

# 变量和基本类型

 C++ Primer第五版

 第2章

【C++ Primer 5th】 第二章.变量与基本类型 笔记 （数据类型  
|  
变量|指针|&|const|auto|typedef|decltype|头文件|预处理））

C++算数类型			
类型	含义	最小尺寸	说明
bool	布尔类型	未定义	取值为真（true）或假（false）
char	字符	8位	1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
wchar_t	宽字符	16位	用于扩展字符集，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	Unicode字符	16位	用于扩展字符集，为 Unicode 字符集服务
char32_t	Unicode字符	32位	用于扩展字符集，为 Unicode 字符集服务
short	短整型	16位	
int	整型	16位	一个 int 至少和 一个 short 一样大
long	长整形	32位	一个 long 至少和一个 int 一样大
long long	长长整形	64位	C++11 中新定义，一个 long long 至少和一个 long 一样大
float	浮点型	6位有效数字	
double	双精度浮点型	10位有效数字	
long double	扩展精度浮点型	10位有效数字	常常用于有特殊浮点需求的硬件

计算机以比特序列存储数据，每个比特非0即1, 可寻址的最小内存块称为“字节（byte）”，内存的基本单元称为“字（word）” 大多数机器的字节由8比特构成，字则由32或64比特构成

### 无符号类型

- int、short、long和long long都是带符号的，前面加上unsigned就可以得到无符号类型，例如unsigned long
- unsigned int可以缩写成unsigned。
- char比较特殊，类型分为三种：char、signed char、unsigned char
  - char是signed char或unsigned char的其中一种（编译器决定）

```
bool b=42;           //b为真
int i=b;             //i的值为1
i=3.14;             //i的值为3
double pi=i;         //pi的值为3.0
unsigned char c=-1;   //假设char占8比特，c的值为255
signed char c2=256;   //假设char占8比特，c2的值未定义
```

注意：char在一些机器上是有符号的，而在另一些机器上是无符号的

切勿混用带符号类型和无符号类型

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

提示：2的32次方为4294967296  
全是1表示的是-1，因为-1+1=0

字面值常量

一个型如42的值呗称为字面值常量（literal）

整形和浮点型字面值

- 可以将整形写成十进制、八进制或十六进制

```
20/*十进制*/  024/*八进制*/  0x14/*十六进制*/
```

- 浮点型字面值是一个double类型的值，表现为一个小数或科学计数法的指数形式

```
3.14159 3.14169E0 0. 0e0 .001
```

- 字符和字符串字面值

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

- 转义序列，C++定义的转义序列包括：

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		

- 泛化的转义序列，其形式是\x后紧跟1个或多个十六进制的数字，或、后面紧跟1/2或3个八进制的数字。假设使用的是Latin-1字符集，以下是一些示例：

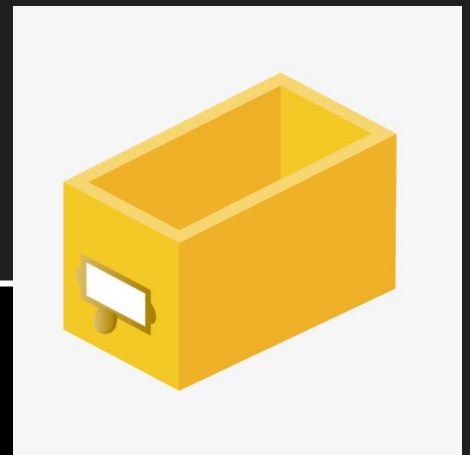
```
\7 （响铃）      \12 （换行符）      \40 （空格）
\0 （空字符）    \115 （字符M）      \x4d （字符M）
```

- 指定字面值的类型

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		

## 变量

- 变量提供一个具有名称的、可供程序操作的存储空间。
- 变量都具有数据类型
  - 通过数据类型可以决定变量所需要的内存空间、布局方式、以及能够表示值的范围。



```
int sum=0,value;//sum、value和unit_sold都是int
units_sold=0;//sum和units_sold初值为0
Sales_item item;//item的类型是Sales_item
//string 是一种库类型，表示一个可变长的字符序列
std::string book("0-2-1-78345-X");//book通过一个string字面值初始化
```

