# Chapter 2 变量和基本类型

## 2.1 基本内置类型

1. C++算数类型表：

| 类型 | 含义 | 最小尺寸（至少） |
| --- | --- | --- |
| bool | 布尔 | Not defined |
| char | 字符 | 8 bit |
| wchar_t | 宽字符 | 16 bit |
| char16_t | Unicode字符 | 16 bit |
| char32_t | Unicode字符 | 32 bit |
| short | 短整型 | 16 bit |
| int | 整型 | 16 bit |
| long | 长整型 | 16 bit |
| long long | 长整型 | 64 bit |
| float | 单精度浮点数 | 至少6位有效数字 |
| double | 双精度浮点数 | 至少10位有效数字 |
| long double | 扩展精度浮点数 | 至少10位有效数字 |

2. 字符类型：

    - 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，也就是说一个char的大小刚好和一个机器字节一样（1 byte）
    - 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
    - char16_t和char32_t为unicode字符集服务

3. 内置类型的机器实现

    - 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）
    - 大多数机器的字节由8 bit组成，字则有32或者64 bit，也就是4或8字节

4. 通常float以一个字（word）来表示，double由两个字（2 word）来表示（64 bit），long double则有3或4个字。long double通常被用于有特殊浮点需求的硬件。

5. 符号类型

    - 带符号类型（signed）可以表示整数、负数或者0，无符号则只能表示大于等于0的数
    - int、short、long和long long都是带符号的，前面加上unsigned就是无符号类型

- **unsigned int可以简写成unsigned**
- char也有char， unsigned char， signed char。但是此处**signed char 和 char类型不一样**，char实际上只有signed char和unsigned char，而char则由编译器决定表示两种中的哪一种
- 无符号类型所有bit都用来存储值（首bit不是符号位）
- 为保证正负值的表示范围平衡，8 bit的signed char虽然理论上可以表示-128 到 128，但实际上是-128 到 127（**0算作正方向上的值从而使得正负范围平衡**）

6. 类型选择的建议：

- 当明确知道数值不为负，选用unsigned
- 优先使用int。short太短，long长度往往和int相同，过长数据则选用long long
- 算术表达式中不要选择char或者bool（因为char在不同的机器上可能表现signed或者 unsigned，如果不得不使用，明确是signed char还是unsigned char）
- 浮点运算优先使用double，float常常精度不够，两者计算代价相差不大，某些机器double更快
- 往往不需要long double精度，而且运行消耗比较大

7. 类型转换

- 当用非布尔类型的算数值为布尔值赋值时，只有值为0时布尔值才为false，其他为true
- 当把浮点数赋值给整数类型时，结果只保留小数点前的部分
- 当我们把一个整数赋给浮点数时，小数记为0，如果整数空间超过浮点类型的容量，可能会导致精度损失
- unsigned char表示0-256，如果赋值超出范围，比如-1，那么我们表示的数与256取模获得的余数为255（关于什么是取模，如何计算出255，看第8条）
- 如果我们赋值给signed char超出范围，那么结果是undefined（未定义的）

```
bool b = 42; // b为真
int i = b; // i为1
i = 3.15; // i为3
double pi = i; // pi为3.0
unsigned char c = -1; // 假设char占8bit，那么c为255
signed char c2 = 256; //假设char占8bit，那么c2为未定义的

int i = 42;
if(i)  // 此时i会被编译器自动转换为true
    i = 0;
```

8. 取模定义

> 对于整数a，b来说，取模运算或者求余运算的方法要分如下两步：
> 1.求整数商：c=a/b
> 2.计算模或者余数：r=a-(c*b)
> 求模运算和求余运算在第一步不同
> 取余运算在计算商值向0方向舍弃小数位

> 取模运算在计算商值向负无穷方向舍弃小数位
> 例如：4/(-3)约等于-1.3
> 在取余运算时候商值向0方向舍弃小数位为-1
> 在取模运算时商值向负无穷方向舍弃小数位为-2
> 所以
> 4rem(-3)=1
> 4mod(-3)=-2

- 根据上方定义，\(-1\) mod 256 第一步计算商为 $\frac{-1}{256}$，取模向负无穷舍弃小数位则为-1，所以此时取模的值就等于-1 - \(-1\) * 256 = 255。

