面向对象编程基础

本课程入选教育部产学合作协同育人项目
课程主页:**http://cpp.njuer.org**
课程老师:陈明 **http://cv.mchen.org**

```
ppt和代码下载地址
git clone https://gitee.com/cpp-njuer-org/book
```

第2章

变量和基本类型

数据类型是程序的基础

- 它告诉我们数据的意义
- 以及我们能在数据上执行的操作

C++支持广泛的数据类型

- 基本内置数据类型
    - 字符 整形 浮点数等
- 自定义数据类型
    - 标准库定义了一些更加复杂的数据类型
        - 如可变长字符串和向量等

数据类型决定了程序中数据和操作的意义

```
i = i + j;
```

- 如果i和j都是整型数，这条语句就是普通加法运算。
- 如果i和j都是上一章的Sales_item类型的数据，则这条语句把两个对象的成分相加。

基本内置变量

- 算数类型
    - 包含字符 、整型数 、 布尔值 、浮点数
- 空类型
    - 不对应具体的值,仅用于特殊场合
        - 如 函数不返回任何值时，使用空类型做返回值。

C++算数类型

类型	含义	最小尺寸
bool	布尔类型	未定义
char	字符	8bits
wchar_t	宽字符	16bits
char16_t	Unicode字符	16bits
char32_t	Unicode字符	32bits
short	短整型	16bits
int	整型	16bits
long	长整型	32bits
long long	长整型	64bits
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

```cpp
//testSize.cpp
//1Byte=8bit
#include<iostream>
using namespace std;
int main(){
    cout<< "int * "<<sizeof(int*)*8<<endl;
    cout<< "bool "<<sizeof(bool)*8<<endl;
    cout<< "char "<<sizeof(char)*8<<endl;
    cout<< "wchar_t "<<sizeof(wchar_t)*8<<endl;
    cout<< "char16_t "<<sizeof(char16_t )*8<<endl;
    cout<< "char32_t "<<sizeof(char32_t )*8<<endl;
    cout<< "short "<<sizeof(short )*8<<endl;
    cout<< "int "<<sizeof(int )*8<<endl;
    cout<< "long "<<sizeof(long )*8<<endl;
    cout<< "long long"<<sizeof(long long)*8<<endl;
    cout<< "float "<<sizeof(float)*8<<endl;
    cout<< "double "<<sizeof(double)*8<<endl;
    cout<< "long double "<<sizeof(long double)*8<<endl;

    return 0;
}
```

```
//根据环境差异 结果可能不同
int * 64
bool 8
char 8
wchar_t 32
char16_t 16
char32_t 32
short 16
int 32
long 64
long long64
float 32
double 64
long double 128
```

内置类型的机器实现

字节Byte
- 可寻址的最小内存块
- 1 Byte = 8 bit（比特）

字 Word
- 存储的基本单元
- 1 Word = 4字节或8字节

内存中每个字节与一个数字（被称为地址address）关联。
我们能够使用某个地址来表示从这个地址开始的大小不同的比特串。
- 如地址736424的那个字 或者地址736424的那个字节
- 必须知道存储在某地址的数据类型，才能赋予内存该地址明确含义
- 类型决定数据所占比特数，以及如何解释这些比特的内容

带符号类型和无符号类型

其它整型
- 除去布尔型和扩展字符型外，其它整型可划分为带符号的和无符号的
- int, short, long, long long 带符号
- 类型名前添加unsigned 得到无符号类型
  - 如unsigned long,
  - unsinged int(可简写为 unsigned)

字符型
- 分三种 char, signed char, unsigned char
- char 表现为 带符号的和无符号的，由编译器决定

如何选择类型

- 1.数值不可能是负数时，选用无符号类型；
- 2.使用int执行整数运算。
    - 一般long的大小和int一样，而short常常显得太小。
    - 超过了int的范围，选择long long。
- 3.算术表达式中不要使用char或bool。
    - 符号容易出问题
- 4.浮点运算选用double。

