复合类型和限定符

引用

1. 引用就是给对象起别名，所以定义一个引用时必须初始化。程序把引用和它的初始值绑定在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它的初始值对象一直绑定在一起。**无法令引用重新绑定到另一个对象**。
2. **定义一个引用后，对其所进行的所有操作都是在与之绑定的对象上进行的**。包括delete操作：对一个引用做delete时，实际上是delete它绑定的变量，而不是将简单的删除了别名。
3. **引用本身并不是一个对象**，所以不能定义引用的引用。
4. 引用本身存放的是引用对象的地址，所以引用所占的内存大小就是指针的大小。

```
int x=5;
int &y=x;
cout<<&x<<' '<<&y;
```

上面代输出了相同的值，但并不代表x和y指向的地址相同。上面代码对应的汇编代码为：

```
int x=5;    // mov dword ptr [bep-4],5
int &y=x;   // lea eax,[ebp-4]  ;  mov dword ptr [ebp-8],eax
```

可以发现引用y有自己的地址[ebp-8]。
所以事实上，我们无法通过&y获取y的地址，因为编译器会将&y解释为：&(*y)=&x。
由此也可知我们使用y时，编译器会将y解释为y=(*y)
这也就是上面所说的，引用是和被引用对象**绑定**在一起的。

5. 不能用非常量的引用绑定一个字面值（1，“ytl”之类）

指针

1. 引用本身不是一个对象，但指针本身就是一个对象。
2. 空指针：C++11新标准引入了nullptr来定义空指针，最好使用nullptr而不是NULL
   - NULL：C语言中的NULL常被定义为 #define NULL ((void*)0)，在把NULL赋值给其他指针时发生了隐式类型转换，把void指针转为相应类型的指针。但在C++中，由于C++是强类型语言，**void*不能隐式转换成其他类型的指针，这与C++的函数重载机制有关**，所以实际上编译器是令NULL=0。所以使用时会有二义性。比如：

```
void func(void* parameter){
    cout<<"func1";
}
void func(int parameter){
    cout<<"func2";
}
```

在执行func(NULL)时，输出的是“func2”，这就是用NULL代替空指针时存在的二义性。

- nullptr：nullptr是为何解决使用NULL时遇到的二义性而提出的

```
//[nullptr的实现](https://segmentfault.com/q/1010000008899545)
```

const限定符

1. 以编译时初始化的方式定义一个const对象时，**编译器将在编译过程中把用到该变量的地方都替换成对应的值**。如定义 const int size=512; 则编译时编译器会将代码中所有用到size的地方都替换成512。**默认情况下，const对象被设定为仅在文件内有效**。要被其他文件共享，就需要添加关键字**extern**。
2. **初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能转换成引用的类型即可**（即允许引用的类型与所引用对象的类型不一致），尤其是允许为一个常量引用绑定非常量对象、字面值。如下代码：

```
double dval=1.24;
const int &ri=dval;
```

为了确保ri绑定到一个整数，编译器把上述代码变成如下形式：

```
const int temp=dval;
const int &ri=temp;
```

因此**ri实际上绑定的是一个临时对象，而不是预期的dval**。所以dval的值改变并不会对ri造成影响，所以可以绑定成功，但没有意义。
但当ri不是常量时，就允许对ri赋值进而改变ri所引用的对象的值。但此时ri绑定的并不是预期的dval，所以C++语法把这种行为视为非法的。

3. 指向常量的指针不能用于改变其所指对象的值；**想要存放常量对象的地址，只能使用指向常量的指针**。
4. 指针的类型必须与其所指对象的类型一致，但有例外：**允许一个指向常量的指针指向一个非常量对象**。（指向常量的指针只是“自以为是”地认为自己指向的对象是常量）。类似常量引用，当常量指针指向一个对象时也允许两者的类型不同。
5. 明确const和*一起定义变量时变量的含义，可以采用从右往左阅读定义的方式：

```
int val=5;
int *const constp=&val;
```

上面代码表示 constp是一个常量————const， 这个常量是一个指针————*，这个指针是一个int型的指针。所以指针是常量，即不变的是指针本身，而不是指针指向的那个值。常用**顶层const**表示指针本身是一个常量，用**底层const**表示指针所指的对象是一个常量。

