Programming Language C++

- 初阶
  - 第一部分：C++基础
    - 变量和基本类型
      - **基本内置类型**
        - ## 空类型 Void
          不对应具体值，仅用于特殊场合，例：当函数不返回任何值时作为返回类型
        - ## 算数类型 ArithmeticType
          尺寸：即该类型数据所占比特数 在不同机器上有所差别 所能表示的数据范围亦不同
          C++标准规定了该尺寸的最小值-P33 ， 同时允许编译器扩充
        - ## 类型转换 Convert
          数据类型包含：1.对象能包含的数据 2.能参与的运算 （猜测其功能底层应该由汇编语言编写，C++和开发环境一直都运行在汇编的抽象层级上）
          大多数数据类型都包含一种运算：转换 convert 类型转换往往是计算机自动完成的
          转换的特点是转换输入的数据，使之符合将被输入的数据类型
          - **Bool与非Bool**
            非Bool对Bool时赋值：初始为0则结果为false,否则结果为true
            Bool对非Bool时赋值：初始值为false则结果为0，初始值为true则结果为1
          - **Float与int**
            Float对int赋值：近似处理，仅保留整数部分
            int对Float赋值：小数部分记为0
          - **Unsigned和signed**
            - 取模
              取模运算（“Modulo Operation”）和取余运算（“Complementation”）
              两个概念有重叠的部分但又不完全一致。主要的区别在于对负整数进行除法运算时操作不同。取模主要是用于计算机术语中。取余则更多是数学概念。
              通常取模运算也叫取余运算，它们返回结果都是余数.
              取模（mod）与取余（rem）的唯一的区别在于:
              当x和y的正负号一样的时候，两个函数结果是等同的；当x和y的符号不同时，rem函数结果的符号和x的一样，而mod和y一样。
            - 赋给无符号类型一个超出它表示范围的值
              结果是初始值对该无符号数据类型 可表示数值总数 取模后的余数
              例：8Bit 的 unsigned char 可以表示0到255区间的值 共计256个
              当我们把-1赋给8bit的 unsigned char时 需要将-1对256取模所得余数作为结果 即255
              从数学角度解读:
              取模运算时,对于负数,应该加上被除数的整数倍,使结果大于或等于0之后,再进行运算.
              也就是：(-1)%256 = (-1+256)%256=255%256=255
              -42在无符号数的框架下得出结果为4294967264

当从无符号数中减去一个值时，结果等于这个无符号数加上无符号数的模。

- 赋给有符号类型一个超出它表示范围的值

结果是未定义的
程序可能工作、崩溃或生成垃圾数据

- **注意**

一、
程序应该避免未定义的行为
程序应该尽量避免依赖于实现环境的行为
例：如果我们把Int的尺寸看成是一个确定不变的已知值 那么这样的程序就是 不可移植的(nonportable)。
二、
1.另一种类型的值将被自动转换为程序该处所需的值
例：if（）条件值必为Bool类型
算数表达式里的Bool类型数据必为0或1
2.由于无符号类型数据被赋负值时会出现取模运算，往往会得到非预计值
我们不会直接赋值，但我们很容易写出这样的表达式
例：当一个算术表达式中既有无符号又有int值时，int值就会被转换为无符号数
另一个有关循环的数据转换的例子P34
3.切勿混用带符号类型和无符号类型
当有符号数值为负值时会出现异常结果，因为有符号数会被转换为无符号数。
例：int a=-1, unsigned b= 1　当a*b时，a会被转换成4294967295（这是2的32次方-1）（结果需要视机器上int所占位数决定)

字面值常量 Literal

该常量的容器形状即为数值本身的值 被称作字面值常量 literal
每个字面值常量的形式和值直接决定了它的数据类型

- **变量**

变量Variable

变量提供一个具名的、可供程序操作的储存空间。
每个变量都有其数据类型，决定着变量所占内存空间大小和布局方式、储存值的范围，变量能参与的运算。
变量Variable和对象Object在C++可以互换

- **定义**

类型说明符 type specifier + 变量名列表（以逗号分隔，以分号结束）
列表中每个变量名的类型都由类型说明符决定
定义时可同时为一个或多个变量赋初值

- **对象 Object**

#对象 在C++很多场合下被使用，对象是指一块能储存数据并具有某种类型的内存空间。
对象的定义在很多人口中有区别：
1.与类相关的场景下使用对象
2.把命名的对象叫做变量，与未命名的对象区别开来
3.对象指能被程序修改的数据，而值指只读的数据
C++Primer中的对象指 某种具有 数据类型的 内存空间
且并不区分是 类 还是 内置类型，也不区分 是否命名或是否只读