9. 当一个算术表达式中既有无符号数又有int值的时候，int值会被自动转换成无符号数。

```
unsigned u = 10;
int i = -42;
cout << i + i <<endl; // 输出-84
cout <<u + i <<endl; // 如果int占32bit，输出4294967264
```

- 此处u + i时，i自动被转换成unsigned int，但-42超出0到$2^{32}$（也就是4294967296）的范围，于是相对于4294967296取模，按照上述方法，先求得商为$\frac{-42}{4294967296}$，向负无穷取值为-1，然后求得模的值为4294967254，加上u的10，算得结果4294967264。
- 由此而言，我们一定要**确保unsigned的数值不能为负数,最好不要混用带符号和不带符号类型**。

10. 当把unsigned值应用于for循环时:

```
for(unsigned u = 10; u >= 0; --u){
  cout << u << endl; // 此时u永远不会小于0，因为当u=0后再减一，会获得求模后的正数
}
```

- 要避免这种情况一般用while代替for循环，同时将判断语句设置为u大于1

```
unsigned u = 11;
while( u > 1){
--u;
cout << u << endl;
}
```

11. 字面值常量：

- 十进制字面值是带符号数，八进制和十六进制既可能带符号也可能不带符号
- 十进制字面值类型是int、long、long long中能容下该数值的最小者
- 八进制和十六进制是int、unsigned int、long、unsigned long、long long、unsigned long long中能容下该数值的尺寸最小者
- 浮点型字面值是double

12. 字面值常量表

    - 前缀

| 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 | char |

    - 后缀（尽量使用L来代替小写的l，因为小写的l像数字1）

| 后缀整型 | 最小匹配类型 |
| --- | --- |
| u or U | unsigned |
| l or L | long |
| ll or LL | long long |

| 后缀浮点型 | 类型 |
| --- | --- |
| f or F | float |
| l or L | long double |

13. 例子：

```
L'a'; // 宽字符字面值，类型是wchar_t
u8"Hi"; // utf-8字符串字面值
42ULL; // 无符号整型字面值，后缀，所以是unsigned long long
1E-3F; // 单精度浮点数字面值，float
3.14159L; // 扩展精度浮点型字面值，类型是long double
```

14. true和false是布尔类型字面值，nullptr是指针字面值

## 2.2 变量

1. 初始化和赋值都使用“=”，然而初始化不是赋值，初始化的含义是创建变量的同时赋予一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值来替代。

2. c++11的新标准加入了“{}”来初始化变量，这种初始化形式被称为列表初始化（list initialization）

3. 当使用列表初始化且初始值存在丢失信息的风险，编译器会报错。

```
long double ld = 3.1415123;
int a{ld}, b{ld}; // int转换成double，报错，会导致信息损失
```

```
int c(ld), d(ld); // 通过，忽略已发生的信息损失
```

4. 当定义变量时没有定义初值，那么变量会被默认初始化。如果**内置变量**未被初始化，且其在任何函数体之外，变量会被初始化为0。例外是，定义在函数体内部的**内置类型变量**将不被初始化，而该值就是未被定义的（undefined），此时拷贝或引用将发生错误。**建议初始化每一个内置类型的变量**。

5. 每个类各自决定其初始化对象的方式，由它自己决定初始化的值到底是什么。例如string类如果没有初始化那么将指定初始值为一个空串。

6. c++是分离式编译机制，允许将程序分割成若干文件且各自可以独立编译，此时就需要文件间共享代码。于是c++将变量声明（declaration）和定义（definition）分离开来：

    - 声明使得名字为程序所知，即如果想使用别处定义的名字则必须在该文件中包含对其的声明
    - 定义则负责创建与名字关联的实体
    - 变量规定类型和名字，定义在此之上还申请了内存空间，也可能直接为变量赋值
    - 如果只想声明，需在名字前加上**extern**关键字，如果此时初始化了则成为定义
    - 函数内部如果初始化一个**extern**关键字标记的变量将引发错误（**不能在函数体内使用extern**）

    ```
    extern int i; // 仅声明变量i
    int j;  // 声明并定义j
    extern int p = 10; // 定义p
    ```

