第2章 变量和基本类型

2.1 基本内置类型

2.1.1 算数类型

1. 一个 `char` 的空间应确保可以存放机器基本字符集中任意字符对应的数字值，其大小和机器字节一样。
2. `wchar_t` 类型用于确保可以存放机器最大扩展字符集中的任意一个字符，类型 `char16_t` 和 `char32_t` 则为 *Unicode* 字符集服务（*Unicode* 是用于表示所有自然语言中字符的标准）。
3. 大多数计算机以 2 的整数次幂个比特 *bite* 作为块来处理内存，可寻址的最小内存块称为字节 *byte*，存储的基本单元称为字 *word*，它通常由几个字节组成。
4. 大多数机器的字节由 8 个比特构成（1*byte* == 8*bite*）
5. 32位：1*word* == 4*byte* == 32*bite*
6. 64位：1*word* == 8*byte* == 64*bite*
7. 为了赋予内存中某个地址明确的含义，必须首先知道存储在该地址的数据的类型。类型决定了数据所占的比特数以及该如何解释这些比特的内容。
8. 通常 `float`：1*word*(32*bite*)、 `double`：2*word*(64*bite*)、 `long double`：3或4*word*(96或128*bite*)

2.1.2 类型转换

1. 当我们赋值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数值总数取模后的余数。（a = b x q + r，其中 |r|<|a|。负数取模：r = a - (a / b) x b, 其中 a/b 在 C++ 中是向0取整）
2. 当我们赋给带符号类型一个超出它表示范围的值时，结果是未定义的，程序可能继续工作、可能崩溃、也可能生成垃圾数据。
3. 程序应该尽量避免依赖与实现环境的行为。
4. 切勿混用带符号类型和无符号类型。

2.1.3 字面值常量

1. 我们可以将整型字面值写作十进制、八进制、十六进制数的形式。以 0 开头的整数代表八进制，以 0x 或者 0X 开头的代表十六进制数。
2. 如果一个字面值连与之关联的最大的数据都放不下，将产生错误。类型 `short` 没有对应的字面值。

3. 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型字面值。
4. 如果两个字符串字面值位置紧邻且仅由空格、缩进和换行符分隔，则它们实际上是一个整体。
5. 转义序列 *escape sequence*

2.2 变量

2.2.1 变量定义

1. 初始化不是赋值，初始化的含义是创建变量时赋予其一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值替代。

2.2.2 变量声明和定义的关系

1. 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` ，而且不要显式地初始化变量。
2. 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。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。

2.2.3 标识符

1. 用户自定义的标识符中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。此外，定义在函数体外的标识符不能以下画线开头。

2.3 复合类型

2.3.1 引用

1. 区分左值引用 *lvalue reference* 和右值引用 *rvalue reference*。
2. 引用本身不是一个对象，所以不能定义引用的引用。
3. 引用只能绑定在对象上，而不能与字面值或某个表达式的计算结果绑定在一起。

2.3.2 指针

1. 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。

2. `NULL` 为预处理变量，这个变量在头文件 `cstdlib` 中定义，它的值就是 `0` 。
3. 在 C++11 新标准下，最好使用 `nullptr` ，同时尽量避免使用 `NULL` 。
4. `void*` 是一种特殊的指针类型，可用于存放任何对象的地址。
5. 利用 `void*` 指针能做的事：
   (1)拿它和别的指针比较
   (2)作为函数的输入或输出
   (3)赋给另外一个 `void*` 指针。
6. 无法访问和操作 `void*` 内存空间中所存的对象。

2.3.3 理解复合类型的声明

1. 面对一条比较复杂的指针或者引用的声明语句时，从右向左阅读有助于弄清楚它的含义。如：

```
int *p;
int *&r = p;     // r是一个指向指针p的引用。
```

2.4 const限定符

1. 利用一个对象去初始化另外一个对象，则它们是不是 `const` 都无关紧要。
2. 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，其实等同于在不同的文件中分别定义了独立的变量。
3. 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。

2.4.2 指针和const

1. 指向常量的指针 *pointer to const*：

```
const double pi = 3.14;
const double * cptr = &pi;
```

2. 常量指针 *const pointer*：把 `*` 放在 `const` 之前用以说明指针是一个常量，即不变的是指针本身的值而非指向的那个值。

```
int errNumb = 0;
int *const curErr = &errNumb;
```

3. 指向常量对象的常量指针：

```
const double pi = 3.14;
const double *const pip = &pi;
```

2.4.3 顶层 const

1. 一般的来说，顶层 const 可以表示任意的对象是常量。底层 const 则与指针和引用等复合类型的基本类型部分有关。
2. 比较特殊的是，指针类型既可以是顶层 const (*top-level const*) 也可以是底层 const (*low-level const*)。顶层 const 表示指针本身是一个常量，底层 const 表示指针所指的对象是一个常量。（ int *const 顶层， const int* 底层）
3. 当执行对象拷贝操作时，是否为顶层 const 不影响拷贝。
4. 当执行对象拷贝操作时，拷入和拷出的对象都必须具有相同的底层 const 资格，或者两个对象的数据类型必须能够转换（非常量可以转换成常量，反之则不行）。

2.4.4 constexpr 和常量表达式

1. 常量表达式 *const expression* 是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。
2. constexpr 会把它所定义的对象置为顶层 const ，即常量对象。
3. 尽管指针和引用都能定义成 constexpr ，但是它们的初始值却受到严格的限制。一个 constexpr 指针的初始值必须是 nullptr 或者 0 ，或者存储于某个固定地址的对象。
4. 在 constexpr 声明中如果定义了一个指针，限定符 constexpr 仅对指针有效，对指针所指的对象无关。

2.5 处理类型

2.5.1 类型别名

1. 传统方法 typedef ：

```
typedef double base, *p;     // base是double的同义词，p是double*的同义词
```

2. 新规范使用别名声明 *alias declaration*：

```
using SI = Sales item;
```

3. 如果某个类型别名指代的是复合类型或者常量：

```
typedef char *pstring;
const pstring cstr = 0;     //cstr是指向char的常量指针，等同于char* const cstr = 0;
const pstring *ps;          // ps是一个指针，它的对象是指向char的常量指针
```

2.5.2 auto类型说明符

1. auto 一般会忽略掉顶层 const ，同时底层 const 则会保留下来。
2. 设置一个类型为 auto 的引用时，初始值中的顶层常量属性仍然保留。如果我们给初始值绑定一个引用，则此时的常量就不是顶层常量了。（此规则理解不清晰）

2.5.3 decltype类型指示符

1. 如果希望从表达式的类型推断出要定义的变量类型，但是不想用该表达式的值初始化变量，使用 decltype （选择并返回操作数的数据类型）。
2. decltype(*p) 结果类型是 int& ，而非 int 。

```
int i = 0, *p = &i;
decltype(*p) z = i;
```

3. 对于 decltype 所用的表达式来说，如果变量名加上一对括号，得到的类型于不加括号时会不一样：

```
// 不加括号，得到的结果就是该变量的类型：
decltype(i) e = 0;       // e是int

// 加括号，编译器会把它当作一个表达式，变量是一种可以作为赋值语句左值的特殊表达式，所以会得到引用类型：
decltype((i)) e = i;     // d是int&（加括号时，结果永远是引用）
```

2.6 自定义数据结构

2.6.1 定义Sales_data类型

1. 不能使用圆括号进行类内初始化。

2.6.3 编写自己的头文件

1. 预处理变量无视 C++ 中作用域的规则。