**二，变量和基本类型**

1，基本内置类型

可寻址的最小内存块称为字节，存储的基本单元称为字，它通常由几个字节组成。

大多数机器的字节由8比特构成，字则由32或64比特构成，也就是4或8字节。

大多数计算机将内存中的每个字节和一个数字（地址）关联起来。

通常float以一个字（32比特）来表示，double以2个字（64比特）来表示，long double以3或4个字来表示。

除去布尔型和扩展的字符型之外，其他类型可分为带符号和无符号的两种。

unsigned是unsigned int的缩写。

参考文档：建议：如何选择类型

1，当明确知晓数值不可能为负时，选用无符号类型。

2，一般使用int进行运算，如果超过int范围请使用long long类型。

3，执行浮点数运算选用double，long double提供的精度在一般情况下没有必要，而且运行时消耗不容忽视。

如果i为0，则条件的值为false，i为非0值条件为true。

当一个算术表达式中即有无符号又有int值时，那个int值就会转换成无符号数。

当从无符号数中减去一个值时，不管这个值是不是无符号数，我们都必须确保结果不能是负数。

0开头的整数代表八进制，0x开头代表十六进制。

默认的浮点型字面值是一个double。

编译器在每个字符串字面值的结尾处添加空字符('\0')，因此字符串字面值的实际长度要比它的内容多1。

整型字面值后缀ll或LL表示long long，浮点型后缀l表示long double

2，变量

定义于任何函数体之外的内置类型变量被初始化为0。定义于函数体内的内置类型的对象如果没有初始化，则其值未定义。不能进行拷贝和访问。

定义于函数体内的类的对象如果没有显示地初始化，则其值由类确定。

声明使得名字为程序所知，而定义负责创建于名字关联的实体。

如果想声明一个变量而非定义它，就在变量名前添加关键字extern，而且不要显示地初始化变量。

变量能且只能被定义一次，但是可以被多次声明。如果要在多个文件中使用同一个变量，就必须将声明和定义分离。

可以看成“定义 = 声明 + 内存分配”。

3，复合类型

C++语言有几种复合类型，本章介绍其中的两种：引用和指针。

一条声明语句由一个基本数据类型和紧随其后的一个声明符列表组成。

引用为对象起了另外一个名字。

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，因为无法令引用重新绑定到另外一个对象，因此引用必须初始化。

引用并非对象，相反的，它只是为一个已经存在的对象所起的另外一个名字。

因为引用本身不是对象，所以不能定义引用的引用。

指针与引用相比有很多不同点：

其一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。

其二，指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

因为引用不是对象，没有实际地址，所以不能定义指向引用的指针。

指定空指针的方法：int \*p1 = nullptr; =0; = NULL;

预处理变量不属于命名空间std，它由预处理器负责管理，因此我们可以直接使用预处理变量而无须在前面加上std::，NULL是一个预处理变量。

任何非0指针对应的条件值都是true。

void\*是一种特殊的指针类型，可用于存放任意对象的地址。

利用void\*指针能做的事儿比较有限：拿它和别的指针比较，作为函数的输入输出，或者赋给另外一个void\*指针。不能直接操作\*void指针所指的对象。

指向指针的指针 int \*\*ppi = &pi;

引用本身不是一个对象，因此不能定义指向引用的指针。但指针是对象，所以存在对指针的引用。

指向指针的引用 int \*p; int \*&r = p;

面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

4，const限定符

const对象一旦创建后其值就不能再改变，所以const对象必须初始化。

默认情况下，const对象被设定为仅在文件内有效。当多个文件中出现了同名的const变量时，其实等同于在不同文件中分别定义了独立的变量。

如果想在多个文件之间共享const对象，必须在变量的定义之前添加extern关键字。

可以把引用绑定到const对象上，就像绑定到其他对象上一样，我们称之为对常量的引用。对常量的引用不能被用作修改它所绑定的对象。

对const的引用可能引用一个并非const的对象。

类似于常量引用，指向常量的指针不能用于改变其所指对象的值。要想存放常量对象的地址，只能使用指向常量的指针。

所谓指向常量的指针和引用，不过是指针或引用“自以为是”，觉得自己指向了常量，所以自觉地不去改变所指对象的值。

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本身定为常量。常量指针必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能改变了。

常量指针的定义：int \*const curErr = &errNumb;

指向常量的指针：const int\* curErr = &errNumb;

指向常量对象的常量指针:const int \*const curErr = &errNumb;

指针本身是不是常量以及指针所指的是不是常量就是两个独立的问题。顶层const表示指针本身是一个常量，底层const表示指针所指对象是一个常量。

当执行对象拷贝的操作时，常量是顶层const还是底层const区别明显，其中顶层const不受什么影响，底层的const的限制却不能忽视。

常量表达式是指不会改变并且在编译过程就能得到计算结果的表达式。const int number = 1;//这是一个常量表达式

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。

声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化。

一般来说，如果你认定变量是一个常量表达式，那就把它声明成constexpr类型。

允许函数定义一类有效范围超出函数本身的变量，这类变量和定义在函数体之外的变量一样也有固定地址，因此，constexpr引用能绑定到这样的变量上，constexpr指针也能指向这样的变量。

在constexpr生命中如果定义一个指针，限定符constexpr仅对指针有效，与指针对象无关。把它所定义的对象置为顶层const。

5 处理类型

有两种方法用于定义类型别名，传统的方法使用关键字typedef，新方法是使用using。

遇到一条使用了类型别名的声明语句时，人们往往会错误地尝试把类型别名替换成它本来的样子，以理解该语句的含义。

auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。

编译器推断出来的auto类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

1，编译器会以引用对象的类型作为auto的类型；

2，auto一般会忽略掉顶层const，同时底层const则会保留下来。

3，如果要定义引用和顶层const，需要手动指明。

decltype的作用是选择并返回操作数的数据类型。在此过程中，并不实际计算表达式的值。

decltype处理顶层const和引用方式与auto有些许不同。

如果decltype使用的是加了一层或多层括号的变量，编译器就会把它当成是一个表达式，得到的将是引用类型。

decltype((variable))的结果永远是引用，而decltype(variable)结果只有当variable本身就是一个引用时才是引用。

6，自定义数据结果

类体右侧的表示结束的花括号后必须写一个分号。

头文件通常包含那些只能被定义一次的实体，如类，const和constexpr变量等。

确保头文件多次包含仍能安全工作的常用技术是预处理器。

预处理命令：#define #ifdef #ifndef #endif

#ifndef SALES\_DATA\_H

#define SALES\_DATA\_H

#include <string>

struct Sales\_data{};

#endif

头文件即使没有被包含在任何其他头文件中，也应该设置保护符。头文件保护符很简单，程序员只要习惯性加上就可以了。

使用#pragma once可以获得相同的效果。但是，这种方式不支持跨平台！