第 2 章 变量和基本类型

2.1 基本内置类型

算术类型（arithmetic type）：

- 字符
- 整型数
- 布尔值
- 浮点数

空类型 （void）：不对应具体的值，仅用于一些特殊场合.。

2.1.1 算术类型

算术类型分为两类：**整型** （integral type，包括字符和布尔类型在内）和浮点型。

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

布尔类型（bool）的取值是真（true）或者假（false）。

带符号类型和无符号类型

除去布尔型和扩展的字符型之外，其他整型可分为 **带符号的（signed）** 和 **无符号的（unsigned）** 两种。缺省时表示 `signed`。

2.1.2 类型转换

对象的类型定义了对象能包含的数据和能参与的运算，其中一种运算被大多数类型支持，即将对象从给定的一种类型 **转换**（convert）为另一种相关类型。

类型所能表示的值的范围决定了转换的过程：

- 非布尔类型的算术值赋值给布尔类型时，初始值为 0 则结果为 false，否则为 true。
- 布尔值赋值给非布尔类型时，初始值为 false，结果为 0，为 true 时，结果为 1。
- 浮点数赋值给整数类型时，进行近视处理。结果值将仅保存浮点数中整数部分。
- 整数赋值给浮点类型时，小数部分记为 0。如果该整数所占的空间超过了浮点类型的容量，精度可能有损失。
- 赋值给无符号类型（unsigned）一个超过它范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。
- 赋值给带符号类型一个超过它范围的值，结果是 **未定义的**。

含有无符号类型的表达式

先进行类型转换，再运算。**向有符号类型进行转换**。

2.1.3 字面值常量

一个形如 42 的值被称为 **字面值常量**（literal），这样的值一望而知。每个字面值常量都对应一种数据类型，字面值常量的形式和值决定了它的数据类型。

整型和浮点型字面值

可将整型字面值写作十进制、八进制或十六进制数的形式。

- 20 // 十进制
- 024 //八进制
- 0x14 // 十六进制

浮点型字面值表现为一个小数或以科学计数法表示的指数，其中指数部分用 E 或 e 表示。默认的。浮点型字面值是一个 double，可以使用后缀来表示其他浮点型。

字符和字符串字面值

由单引号括起来的一个字符称为 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字面值。字符串字面值实际上是由常量字符构成的数组（array）。编译器在每一个字符串的结尾处添加空字符（'\0'），故字符串字面值的实际长度比它的内容多 1。

```
'a' // 字符字面值 "Hello world" // 字符串字面值
```

如果两个字符串字面值位置紧临且仅由空格、缩进和换行符分隔，则他们实际上是一个整体。如下，可将一个长字符串分成多行。

```
    std::cout << "a really, really long string literal "
                 "that spans two lines" << std::endl;
```

转义序列

两类字符不可直接使用：不可打印 字符，有特殊含义的字符。此时需要转义序列： 换行符 \n 横向制表符 \t 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

通过添加前缀或后缀，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	Unicode16 字符	char16_t
U	Unicode32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8 仅用于字符串字面值常量	char

整型字面值

后缀	最小匹配类型
u or U	unsigned
l or L	long
ll or LL	long long

浮点型字面值

后缀	类型
f or F	float
l or L	long double

布尔字面值和指针字面值

true 和 false 是布尔类型的字面值。 bool test = true;

nullptr 是指针字面值。

2.2 变量

变量提供一个具名的、可供程序操作的存储空间。C++ 中的每个变量都有其数据类型。『变量』和『对象』一般可以互换使用。

2.2.1 变量定义

基本形式：类型说明符 + 一个或多个变量名组成的列表，变量名以逗号分隔，以分号结束。列表中的每个变量名的类型都由类型说明符指定。 int num = 0, value, units_sold = 0; Sales_item item; std::string book('fdfdf')

初始值

对象在创建时获得一个特定的值，说对象被 **初始化**。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。 double price = 109.99, discount = price * 0.16;

值得注意的是 *初始化* 和 *赋值* 的区别。

- 初始化的含义是创建变量时赋予其一个初始值
- 赋值的含义是把对象当前的值擦除，而以一个新的值替代

列表初始化

C++ 语言定义了初始化的几种不同形式。 int units_sold = 0; int units_sold = {0}; int units_sold{0}; int units_sold(0);

在 C++11 新标准下，用花括号进行初始化得到了全面应用。这种初始化形式被称为 **列表初始化**。

这种初始化方法用于内置类型变量时：如果我们使用列表初始化且初始值存在丢失信息的风险，则编译器报错。 long double ld = 3.1415926536 int a{ld}, b = {ld}; // 报错，拒绝进行有风险的转换 int c(ld), d = ld; // 不报错，且进行了转换，丢失信息

