

第2章 C++ 基础

教学目标：掌握 C++ 语言中的基本概念和非面向对象的语言机制，了解 C++ 新语言标准与新语言特点，具备应用 C++ 新语言特性编写 C++ 程序的基本能力。

教学重点：C++ 的数据类型、类型推断、类型转换、常量、引用、指针、函数原型、默认参数、函数重载、内联函数、Lambda 函数、范围 for、命名空间及其应用、变量、文件。

教学难点：常量与指针、引用结合应用，智能指针、Lambda 函数、文件、类型转换、函数参数、移动函数、静态变量的作用域与生存期。

2.1 C++语言对C语言的数据类型扩展和类型定义

1、C数据类型在C++中继续可用

2、C++对C的struct、enum、union进行了扩展

C： 结构名、枚举、联合不是类型

```
struct some_struct {.....};
```

```
struct some_struct struct_var;
```

```
typedef struct some_struct struct_type;
```

– **C++：** 结构名、联合名为类型

```
struct some_struct {.....} ;
```

```
some_struct struct_var;
```

– **C++**允许在struct、union内部设置函数

2.1 C++语言对C语言的数据类型扩展和类型定义

3、enum和enum class

```
enum color{black, white, red, blue, yellow};           // C, C++ 98/03
enum class color1{black1, white1, red1, blue1, yellow1}; // C++ 11
bool black = false;                                     // L1, 错误, 不能通过编译
bool black1 = false;                                    // L2, 正确
```

- **enum**是C语言中不限作用域的枚举，称为枚举（**enumeration**）
- **enum class**是C++ 11标准**enum**的补充，称为枚举器（**enumerator**）
- 区别：
 - **enum**的作用域不受定义它的“{ }”限制，在上一级作用域内仍然有效。因此，**color**定义的**black**标识在语句L1处仍然有效，再定义**black**就属于重定义错误。
 - **enum class**的作用域局限在定义它的“{ }”中。因此，语句L2定义**black1**时并不会与**color1**中的枚举常量**black1**发生冲突，因为**black1**的作用域被限定在定义它的“{ }”中，不会延伸到语句L2处

2.2 C++程序变量设计的基本思想

1.面向过程与面向对象程序变量设计思想的主要区别

面向过程程序设计的基本思想

“程序=数据结构+算法”

程序设计基本方法:

① 定义程序要用到的数据结构和全局变量;

② 定义操作数据和全局变量的若干函数, 定义函数时, 也是先定义好要使用的所有局部变量才开始编写函数执行代码;

③ 在主程序(主函数)中组织执行流程, 按次序调用各函数, 进行全局变量的运算和修改, 实现程序功能。

```
struct A{ ... };  
int x, y, z;  
  
int f1(...) {  
    int i, j, k;  
  
    for(i = 0; i < n; i++) {  
        x += 10;  
        .....  
    }  
}  
int f1(...) { ... }  
void main() {  
    int a, b, c;  
    struct A s;  
    f1(a, b, c);  
    .....  
    f2(s);  
}
```

1.先定义数据结构和全部变量

2.再定义函数, 函数也是先定义变量, 再写操作语句

主要问题:

- 1.全局变量生存期长, 耗占内存时间长
- 2.全局变量可能与局部变量冲突
- 3.在编写大程序时代码量巨大, 查找最前面的全局变量定义较难。

2.2 C++程序变量设计的基本思想

1.面向过程与面向对象程序变量设计思想的主要区别

面向对象程序变量的基本思想

“尽量减小变量的作用域范围，少用（甚至不用）全局变量，变量应就近定义，就近使用”。

面向对象程序设计语言允许在任何语句位置定义变量。包括for、while、do-while循环语句内部以及switch和if等复合句中都可以定义变量。

【例2-2】在C++中，在for循环内部定义局部变量。

// Eg2-2.cpp

#include <iostream>

using namespace std;

void main() {

int n = 1;

for(int i = 1; i <= 10; i++)

{

int k;

n = n*i;

} // i、k作用域结束

int i=0;

} // n作用域结束

1.变量就近定义，就近使用

2.随用随定义

优点：

1.对象通常较大（数据成员多），减少全局变量可减少其对内存的长期耗占

2.减少了与局部变量的冲突

3.变量随用随定义，写大程序时不须找最前面的全局变量。

2.3 左值、右值

变量的左值和右值

int x,y; 定义了变量的左、右两个值

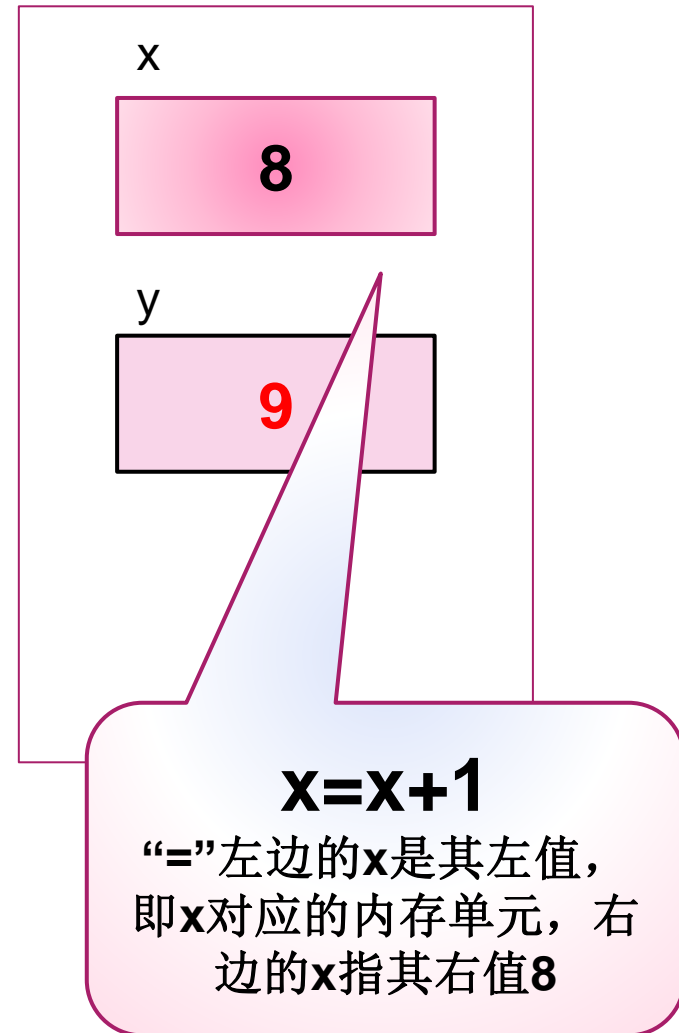
- **左值**

- 代表变量名对应的内存区域，是“能够放在赋值语句左边的值”
- 变量：**x**（指**x**对应的内存区域）

- **右值**

- 指变量对应内存区域中的值，是“放在赋值语句右边的值”
- 常量：**2**，**-3**，
- 表达式：**7+34.7**
- 变量：**x**（指在**x**对应内存区域中存放的值）
- 有变量的表达式：**x+7+y**