类型转换

类型所能表示值的范围决定转换过程：

- 非布尔型赋给布尔型，初始值为0则结果为false，否则为true。
- 布尔型赋给非布尔型，初始值为false结果为0，初始值为true结果为1。
- 浮点数赋给整型，近似处理，保留小数点前的部分。
- 整型赋给浮点型，小数部分记0，若整型超过浮点型容量则精度可能有损失。
- 超出范围值赋给无符号数，结果是初始值对无符号类型表示数值总数取模余数。
- 超出范围值赋给有符号数，结果未定义。

```
//使用算数类型的值，而需要的是其它类型的值，就会执行类型转换
//testV.cpp
int i=42;
if (i)
  i=0;
```

```
//含有无符号类型的表达式
//testV2.cpp
#include<iostream>

int main(){
    unsigned  u =10;
    int i=-42;
    std::cout<<i+i<<std::endl; // -84
    std::cout<<u+i<<std::endl; // ?4294967264

    return 0;
}
```

```cpp
//testV3.cpp
#include<iostream>

int main(){
    for(int i=10;i>=0;--i){
        std::cout<<i<<std::endl;
    }
    return 0;
}
```

```cpp
//testV4.cpp
//无符号数不会小于0
//死循环 按ctrl+c退出
#include<iostream>

int main(){
    for(unsigned u=10;u>=0;--u){
        std::cout<<u<<std::endl;
    }
    return 0;
}
```

切勿混用带符号和无符号类型

练习

```cpp
//q2_3.cpp
#include<iostream>

int main(){
    unsigned u = 10, u2 = 42;
    std::cout << u2 - u << std::endl;
    std::cout << u - u2 << std::endl;
    int i = 10, i2 = 42;
    std::cout << i2 - i << std::endl;
    std::cout << i - i2 << std::endl;
    std::cout << i - u << std::endl;
    std::cout << u - i << std::endl;

    return 0;
}
```

```
/*
32
4294967264
32
-32
0
0
*/
```

字面值常量(LITERAL)

- 整型字面值。
    - 整型字面值 10进制
    - 0开头 8进制
    - 0x或0X开头 16进制
- 浮点型字面值
    - 一个小数 或科学计数法
    - 3.14 3.14E0 0. 0e0 .001

字面值常量(LITERAL)

- 字符和字符串字面值。
    - 字符字面值：单引号， 'a'
    - 字符串字面值：双引号， "Hello World"
        - 实际类型是字符数组，结尾添加'\0'字符
        - 字符串型实际上时常量字符构成的数组，结尾处以’ \0’ 结束，所以字符串类型实际上长度比内容多1。
    - 分多行书写字符串。由空格 换行 缩进符分隔

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

字面值常量(LITERAL)

转义序列

```
换行\n 横向制表符\t 响铃\a
纵向制表符\v 退格符\b 双引号\"
反斜线\\ 问号\? 单引号\'
回车\r 进纸\f
## 泛化转义序列，\x跟1个多个16进制数，或\后跟1、2、3个8进制数
\7响铃 \12换行 \40空格
\0空字符 \115字符M \x4d字符M
```

```cpp
//testV5.cpp
#include<iostream>
int main(){
    std::cout<< '\n';//换行
    std::cout<< "\tHi!\n";//制表符Hi! 换行
    std::cout<< "\v\?\abc\b\n";//\v纵向制表符，响铃，?b ，退格,换行
    std::cout<< "Hi \x4dO\115!\n";//Hi MOM!
    std::cout<< '\115'<<'\n';//M,换行
    return 0;
}
```

指定字面值类型

```
# 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(字符串字面常量) 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

- 布尔字面值。`true`，`false`。
- 指针字面值。`nullptr`

练习

下面两组定义是否有区别，如果有，请叙述之：

```
//testV5.cpp
#include<iostream>
int main(){
    int month = 9, day = 7;
    int month1 = 09, day1 = 07;
    return 0;
}
```

练习

下面两组定义是否有区别，如果有，请叙述之：