7. **变量只能被定义一次，但可以多次声明。当变量在多个文件中被同时使用时，需要在一个文件中定义，其他的都将只能且必须声明**

8. c++标志符：

    - 必须以字母或下划线开头（定义在函数体外的标识符不能以下划线开头）
    - 对大小写敏感
    - 不能使用c++保留字
    - 不能连续出现两个下划线
    - 不能以下划线连接大写字母开头

9. 命名规范：

    - 标识符要能体现实际含义
    - 变量名一般用小写字母
    - 自定义类一般以大写字母开头
    - 多个单词构成时中间应该有明显区分（例如下划线）

10. c++中大多数作用域都以花括号分隔。名字的有效区域始于名字的声明，以声明语句所在的作用域末端为结束

11. 如果全局变量和函数内局部变量命名冲突，则在函数内未定义局部变量前使用全局，定义同名局部变量后优先使用局部变量（不建议定义同名变量）

```
int i = 42;
int main(){
    int i = 100;
    int j = i; // j = 100;
    return 0;
}
```

12. 一般来说，在对象第一次使用的地方附近定义它是一种好的选择，这样做有助于我们更容易找到他们。

## 2.3 复合类型

1. 引用就是为对象起了另一个名字。**引用不是对象，没有实际地址**，为引用赋值就是把值赋给引用绑定的对象，获取引用的值实际上是获取引用的对象的值。

2. 一般在初始化变量时初始值会被拷贝到新建的对象中，然而定义引用时，程序把引用和他的初始值绑定（bind）在一起，而不是将初始值拷贝给引用。因此**引用必须初始化且一旦初始化无法重新绑定到另一个对象。**

3. 大部分情况下所有引用类型都必须和与之绑定的对象严格匹配，**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。**

```
int &r = 10; // 错误，引用类型初始值必须是一个对象
double dd = 3.14;
int &rr = dd; // 错误，对象类型不匹配
```

4. 指针和引用的区别：

    - 指针本身就是一个对象，允许对指针进行赋值和拷贝，在生命周期内可以更改指向到其他对象
    - 指针无需在定义时初始化。如果在区块内没有被初始化则拥有一个不确定的值。

5. 引用没有地址，所以不能定义指向引用的指针。

6. 指针的类型都要和他所指的对象严格匹配。

7. 指针值：

    - 指向一个对象
    - 指向紧邻对象的所占空间的下一个位置
    - 空指针
    - 无效指针，上述情况之外的值

8. 试图拷贝或访问无效指针的值都将引发错误。编译器也不负责检查此类错误。这点和访问未初始化的指针类似。

9. 解引用操作（操作符*）仅适用于那些确实指向了某个对象的有效指针。

10. 空指针用nullptr来初始化指针。另一种方法是通过字面值0来初始化指针。注意：直接为指针赋值int型变量0是错误的。

```
int zero = 0;
ptr = zero; // 错误，不能把int变量直接赋值给指针
```

11. 过去我们会用NULL这样一个预处理变量来给指针赋值，其值就是0。预处理器是运行于编译过程之前的一段程序。我们可以直接使用预处理变量而不需加上std::。预处理变量会被预处理器自动转换成实际值。

12. **建议初始化所有指针，以避免访问未初始化的指针时出错。**

13. 为指针赋值0也可以让指针指向空。

```
int i = 1;
int *pi = &i;
int *pi2; // 此时如果pi2在块内，则其值无法确认，不推荐。
pi2 = pi; // pi2也指向i
pi = 0; // pi不再指向任何对象
```

14. 只有空指针在条件判断中为false，其他值得指针都为true。两个指针可以使用"=="和"!="来判断他们存放的地址是否相同。

15. void指针是一种特殊的指针，可用于存放任意对象的地址。void*存放地址，但我们不了解该指针存放地址的类型是什么。

    - 我们不能直接操作void*指针所指的对象，因为我们并不知道这个对象到底是什么类型，也无法知道在这个对象上做哪些操作

16. 指针是对象，所以存在对指针的引用

```
int i = 42;
int *p;
int *&r = p; // r是一个对指针p的引用
r = &i; // r引用了了一个指针，因此赋值给r一个i的地址也就是令p指向i
*r = 0; // 解引用r得到i，也就是p指向的对象将i的值变为0
```