内存区域



2.4 指针

- 本节要掌握的内容
 1. C++的指针
 2. new、delete
 3. 指针与常量之间的关系
 4. 0指针、void指针、智能指针

2.4.1 指针概述

1、C++内存分配方式

– 静态分配（静态变量）

- 编译器在处理源代码时为变量分配内存，其效率较高，但缺少灵活性（要求程序执行之前就知道变量所需的内存类型和数量）

– 动态分配（动态变量）

- 程序执行时调用运行时刻库函数来分配变量的内存。

– 两者的区别

- 静态变量是有名字的变量，可以通过名字对它所代表的内存进行操作；动态变量是没有名字的内存变量，只能通过指针进行操作。
- 静态变量的分配和释放由编译器自动处理，动态变量的分配与释放必须由程序员控制。

2.4.1 指针概念的回顾

2、动态内存分配---指针

- 对类型**T**，**T ***是“到**T**的指针”，即一个类型为**T ***的变量，能存一个类型**T**的对象的地址

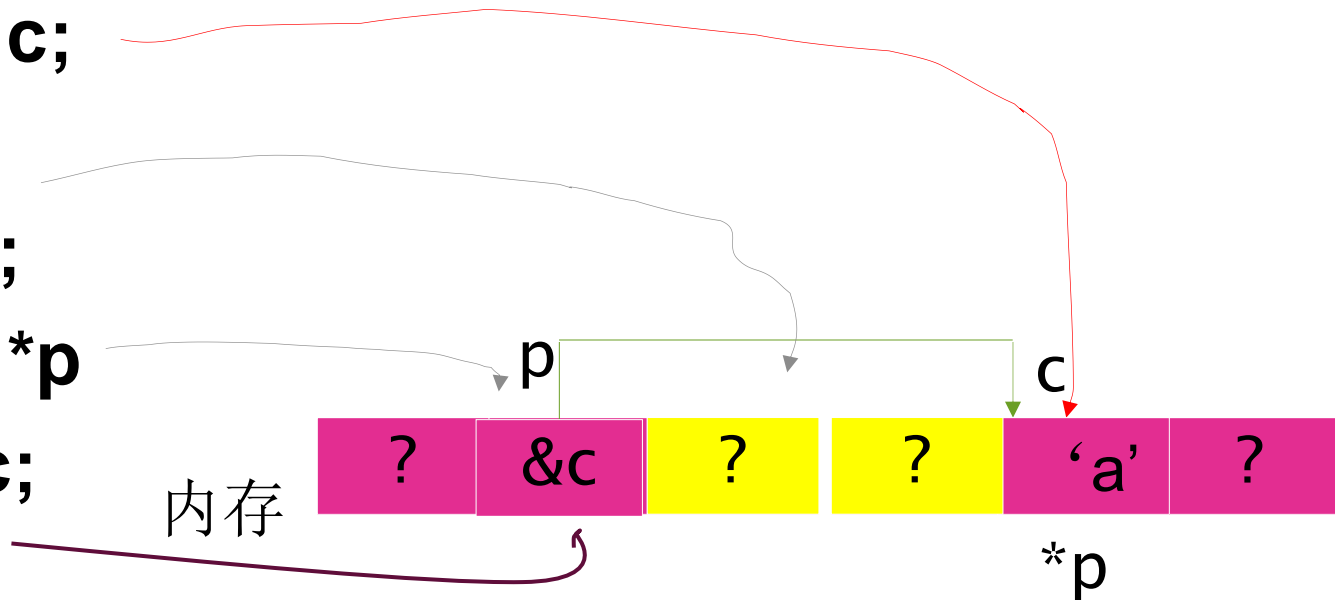
char c;

.....

c='a';

char *p

p=&c;



局部变量与指针在堆栈中的分配

```
#include<iostream>
using namespace std;
void main(int argc, char* argv){
    double d,*pd;           //L1
    int n=0,*p;              //L2
    d=3.2;                   //L3
    p=&n;                     //L4
    pd=&d;                    //L5
    *p=10;                   //L6
    cout<<" &d:"<<&d<<"\td : "<<d<<endl;
    cout<<"&pd:"<<&pd<<"\tpd:"<<pd<<endl;
    cout<<" &n:"<<&n<<"\tn : "<<n<<endl;
    cout<<" &p:"<<&p<<"\tp : "<<p<<endl;
    p=new int(8);
    cout<<"*p:"<<&*p<<endl;
```

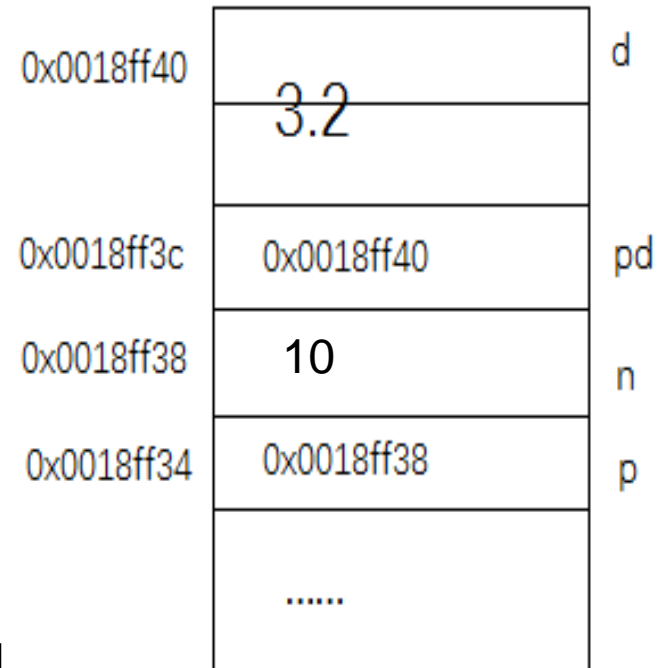


图2-1 VC6.0中建立的堆栈

//动态内存分配

}

2.4.1 指针概述

- 指针是一个复杂的概念，它能够指向（保存）不同类型变量的内存地址。例如：

`int *pi;` `// pi是指向int的指针`

`int **pc;` `// pc是指向int指针的指针`

`int *pA[10];` `// pA是指向int的指针数组`

`int (*f)(int,char);` `// f是指向具有两个参数的函数的指针`

`int *f(int)` `// f是一个函数，返回一个指向int的指针`

2.4.2 空指针，**void***，获取数组首、尾元素位置的指针

1. 空指针

- 空指针是没有指向任何内存单元的指针。
- **C++11**中NULL，0，nullptr意义等价，用于将指针设置为空指针：

T *ptr=0;