```
//testV5.cpp
#include<iostream>
int main(){
    int month = 9, day = 7;
    int month1 = 09, day1 = 07;
    //error: invalid digit "9" in octal constant
    return 0;
}
```

变量

变量提供一个**具名**的、可供程序操作的存储空间。 C++中**变量**和**对象**一般可以互换使用。

变量定义（DEFINE）

- 定义形式：类型说明符（type specifier） + 一个或多个变量名组成的列表;
- 初始化（initialize）：对象在创建时获得了一个特定的值。
  - 初始化不是赋值
    - 初始化 = 创建变量 + 赋予初始值
    - 赋值 = 擦除对象的当前值 + 用新值代替
  - 列表初始化：使用花括号{}
    - 若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。
    - 如 `int a{3.14};// error: narrowing conversion from ‘double’ to ‘int’`
  - 默认初始化：定义时没有指定初始值会被默认初始化；
    - 在函数体内部的内置类型变量将不会被初始化。
    - 类的对象如果没有显示初始化，其值由类定义。

练习

下列变量的初值分别是什么？

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

练习

下列变量的初值分别是什么?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

global_str和global_int是全局变量，所以初值分别为空字符串和0。
local_int是局部变量并且没有初始化，它的初值是未定义的。
local_str 是 string类的对象，它的值由类确定，为空字符串。

未初始化变量引发运行时故障。

- 建议初始化每一个内置类型的变量。

变量声明和定义的关系

- 把程序拆分成多个逻辑部分来编写
  - C++支持分离式编译机制
  - 程序分割为若干文件，每个文件可被独立编译
- 为了支持分离式编译，C++将声明和定义区分开。
  - 声明使得名字为程序所知。定义负责创建与名字关联的实体。
  - 声明规定了变量类型和名字。定义申请存储空间，也可能为变量赋初值。
- 想声明一个变量而非定义它，在变量名前加关键字extern,不要显示初始化变量。
  - `extern int i;//声明i而非定义`
  - `int j;//声明并定义j`
  - 包含了显示初始化的声明，就变成了定义：
    - `extern double pi = 3.14;//定义`
    - 函数体内部，试图初始化一个extern关键字标记的变量，将引发错误。

变量声明和定义的关系

- 变量只能被定义一次，但是可以多次声明。
  - 定义只出现在一个文件中，其他文件使用该变量时需要对其声明。

练习

指出下面的语句是声明还是定义：

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
```

练习

指出下面的语句是声明还是定义：

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
```

```
(a): 定义
(b): 定义
(c): 声明
```

标识符

由字母、数字、下划线组成，必须以字母或下划线开头。

- 长度没有限制。
- 大小写敏感。

保留名字不能用做标识符。

用户自定义标识符

- 不能连续出现两个下划线
- 不能以下划线紧跟大写字母开头
- 定义在函数体外的标识符不能以下划线开头

```
//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
char16_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
```

变量命名规范

- 体现实际含义
- 变量名一般用小写字母
  - 如index，不要用Index或INDEX
- 自定义类名大写字母开头
  - 如Sales_item
- 若标识符有多个单词组成，单词间应有明显区分
  - 如student_loan,studentLoan,不要用studentloan

练习

请指出下面的名字中哪些是非法的？

```
(a) int double = 3.14;
(b) int _;
(c) int catch-22;
(d) int 1_or_2 = 1;
(e) double Double = 3.14;
```

(a), (c), (d) 非法。

- 名字的作用域（namescope）。以{}分隔。
    - 全局作用域。块作用域。
    - **第一次使用变量时再定义它。**
        - 更容易找到。赋予比较合理的初始值。

```cpp
//scope.cpp
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42;  // reused has global scope
int main()
{
    int unique = 0; // unique has block scope
    // output #1: uses global reused; prints 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << " " <<  unique << std::endl;
    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << " " <<  unique << std::endl;
    return 0;
}
```

- 嵌套的作用域
    - 同时存在全局和局部变量时，已定义局部变量的作用域中可用::显式访问全局变量。
    - **用到全局变量时，尽量不使用重名的局部变量。**

练习

```
//下面程序中j的值是多少？
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
//j的值是100，局部变量i覆盖了全局变量i

//下面的程序合法吗？如果合法，它将输出什么？
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
//合法。输出 100 45 。
```

复合类型

基于其它类型定义的类型

- c++语言有几种复核类型
- 这里介绍两种: 引用和指针

左值和右值

- **左值**（l-value）**可以**出现在赋值语句的左边或者右边，比如变量；
- **右值**（r-value）**只能**出现在赋值语句的右边，比如常量。

引用

一般说的引用是指的左值引用

- **引用**：引用是为对象起了另外一个名字，引用类型引用（refer to）另外一种类型。
    - `int &refVal = val;`
- 引用必须初始化。
    - `int &refVal2; //报错，引用必须被初始化`
- 引用和它的初始值是**绑定bind**在一起的，而**不是拷贝**。一旦定义就不能更改绑定为其他的对象
- 引用类型与绑定对象匹配。
- 引用只能绑定在对象上，不能与字面值或表达式计算结果绑定。

练习