## 2.4 const限定符

1. 因为const对象一旦创建后就无法改变，所以**一定要进行初始化**（类似于引用的初始化性质）

2. 如果用一个对象**去初始化另一个对象**，则他们**是不是const都无关紧要**：

```
const int k; // 不合法，const必须初始化
int i = 42;
const int ci = i; // 合法
int j = ci; // 合法
```

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

```
// 在file.cc中定义并初始化一个常量，该常量能被其他文件访问
extern const int i = 10;
// 在另一个文件file.h中声明该变量，表明该变量于cc文件里的i是同一个变量
extern const int i;
```

4. 常量对象只能用常量引用来引用它,而不能使用一般引用

```
const int i = 42;
const int &r = i;
r = 20; // 错误，常量引用无法修改值
int &r2 = i; // 错误，无法用非常量引用来引用一个常量值
```

5. 相反的，允许常量引用绑定一个非常量的对象或字面值。**常量引用可以引用一个字面值，或是一般表达式。**

```
int i = 10;
const int &r1 = i; // 正确，常量引用可以绑定非常量
const int &r2 = 1; // 正确，常量引用可以引用一个字面值，如果去除const则不正确
const int &r3 = r1 * 2; // 正确，常量引用可以引用一般表达式
int &r4 = r1 * 2; // 错误，非常量引用无法绑定一个表达式的计算结果
```

6. 临时量（temporary）:所谓临时量对象就是当编译器需要一个空间来暂存表达式的求值结果时临时创建的一个未命名的对象。

```
double v1 = 3.14;
// const引用可以用任意表达式作为初始值，只要该表达式的结果可以转换成引用类型
const int &r = v1;
/*
* 原因在于此处应该r无法修改所引用的内容，相对来说仍然是安全的，但如果r不是const的，
* 那么就会导致r可以修改内容，但是所引用的内容是double，而引用本身却是int，
* 此时就会产生错误。
*/

// 上述相当于下式


const int temp = v1; // 此处temp为临时量
const int &r = temp;
```

7. 同样的，常量的地址只能被指向常量的指针（pointer to const， 将const放在类型前面，例如const int _ptr）存储，但*指向常量的指针也可以指向一个非常量对象，只是该指针不能修改其内容。指向常量的指针可以不进行初始化。

8. 被指向常量的指针或是常量引用绑定的非常量对象，虽然无法通过指针或者引用改变值，但可以通过其他方式改变值。

9. 常量指针（将const放在_的后边，例如int_ const ptr）**必须初始化**，而且一旦初始化完成它所存储的地址值就无法改变了。然而在该地址上存储的值却是依赖于其类型，如果该值不是const类型，也可以改变，只是改变的结果依然存储在该地址，所以此时常量指针所存储的值也就是该值的地址没有发生改变。

```
int *const cp; // 不合法，常量指针必须初始化
```

10. 例题2.28

```
int i, *const cp;       // 不合法, const 指针必须初始化
int *p1, *const p2;     // 不合法, const 指针必须初始化
const int ic, &r = ic;  // 不合法, const int 必须初始化
const int *const p3;    // 不合法, const 指针必须初始化
const int *p;           // 合法. 一个指针，指向 const int。指向常量的指针可以不初始
```

11. 例题2.29

```
//假设已有上一个练习中定义的那些变量，则下面的哪些语句是合法的？请说明原因。
i = ic;     // 合法, 常量赋值给普通变量
p1 = p3;    // 不合法, p3 是const指针不能赋值给普通指针
p1 = &ic;   // 不合法, 普通指针不能指向常量
p3 = &ic;   // 合法, p3 是常量指针且指向常量
p2 = p1;    // 合法, 可以将普通指针赋值给常量指针
ic = *p3;   // 不合法, ic是const int，初始化后不能再改变其值
```

12. 顶层const（top-level const) 表示指针**本身是一个常量**；底层const（low-level const）表示指针**所指的对象是一个常量**。