*ptr=NULL;

*ptr=nullptr;

C++11新定义

T是指任意数据类型

2.4.2 空指针，void*，获取数组首、尾元素位置的指针

2. void空指针

(1) 指针与地址的关系

- 每个指针都是一个内存地址，但都有一个相关的类型指示编译器怎样解释指针所指定内存区域的内容，以及该内存区域应该跨越多少个内存单元。相同类型的指针进行比较或相互赋值才有意义。

(2) void *指针

- void*指针只表示与它相关的值是个内存地址，但该内存的数据类型是未知的。void *是能够接受任何数据类型的特殊指针。
- void*最重要的用途是作为函数的参数，向函数传递一个类型可变的对象。另一种用途就是从函数返回一个无类型的对象。
- 在使用void*指针之前，必须显式地将它转换成某种数据类型的指针后使用，其他操作都不允许。

2.4.2 空指针， void*， 获取数组首、尾元素位置的指针

【例2-4】 void*指针的应用。

```
#include<iostream>
using namespace std;
void main(){
    int i=4,*pi=&i;
    void* pv;
    double d=9,*pd=&d;
    pv=&i;
    pv=pi;
    // cout<<*pv<<endl;
    pv=pd;
    cout<<*(double*)pv;
}
```

void * pv 分别指向了int、double类型的数据，但使用前必须进行类型转换！

- //L1: 正确
- //L2: 正确
- //L3: 错误
- //L4: 正确
- //L5: 正确，输出9

2c	4	i
28	2c	pi
24		pv
	9	
1c		d
18	1c	pd

2.4.2 空指针，**void***，获取数组首、尾元素位置的指针

3. **begin()**和**end()**

C++11

<iterator>头文件的两个函数，用于确定指向数组首元素和尾元素后一位置的指针，方便遍历数组。

```
int a[] = { 1,2,3,4,5,6,7,8,9,10 };  
for (int *p = begin(a); p != end(a); p++)  
    cout << *p << ", ";  
cout << endl;
```

for 循环依次输出数据组**a**的元素值！

2.4.3 内存的分配 和释放

1、动态存储管理的概念

系统为每个程序提供了一个可在程序执行时期申请使用的内存空间，这个内存空间被称为**空闲存储区**或**堆（heap）**，运行时刻的内存分配就称为**动态内存分配**。

2、C用malloc和free进行动态内存分配，操作麻烦

```
#include<stdlib.h>           //malloc和free定义于此头文件中
void main(){
    int *p;
    //从堆中分配1个int对象需要的内存并将转换为int类型
    p=(int*)malloc(sizeof(int));
    *p=23;
    free(p);                   //释放堆内存
}
```


2.4.3 内存的分配 和释放

3. C++动态内存分配可由new,delete运算符完成

- **New**用于从内存中分配指定大小的内存
 - 用法1: `p=new type;`
 - 用法2: `p=new type(x);` //分配并设初值x
 - 用法3: `p=new type[n];` //数组
- **delete**用于释放new分配的堆内存
 - 用法1: `delete p;`
 - 用法2: `delete []p;` //回收数组

4、【例2-5】 用new和delete分配与释放堆内存。

```
#include <iostream>
using namespace std;
void main(){
    int *p1,*p2,*p3;
    p1=new int;           //分配一个能够存放int类型数据的内存区域
    p2=new int(10);       //分配一个int类型大小的内存区域，并将10存入其中
    p3=new int[10];       //分配能够存放10个整数的数组区域
    if(!p3)               //程序中常会见到这样的判定
        cout<<"allocation failure"<<endl;    //分配不成功显示错误信息
    *p1=5;  *p3=1;
    p3[1]=2; p3[2]=3;     //访问指向数组的数组元素
    cout<<"p1  address: "<<&p1<<" value: "<<p1<<endl;
    cout<<"p2  address: "<<&p2<<" value: "<<p2<<endl;
    cout<<"p3[0] address: "<<&p3<<" value: "<<p3<<endl;
    cout<<"p3[1] address: "<<&p3[1]<<" value: "<<p3[1]<<endl;
    delete p1; delete p2; //释放指向的内存
    //delete p3;           //错误，只释放了p3指向数组的第1个元素
    delete []p3;          //释放p3指向的数组
}
```

2.4.3 内存的分配 和释放

5、new、delete和malloc、free的区别

- new能够自动计算要分配的内存类型的大小，不必用sizeof计算所要分配的内存字节数
- new不需要进行类型转换，它能够自动返回正确的指针类型。
- new可以对分配的内存进行初始化。
- new和delete可以被重载，程序员可以借此扩展new和delete的功能，建立自定义的存储分配系统。
- sizeof()被改为sizeof

2.5 引用（Reference）

1. 概念

- “引用”即“别名”，即是某对象的另一个名字，引用的主要用途是为了描述函数的参数和返回值。特别是用于运算符的重载。

2. 类型

- 左值引用
- 右值引用

3. 定义

- 类型 **&左值引用名** = 变量名;
- 类型 **&&右值引用名** = 表达式;

左值引用

：为变量对应的内存区域定义的别名，代表内存区域本身

右值引用

：为变量对应内存区域中的值定义的别名，代表内存区域内的数据（8）本身



2.5.1 左值引用

1. 概念

- “左值引用”即“变量的别名”，代表变量对应的内存区域，与原变量系同一内存区域，具有完全相同的操作方法。由于历史原因，也称之为引用。

2. 定义

类型 &引用名=变量名;

- 例如：

```
int i=9;           //L1
```

```
int &ir=i;          //L2  ir 与 i是同一实体的不同名称
```

3. 与指针区别

```
int * ip = & i;
```

```
int & ir = i;
```

2.5.1 左值引用

【例 2 -9】 引用的简单例子。

```
//Eg2.9.cpp
#include<iostream.h>
void main(){
    int i=9;
    int& ir=i;
    cout<<"i= "<<i<<"    "<<"ir="<<ir<<endl;
    ir=20;
    cout<<"i="<<i<<"    "<<"ir="<<ir<<endl;
    i=12;
    cout<<"i="<<i<<"    "<<"ir="<<ir<<endl;
    cout<<"i 的地址是: "<<&i<<endl;
    cout<<"ir的地址是: "<<&ir<<endl;
}
```