```
//下面的哪个定义是不合法的？为什么？
(a) int ival = 1.01;
(b) int &rval1 = 1.01;
(c) int &rval2 = ival;
(d) int &rval3;
```

(b)和(d)不合法，(b)引用必须绑定在对象上，(d)引用必须初始化。

指针

- 是一种"指向（point to）"另外一种类型的复合类型。
    - 本身就是一个对象
    - 无需定义时赋值

定义指针类型

声明符写成*d的形式，d是变量。

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

获取对象的地址

- 指针存放某个对象的**地址。**
- 获取对象的地址： `int i=42; int *p = &i;`
    - &是**取地址符。**
- 指针的类型与所指向的对象类型必须一致（均为同一类型int、double等）

获取对象的地址

```
int ival = 42;
int *p  = &ival; //p存放ival的地址，p是指向val的指针

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，试图把double对象的地址赋给int指针
```

- 指针的值(即地址)的四种状态：
    - 1.指向一个对象；
    - 2.指向紧邻对象的下一个位置；
    - 3.空指针；
    - 4.无效指针。

> 对无效指针的操作均会引发错误；
> 第二种和第三种虽为有效的，访问指针对象的行为后果无法预计。

指针访问对象：

```
//如果指针指向一个对象，使用解应用符（操作符*）来访问对象
int ival = 42;
int *p = &val;
cout << *p;// 输出p指针所指对象的数据，  *是解引用符。

*p = 0;
cout <<*p;// 0
```

解引用操作仅适用于确实指向某个对象的有效指针。

空指针

空指针不指向任何对象。

```
int *p1=nullptr;//使用空指针。
int *p2 = 0;//p2 初始化为字面常量0
// include<cstdlib>
int *p3 = NULL;//int *p3=0. NULL预处理变量

int zero = 0;
p1 = zero; //错误，不能把int变量直接赋给指针
```

建议初始化所有指针。

指针和引用的区别：

- 引用本身并非一个对象，引用定义后就不能绑定到其他的对象了；
- 指针并没有此限制

```
int i = 42;
int *pi = 0;//pi 初始化，没有指向任何对象
int *pi2 = &i;//pi2指向i
int *pi3;//如果pi3定于于块内，pi3值无法确定

pi3=pi2;//pi3与pi2指向同一对象i
pi2=0;//pi2 不指向任何对象

//赋值语句永远改变的是左侧的对象。
pi = &ival;//pi指向了ival
*pi = 0;//ival值改变，pi没改变
```

赋值语句永远改变的是**左侧**的对象。

其它指针操作

```
int ival = 1024;
int *p1=0;
int *p2=&ival;
//指针值0，条件取false
if(p1)
    //...
//p2指针非0，条件值是true
if(p2)
    //...
//指针比较
if(p1==p2)
    //...
```

非法指针会引发不可预计的后果。

VOID指针

可以存放任意对象的地址。因无类型，仅操作内存空间，不能直接操作void * 指针所指的对象。

```
double obj =  3.14, *pd = &obj;
void *pv = &obj; //obj 可以是任意类型的对象
pv = pd;//void* 可存任何类型
```

练习

说明指针和引用的主要区别

引用是另一个对象的别名，而指针本身就是一个对象。
引用必须初始化，并且一旦定义了引用就无法再绑定到其他对象。
而指针无须在定义时赋初值，也可以重新赋值让其指向其他对象。

请解释下述定义。在这些定义中有非法的吗？如果有，为什么？

```
int i = 0;
(a) double* dp = &i;
(b) int *ip = i;
(c) int *p = &i;
```

(a): 非法。不能将一个指向 `double` 的指针指向 `int` 。
(b): 非法。不能将 `int` 变量赋给指针。
(c): 合法。k

理解复合类型的声明

