C++primer学习

第2章 变量和基本类型

2.1 基本内置类型

1.c++定义了一套包括**算数类型**和**空类型**在内的基本数据类型

算数类型：整型、浮点型

表 2.1：C++：算术类型

类型	含义	最小尺寸
bool	布尔类型	未定义
char	字符	8 位
wchar_t	宽字符	16 位
char16_t	Unicode 字符	16 位
char32_t	Unicode 字符	32 位
short	短整型	16 位
int	整型	16 位
long	长整型	32 位
long long	长整型	64 位
float	单精度浮点数	6 位有效数字
double	双精度浮点数	10 位有效数字
long double	扩展精度浮点数	10 位有效数字

2.除去布尔型和扩展的字符型外，其他整型可以划分为**带符号的(signed)**和**无符号的(unsigned)**，带符号的可以表示正数、负数和0，无符号类型仅能表示大于等于0的值。

3.**类型转换**注意的几点：

- 把浮点数赋给整数类型时，小数点后的部分会舍去。
- 把整数类型赋给浮点类型时，小数部分会记为0。
- 赋给无符号类型一个超过它表示范围的值时，结果是初始值对无符号类型表示总数取模后的余数。
- 当我们赋给带符号类型一个超出他表示范围的值时，结果是未定义的。此时程序可能继续工作、可能崩溃、也可能产生垃圾数据。

4.含有无符号类型的表达式（一般不要混用带符号类型和无符号类型）

- 当一个算术表达式中既有无符号数又有整数类型时，会将整数类型转化为无符号数进行计算。

- 当

字符和字符串字面值

前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8（仅用于字符串字面常量）	char

整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f或F	float
l or L	long	l或L	long double
ll or LL	long long		

从无符号数中减去一个值时，不管这个值是不是无符号数，我们都要确保结果不能是一个负值。

5.字面值常量

- 整型和浮点型字面值

整型：20 /*十进制* 024 /*八进制* 0x14 /*十六进制* 浮点型：表现为一个小鼠或以科学计数法表示的指数，其中指数部分用E或e标识：3.14159 3.14159E0 .001 0.

- 字符和字符串字面值

由单引号括起来的一个字符成为char型字面值，双引号括起来的零个或多个字符则构成字符串型字面值：'a' /*字符字面值* "Hello world!" /*字符串字面值*

- 转义序列

c++中有两类字符不能直接使用：一类是**不可打印的字符**；另一类是在c++语言中有**特殊函数的字符。**在这些情况下要用到转义字符(escape sequence)，转义序列均已反斜线开始，包括：

换行符	\n	横向制表符	\t	报警（响铃）符	\a
纵向制表符	\v	退格符	\b	双引号	\"
反斜线	\\	问号	\?	单引号	\'
回车符	\r	进纸符	\f		

- 指定字面值的类型

通过添加下表中的前缀和后缀，可以改变整型、浮点型和字符型字面值的默认类型。

字符和字符串字面值

前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8（仅用于字符串字面常量）	char

整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f或F	float
l or L	long	l或L	long double
ll or LL	long long		

- 布尔字面值和指针字面值

true和false是布尔类型的字面值；nullptr是指针字面值。

2.2 变量

1.变量的基本形式：首先是类型说明符，随后紧跟着由一个或多个变量名组成的列表，其中变量名以逗号分隔，最后以分号结束

```
int sum = 0, value, units_sold = 0;
```

2.初始化：当对象在创建时获得了一个特定的值，我们就说这个对象被初始化了。

ps：初始化不是赋值，初始化的含义是创建变量时赋予变量一个初始值，而赋值的含义是把对象的当前值擦除，而已一个新的值来替代。

3.默认初始化：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”，由变量类型决定。

ps：一种例外情况，定义在函数体内的内置类型变量不被初始化，其值未定义。类的对象如果没有显示的初始化，则其值由类确定。

4.c++语言支持分离式编译，将声明和定义区分开来。

声明：是的名字为程序所知，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：负责创建与名字关联的实体。

```
extern int i;  //声明i而非定义i
int j;  //声明并定义了j
extern int k = 0;  //赋了一个初始值，抵消了extern关键字，声明并定义了k
```

ps：变量能且只能被定义一次，但是可以被声明很多次。

5.标识符：c++的标识符由字母、数字和下划线组成，其中必须以字母或者下划线开头；长度没有限制，但对大小写敏感。

变量命名规范

- 变量名一般用小写字母，如index
- 用户自定义的类名一般以大写字母开头，如Sales_item
- 如果标识符由多个单词组成，则单词间应有明显区分，日student_loan或studentLoan

6.作用域：大多数由大括号{}分离 作用域能嵌套着彼此，被包含的被称为内层作用域，包含着别的作用域的被称为外层作用域。

2.3 复合类型

1.**引用（目前指左值引用）**

引用为对象起了另一个名字，引用类型引用另外一种类型。通常将声明符携程&d来定义引用类型，其中d是声明的变量名。 **（引用必须初始化） ps：引用并非对象，只是对一个已存在的对象所起的另一个名字。**

2.**指针**

指针是“指向”另外一种类型的复合类型，也实现了对其他对象的间接访问。通常用*d，其中d是变量名。

- 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他可以先后只想几个不同的对象。
- 指针无须在定义时赋初值。
- 指针存放某个对象的地址，若想获取该地址，需要使用**取地址符 &**
- 如果指针指向了对象，则允许使用解引用符（操作符*）来访问该对象