2.5.1 左值引用

3、使用引用应该注意的事情

- ① 引用不是值，**不占用存储空间**
- ② 引用在声明时**必须初始化**，否则会产生编译错误
- ③ 引用的初始值可以是一个变量或另一个引用
- ④ 引用可以视为“隐式指针”，但不分配存储空间
- ⑤ 引用由**类型标准符**和**一个取地址操作符来定义**，必须被初始化，且**不可**重新赋值
 - `int i,k;`
 - `int &r=i;`
 - `int &m=r; // ③`
 - `r=&k // error`
 - `r=k; //ok`
- ⑥ 引用的地址就是其所引用的变量的地址
 - `int num=50;`
 - `int &rnum=num;`
 - `int *p=# //p是指向num的指针，非引用;`
 - `int *rp=rnum; //err, 当为int *rp=&rnum;`

2.5.1 左值引用

⑦ 若一个变量声明为**T&**，必须用一个**T**类型的变量或对象，或能够转换为**T**类型的对象进行初始化

- `int i;`

- `double &rr=i;` //err, 类型不匹配

⑧ 可以有指针变量的引用，不能有指向引用的指针

- `int *p;`

- `int *&rp=p;` //ok rp是一个引用，它引用的是指针

- `int &*ra=a;` //error, ra是一指针，指向一个引用。

⑨ 变量名只能是左值引用，不能作右值引用

- `int a=1;`

- `int &&ra=a;` //error

2.5.1 左值引用

⑩ 引用与数组

可以建立数组或数组元素的引用，但不能建立引用数组

原因：引用不分配内存，一次只能为一个已有变量定义一个别名，而数组一次需要定义多个元素，所以不能定义引用数组

```
int i = 0, a[10] = { 1,2,3,4,5,6,7,8,9,10 }, *b[10];
```

```
int (&ra)[10] = a;      //L1: 正确，ra是具有10元素的整型数组的引用
```

```
int &aa = a[0];         //L2: 正确，数组元素的引用
```

```
int *(&rpa)[10] = b;    //L3: 正确，rpa是具有10个整型指针的数组的引用
```

```
int &ia[10]=a;          //L4: 错误，ia是引用数组，每个数组元素都是引用
```

```
ra[3] = 0;              //L5: 正确，数组引用的用法
```

```
rpa[3] = &i;            //L6: 正确
```

2.5.2 右值引用、移动及其语义

1. 概念

- 右值引用就是绑定到右值上的引用。
- 右值引用是C++11为了支持移动操作而引入的新型引用类型，其重要特点就是只能绑定到即将销毁的临时对象上，比如常量或表达式。通过右值引用可以方便地将它引用的资源“移动”到另一个对象上。

2. 定义

- 类型 &&引用名=表达式;

double r=10;

double &lr1=r; //正确，变量名代表左值

double &lr2=r+10; //错误，引用只能是变量

double &&rr=r; //错误，变量名代表左值，而&&需要右值

double &&rr=r+10; //正确，rr是保存“r+10”计算结果20的临时内存单元的别名

2.5.2 右值引用、移动及其语义

【例2-11】右值引用的定义和使用。

```
#include <iostream>
using namespace std;
void main(){
    int x = 10;
    int &r = x;
    //int &&ar = x;           //L1:错误，变量名只能绑定到左值
    int &&rx = x + 10 * 3;     //L2:正确，rx为右值引用，保存表达式值
    cout << "x=" << x << "\t rx=" << rx << endl;    //L3
    x = 20;                   //x变化不影响rx，
    cout << "x=" << x << "\t rx=" << rx << endl;    //L4      rx=40
    int y = rx;                //L5      y=40
    cout << "y=" << y << endl;    //L6
}
```

2.5.2 右值引用、移动及其语义

• 左值引用&右值引用的区别

- **关键理解**：只要是引用（无论左引用还是右引用），都是只能是某内存单元的别名，都对应的是**某内存变量的地址**。
- 左值只能够绑定到变量名（对应变量的内存区域），而且**具有持久性**（引用与其对应变量的生存期相同）；
- 右值只能**绑定到常量，或者表达式求值过程中创建的临时对象**上，本来该临时对象是短暂的，用完就会被销毁，而右值引用“接管”了该临时对象，使它可再次被使用（**右值引用的作用域和生存期长于它所对应的临时变量**）。

左值持久
右值临时

```
int x=9;  
int f(){..... return y;}  
int &lr=x;  
int &&rr1=x; //err  
int &&rr1=move(x);  
int &&rr2=f();  
int &&rr3=x+9;
```

std::move(x)是
一个类型转换函
数，将参数转换
为右值引用

x=f();
VS
x=rr2;

2.6 **const**和**constexpr**常量


2.6.1、常量的定义

– C

- **#define** 常量名称 常量

– C++

- **const** 类型 **常量名 = 常量表达式;** // 执行期
- **constexpr** 类型 常量名=常量表达式; // 编译期



常量在定义
时就必须初
始化

2.6.1、常量的定义

1、常量注意事项

① 常量一经定义就不能修改

<code>const int i = 5;</code>	<code>// 定义常量i</code>
<code>constexpr int k = 9;</code>	<code>// 定义常量k</code>
<code>i = 5;</code>	<code>// 错误, 修改常量</code>
<code>k++;</code>	<code>// 错误, 修改常量</code>

② 常量必须在定义时初始化

<code>const int n;</code>	<code>//错误, 常量n未被初始化</code>
<code>constexpr int k;</code>	<code>//错误, 常量n未被初始化</code>

③ 表达式可以出现在常量定义语句中, `const`中可以有变量名, 但 `constexpr`的表达式中不能有变量

<code>int j, k=9;</code>	<code>//L1</code>
<code>const int i1=10+k+6;</code>	<code>//正确</code>
<code>constexpr int i1=10+k+6;</code>	<code>//错误, k是变量</code>

2.6.1、常量的定义

2. `const`和`constexpr`的区别

- `constexpr` 在编译时进行初始化，`const` 在运行时初始化。
 - `constexpr`常量的初始化表达式中的每部分值都是程序运行之前就可以确定的字面值常量。
 - `const`常量初始化表达式中可以有变量，其初始值可以在程序运行期取得。但一经取得，就不可再被修改。