```
//一条语句定义不同类型变量
int i=1024,*p=&i,&r=i;

int* p;//合法但容易产生误导
int* p1,p2;//p1是指针 p2是int

int *p1,*p2;//p1,p2都是指针
```

或

```
int *p1;
int *p2;
```

理解复合类型的声明

指向指针的指针

```
//ppi.cpp
int ival =1024;
int *pi =&ival;//pi指向int型
int **ppi = &pi;//ppi指向int型指针
// 解引用
cout << "The value of ival\n"
    << "direct value: " << ival << "\n"
    << "indirect value: " << *pi << "\n"
    << "doubly indirect value: " << **ppi
    << endl;

/*
The value of ival
direct value: 1024
indirect value: 1024
doubly indirect value: 1024
*/
```

指向指针的引用

从右向左阅读r的定义

```
int i=42;
int *p;
int *&r = p;//r是对指针p的引用

r = &i;//p指向i
*r=0;//i=0
```

练习

说明下列变量的类型和值。

```
(a) int* ip, i, &r = i;
(b) int i, *ip = 0;
(c) int* ip, ip2;
```

(a): ip 是一个指向 int 的指针，i 是一个 int，r 是 i 的引用。
(b): i 是 int ，ip 是一个空指针。
(c): ip 是一个指向 int 的指针，ip2 是一个 int。

CONST限定符

const：定义一些不能被改变值的变量。
- const对象一旦创建值不再改变，所以必须初始化，且不能被改变。

```
const int bufSize = 512;
bufSize = 512;//error，试图写值到const对象

const int i= getSize();
const int j=42;
const int k;//error,k未初始化常量

int i=42;
const int ci=i;
int j=ci;
```

CONST对象仅在文件内有效

- 当多个文件出现同名的const变量，等同于在不同文件分别定义独立变量。
- 要想在多个文件中使用const变量共享，定义和声明都加extern关键字即可。

```
//file_1.cc定义初始化常量，能被其它文件访问
extern const  int bufSize = fcn();
//file_1.h 头文件
extern const  int bufSize ;//与file_1.cc中定义的bufSize是同一个
```

练习

下面哪些语句是合法的？如果不合法，请说明为什么？

```
const int buf;       // 不合法，const 对象必须初始化
int cnt = 0;         // 合法
const int sz = cnt; // 合法
++cnt; ++sz;         // 不合法，const 对象不能被改变
```

CONST的引用

- 把引用绑定到const对象上，称之为对常量的引用。
- 与普通引用不同，对常量的引用不能被用作修改它绑定的对象。

```
const int ci=1024;
const int &r1=ci;
r1=42;//error r1是对常量的引用
int &r2 = ci;//error,试图让非常量引用指向一个常量对象
```

初始化和对CONST的引用

引用类型必须与其所用对象类型一致

- 例外：初始化常量引用时，允许用任意表达式做初始值，只要表达式的结果能转换成引用的类型。
- 允许为一个常量引用绑定非常量的对象、字面值、一个一般表达式。

```
int i=42;
const int &r1=i;//允许将const int&绑定到int对象
const int &r2=42;//r2 是个常量引用
const int &r3=r1*2;//r3是个常量引用
int &r4=r1*2;//error，r4是一个普通的非常量应用。
```

```
//当一个常量引用被绑定到另外一种类型发生了什么
double dval = 3.14;
const int &ri=dval;
//编译器将上述代码改为如下形式
const int temp=dval;//生成临时整型变量
const int &ri=temp;//让ri绑定这个临时变量
```

- 临时量（temporary）对象：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。
- 对临时量的引用是非法行为。

```
    double dval = 3.14;
    int &ri=dval;//error
```

对CONST的引用可能引用一个非CONST对象

- 常量引用仅对引用可参与的操作作出限定，对引用的对象本身是不是常量未做限定。

```
int i=42;
int &r1=i;//r1 绑定i
const int &r2 = i;//r2绑定i；不允许通过r2修改i
r1=0;//ok r1非常量
r2=0;//error r2是一个常量引用
```

和常量引用一样，指向常量的指针同理也没有规定所指对象必须是常量。仅要求不能通过指针改变对象值。

CONST 指针

常量指针必须初始化,允许把指针定义成常量.