- **初始化 Initialized**

当对象在创建时获得了一个特定的值，称其被初始化了。
用于初始化变量的值可以是任意复杂的表达式。
定义后的变量的名字就可以使用了，因此在同一条定义语句中，可以用先定义的变量值去初始化后定义的其他变量。

例：double price = 109.99, discount = price*0.16;
price 先被定义并赋值，随后被用于初始化。
初始化是个很复杂的问题，初始化和赋值是两个完全不同的操作。但在很多编程语言中两者的区别几乎可以忽略不计。
初始化不是赋值，初始化是创建变量时赋予其一个初始值，而赋值的含义是把当前的对象的值擦除，而以一个新值来代替。

- **列表初始化 List initialization**

  初始化具有多种形式，这也是初始化问题复杂性的一个体现
  用花括号来初始变量的初始化形式被称为列表初始化
  这种形式可以初始化对象或者为对象赋新值
  例：int units_sold{0}; int units_sold = {0};
  当用于内置类型的变量时，这种初始化形式有一个重要的特点：如果使用列表初始化，且初始值存在丢失信息的风险，则编译器将报错。
  long double ld = 3.141592653536
  int a{ld}, b = {ld}; 错误：转换未执行，因为存在丢失信息的危险
  int c{ld}, d = ld; 正确：转换执行，且确实丢失了部分值
  这种初始化有可能在不经意间发生。

- **默认初始化 Default initialized**

  如果定义变量时没有指定初值，则变量被默认初始化，此时变量被赋予了“默认值”。
  默认值到底是什么由变量类型决定，同时定义变量的位置也会对此有影响。
  定义于任何函数体之外的变量被初始化为0.
  若变量类型为内置类型且未被显式初始化，它的值由定义的位置决定。
  一种例外情况：定义在函数体内部的内置类型变量直到该函数执行前都将不被初始化（uninitialized）
  一个未被初始化的内置类型变量的值是未定义的，如果试图拷贝或以其他形式访问此类值将引发错误。

- **初始化故障**

  为初始化的变量都含有一个不确定的值，使用未初始化变量的值是一种错误的编程行为并且很难调试。
  尽管大多数编译器都能对一部分使用未初始化变量的行为提出警告，但严格来说编译器并未被要求检查此类错误。
  使用未初始化的变量将带来无法预计的后果。其错误往往是隐藏的，但程序必定因此仍然有错。

变量声明 Declaration 和定义 Definition 的关系

- **变量声明和定义的区别**

  为允许程序分为多个逻辑部分来编写，C++语言支持分离式编译（separate compilation）机制。该机制允许将程序分割为若干个文件，每个文件可被独立编译。
  如果将程序分为多个文件，则需要有在文件间共享代码的方法。
  例如，一个文件的代码可能需要使用另一个文件中定义的变量。例：std::cout 和 std::cin,它们定义于标准库，却能被我们写的程序所用。
  为了支持分离式编译，C++创造出了声明和定义两种不同的东西。
  #声明（declaration）使得名字为程序所知，一个文件如果想使用别处定义的名字，则必须包含对那个名字的声明。
  #定义（definition）负责创建与名字关联的实体。
  变量声明声明了变量的名字和类型。而变量定义还申请储存空间，也会为变量赋一个初始值。

- **声明和定义的使用**

  如果想声明一个变量的类型而非定义它，就在变量名前添加关键字 extern
  而不要显示地初始化变量：extern int i;

任何包含了显式初始化的声明即成为定义。给由extern关键字标记的变量赋一个初始值就抵消了extern的作用。
在函数体内部如果试图初始化一个由extern关键字标记的变量，将引发错误。

变量只能被定义一次，但是可以被多次声明。
如果要在多个文件中使用一个变量，就必须声明和定义分离
此时变量的定义必须出现在且只能出现在一个文件中 而其他用到该变量的文件必须对其进行声明，却绝对不能重复定义。

- **静态类型 Statically typed**

C++是一种静态类型语言，即：在编译阶段检查类型。检查类型的过程称为类型检查（type checking）
编译器负责检查数据类型是否支持要执行的运算，若不支持将会报错且不会生成可执行文件。
类型检查的前提是编译器必须知道每一个实体对象的类型，这样才能检测数据类型是否支持要执行的运算，这就要求我们在使用某个变量之前必须声明其类型。