`const int n=size();` //L1: 正确，但n值的取得是在执行函数时。

`constexpr int m=size();` //L2: 错误，程序编译时不知道size()的值

`const int i = 10;`

`int j = 21;`

`const int i1 = i + 10;` //L3: 正确

`const int j1 = j + 10;` //L4: 正确

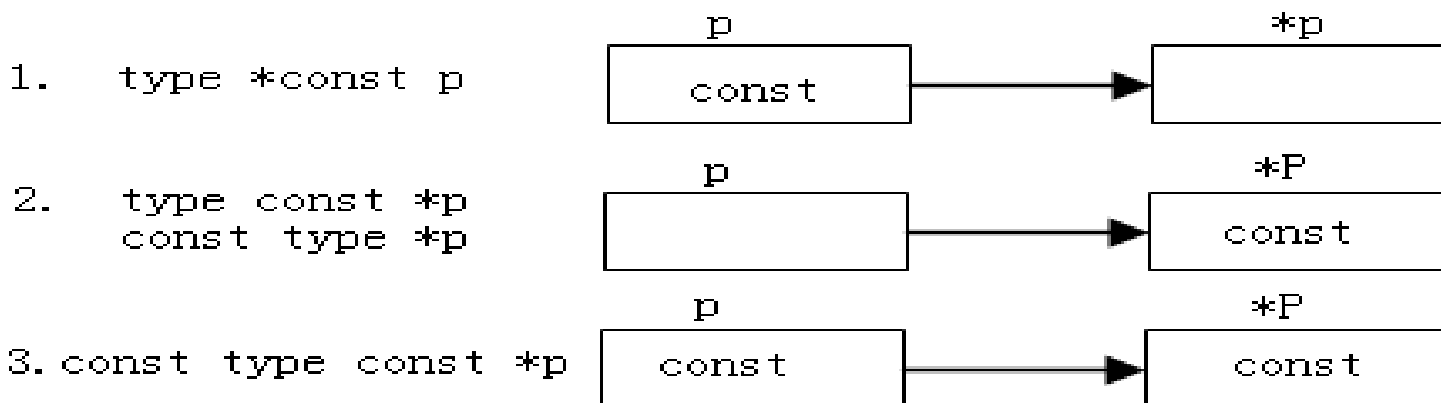
`constexpr int i2 = i + 10;` //L5: 正确，编译时可确定i值为10

`constexpr int j2 = j + 10;` //L6: 错误，j是变量。

2.6.2 **const**、**constexpr**与指针

1、**const**、**constexpr**与指针的限定关系

- 指针与**const**的限定关系



- 指针与**constexpr**的限定关系

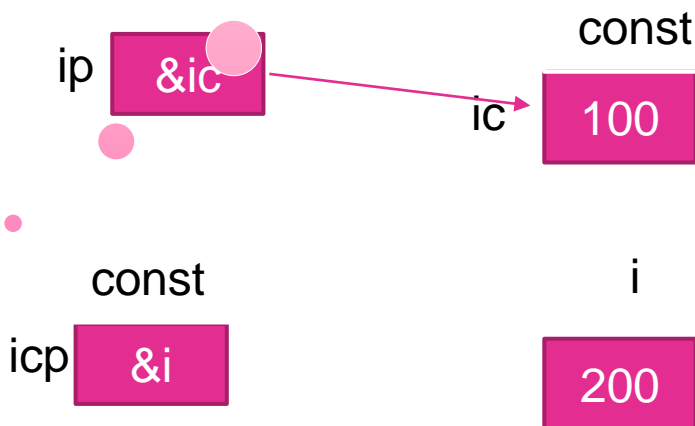
- **constexpr**限定指针时，它只限制指针变量本身是常量，与它所指的变量没有关系。
- **constexpr**指针的初始值必须是`nullptr`、`0`，或存储某个固定地址的对象。

2.6.2 **const**、**constexpr**与指针

2、const限制变量访问，避免非本意的数据修改举例

```
#include<iostream>
using namespace std;
main(){
    int i;
    const int ic = 100;
    const int * ip = & ic;
    ip=&i;    //ok
    *ip=0    //err
    int * const icp = & i;
    //icp = &j;    //err
    *icp = 200;
    cout<<"i="<<i<<endl;
    cout<<"*ip="<<*ip<<endl;
    cout<<"*icp="<<*icp<<endl;
}
```

icp是一个常量地址,此处企图修改它



2.6.2 **const**、**constexpr**与指针

3、指向常量的指针

- 在指针定义前加**const**，表示指向的对象是常量。

```
const int a=78;
```

```
const int b=28;
```

```
int c=18;
```

```
const int *pi=&a; //定义指向常量的指针
```

```
*pi=58;           //error, 不能修改指针指向的常量
```

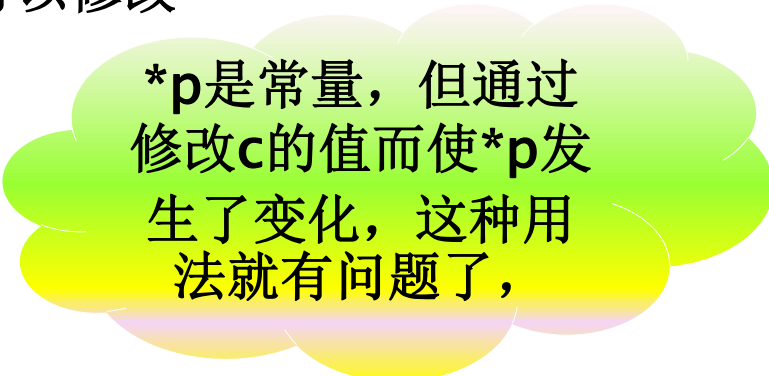
```
pi=&b;             //ok, 指针值可以修改
```

```
*pi=68;           //error
```

```
pi=&c;             //ok
```

```
*pi=98;           //error
```

```
c=98;             //ok
```



p**是常量，但通过修改**c**的值而使p**发生了变化，这种用法就有问题了，

2.6.2 **const**、**constexpr**与指针

– 例题:限制函数修改参数

//eg.cpp

```
#include <iostream.h>
```

```
void mystrcpy(char * Dest,const char *Src)
```

```
{ while(*Dest++=*Src++); }
```

```
void main(){
```

```
    char a[20]="How are you!";
```

```
    char b[20];
```

```
    mystrcpy(b, a);
```

```
    cout<<b<<endl;
```

```
}
```