13. 指针类型既可以是顶层const，也可以是底层const。**用于声明引用（&）的const都是底层const**。

14. 执行拷贝操作并不会改变被拷贝对象的值，所以考入和考出的对象对于顶层const是否是常量都没什么影响。然而对于底层const，考入考出的拷贝对象**必须具有相同的底层const，或者两个对象之间必须能够转换**（一般来说，非常量可以转换成常量，反之则不行）：

```
int i = 0;
int *const p1 = &i; // 不能改变p1的值，这是顶层const
const int ci = 42; // 不能改变ci的值，这是顶层const
const int *p2 = &ci; // 允许改变p2的值，这是一个底层const
```

```
const int *const p3 = p2; // 靠右的const是顶层const，靠左的是底层const
const int &r = ci; // 用于声明引用的const都是底层const

i = ci; // 正确：拷贝ci的值，ci是一个顶层const，操作无影响
p2 = p3; // 正确：p2和p3指向的对象类型相同，p3顶层const部分不受影响
int *p = p3; // 错误：p3包含底层const，p没有，不能用const指针为非const初始化
p2 = &i; // 正确：此时int*被转换成const int*
int &r2 = ci; // 错误：普通引用无法绑定const值
const int &r3 = i; // 正确
```

15. 习题2.30

```
//对于下面的这些语句，请说明对象被声明成了顶层const还是底层const？
const int v2 = 0; int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
//v2 是顶层const，p2 是底层const，p3 既是顶层const又是底层const，r2 是底层const。
```

16. 习题2.31

```
r1 = v2; // 合法，顶层const在拷贝时不受影响。
         // 注意此处不是为r1赋值，而是将v2的值拷贝给r1引用的对象
p1 = p2; // 不合法，p2 是底层const，如果要拷贝必须要求 p1 也是底层const
p2 = p1; // 合法，int* 可以转换成const int*
p1 = p3; // 不合法，p3 是一个底层const，p1 不是
p2 = p3; // 合法，p2 和 p3 都是底层const，拷贝时忽略掉顶层const
```

17. 常量表达式：**在编译过程就能获得计算结果的表达式。**

```
const int max = 20; // 常量表达式
const int limit = max + 10; // 常量表达式
int staff = 10; // 非常量表达式，staff只是普通int，后续程序可能会改变它的值
const int sz = get_size(); // 非常量表达式，只有运行getsize方法是才知道他的值
```

18. c++11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，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 + 10; // mf+10是常量表达式
constexpr int sz = get_size(); // 只有当get_size是一个constexpr函数时此条正确
```

19. const和constexpr的区别：

> const修饰的是类型，constexpr修饰的是用来算出值的那段代码。

> constexpr表示这玩意儿在编译期就可以算出来（前提是为了算出它所依赖的东西也是在编译期可以算出来的）。而const只保证了运行时不直接被修改（但这个东西仍然可能是个动态变量）。

20. 一般来说，如果你认定一个变量是一个常量表达式，就把它声明成constexpr类型

21. 常量表达式要在编译过程都能获得计算结果，因此对声明constexpr时用到的类型必须是"字面值类型"（literal type）。目前接触过的类型中算数类型、引用、指针都属于字面值类型。自定义的类、IO库、string类都**不能被定义成constexpr**。

22. **constexpr只能将指针声明成const，而对该指针指向的对象无影响**。(也就是说constexpr是定义一个顶层const)，此时constexpr指针既可以指向常量又可以指向非常量。

```
const int *ptr = nullptr; // 指向常量的指针
constexpr int *ptr2 = nullptr; // 常量指针，指针本身是常量
```

## 2.5 处理类型

1. 类型别名：类型别名和名字等价。只要类型名字出现的地方别名就可以替代。

   - 传统方法：typedef

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

   - c++11新方法：using（推荐使用using）

     ```
     using SI = Sales_item; // SI是Sales_item的同义词
     ```

2. 当类型别名指代复合类型或常量时（**此处要特别注意**）：

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

   - pstr实际上是指向char的指针，因此const pstr就是指向char的**常量指针**，此处const修饰pstr这个指针，而非**指向常量字符的指针**。

3. auto类型说明符，能让编译器代替我们去分析表达式所属的类型。auto让编译器通过初始值来推算变量的类型，所以**auto定义的变量必须有初始值**。

4. auto在一条语句中声明多个变量，这些变量**只能有一个基本数据类型**。