默认初始化

定义变量时没有指定初始值，则变量被 **默认初始化**。默认值由变量类型决定，同时受到定义变量位置的影响。

内置类型的默认初始化时，值由定义位置决定：

- 定义于函数之外的变量被初始化为 0。
- 定义于函数体内部的 **不被初始化**，其值是未定义的。

每个类各自决定其初始化对象的方式，是否允许默认初始化及默认初始化的值由类自身决定。

建议：初始化每一个内置类型的变量。

2.2.2 变量声明和定义的关系

C++ 语言支持 **分离式编译** 机制，允许将程序分割为多个文件，每个文件可被独立编译。显然，需要文件间共享代码的方法。

为支持分离式编译，C++ 语言将声明和定义区分开来。

- **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。

```
extern int i;   // 声明
int j;          // 声明且定义
```

注意：变量能且只能被定义一次，但可被多次声明。

2.2.3 标识符

C++ 的标识符由字母、数字和下划线组成，必须以字母或下划线开头。长度无限制，且对大小写敏感。

- 一些 C++ 保留字不能被使用为标识符。
- 自定义的标识符不能连续出现两个下划线。
- 自定义的标识符不能以下划线紧连大写字母开头。
- 定义在函数体外的标识符不能以下划线开头。

变量命名规范

若能坚持，必将有效。

- 体现实际含义。
- 变量名一般用小写字母。
- 自定义类名以大写字母开头。
- 标识符由多个单词组成，单词间有明显区分。

2.2.4 名字的作用域

作用域（scope）是程序的一部分，C++ 中大多数作用域都以花括号为分隔。

- **全局作用域**
- **块作用域**

建议：第一次使用变量时再定义它。

嵌套的作用域

作用域能彼此包含，此时出现了 **内层作用域** 和 **外层作用域**。

作用域中定义的名字，在其嵌套的所有作用域中都能访问，但允许在内存作用域中重新定义外层作用域已有的名字。

警告：如果函数有可能用到某全局变量，则不宜再定义一个同名的局部变量。

2.3 复合类型

复合类型（compound type）指基于其他类型定义的类型，在 C++ 中有几种复合类型，如：引用和指针。

声明语句 = **基本数据类型**（base type） + **声明符**（declarator） 列表。每个声明符命名了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。

2.3.1 引用

Note：右值引用和左值引用区别。使用术语『引用』一般指左值引用。

引用（reference） 为对象起了另外一个名字，引用类型引用（refers to）另外一种类型。

```
int ival = 1024;
int &refVal = ival;     // refVal 指向 ival
int &refVal2;           // 报错 引用需初始化
```

初始化变量时，初始值被拷贝到新建的对象中。定义引用时，引用和初始值 **绑定** 在一起，且这种绑定对引用来说无法改变，即无法令引用重新绑定到另外一个对象。

引用即别名

Note：引用并非对象，它只是为一个已经存在的对象所起的另外一个名字。

引用本身不是对象，不能定义引用的引用。

引用的定义

允许在一条语句中定义多个引用，每个引用标识符都必须以符号 & 开头。

```
int i = 1024, i2 = 2048;
int &r = i, r2 = i2;
int i3 = 1024, &ri = i3;
int &r3 = i3, &i4 = i2;
```

引用的类型都要和与之绑定的对象严格匹配，且只能绑定在对象上，不能与字面值或计算结果绑定。

2.3.2 指针

指针 是『指向』另外一种类型的复合类型。与引用类似，指针也实现了对其他对象的间接访问。但：指针本身就是对象，允许值的拷贝和赋值，且可以指向不同的对象；其二，指针无需在声明时赋初值。

```
int *ip1, *ip2;
```

获取对象的地址

指针存放某个对象的地址，要想获取该地址，需要使用 **取地址符**：&。

```
double dval;
double *pd = &dval;     // 正确
double *pd2 = pd;       // 正确
/*-------------------------*/
int *pi = pd;           // 错误，类型不匹配
pi = &dval;             // 错误，类型不匹配
```

注意类型匹配。

指针值

指针值处于如下四种状态：

1. 指向一个对象。
2. 指向紧邻对象所占空间的下一个位置。
3. 空指针，没有指向任何对象。
4. 无效指针。

试图拷贝或者以其他形式访问无效指针的值都将引发错误，引发的后果无法预计。

利用指针访问对象

若指针指向一个对象，允许使用 **解引用符**（*）来访问该对象。

```
int ival = 42;
int *p = &ival;
std::cout << *p;
```