```
int errNumb=0;
int *const curErr=&errNumb;// curErr是常量，一直指向errNumb
const double pi=3.1415;
const double *const pip= &pi;//pip是常量，*pip也是常量，
                             //pip 是指向常量对象的常量指针
*pip = 2.72;//error *pip 是常量
if(*curErr){
    errorHandler();
    *curErr=0;//ok *curErr不是常量
}
```

练习

```
//下面的哪些初始化是合法的？请说明原因。
int i = -1, &r = 0;          // 不合法，r 必须引用一个对象
int *const p2 = &i2;         // 合法，常量指针
const int i = -1, &r = 0;    // 合法
const int *const p3 = &i2;   // 合法
const int *p1 = &i2;         // 合法
const int &const r2;         // 不合法，r2 是引用，没有顶层const
const int i2 = i, &r = i;    // 合法
```

练习

```
//说明下面的这些定义是什么意思，挑出其中不合法的。
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
```

练习

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;    // 合法，对 p3 取值后是一个 int 然后赋值给 ic
```

顶层CONST

- 顶层const：指针本身是个常量。
- 底层const：指针指向的对象是个常量。拷贝时严格要求相同的底层const资格。

```
int i=0;
int *const p1=&i;//p1常量，顶层const
const int ci=42;//ci常量，顶层const
const int *p2=&ci;//*p2常量，底层const
const int *const p3=p2;//靠右顶层，靠左底层const
const int &r =ci;//用于声明应用的const都是底层const


//拷贝时，顶层const不受影响
i=ci;//拷贝ci的值，ci顶层const，无影响
p2=p3;//p2 p3所指对象类型相同 p3顶层const部分不影响

//拷贝时严格要求相同的底层const资格,数据类型能够转换。
int *p =p3;//error p3包含底层const定义，p没有
p2=p3;//ok p2 p3都是底层const
p2=&i; //ok int* 转const int*
int &r=ci;//error 普通int& 不能绑定到int 常量
const int &r2=i;//ok const int& 可以绑定到int
```

练习

```
//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。

```
//假设已有上一个练习中所做的那些声明，则下面的哪些语句是合法的？
//请说明顶层const和底层const在每个例子中有何体现。
r1 = v2; // 合法，顶层const在拷贝时不受影响
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
```

constexpr和常量表达式

- 常量表达式：指值不会改变，且在编译过程中就能得到计算结果的表达式。

```
//一个对象（表达式）是不是常量表达式由数据类型和初始值共同决定
const int max_file=30;//max_file是常量表达式
const int limit = max_file+1;//limit是常量表达式
int staff_size = 7;//staff_size不是常量表达式
const int sz = get_size();//运行时获取，sz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的表达式。

```
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 = get_size();//取决于get_size()是constexpra函数否
```

字面值类型

常量表达式的值编译时就得到计算，类型简单，值容易得到，称为为“字面值类型”（literal type）

- 算数类型
- 引用 指针 nullptr 0或固定位置

指针和CONSTEXPR

- constexpr把所定义的对象置为顶层

```
const int *p=nullptr;//p是一个指向常量的指针
constexpr int *q=nullptr;//q是常量指针

//constexpr把所定义的对象置为顶层
constexpr int *nq=nullptr;//nq是常量指针
int j=0;
constexpr int i=42;//i是常量
//i,j定义在函数外
constexpr const int *p=&i;//p是常量指针,指向整型常量i
constexpr int *p1=&j;//p1是常量指针，指向整数j
```

处理类型

- 程序越来越复杂，类型越来越复杂。类型难于拼写，明确目的含义，搞不清需要什么类型。

类型别名

- 传统别名：使用**typedef**来定义类型的同义词。 `typedef double wages;`
- 新标准别名：别名声明（alias declaration）： `using SI = Sales_item;` (C++11)

```
wages hour,weekly;//double hour,weekly;
SI items;//Sales_item items;
```

指针、常量和类型别名

```
// 对于复合类型（指针等）不能代回原式来进行理解
typedef char *pstring;  // pstring是char*的别名
const pstring cstr = 0; // cstr是指向char的常量指针
const pstring *ps;//ps是一个指针，对象是指向char的常量指针
//基本数据类型是指针而不是const char。
```