- **总结**

①变量定义：用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。
②变量声明：用于向程序表明变量的类型和名字。
③定义也是声明：当定义变量时我们声明了它的类型和名字。
④extern关键字：通过使用extern关键字声明变量名而不定义它。
1.定义也是声明，extern声明不是定义，即不分配存储空间。extern告诉编译器变量在其他地方定义了。
例如：extern int i; //声明，不是定义
　　int i; //声明，也是定义
2.如果声明有初始化式，就被当作定义，即使前面加了extern。只有当extern声明位于函数外部时，才可以被初始化。
例如：extern double pi=3.1416; //定义
3.函数的声明和定义区别比较简单，带有{ }的就是定义，否则就是声明。
例如：extern double max(double d1,double d2); //声明
4.除非有extern关键字，否则都是变量的定义。
例如：extern int i; //声明
　　int i; //定义
程序设计风格：
　1. 不要把变量定义放入.h文件，这样容易导致重复定义错误。
　2. 尽量使用static关键字把变量定义限制于该源文件作用域，除非变量被设计成全局的。
　3. 可以在头文件中声明一个变量，在用的时候包含这个头文件就声明了这个变量。
总结：
变量在使用前就要被定义或者声明。
在一个程序中，变量只能定义一次，却可以声明多次。
定义分配存储空间，而声明不会。
C++程序通常由许多文件组成，为了让多个文件访问相同的变量，
C++区分了声明和定义。
　　变量的定义（definition）用于为变量分配存储空间，还可以为变量指定初始值。在程序中，变量有且仅有一个定义。
　　声明（declaration）用于向程序表明变量的类型和名字。定义也是声明：当定义变量的时候我们声明了它的类型和名字。可以通过使用extern声明变量名而不定义它。不定义变量的声明包括对象名、对象类型和对象类型前的关键字extern。
　　extern声明不是定义，也不分配存储空间。事实上它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次。
　　只有当声明也是定义时，声明才可以有初始化式，因为只有定义才

分配存储空间。初始化式必须要有存储空间来进行初始化。如果声明有初始化式，那么它可被当作是定义，即使声明标记为extern。
任何在多文件中使用的变量都需要有与定义分离的声明。在这种情况下，一个文件含有变量的定义，使用该变量的其他文件则包含该变量的声明（而不是定义）。

标识符 Identifier

- **命名规则**

  C++的标识符由字母、数字和下划线组成，其中必须以字母或下划线开头。
  自定义标识符不能连续出现两个下画线，也不能以下画线紧连大写字母开头。
  定义在函数体外的标识符不能以下画线开头。
  标识符的长度没有限制。但是对大小写敏感。
  C++保留了关键字、标准库和操作符代替名 供语言本身使用，这些名字不能被用作标识符。表P43

- **命名规范**

  变量命名有许多约定俗成的规范，下面的这些规范能有效的提高程序的可读性
  1.体现实际含义
  2.一般用小写字母，如index,不要使用Index 或 INDEX
  3.用户自定义的类名一般以大写字母开头，如Sales_item.
  4.如果标识符由多个单词组成，则单词间应有明显区分，如student_loan.
  *对于命名规范来说，若能坚持，必将有效。

名字的作用域 Scope

- **作用域**

  名字一定会指向一个特定实体
  如果同一个名字出现在程序的不同位置，也有可能指向不同实体
  作用域是程序的一部分，C++语言中大部分作用域都用花括号分隔。
  同一个名字在不同的作用域中可能指向不同的实体。
  名字的有效区域始于名字的声明语句，以声明语句所在的作用域末端为结束。
  例：名字main定义于所有花括号之外，它和大多数定义在函数体之外的名字一样拥有全局作用域（global scope）在整个程序的范围内可使用
  定义在函数体内部的变量拥有块作用域（block scope）
  定义在for语句内的名字在for语句中可以使用，但在函数体的其他部分则不行。
  #建议：在第一次使用变量时再定义它。

- **嵌套的作用域**

  作用域可以相互包含，外层作用域（outer scope）内层作用域（inner scope）
  作用域中一旦声明了某个名字，它所嵌套着的所有作用域中都能访问该名字
  同时允许在内层作用域中重新定义和覆盖外层作用域已有的名字和作用域规则 例P49
  还可以使用：：（作用域操作符 左侧为空）来向全局作用域发出请求获取操作符右侧名字对应的变量。