```
auto i = 0, *p = &i; // 正确，i和指针p的基本类型都是int
auto sz = 0, pi = 3.14; // 错误，sz和pi基本类型不同
```

5. auto有时会适当改变结果来使推断的类型更符合初始化规则。

   - 使用引用实际是初始化引用对象的值，因此编译器会以引用对象的类型作为auto类型：

     ```
     int i = 0, &r = i;
     auto a = r; // 此时a的类型为int
     ```

   - auto会忽略掉顶层const，同时底层const则会保留下来，比如当初始值是一个指向常量的指针时。

     ```
     const int ci = i, &cr = ci;
     auto b = ci; // 此时b为int型（顶层const被忽略）
     auto c = cr; // 此时c也为int型
     auto d = &i; // d是一个整型指针
     auto e = &ci; // e也是一个指向整型常量的指针（对常量取地址是一种底层const）
     ```

   - 如果希望推断出的auto类型是一个顶层const，必须在前面加上const（const auto）

     ```
     const auto f = ci; // ci的推演类型是int，f是const int
     ```

   - 还可以**将引用的类型设置为auto，此时原来的初始化规则仍然使用**

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

6. decltype:选择并返回操作数的数据类型。

   ```
   decltype(f()) sum = x; // sum的类型就是函数f的返回类型
   ```

7. decltype处理顶层const和引用的方式与auto不同，decltype会返回变量的类型包括顶层const和引用在内：

   ```
   const int ci = 0, &cj = ci;
   decltype(ci) x = 0; // x为const int，此处decltype并没有忽略顶层const
   decltype(cj) y = x; // y为 const int&
   decltype(cj) z; // 错误，z是一个引用，必须初始化
   ```

8. decltype使用的表达式如果不是一个变量，那么decltype则会返回表达式结果对应的类型

```
int i = 42, *p = &i, &r = i;
int a = 1, e = 3;
decltype(r + 0) b; // b为int，r+0返回int型
decltype(*p) c; // c为int&，此处错误，必须初始化。*p是解引用操作，也就是返回引用
decltype(a = e) d; // d为int&，赋值操作是一种引用类型。此处错误，未初始化
```

9. decltype和auto的区别：
    - 处理顶层const和引用的方式不同，auto会忽略顶层const，对于引用返回引用对象类型，而decltype不会忽视，对于引用直接返回引用类型）
    - decltype直接跟上变量则直接返回变量类型，如果decltype加上一或多层括号则会将该变量视为表达式（**decltype的双括号结果永远视为引用**）

```
int i = 1;
decltype((i)) d; // 双括号，类型为int&
decltype(i) e; // 类型为int
```

## 2.6 自定义数据结构

1. 类体右侧的表示结束的花括号必须写一个分号，这是因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号**必不可少**：

```
struct Sales_data{    } accum, trans, *salesptr; // 后面直接跟要声明的对象名
//下方代码与上一条语句等价，但更好一点
struct Sales_data{    };
Sales_data accum, trans, *salesptr;
```

2. c++11新标准规定我们可以为数据成员（data memeber）提供一个类内初始值（in-class initializer）。没有初始化的值将被默认初始化（函数体外的变量默认初始化都为0，函数体内的为undefined）。
3. 为了确保各个文件中的类定义一致，类通常被定义在头文件中，而且**类所在头文件的名字应该与类的名字一样**。例如库类型string在名为string.h的头文件中定义。
4. 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。
5. 预处理器：确保头文件多次包含仍能安全工作。由c++从c语言继承而来。
    - **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**
6. 头文件保护符（header guard）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当且仅当预处理变量已定义时为真。#ifndef当且仅当预处理变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到#endif指令为止。否则跳过。

```
// 判断SALES_DATA_H是否定义，如果已定义则跳过直到#endif符号，否则执行到#endif
#ifndef SALES_DATA_H
```

```
#define SALES_DATA_H // 将SALES_DATA_H设置为预处理变量
#include <string>
struct Sales_data{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
}
#endif // !SALES_DATA_H
```

7. 整个程序中的预处理变量包括头文件保护符必须唯一。通常做法是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。**一般把预处理变量的名字全部大写**。如上述代码中的SALES_DATA_H。