AUTO类型说明符 C++11

清楚的知道表达式类型并不容易，因此引入auto

- auto类型说明符：让编译器自动推断类型。显然auto定义变量必须有初始值
- 一条声明语句只能有一个基本数据类型
    - `auto sz = 0, pi =3.14; //错误,类型不一致`

复合类型、常量和AUTO

```
- int i = 0, &r = i; auto a = r; 推断a类型是int
- 会忽略顶层const; const int ci=i;auto b=ci;//推断int，忽略顶层const
- const int ci = 1; const auto f = ci;
//推断类型是int，如果希望是顶层const需要自己加const
```

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。
- **decltype**：选择并返回操作数的**数据类型**,不计算表达式的值。
- decltype(f()) sum = x; 推断sum的类型是函数f的返回类型。
- 不会忽略顶层const。
- 如果对变量加括号，编译器会将其认为是一个表达式，decltype((i))得到结果为引用。
- 赋值是会产生引用的一类典型表达式，引用的类型就是左值的类型。也就是说，如果 i 是 int，则表达式 i=x 的类型是 int&。

自定义数据结构

STRUCT

尽量不要把类定义和对象定义放在一起。如
```
struct Student{} xiaoming,xiaofang;
```

- 类可以以关键字struct开始，紧跟类名和类体。
- 类数据成员：类体定义类的成员。
- C++11：可以为类数据成员提供一个**类内初始值**（in-class initializer）。

- 定义Sales_data类

```
//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
```

- 使用Sales_data类

```cpp
//Sales_data.cpp
#include <iostream>
#include <string>
#include "Sales_data.h"

int main()
{
    Sales_data data1, data2;

    // code to read into data1 and data2
    double price = 0;  // price per book, used to calculate total revenue

    // read the first transactions: 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;

    // read the second transaction
    std::cin >> data2.bookNo >> data2.units_sold >> price;
    data2.revenue = data2.units_sold * price;
```

- 使用Sales_data类

```
    // code to check whether data1 and data2 have the same ISBN
    //        and if so print the sum of data1 and data2
    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
    }
}
```

编写自己的头文件

- 头文件通常包含哪些只能被定义一次的实体：类、const和constexpr变量。

预处理器概述：

- **预处理器**（preprocessor）：确保头文件多次包含仍能安全工作。
- 当预处理器看到#include标记时，会用指定的头文件内容代替#include
- **头文件保护符**（header guard）：头文件保护符依赖于预处理变量的状态：已定义和未定义。
    - #indef已定义时为真
    - #inndef未定义时为真
    - 头文件保护符的名称需要唯一，且保持全部大写。养成良好习惯，不论是否该头文件被包含，要加保护符。

```
#ifndef SALES_DATA_H  //SALES_DATA_H未定义时为真
#define SALES_DATA_H
strct Sale_data{
    ...
}
#endif
```

练习

根据你自己的理解重写一个`Sales_data.h`头文件，并以此为基础编写书店程序。

实践课

- 本场景将使用一台配置了Aliyun Linux 2的ECS实例（云服务器）
    - 使用Vim编辑C++代码
      根据你自己的理解重写一个Sales_data.h头文件，并以此为基础编写书店程序。
    - 使用g++编译运行这段代码
    - 编辑一个 **README.md** 文档，键入本次实验心得。
    - 使用git进行版本控制

```
- 云服务器（Elastic Compute Service，简称ECS）
- Aliyun Linux 2是阿里云推出的 Linux 发行版
- Vim是从vi发展出来的一个文本编辑器。
- g++ 是c++编译器
```

从课程主页cpp.njuer.org 打开实验二链接
使用Vim编辑c++代码和markdown文档，使用git进行版本控制
●单击屏幕右侧创建资源
●资源创建完毕后， 使用命令安装git工具和g++工具
//本地虚拟机这些工具已经装好，不必运行这两行
●yum install -y git
●yum install -y gcc-c++
●使用git工具进行版本控制，使用VIM编辑一个markdown文档，可参照左侧git说明帮助
●mkdir test 建立文件夹test
●cd test 进入文件夹test
●git init 表示文件夹版本库初始化
●vim test.md 按i键入第一版内容（简述这堂课学过的c++数据类型）,按ESC 再按：wq退出
//根据你自己的理解重写一个Sales_data.h头文件，并以此为基础编写书店程序test.cpp
vim Sales_data.h
vim test.cpp
●g++ ./test.cpp 编译
●./a.out 执行程序