对指针解引用会得到指针指向的对象，对指针解引用的对象赋值，即是对对象赋值。

Note：解引用操作仅适用于那些确实指向了某个对象的有效指针。

空指针

空指针 不指向任何对象，使用指针前可检查其是否为空指针。

用字面值 **nullptr** 来初始化指针得到空指针，也可在初始化直接令指针为 0。

```
int *p1 = nullptr;
int *p2 = 0;
// #include <cstdlib>
int *p3 = NULL;
```

建议：初始化所有指针。

赋值和指针

赋值永远改变的是等号左侧的对象。

其他指针操作

只要指针拥有合法值，就能将它用在条件表达式中。

对类型相同的指针，可以进行逻辑运算：== 或 !=。

void* 指针

void* 是一种特殊的指针类型，可用于存放 **任意对象** 的地址。

不能直接操作该指针类型的指针指向的对象。以 void* 的视角来看内存空间也仅仅是内存空间，没办法访问内存空间中所存的对象。

2.3.3 理解复合类型的声明

变量的定义 = 基本数据类型 + 一组声明符。不要迷惑于基本数据类型和类型修饰符的关系，后者不过是声明符的一部分罢了。

定义多个变量

坚持一种写法，将 * 或 & 与变量名连在一起。

指向指针的指针

声明符中修饰符的个数没有限制，故存在指向指针的指针，指向指针的指针的指针，以此类推。

```
int ival = 0;
int *p1 = &ival;
int **p2 = &p1;
int ***p3 = &p2;
```

指向指针的引用

引用不是对象，而指针是对象，故不能定义指向引用的指针，但存在对指针的引用。

```
int i =42;
int *p;
int *&r = p;    // 从右向左看

r = &i;
*r = 0;
```

从右向左阅读变量 r 的定义，离它最近的修饰符对其有最直接的影响，如上可看到，r 是一个引用，再外层的修饰符 * 表示该引用引用了一个指针。

2.4 const 限定符

用 **const** 符限定，变量的值不能被改变，故在声明时必须进行初始化。

```
const int j = 42;
```

初始化和 const

只能在 const 类型的对象上执行不改变其内容的操作，初始化也是这种操作，不改变内容。

默认 const 对象仅在文件内有效

编译器在编译时把所有的 const 类型的变量用其初始值代替。为避免对同一变量的重复定义，默认 const 对象被设定为仅在文件内有效。

Note：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。

2.4.1 const 的引用

将引用绑定到 const 类型上，称为 **对常量的引用**。显然，不能修改引用对象的值。

```
const int ci =1024;
const int &ri = ci;

int &r2 = ci;       //错误
```

初始化对 **const** 的引用

一个常量引用被绑定到另外一种类型上发生了什么，这导致这种操作为非法。

对 **const** 的引用可能引用一个并非 **const** 的对象

常量引用仅对引用可参与的操作进行了界定，对于引用的对象本身是不是一个常量没有做界定，该对象也可以不是常量，可以通过其他途径修改值。

2.4.2 指针和 const

可令指针指向常量或非常量，**指向常量的指针** 不能用于改变其所指对象的值。

```
const double pi = 3.14;
double *ptr = &pi;          // 错误
const double *cptr = &pi;   // 正确
*cptr = 42;                 // 错误
```

同常量引用一样，指向常量的指针也可在初始化时指向非常量。

const 指针

指针是对象，故允许把指针定义为常量，即 **常量指针**，同样的，必须在声明时就进行初始化。把 * 放在关键字 const 之前说明指针是一个常量。

```
int errNumb = 0;
int *const curErr = &errNumb;
const double pi = 3.14;
const double *const pip = &pi;
```

从右向左读。

2.4.3 顶层 const

用名词 **顶层 const** 表示指针本身是一个常量，而名词 **底层 const** 表示指针指向的对象是一个常量。

更一般的，顶层 const 可以表示任意对象是常量，这对任何数据类型都适用。底层 const 则与指针和引用等复合类型的基本类型部分有关。

```
int i = 0;
int *const p1 = &i;     // 顶层 const
const int ci = 42;      // 顶层 const
const int *p2 = &ci;    // 底层 const
const int &r = ci;      // 用于声明引用的 const 都是底层 const
```

执行拷贝操作时，常量是顶层 const 还是底层 const 区别明显：

- 顶层 const：几乎不受什么影响
- 底层 const：拷贝操作时，拷入和拷出的对象必须具有相同的底层 const，或者两个对象的数据类型必须能够转换。

2.4.3 constexpr 和常量表达式

常量表达式 指值不会改变且在编译过程就能得到计算结果的表达式。字面值属于常量表达式，用常量表达式初始化的 const 对象也是常量表达式。

一个对象（或表达式）是不是常量表达式由它的数据类型和初始值共同决定。

```
const int max_files = 20;           // 是
const int limit = max_files + 1;    // 是
int staff_size = 27;                // 不是
const int sz = get_size();          // 不是
```

constexpr 变量

很难（几乎不能）分别一个初始值到底是不是常量表达式。

C++11 新标准规定，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。

```

```

```
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();      // 当 size 是一个 constexpr 函数 正确
```