- **复合类型**

复合类型 Compound Type

一条声明语句由一个基本数据类型 (base type) 和紧随其后的一个声明符（declarator）列表组成
声明符命名一个变量并指定该变量为与基本数据类型有关的某种类型
简单的声明语句中 声明符就是变量名

更复杂的基于基本数据类型来指定变量

引用 Reference

- **定义**

  引用是一种基于基本数据类型的复合类型 来指定变量
  引用类型 引用另外的一种类型（这就是它本身的类型属性）
  将声明符写成 &+变量名 的形式（与绑定对象相符的数据类型+&+变量名）来定义引用类型 其中变量名所指对象必须是被初始化过的

- **性质**

  引用即别名。
  定义引用时程序把引用和其初始值绑定（bind）而并不拷贝，因此引用必须被初始化
  引用始终只能绑定一个对象无法重新绑定其他对象
  引用并非对象，而是已存在对象的别名

  所有对引用的操作都是对其绑定对象进行的
  赋值、获取、以引用作为初始值等等

- **注意**

  引用本身不是对象，所以不能定义引用的引用
  允许在一条语句中定义多个引用 例：int &r3 = i3, &r4 = i2;
  引用的类型必须与绑定对象严格一致 有两种例外：

  引用只能绑定在对象上，而不能与常量 即不能与字面值或某个表达式的计算结果绑定在一起

指针 Pointer

与引用类似，指针也实现了对其他对象的间接访问
与引用的不同点：
1.指针本身就是对象，就是一块 有数据类型的 内存空间。允许对指针赋值和拷贝。
2.指针可以在其生命周期之内先后指向几个不同的对象。
3.指针正如对象，无需在定义时赋初值，类似内置类型，在块作用域内定义的未被初始化的指针将拥有一个不确定的值。

定义方法：数据类型 + * 变量名
可以在一条语句中定义多个指针 形式与引用类似。
声明形式：只需写出指针名

- **获取对象的地址**

  指针存放有某个对象的地址，获取该地址需要取地址符 &（操作符）
  注意：数据类型+&是引用，单独一个&是取地址符
  &+变量名 即为该变量所在地址
  int *p = &ival；即把p定义为一个指向int（int是int类型的数据）的指针，随后将ival的地址存放在指针p里，p被初始化。
  因为引用不是对象，所以没有实际地址，不能定义指向引用的指针。

  指针类型需要与其所指对象严格匹配 （虽然指针的主要存放功能是存放地址，但指针的底层是由汇编或C编写的复杂结构体）
  有两种情况除外：

- **指针值**

  指针的值（即地址）应属于下列4种状态之一
  1.指向一个对象

2.指向紧邻对象的所占空间的下一个位置
3.空指针，意味指针没有指向任何对象
4.无效指针，上述情况之外的其他值

程序员必须清楚指针是否有效
因为类似拷贝和访问未初始化变量一样
访问和拷贝无效指针引发的后果无法预计，且编译器并不负责检查此类错误
尽管第2种和第3种指针是有效的 但因为其并未指向任何具体对象 所以访问这些指针的后果也无法预计

- **利用指针访问对象**

  如果指针指向了一个对象
  则可用 解引用符 （操作符* ）来访问该对象：
  int ival = 42;
  int *p = &ival;
  cout << *p;

  注意：
  对指针 解引用 会得出所指的对象，因此如果给解引用的结果赋值，实际上也就是给指针所指的对象赋值：
  *p= 0;
  经由指针p给变量ival赋值
  解引用只适用于那些确实指向了某个对象的有效指针

- **符号的多重含义**

  符号的上下文和符号共同组成其含义
  像&和*既能作 表达式里的运算符 也能作声明的一部分
  &紧随类型名一起出现 则这是一个引用
  *紧随类型名出现 则这是一个指针
  &出现在表达式中 是一个取地址符
  *出现在表达式中 是一个解引用符
  int &r2 = *p;
  上例：&是声明的一部分，*是一个解引用符
  在声明语句中，&和*用于组成复合类型，在表达式中它们是运算符

- **空指针 Null Pointer**

  不指向任何对象
  生成空指针的方法：
  int *p = nullptr; //等价于 int *p1 = 0；C++新标准刚引入的方法：nullptr是一种特殊类型的字面值，它可以转化为指针类型。
  int *p2 = 0; // 直接将p2初始化为字面值常量0
  int *p3 = NULL; // 等价于 int *p3 = 0；需要首先 # include cstdlib 头文件cstdlib中定义，它的值就是0。