3.指针值（既地址）应属下列4种状态之一：

- 指向一个对象
- 只想紧邻对象所占空间的下一个位置
- 空指针，意味着指针没有指向任何对象
- 无效指针，上述情况以外的其他值

4.void*指针：是一种特殊的指针类型，可以存放任意对象的地址

5.指向指针的指针：** ，指向指针的指针的指针：***

```
int ival = 1024;
int *pt = &ival;
int **ppi = &pi;
```

6.指向指针的引用：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，所以存在指向指针的引用：

```
int i = 42; int *p;  //p是一个int指针
int *&r = p;  //r是一个对指针p的引用
r = &i;  //r引用了i
*r = 0;  //解引用r得到i，也就是p指向的对象，将i变为0
```

2.4 const限定符

1.定义一种变量，它的值不能改变，可以用const加以修饰

```
const int bufSize = 512;
```

2.默认状态下，const对象仅在文件内有效，加上extern关键字就可以定义一次就可以在文件间共享const变量。

```
extern const int bufSize = 512;
```

3.const引用：把引用绑定在const对象上，不能修改他所绑定的对象

ps：允许一个const引用绑定到非const对象上

4.指针和const：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：

```
const double pi = 3.14;
double *ptr = &pi;  //错误：ptr是一个普通指针
const double *cptr = &pi;  //正确：cptr是一个常量指针
```

ps：和常量引用一样，指向常量的指针也没规定其所指的对象必须是一个常量。只是要求不能通过该指针改变对象的值，而没规定那个对象的值不能通过其他途径进行改变。

5.**const指针**：把*放在const关键字之前以说明指针是一个常量，意味着不变的是指针本身的值而不是指向的那个值

```
int errBumb = 0;
int *const curErr = &errBumb;   //curErr是一个常量指针，可以修改常量指针的指向的值
const double pi = 3.14159;
const double *const pip = &pi;  //pip是一个指向常量对象的常量指针，都不可改变
```

6.**顶层const**：用名词**顶层const**表示指针本身是个常量，而用**底层const**表示指针所指的对象是一个常量。

ps：比较特殊的是，指针类型既可以是顶层const也可以是底层const

```
int i = 0;
int* const p1 = &i;  //不能改变p1的值，这是一个顶层const
const in ci = 42;    //不能改变ci的值，这是一个顶层const
const int *p2 = &ci; //能改变p2的值，这是一个底层const
const int *p3 = p2;  //靠右的const是顶层const，靠左的const是底层const
const int &r = ci;   //用于声明引用的const都是底层const
```

执行对象的拷贝操作时，拷入和拷出的对象必须具有相同的底层const资格，或者两个对象的数据类型能够转换。一般来说非常量可以转换为常量，反之不行。

```
int *p = p3;        //错误：p3包含底层const含义，而p没有
p2 = p3;            //正确：p2和p3都是底层cosnt
p2 = &i;            //正确：int*能转换为const int*
int &r = ci;        //错误：普通的int&不能绑定到const常量上
const int &r2 = i;  //正确：const int&可以绑定到一个普通int上
```

7.constexpr和常量表达式：指值不会改变并且在编译过程中就能得到计算结果的表达式。显然，字面值属于常量表达式。

constexpr变量：c++11种允许将变量声明为constexpr类型以便由编译器来检验变量的值是否是一个常量表达式。声明constexpr的变量一定是一个常量，而且必须用常量表达式初始化。

```
constexpr int mf = 20;         //20是常量表达式
constexpr int limit = mf + 1;  //mf+1是常量表达式
constexpr int sz = size();     //只有当size是一个constexpr函数时才是一条正确的语句
```

指针和constexpr：在constexpr生命中如果定义了一个指针，限定符constexpr仅对指针有效，与指针所指对象无关。

```
const int *p = nullptr;      //p是一个指向整型常量的指针
constexpr int *q = nullptr;  //q是一个指向整型的常量指针
```

2.5 处理类型

1.类别别名：是一个名字，是某种类型的同义词。

- 传统方法：使用关键字typedef

```
typedef double wages;
typedef wages base, *p;
```

- c++11规定了一种新的方式：使用**别名声明**来定义类型的别名：

```
using SI = Sales_item;
```

2.auto类型说明符：能让编译器替我们去分析表达式所属的类型

3.decltype类型指示符：选择并返回操作数的数据类型

ps：使用auto和decltype时一定注意他们的结果类型和表达式形式密切相关。如果decltype使用的是一个不加括号的变量，则得到的结果是该变量的类型；如果加上了一层或多层括号，decltype会得到一个引用类型：

```
decltype ((i)) d;  //错误：d是int&类型，必须初始化
decltype (i) e;    //正确：e是一个未初始化的int
```

2.6 预处理器

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（preprocessor），它由C++语言从C语言继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前已经用到了一项预处理功能#include，当预处理器看到#include 标记时就会用指定的头文件的内容代替#include。

C++程序还会用到的一项预处理功能是头文件保护符（header guard），头文件保护符依赖于预处理变量。预处理变量有两种状态:已定义和未定义。#define 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：#ifdef 当且仅当变量已定义时为真，#ifndef 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到#endif指令为止。

ps：预处理变量无视c++语言中关于作用域的规则