对于C++程序员来说，变量（**variable**）和对象（**object**）一般是可以互换的。

## 初始化

- 作为C++新标准的一部分，使用花括号来初始化变量得到了全面应用。

```
int units_sold = 0;
int units_sold = {0}; //列表初始化
int units_sold{0}; //列表初始化
int units_sold(0);
```

- 如果我们使用列表初始化，且初始值存在丢失信息的风险，则编译器将报错。

```
long double ld = 3.1415926536;
int a{ld},b={ld}; //错误：转换未执行，因为存在丢失信息的风险
int c(ld),d=ld; //正确：转换执行，且确实丢失了部分值
```

## 默认初始化

- 如果定义变量没有定义初始值，则变量被赋予默认值。
- 默认值是由变量类型决定的，同时定义变量的位置也会有影响。
  - 内置类型：由定义的位置决定，函数体之外初始化为0
  - 每个类各种决定其初始化对象的方式

```
std::string empty;//empty非现实地初始化一个空串
Sales_item item;//被默认初始化的Sales_item对象
```

未初始化的变量含有一个不确定的值，将带来无法预计的后果，应该避免。

变量的定义与声明

C++是一种静态类型语言，其含义是在编译阶段检查类型。这就要求我们在使用某个变量之前必须先声明。  
如果想声明一个变量而非定义它，就在变量名前添加extern关键字，而且不要显示的初始化。

```
extern double pi = 3.14; //定义,不能放在函数体内部
int main(){
    extern int i; //声明i而非定义i
    int j; //声明并定义j
}
```

变量能且只能定义一次，但可以被声明多次。

标识符

C++标识符（identifier）由字母、数字和下划线组成，其中必须以字母或下划线开头。标识符的长度没有限制，但对大小写敏感。  
C++语言保留了一些名字供语言本身使用，这些名字不能作为标识符。

C++关键字					
alignas	continue	friend	register	true	
alignof	decltype	goto	reinterpret_cast	try	
asm	default	if	return	typedef	
auto	delete	inline	short	typeid	
bool	do	int	signed	typename	
break	double	long	sizeof	union	
case	dynamic_cast	mutable	static	unsigned	
catch	else	namespace	static_assert	using	
char	enum	new	static_cast	virtual	
char16_t	explicit	noexcept	struct	void	
char32_t	export	nullptr	switch	volatile	
class	extern	operator	template	wchar_t	
const	false	private	this	while	
constexpr	float	protected	thread_local		
const_cast	for	public	throw		

C++操作符替代名					
and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	



## 名字的作用域 ( scope )

同一个名字如果出现在程序的不同位置，也可能指向不同的实体。

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语句，以声明语句所在的作用域末端为结束。

```
#include <iostream>

int main(){
    int sum = 0; //sum用于存放1到10所有数的和
    for(int val = 1; val<=10; ++val)
        sum+=val; //等价于sum=sum+val

    std::cout<<"sum of 1 to 10 inclusive is "<<sum<<std::endl;
    return 0;
}
```

全局作用域

块作用域

块作用域

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。

```
#include <iostream>

// 该程序是一个不好的示例，仅用于展示作用域
int reused = 42; // reused 拥有全局作用域

int main()
{
    int unique = 0; // unique 拥有块作用域

    // output #1: 42 0
    std::cout << reused << " " << unique << std::endl;

    int reused = 0; // 同名的新建局部变量，覆盖了全局变量

    // output #2: 0 0
    std::cout << reused << " " << unique << std::endl;

    // output #3: 显式地访问全局变量，打印 42 0
    std::cout << ::reused << " " << unique << std::endl;

    return 0;
}
```

## 复合类型 ( compound type )

是指基于其他类型定义的类型。

C++11中新增了“右值引用”；当我们使用术语“引用”时，一般指的是“左值引用”