2.6.2 **const**、**constexpr**与指针

4、指针常量

在指针定义语句的指针名前加**const**，表示指针本身是常量。

`char * const pc = "aaaa";` //定义指针常量时必须初始化

`pc = "bbbb";` //error, 指针常量不能改变其指针值

`*pc = "a";` //error, 类型不匹配

`*pc = 'a';` //ok,在VC环境下产生运行错误

`*(pc + 1) = 'b';` //ok,在VC环境下产生运行错误

`pc++ = 'y';` //error

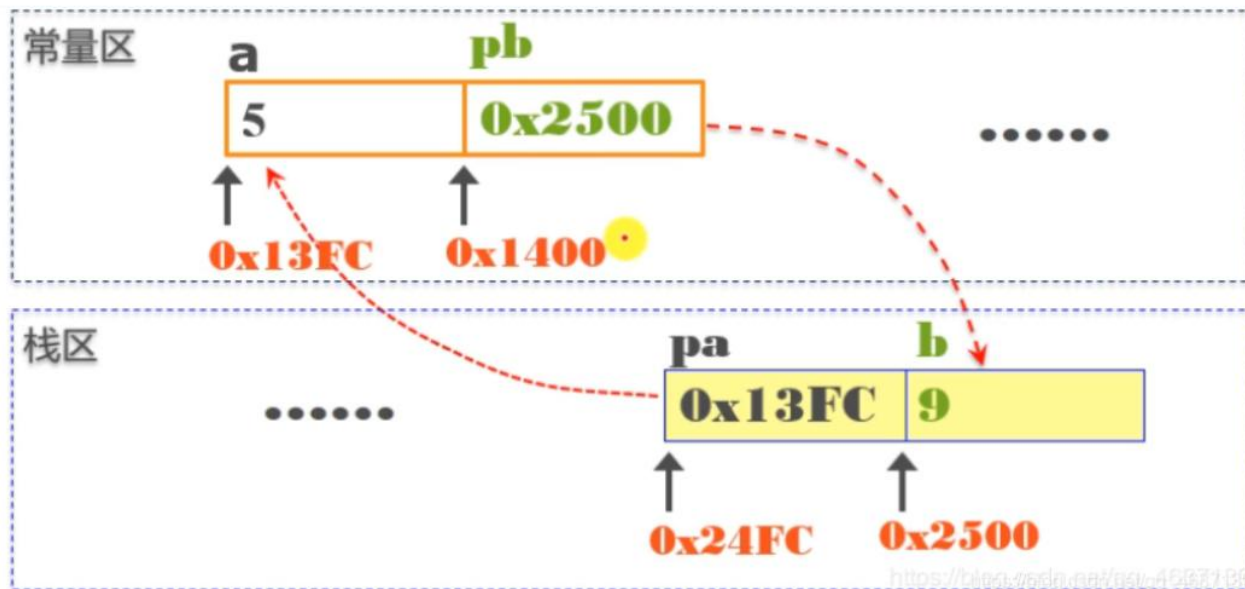
`const int b = 28;`

`int * const pi = &b;` //error, pi不能变,但它所指的内存单元内容却可以改

//变,此处却将它指向了一个不可变的内存单元

//,即:不能将**const int ***转换成**int ***

```
const int a=5;  
int b=9;  
const int *pa=&a;  
int * const pb=&b;
```



2.6.2 **const**、**constexpr**与指针

5、指向常量的**常指针**

- 可以定义一个指向常量的常指针变量，它必须在定义时进行初始化。例如：

```
const int ci=7;
```

```
int ai=8;
```

```
const int * const cpc=&ci;
```

```
const int * const cpi=&ai;
```

```
cpi=&ci;
```

```
*cpi=39;
```

```
ai=39;
```

//指向常量的指针常量

//ok

//error,

//error, 编译报错

//ok,

2.6.2 **const**、**constexpr**与指针

6、**const**、指针与变量赋值

- **const**对象的地址只能赋值给指向**const**对象的指针
- 但是，指向**const**对象的指针**可以指向常量对象，也可以指向非常量对象。**

const double minWage=9.60; //只能由指向常量的指针指向

const double *ptr=&minWage; //ptr既可以指向常量，也可以指向非常量

double dval=3.14;

ptr=&dval;

***ptr=23;**

dval=23;

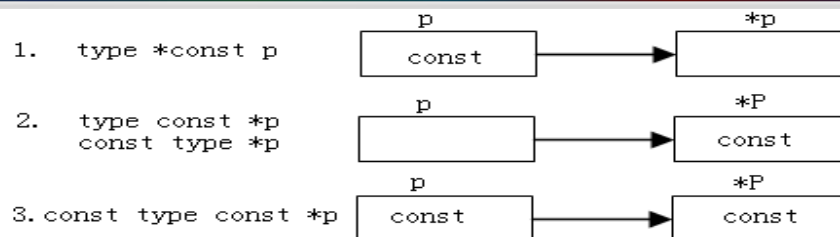
2.6.2 const、constexpr与指针

【例2-12】const与指针的关系。

```
#include <iostream>
using namespace std;
int main(){
```

```
    char *const p0;           // L1 错误, p0是常量, 必须初始化
    char *const p1 = "dukang"; // L2 正确
    char const *p2;           // L3 正确
    const char *p3 = "dukang"; // L4 正确
    const char *const p4 = "dukang"; // L5 正确
    const char *const p5;     // L6 错误, p5是常量, 必须初始化

    p1 = "wankang";           // L7 错误, p1是常量, 不可改
    p2 = "wankang";           // L8 正确, p2是变量, 可改
    p3 = "wankang";           // L9 正确, p3是变量, 可改
    p4 = "wankang";           // L10 错误, p4是常量, 不可改
    p1[0] = 'w';              // L11 正确
    p2[0] = 'w';              // L12 错误, *p2是常量, 不可改
    p3[0] = 'w';              // L13 错误, *p3是常量, 不可改
    p4[0] = 'w';              // L14 错误, *p4是常量, 不可改
    return 0;
```



}

2.6.3 **const**与引用

1、概念与功能

- 在定义引用时，可以用**const**进行限制，使它成为不允许被修改的常量引用。

```
int i=9,
```

```
int &rr=i;
```

```
const int &ir=i;
```

```
rr=8;
```

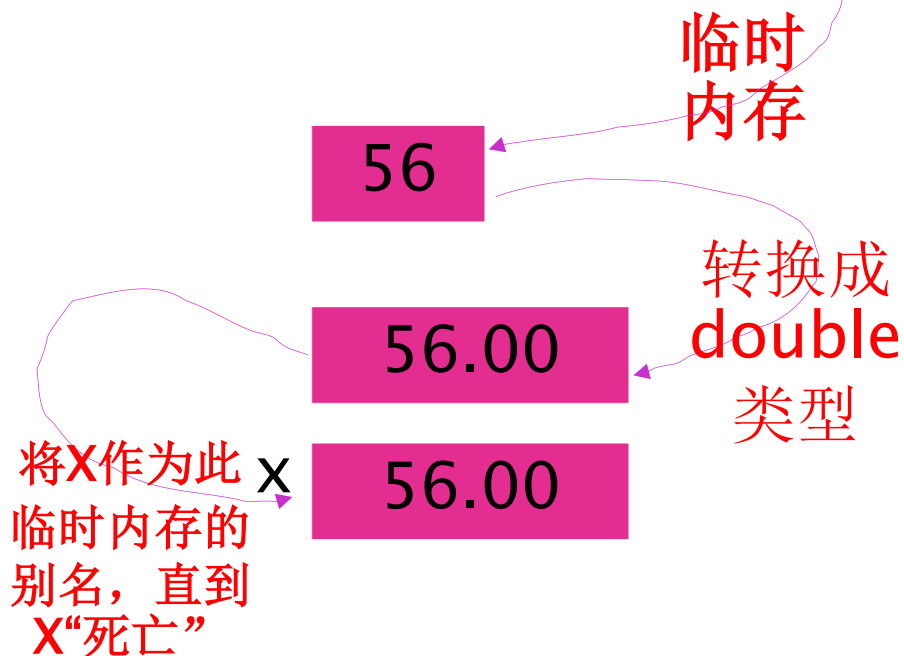
```
ir=7;
```

//错误，ir为const

2.6.3 **const**与引用

2、用常数或表达式初始化

```
#include<iostream.h>
void main(){
    int i=10;
    const double &x=23+23+i;
    cout<<"x="<<x<<endl;
}
```