6. C++11允许将变量声明为constexpr类型以便由编译器来验证变量的值是否是一个常量表达式。常量表达式的值必须在编译时就得到计算。**用constexpr定义一个指针时，限定符constexpr只对指针有效，与指针所指的对象无关**（即指针是常量，其指向的对象不一定是常量）。
7. C和C++中const的差别：

```
const int i=10;
int *p=&i;
*p=5;
```

上面代码在C语言中可以正常执行，且执行完成后i=5。但在C++中编译时会报错，因为**C++中，想要存放常量对象的地址，只能使用指向常量的指针**。
C语言中const表示只读变量，既然是变量那么内存中就会有存储该变量的空间，所以通过指向该变量的指针我们可以间接改变其执行的内存空间的值。
而C++中把const定义的变量看作是常量，在编译时就已经用字面值替换了。但是C++只对内置数据类型做常数替换，对于我们自己定义的类和结构体，编译器不知道如何替换。

类型别名

1. C++11中使用别名声明来定义类型的别名：**using 类型别名=类型名**; 把“=”左侧的名字规定成右侧类似的别名。
2. 别名与限定符连用的时候需要注意：

```
typedef char *pstring;  //定义pstring为char*的别名
const pstring cstr=0;   //cstr是pstring类型，该类型是const
const pstring *ps;      //ps是一个指针，指针的类型是pstring,pstring是const
```

所以上面代码中cstr是一个char类型的常量指针，ps是指向char类型的常量指针的指针。

auto类型说明符

1. C++11引入auto类型说明符，让编译器通过初始值来推算变量的类型，所以auto定义的变量必须有初始值。
2. auto可以在一条语句中声明多个变量，但因为一条声明语句只能有一个基本数据类型，所以该语句中所有变量的初始基本数据类型都必须一样。

3. auto一般会忽略顶层const，保留底层const。如：

```
const int i=1;
auto b=i;   //这里的b是一个整数（i顶层的const被忽略了）
```

若要保留顶层const，只能自己声明：const auto

decltype类型指示符

1. C++11引入decltype用于**选择并返回操作数的数据类型，但不返回具体的值**。如下代码：

```
decltype(f()) x=y;  //x的类型就是函数f返回的类型
const int ci=1,&cj=ci;
decltype(ci) a=0;   //a的类型是const int
decltype(cj) b=a;   //b的类型是const int&
int i=5,*p=&i,&r=i;
decltype(r+0) c;    //r+0的返回值是一个int整数，所以c的类型是int
decltype(*p) d=i;   //d的类型是int& 必须初始化
decltype((i)) e=i;  //e的类型是int&
```

编译器**并不实际调用函数f**，而是使用当调用发生时f的返回值类型作为x的类型。
如果表达式的内容是解引用操作，则decltype将得到引用类型。因为解引用指针可以得到指针所指向的**对象**，而且还能给这个对象赋值。
给变量加上括号，则编译器就会把它当成是一个表达式，变量是一种可以作为赋值语句左值的特殊表达式，所以这样的decltype会得到**引用类型**。

强制类型转换

1. 一个命名的强制类型转换具有如下形式：cast-name<type>(expression)。其中type是转换的目标类型；expression是要转换的值，若type是引用类型则结果是左；cast-name指定执行了哪种转换。
2. static_cast：任何具有明确定义的类型转换，**只要不包含底层const**，都可以用static_cast进行转换。当需要把一个较大的算术类型赋值给较小的类型时，static_cast告诉编译器不用在乎潜在的精度损失。static_cast也常用于对空指针的类型转换。

```
const int a=0;
double b=static_cast<double>(a);    //可以 const并非底层（仅一层）

int i=0;
const int *j=&i;
//不可行 包含底层const 无论是k的类型还是j的类型 都不可以包含底层const
const double *k=static_cast<const double*>(j);
```

3. const_cast：只能改变运算对象的底层const，是**将常量对象转换成非常量对象的行为**。而且const_cast转换的目的类型必须是指针、引用或者指向类成员对象的指针。