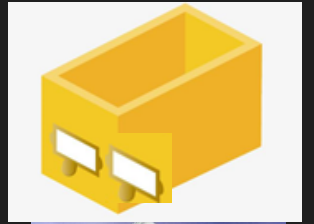
**引用 ( reference )** 为对象起的另一个名字

定义引用时，程序把引用和它的初始值绑定在一起，而不是将初始值拷贝给引用  
引用本身并不是对象，所以不能定义引用的引用

```
int ival = 1024;
int &refVal = ival; // refVal指向ival(是ival的另一个名字)
int &refVal2; // 报错：引用必须初始化
```

```
refVal = 2; // 把2给refVal指向的对象，此处即是赋给了ival
int li = refVal; // 等同于li=ival
```

```
// 正确：refVal13绑定到了那个与refVal绑定的对象上，即绑定了ival
int &refVal3 = refVal;
// 利用与refVal绑定的对象的值初始化变量i
int i = refVal; // 正确：i被初始化为ival的值
```



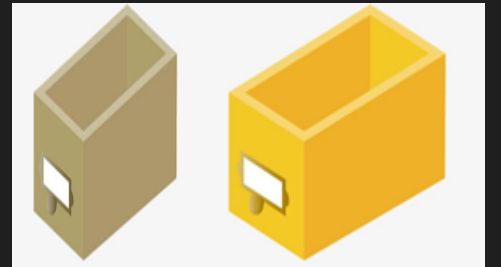
名：赵云  
字：子龙

**指针 ( pointer )** 对地址的封装，本身就是一个对象

- 定义指针类型的方法是将声明符写成\*d的形式
- 如果一条语句中定义了几个指针变量，每个变量前面都必须加上\*符号
- 和其他内置类型一样，在块作用域内定义指针如果没有初始化，将拥有一个不确定的值

引用不是对象，不存在地址，所以不能定义指向引用的指针。

```
int *ip1, *ip2; // ip1和ip2都是指向int型对象的指针
double dp, *dp2;
```



- 可以使用**取地址符**（操作符&）获取指针所封装的地址：

```
int ival = 42;
int *p = &ival; // p是指向ival的指针
double *dp = &ival; // 错误：类型不匹配
```

在声明中，&和\*用于组成复合类型；在表达式中，他们是运算符。含义截然不同。

- 可以使用**解引用符**（操作符\*）利用指针访问对象：

```
int ival = 42;
int *p = &ival; // p是指向ival的指针
std::cout << *p; // 输出42
*p = 0;
std::cout << *p; // 输出0
```



## 空指针 ( null pointer ) 不指向任何对象

在使用一个指针之前，可以首先检查它是否为空。

```
int *p1 = nullptr; //C++11
int *p2 = 0;
int *p3 = NULL; //需要#include cstdlib
```

```
int zero = 0;
p1 = zero; //错误：类型不匹配
```

## void \*指针

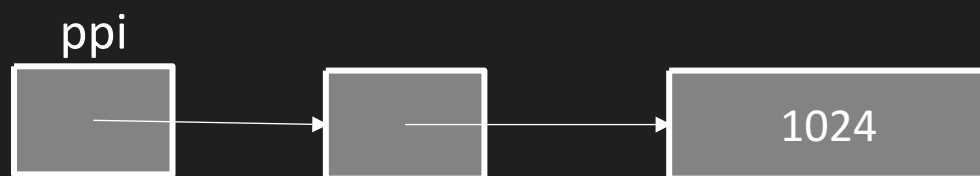
纯粹的地址封装，与类型无关。可以用于存放任意对象的地址。

```
double obj = 3.14, *pd = &obj;
void *pv = &obj;
pv = pd;
```

## 指向指针的指针

通过\*的个数可以区分指针的级别。

```
int ival = 1024;
int *pi = &ival;
int **ppi = &pi; //ppi指向一个int型的指针
```



## 指向指针的引用

指针是对象，可以定义引用。

```
int i = 1024;
int *p;
int *&r = p; //r是一个对指针p的引用

r = &i; //r引用了一个指针，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的值改为0
```