对于形如**T&x**的普通引用，必须用一个类型T的左值初始化x。

对于一个形如**const T&x**的初始化，则不必是一个左值，甚至可以不是T类型的，其处理过程如下：

- 1、首先，如果需要，将应用T的类型转换。
- 2、而后将结果存入一个类型T的临时变量。
- 3、最后将此临时变量用作初始式的值。

2.6.3 **const**与引用

3、用变量初始化**const**引用

```
double dval=1024;  
const int &ri=dval;  
dval=90;  
cout<<"dval="<<dval<<endl;  
cout<<"ri="<<ri<<endl;  
ri=90;    //错误，修改常量
```

编译器将此定义转换成类似如下的代码：

```
int temp=dval;  
const int &ri=temp;
```

对**dval**的修改不会影响到**temp**，所以**ri**的值不会因**dval**而变动。

程序输出结果：

dval=90

ri=1024?

2.6.4 顶层const与底层const

• 1、概念

- 指针实际上定义了两个对象：指针本身和它所指的对象。这两个对象都可以用const进行限定，当指针本身被限定为常量时，称指针为顶层const；当所指的对象被限定为常量，而指针本身未被限时，称指针为底层const；
- 当指针和所指对象两者都被限定为常量时，则指针为顶层const，所指对象为底层const。
- 例如：

```
int i = 0;
```

```
const int ic = 32;
```

```
int *const p1 = &i;
```

//p1为顶层const

```
const int *p2;
```

//P2为底层const

```
const int *const p3 = &ic;
```

//p3为顶层const, p3也为底层const

2.6.4 顶层const与底层const

• 2、概念延伸

- 一般地，**顶层const**其实是指不可被修改的常量对象，此概念可以推广为任意的数据类型定义的常量对象都是顶层const。
- 底层const则与指针和引用这样的复合类型定义有关，其中指针比较特殊，既可以顶层const，也可能是底层const。而所有声明为const的**引用都是底层const**。
- 总结：指向常量的指针（指针本身非常量）和常量引用是**底层const**，其它都是顶层const指针。

```
int i=3;
```

```
const double d=9.0
```

```
//d为顶层const
```

```
const int ic = 32;
```

```
//ic为顶层const
```

```
const int &ri = i;
```

```
//ri为底层const
```

```
const int &ric = ic;
```

```
//ric为底层const
```

2.6.4 顶层const与底层const

3、应用

- (1) 复制顶层 const 不受影响。

– 由于执行复制时不影响被复制对象的值，因此它是否为常量对复制没有影响。例如，

```
int i=3;
```

```
const double d=9.0
```

```
//ic为顶层const
```

```
const int ic = 32;
```

```
//ic为顶层const
```

```
const int &ric = ic;
```

```
const int *p2;
```

```
//p2为底层const
```

```
const int *const p3 = &ic;
```

```
i = ic;           // 正确： ic 是一个顶层 const，对此操作无影响
```

```
p2 = p3;          // 正确： p2 和 p3 指向的对象类型相同，p3 顶层 const 部分不影响
```

2.6.4 顶层const与底层const

- (2) 底层const的复制是受限制的。
- 要求拷入和拷出的对象有相同的底层 const 或者能够转换为相同的数据类型，一般而言，非常量能够转换成常量，反之则不行。
- 例如，针对前面的语句组，执行下面的复制操作。

```
int i=3;
const double d=9.0;           //d为顶层const
const int ic = 32;             //ic为顶层const
const int &ric = ic;
const int *p2;                  //p2为底层const
const int *const p3 = &ic;
p2 = p3;                        // 正确: p2为底层const, p3是顶层也是底层const, 且类型相同。
p2 = &i;                        // 正确: p2为底层const, &i为int*, 且能转换成 const int*
p2= &ic;                        // 正确: p2为底层const, &ic为const int *
p2 = &ric;                      // 正确: p2和ric为相同类型的底层const
int *p = p3;                  // 错误: p3 包括底层 const 定义, 而 p 没有
const int &r2 = i;              // 正确: r2为底层const, 而 i 可以转换成底层const
int &r = ic;                  // 错误: 常量不能转换为非常量
```

2.6.4 顶层const与底层const

(3) 常见应用——限定函数参数

- 在C++库函数中，常见到许多函数的形参用const限定，可以简单理解为：提供了更多的调用形式！因为，
 - （1）用const限定的参数，可以接受const和非const类型的参数（非const可以转换成const）
 - （2）若函数参数为非const类型，则不能接受const类型的实参。

小结：const与引用、指针

	底层const	顶层const
Code::Block(Review案例)	const *int p;	int *q=new int(8); int * const p1=q; const int * const p2=q;
	int r=0; const int &rr=r;	推广：底层const外，所有其它的const对象都是顶层const
复制（赋值和函数参数传递时，拷入、拷出要求。 非常量能够转换为常量。	为同类型的const	没有影响
int j = 9; const int i = 0;	int f1(int& x); f1(j) f1(i) int f2(const int& x); f2(j) f2(i) int p1(int *x) p1(&j) p1(&i) int p2(const int *x) p2(&j) p2(&i)	int f1(int x); f1(j) f1(i) int f2(const int x); f1(j) f1(i) int p1(int *x) p1(&j) p1(&i) int p2(const int *const x) p2(&j) p2(&i)
	红色函数调用是错误的！	红色函数调用是错误的！

2.10 函数

- 本节主要介绍**C++**函数的相关知识，应着重了解函数**默认参数**、**引用参数及返回值**、**重载函数**及其解析过程等方面的知识。

2.10.3 函数默认参数

1、概念

- C++允许为函数提供默认参数。在调用具有默认参数的函数时，如果没有提供调用参数，C++将自动把默认参数值作为相应参数的值。

2、规则

- 只能默认全部或部分右边的参数
- 函数声明和定义同时存在时，仅声明中才能出现默认的说明。
- 默认参数的声明必须先于函数调用。

2.10.3 函数默认参数

【例2-23】设计函数`sqrt()`，计算给定数字的平方，默认计算1.0的平方。

```
#include <iostream>
using namespace std;
double sqrt(double f=1.0);
void main(){
    cout<<sqrt()<<endl;           //采用默认参数
    cout<<sqrt(5)<<endl;
}
double sqrt(double f) {
    return f*f;
}
```