  建议：初始化所有指针 访问未经初始化的指针造成的错误难以定位
  大多数编译器环境下 如果使用了未初始化的指针 指针所占内存空间的当前内容将被看为一个地址值 难以区分合法性
  建议等定义了对象之后再定义指向它的指针
  如果实在不知道指针指向 就把它初始化nullptr或者0

- **其他指针操作**

  指针运用在条件表达式中 指针存放值只要不为0 则条件值为ture 指针存放值为0 则条件表达式为false
  对比两个相同类型的合法指针 可以用 （==）相等操作符 或不相等操作符 （！=）来比较 比较的结果是布尔类型的值

若两个指针存放的地址值相同 有三种可能：
1.指针都为空
2.指向同一个对象
3.指向同一个对象的下一个地址

注意：
一个指针指向对象的下一个地址 一个指针指向该对象 此时也有可能出现两个指针值相同的情况
无论是作为条件出现还是参与比较运算 都必须使用合法指针 使用非法指针作为条件或进行比较 都会引发不可预计的后果

- **Void*指针**

  可用于存放无需或未确定类型对象的地址
  可用于存放任意类型的指针
  由于所指对象类型未知 不能直接操作void*指针所指的对象 也就无法确定能对这个对象所哪些操作
  利用void*能做的事情：
  1.与其他指针比较
  2.作为函数的输入输出
  3.赋值给另外一个对象
  以void*视角所看的内存空间仅仅就是内存空间 无法访问对象

理解复合类型的声明

类型修饰符只是声明的一部分
数据类型是一块内存中内容物格式

- **定义多个变量**

  类型修饰符仅仅作用于紧随变量
  int　*p 强调变量具有复合类型
  int*　p 强调本次定义了一个复合类型的变量

- **指向指针的指针**

  声明符中 修饰符的个数无限制 可连写
  **表示指向指针的指针
  以指针为例 指针是内存中的对象 其本身也有地址 因此允许把指针的地址再次存放在别的指针中
  但每次再次存放 *加一
  则若要访问原对象需要两次解引用

- **指向指针的引用**

  引用不是对象 不能定义指向引用的指针
  指针是对象 可以定义对指针的引用
  例：
  int i = 42; 定义了int类型的变量
  int *p; 定义了int型指针
  int *&r = p; 定义了对指针p的引用r
  r = &i; 把引用r的值修改为变量i的地址，而r是对指针p的引用，即把指针p的值修改为变量i的地址，现在指针p指向变量i
  *r = 0; 解引用r，由于r只是别名，对r的解引用就是对指针p的解引用，就是访问指针的所指对象 变量i，同时把i的值赋值为0。

- **const限定符**

  是限定程序无法修改变量值，在定义const之后人也不能轻易修改
  所以一定要在定义时初始化 可以是任意复杂的表达式
  const只能执行不改变其内容的操作
  const对象仅在文件内有效

extern 前缀加 const 后可以被其他文件访问

- ## const的引用

  术语：常量引用就是指对const的引用
  虽然常量引用对其可参与操作做出限定 但是未对对象本身是不是一个常量做出限定
  正常的const形式： const int i = 1;
  引用非常量的const: const int &r1 = i;
  不允许通过r2修改i的值 ， i的值允许通过其他途径修改

- ## 指针和const

  pointer to const 指向常量的指针
  不能改变所指对象值 只存放常量对象的地址
  指针类型一致性的例外：
  所谓指向常量的指针仅仅要求不能通过该指针改变对象的值 而没有规定那个对象的值不能通过其他途径改变

  const pointer 常量指针
  指针本身是对象，可以常量化，但必须初始化
  *const 是指不变的是指针本身的值 而非指向的值
  常量指针本身不能变 但是常量指针所指对象（若是非常量的话）可以通过指针改变

- ## 顶层const

  指针本身是不是常量和指针所指是不是常量就是两个相互独立的问题
  因此用 顶层const (top-level const)表示指针本身是常量
  底层const (low-level const)表示指针所指对象是常量

- ## constexpr和常量表达式

- **处理类型**
  - ## 类型别名
  - ## auto类型说明符
  - ## decltype类型指示符
- **自定义数据结构**
  - ## 定义Sales_data类型
  - ## 使用Sales_data类
  - ## 编写自己的头文件

- # 字符串 向量和数组
  - **命名空间的using声明**
  - **标准库类型string**
  - **标准库类型vector**
  - **迭代器介绍**
  - **数组**
  - **多维数组**
- # 表达式

- 进阶
  - 第三部分：类设计者的工具
  - 第四部分：高级主题

幕布 - 思维概要整理工具