引用类型时，可以返回局部对象，但注意：在返回指针时，不要返回一个局部对象的地址（可以返回一个堆地址）。

1. 返回引用类型

返回引用类型时，不会创建临时对象，所以千万不能返回局部对象的引用。

例：程序quote\_test6

int& func1();

int main(int argc, char\* argv[])

{

int& a = func1();

std::cout << a << std::endl; // 输出a为随机乱数

return 0;

}

int& func1()

{

int a = 5;

return a; // 返回了局部对象给引用

}

在Visual Studio中运行中虽然不会报错，但并不建议这样做，因为在其它编译器中就有可能报错。

float f1(float r)

{

float temp;

temp = r \* r \* 3.14;

return temp;

}

void main()

{

float& a=f1(5.0); // error，原因是函数f1返回temp时，先创建一个临时对象，将temp存入，主函数再从临时对象里取返回值，执行完float& a = f1(5.0)后，临时对象马上析构。引用a的实体就不存在了，所以会出错。

}

函数返回类型为引用时，函数本身可以作为左值。

例：程序quote\_test6

int n = 8;

int& func2();

int main(int argc, char\* argv[])

{

func2() = 9; // 返回类型为引用时，函数可以作为左值

std::cout << n << std::endl; // n为9

return 0;

}

int& func2()

{

return n;

}

函数传递参数时，之所以要使用指针传递和引用传递，很大一部分原因是如果参数的类型非常复杂（比如为类类型）时，采用直接的值传递方式需要复制实参的副本，这是很不可取的，传递引用不需要复制副本，而传递指针只需要复制实参指针的副本。

引用传递和指针传递可能会出现的问题就是：有时程序不允许在函数中修改指针指向的对象或引用绑定的对象，这种情况可以用const关键字进行限定。

void func(const int& a);

void func(const int\* p)

堆中变量的引用：

例：程序quote\_test7

int\* p = new int[3];

int& a = \*p;

std::cout << p << std::endl; // 0x006C8100

std::cout << &a << std::endl; // 0x006C8100

delete [] p; // 等效于delete [] &a;

p = NULL;