●git add . 加入当前文件夹下所有文件到暂存区
●git config --global user.email "you@example.com"
●git config --global user.name "Your Name"
●git commit –m "test1" 表示提交到本地,备注test1
●vim test.md 键入新内容（实验感想),按ESC 再按：wq退出
●git add .
●git commit –m "test2" 表示提交到本地,备注test2
●git log 可看git记录
●git diff HEAD^ -- test.md 可查看test.md文档这一版与上一版区别
●git reset --hard HEAD^ 回滚到上一版本
●键入命令并截图，并提交到群作业。

```
cat test.md test.cpp
git log
```

附加题：

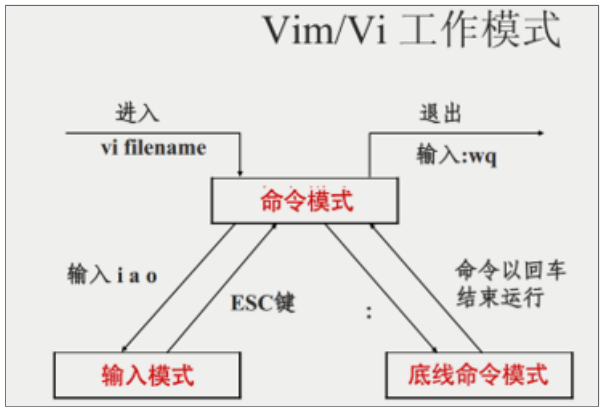
在gitee.com上注册用户名，并建立test项目。
把阿里云平台的本次作业test目录git push到你的git仓库
地址一般为： **https://gitee.com/你的用户名/test.git**

```
//写完作业后
确认自己当前目录里就是作业目录test。不是的话可用cd ..和cd 文件夹改变当前目录。
git remote add origin https://gitee.com/你的用户名/test.git
git push -u origin "master"
输入用户名密码 回车
浏览器打开 https://gitee.com/你的用户名/test 查看作业保存情况。
```

提交

- 截图或复制文字，提交到群作业。
- 填写网页实验报告栏，并将报告链接填入 **https://www.aliwork.com/o/cpphomework**
- 填写问卷调查 **https://rnk6jc.aliwork.com/o/cppinfo**

VIM 共分为三种模式



- 命令模式
  - 刚启动 `vim`，便进入了命令模式。其它模式下按`ESC`，可切换回命令模式
    - `i` 切换到输入模式，以输入字符。
    - `x` 删除当前光标所在处的字符。
    - `:` 切换到底线命令模式，可输入命令。
- 输入模式
  - 命令模式下按下`i`就进入了输入模式。
    - `ESC`，退出输入模式，切换到命令模式
- 底线命令模式
  - 命令模式下按下`:`（英文冒号）就进入了底线命令模式。
    - `wq` 保存退出

VIM 常用按键说明

除了 i, Esc, :wq 之外，其实 vim 还有非常多的按键可以使用。命令模式下：

- 光标移动
  - j下 k上 h左 l右
  - w前进一个词 b后退一个词
  - Ctrl+d 向下半屏 ctrl+u 向上半屏
  - G 移动到最后一行 gg 第一行 ngg 第n行
- 复制粘贴
  - dd 删一行 ndd 删n行
  - yy 复制一行 nyy复制n行
  - p将复制的数据粘贴在下一行 P粘贴到上一行
  - u恢复到前一个动作 ctrl+r重做上一个动作
- 搜索替换
  - /word 向下找word ? word 向上找
  - n重复搜索 N反向搜索
  - :1,$s/word1/word2/g从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2

VIM 常用按键说明

底线命令模式下：

- :set nu　显示行号
- :set nonu 取消行号
- :set paste　粘贴代码不乱序

【注：把caps lock按键映射为ctrl，能提高编辑效率。】

MARKDOWN 文档语法

```
# 一级标题
## 二级标题
*斜体* **粗体**
- 列表项
  - 子列表项
> 引用
[超链接](http://asdf.com)
![图片名](http://asdf.com/a.jpg)

|表格标题1|表格标题2|
|-|-|
|内容1|内容2|
```

谢谢