## const限定符：把变量定义成一个常量

使用const对变量的类型加以限定，变量的值不能被改变。

```
const int bufSize = 512; //输入缓冲区大小
bufSize = 512; //错误：试图向const对象写值
```

const对象必须初始化（其他时候不能出现在等号左边）。

```
const int i = get_size(); //正确：运行时初始化
const int j = 42; //正确：编译时初始化
const int k; //错误：k是一个未经初始化的常量
```

```
const int bb=0;
void * a= bb;//在编译的时候，会把bb编程常量
```

默认状态下，const对象仅在文件内有效  
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

```
//file_1.cc定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
//file_1.h头文件
extern const int bufSize;
```

## const的引用：对常量的引用

```
const int ci = 1024;
const int &r1 = ci; //正确：引用及其绑定的对象都是常量

r1 = 42; //错误，相当于c1=42，试图修改常量
int &r2 = ci; //错误：ci是常量，存在通过r2改变ci（const）的风险
```

```
int i = 42;
const int &r1 = i; //正确：i依然可以通过其他途径修改
const int &r2 = 42;
const int &r3 = r1*2;
int &r4 = r1*2; //错误：不能通过一个非常量的引用指向一个常量
```

## 指针和const

- 指向常量的指针

```
const double pi = 3.14;
double *ptr = &pi; //错误：存在通过ptr指针修改pi的风险
const double * cptr = &pi;
*cptr = 42; //错误

double dval = 3.14;
cptr = &dval; //正确：但不能通过cptr修改dval的值
```

- const指针：指针是对象，也可以限定为常量（必须初始化）
  - 把\*放在const之前，说明指针是一个常量
  - 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;
const double pi = 3.14159;
const double *const pip = &pi; //指向常量的常量指针

*pip = 2.71; //错误： 试图修改常量pi

if(*curErr){
    errorHandler();
    *curErr = 0; //正确：试图修改变量errNumb
}
```

const 指针的  
用法类似引用

## 顶层const

- 顶层const：表示变量本身是一个常量
- 底层const：表示指针所指向的对象是一个const

```
int i = 0;
int *const p1 = &i; //顶层
const int ci = 42; //顶层
const int *p2 = &ci; //底层
const int *const p3 = p2; //（左：底层），（右：顶层）

i = ci; //正确
p2 = p3; //正确

int *p = p3; //错误：存在通过*p修改*p3（const）的风险
p2 = &i; //正确：只是不能通过p2修改i而已
int &r = ci; //错误：存在通过r修改ci（const）的风险
const int &r2 = i; //正确：只是不能通过r2修改i而已
```

## constexpr和常量表达式

- 常量表达式（const expression）是指：值不会改变并且在编译过程就能得到计算结果的表达式。

```
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

- C++11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。

- 一定是一个常量
- 必须用常量表达式初始化

需要在编译时就得到计算，声明constexpr时用到的类型必须显而易见，容易得到（称为：字面值类型）。

```
constexpr int mf = 20;
constexpr int limit = mf + 1;
constexpr int sz = size(); //只有当size是一个constexpr函数时才正确
```

自定义类型（例如：Sales\_item）、IO库、string等类型不能被定义为constexpr

- 指针和constexpr
  - 限定符仅对指针有效，对指针所指的对象无关

```
constexpr int *np = nullptr; //常量指针
int j = 0;
constexpr int i = 42;

constexpr const int *p = &i; //p是常量指针，指向常量
constexpr int *p1 = &j; //p1是常量指针，指向变量j
```

## 处理类型：

随着程序越来越复杂，程序中的变量也越来越复杂。

- 拼写变得越来越困难。
- 搞不清楚变量到底需要什么类型。

## 类型别名：提高可读性

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

using SI = Sales_item; //C++11,别名声明
wages hourly, weekly;
SI item; //等价于Sales_item item
```

对于指针这样的复合类型，类型别名的使用可能会产生意想不到的结果：

```
typedef char *pstring;
const pstring cstr = 0; //指向char的常量指针
const pstring *ps; //ps是指针变量，它的对象是指向char的常量指针

const char *cstr = 0; //是对const pstring cstr = 0 的错误理解
```

## auto类型说明符：C++11，让编译器通过初始值推断变量的类型