字面值类型

声明 constexpr 时用到的类型被称为『字面值类型』。

算术类型、引用和指针都输入字面值类型。constexpr 指针的初始值必须是 nullptr 或 0 或是存储于某个固定地址中的对象。

定义在函数体内的对象地址不固定，故不能用来初始化 constexpr 指针，在函数体外的对象其地址固定不变，可用来初始化 constexpr 指针。

指针和 constexpr

限定符 constexpr 仅对指针有效，与指针指向的对象无关。

2.5 处理类型

处理类型越来越复杂的问题。

2.5.1 类型别名

类型别名 是一个名字，是某种类型的同义词。有两种方法定义类型别名。

1. 使用关键字 **typedef**。

   typedef double wages;

   typedef wages base, *p;

2. 使用 **别名声明**：using + 别名 + = + 真实名字。

   using SI = Sales_item;

指针、常量和类型别名

2.5.2 auto 类型说明符

C++11 引入了 **auto** 类型说明符，让编译器去替我们分析表达式所属的类型。auto 定义的变量必须有初始值。

```
auto item = val1 + val2;
```

auto 也可在一条语句中声明多个变量，不过所有变量的数据类型必须相同。

复合类型、常量和 auto

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

- 推断引用类型时，得到的是引用所引用的类型，而不是引用类型。
- auto 会忽略掉顶层 const，同时会保留底层 const。
- 希望推断出的 auto 类型是一个顶层 const，需明确指出。
- 还可将引用类型设为 auto。

```
    int i = 0, &r = i;
    auto a = r;         // a 是 int 整型 而不是 引用类型

    const int ci = i, &cr = ci;
    auto b = ci;                // b 为 int 忽略了 const
    auto c = cr;                // c 为 int
    auto d = &i;                // d 为 int 的指针
    auto e = &ci;               // e 为 指向 const int 的指针 对常量取地址 视为 底层
const

    const auto f = ci;          // f 为 const int

    auto &g = ci;               // g 为 const int 引用 引用 ci
    auto &h = 42;               // 错误
    const auto &j = 42;         // 正确 可以为常量引用绑定字面值
```

2.5.3 decltype 类型指示符

从表达式的类型推断出要定义的变量的类型，但是不希望用该表达式的值初始化变量。可使用第二种类型说明符 **decltype**，其作用是选择并返回操作数的数据类型。

```
    decltype(f()) sum = x;
```

decltype 处理顶层 const 引用的方式与 auto 有所不同。如果 decltype 使用的表达式是常量，则 decltype 返回该变量的类型（包括顶层 const 和引用在内）。

decltype 和引用

若 decltype 使用的表达式不是一个变量，则 decltype 返回表达式结果对应的类型。

```
    int i = 42, *p = &i, &r = i;
    decltype(r + 0) b;              // 正确 int
    decltype(*p) c;                 // 错误 c 是 int& 必须进行初始化
```

如果表达式的内容是解引用操作，则 decltype 将得到引用类型。

decltype 的结果类型与表达式形式密切相关：对于 decltype 所用的表达式来说，如果变量名加上了一对括号，则得到的类型与不加括号会不同。

- 如果 decltype 使用的是一个不加括号的变量，则得到的结果就是该变量的类型。
- 如果给变量加了一层或多层括号，编译器会把它当成是表达式。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。

```
decltype((i)) = d;  // 错误 d 是 int& 必须初始化
decltype(i) = e;    // 正确
```

2.6 自定义数据结构

数据结构是把一组相关的数据元素组织起来然后使用它们的策略和方法。

2.6.1 定义 Sales_data 类型

以关键字 **struct** 开始，紧跟类名和类体，就像 C 语言定义结构体一样。

类数据成员

类体定义类的成员，我们的类暂时只有 **数据成员**，可以为数据成员提供一个 **类内初始值**。

2.6.2 使用 Sales_data 类

2.6.3 编写自己的头文件

为确保各个文件中类的定义一致，类通常被定义在头文件中，且类所在头文件的文件名应该和类的名字相同。

头文件通常包含那些只被定义一次的实体，如 类、const 和 constexpr。

要在书写头文件时做适当处理，使其在遇到多次包含时也能正常工作。

Note：头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

预处理器概述

确保多次包含仍能正常工作的是 **预处理器**，这是在编译前执行的一段程序。

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，**#ifdef** 当且仅当变量已定义时为真，**#ifndf** 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直到遇到 **#endif** 指令为止。

整个程序中预处理变量包括头文件保护符必须唯一。