```
auto item = val1 + val2;

auto i = 0, *p = &i; //正确
auto sz = 0, pi = 3.14; //错误：auto已经被推断为int型，却需要被推断为double型
```

```
int i = 0, &r = i;
auto a = r; //a是int型

const int ci = i, &cr = ci;
auto b = ci; //b是int型，ci的顶层const被忽略
auto c = cr; //c是int型，ci的顶层const被忽略
auto d = &i; //d是整形指针，整数的地址就是指向整形的指针
auto e = &ci; //e是指向整数常量的指针（底层const没有被忽略）

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


auto &g = ci; //g是一个整形常量引用，绑定到ci，（底层const没有被忽略）

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

auto k = ci, &l = i;
auto &m = ci, *p = &ci;
auto &n = i, *p2 = &ci; //错误：类型不一致
```



## decltype类型说明符：选择并返回操作数的数据类型 只要数据类型，不要其值

@阿西拜-南昌 

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

```
const int ci = 0, &cj = ci;
```

```
decltype(ci) x = 0; // x的类型是const int
```

```
decltype(cj) y = x; // y的类型是const int &
```

```
decltype(cj) z; //错误：z是一个引用，必须初始化
```

引用从来都是作为其所指对象的同义词出现，只有用在decltype处是一个例外

```
int i = 42, *p = &i, &r = i;
```

```
decltype(r+0) b; //正确：b为int型
```

//注意：下面的\*不是出现在声明中，而是表达式中的解引用运算符

```
decltype(*p) c; //错误：解引用表达式，c的类型为引用，需要初始化
```

//变量如果是加上括号，decltype的结果将是引用

```
decltype( (i) ) d; //错误：d是int&类型,必须初始化
```

```
decltype( ( (i) ) ) d1 = i; //正确：d1是int&类型，绑定为了i
```

```
decltype(i) e; //正确：e是一个（未初始化的）int
```

如果表达式的内容是解引用操作，则decltype将得到引用类型

decltype( (variable) )的结果永远为引用，variable本身也可以是一个引用

**类定义：**类定义可以使用关键字class或struct

- 二者默认的继承访问权限不同
- struct是public的，class是private的

```
struct Sales_data{
    std::string bookNo;
    unsigned units_sold = 0; //C++ 11
    double revenue = 0.0;
};//类定义的最后需要加上分号
```

数据成员定义了类的对象的具体内容，每个对象有自己的一份拷贝。

**类使用：**

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main(){
    Sales_data data1, data2;
    double price = 0; // 书的单价, 用于计算总收入

    // 读取第1条交易记录: ISBN, number of books sold, price per book
    std::cin >> data1.bookNo >> data1.units_sold >> price;
    // calculate total revenue from price and units_sold
    data1.revenue = data1.units_sold * price;

    // 读取第2条交易记录:
    std::cin >> data2.bookNo >> data2.units_sold >> price;
    data2.revenue = data2.units_sold * price;

    if (data1.bookNo == data2.bookNo) {
        unsigned totalCnt = data1.units_sold + data2.units_sold;
        double totalRevenue = data1.revenue + data2.revenue;

        // print: ISBN, total sold, total revenue, average price per book
        std::cout << data1.bookNo << " " << totalCnt << " " << totalRevenue << " ";
        if (totalCnt != 0) std::cout << totalRevenue/totalCnt << std::endl;
        else std::cout << "(no sales)" << std::endl;

        return 0; // indicate success
    } else { // transactions weren't for the same ISBN
        std::cerr << "Data must refer to the same ISBN" << std::endl;
        return -1; // indicate failure
    }
}
```

C:\Windows\system32\cmd.exe

```
1-1 5 5
1-1 10 4.5
1-1 15 70 4.66667
```

## 编写自己的头文件：类通常定义在头文件中

Sales\_data.h

```
#ifndef SALES_DATA_H  
#define SALES_DATA_H
```

```
#include <string>
```

```
struct Sales_data {  
    std::string bookNo;  
    unsigned units_sold = 0;  
    double revenue = 0.0;  
};
```

```
#endif
```

预处理器（preprocessor），  
在编译之前执行的代码。  
这里的作用是：头文件保护