2.10.3 函数默认参数

3、注意

- ① 一旦某个参数开始指定默认值，它右边的所有参数都必须指定默认。

```
int f(int i1,int i2=2,int i3=0);    //正确
```

```
int g(int i1,int i2=0,int i3);      //错误，i3没有缺省值
```

```
int h(int i1=0,int i2,int i3=0);    //错误，i1缺省后，其右i2没有缺省
```

- ② 在调用具有默认参数值的函数时，若某个实参默认，其右边的所有实参都应默认。例如：

```
int f(int i1=1,int i2=2,int i3=0){ return i1+i2+i3; }
```

针对此函数，有如下调用：

```
f();                //正确，i1=1,i2=2,i3=0
```

```
f(3);              //正确，i1=3,i2=2,i3=0
```

```
f(2,3);            //正确，i1=2,i2=3,i3=0
```

```
f(4,5,6);          //正确，i1=4,i2=5,i3=6
```

```
f(2,3);            //错误，i1缺省，而右边的i2,i3没有
```

2.10.3 函数默认参数

③ 可以用表达式作为默认参数，只要表达式可以转换成形参所需要的类型即可。但是，局部变量不能作为默认参数值。

【例2-24】 设计函数dog，默认狗名为tom，0.8米高，1.1米长。

```
#include<iostream>
#include<string>
using namespace std;
string name="tom";
double h = 0.8, len = 1.1;
void dog(string dogname = name, double high = h, double lenth = len)
{
    cout << "Dogname:" << dogname << "\tHigh:" << h
        << "\tLenth:" << len << endl;
}
int main(){
    name = "Jake";        //L1: 修改全局变量，改变默认实参值
    double h = 2.1;       //L2: h隐藏了全局变量h，对dog参数h的默认值无影响
    dog();
}
```

2.10.5 函数重载

1、函数重载的概念

- 函数重载就是允许在同一程序中（确切地讲是指在**同一作用域内**）**定义多个同名函数**，这些同名函数可以有不同的返回类型、参数类型、参数个数，以及不同的函数功能。

2、引用函数重载的原因

- 实现简单的“多态”：单接口、多实现。减少程序应用人员的负担：不用定义和记忆更多的函数名
- e.g: 求两个数中的大数
 - `max_i(int,int)`, `max_f(float,float)`.....。

2.10.5 函数重载

【例2-28】 重载计算int、float、double三种类型数据绝对值的函数

//Eg2.28.cpp

```
#include<iostream>
```

```
#using namespace std;
```

```
int abs(int x) {return x>0?x:-x;}
```

```
float abs(float x) {return x>0?x:-x;}
```

```
double abs(double x) {return x>0?x:-x;}
```

```
void main(){
```

```
    cout<<abs(-9) <<endl;
```

```
    cout<<abs(-9.9f) <<endl;
```

```
    cout<<abs(-9.8) <<endl;
```

```
}
```


2.10.5 函数重载

3、函数重载解析过程

按下面的3个步骤的先后顺序找到并调用函数：

(1) **准确匹配**：无须任何转换或只须做平凡转换（如数组名到指针、函数名到函数指针、T到**const T**等）的匹配。

(2) **利用提升**，包括整数的提升

例：**bool->int, char ->int, short-> int , float ->double**

(3) **标准转换**

如：int<->double, double<->long double.

Derive *->base *, T*->void *, int ->unsigned int

(4) **通过一个用户定义的转换**（第6章介绍）

(5) **利用在函数声明中的省略号...**

【例2-29】 函数重载解析的例子。

```
#include <iostream>
using namespace std;
void f(int i){cout<<i<<endl;}
void f(const char*s){cout<<s<<endl;}
void main(){
    char c='A';    int i=1; short s=2;
    double ff=3.4;
    char a[10]="123456789";
    f(c);           //f(int i) 提升
    f(i);           //f(int i) 精确匹配
    f(s);           //f(int i) 提升
    f(ff);          //f(int i) 转换
    f('a');         //f(int i) 提升
    f(3);           //f(int i) 精确匹配
    f("string");    //f(const char*s) 精确匹配
    f(a);           //f(const char*s) 精确匹配
}
```

2.10.5 函数重载

4、函数重载的注意事项

(1) 重载函数原形的要求

每个函数的参数表唯一就行（参数个数、参数类型、或参数顺序上有所不同，不包括函数返回类型）

```
int f(int,int);
```

```
double f(int);
```

```
int f(char);
```

下面两个f函数只有返回类型不同，是错误的：

```
int f(int);
```

```
double f(int);
```

2.10.5 函数重载

② 在定义和调用重载函数时，要注意它的二义性。

```
int f(int& x) { /* ..... */ }
```

```
double f(int x) { /* ..... */ }
```

```
int g(unsigned int x) { return x; }
```

```
double g(double x) { return x; }
```

- **f**和**g**是正确的重载函数，调用不当会有二义性，例如：

```
int a=1;
```

```
f(a); //错误，产生二义性
```

```
g(a); //错误，产生二义性
```

于精确匹配和提升对于**g(a)**的调用都会失败，因此就会使用转换的原则调用**g(a)**，但**int**既可以转换成**unsigned int**，也可以转换成**double**，则**g(a)**调用**g(unsigned int x)**或**g(double x)**都是正确的，因此会产生二义性

2.10.5 函数重载

③ 重载函数和const形参

– 顶层const参数不影响实参的传入，易产生二义性

- 即，顶层const形参，可以接受const和非const的同类型实参，容易在函数调用时，则会产生二义性重定义编译错误。例如，

```
int f(int x, int y) { cout << "fa" << endl;}
```

```
int f(const int x, const int y) { cout << "fb" << endl;}
```

– 底层const则是可区分的，能够正常使用

- 拥有指针或引用参数的函数和拥有底层const指针或引用的同名函数属于重载函数。

2.10.5 函数重载

【例2-30】设计通过底层**const**引用区分重载函数**f**，以及通过底层**const**指针区分的函数**g**。

```
#include <iostream>
using namespace std;
void f(int &x) { cout << "f(int &)" << endl;}
void f(const int &x) { cout << "f(const int& )" << endl; }
void g(const int * x) { cout << "g(const int *)" << endl;}
void g(int * x) { cout << "g(int *)" << endl; }
void main(){
    int x = 10;
    const int y = 9;
    f(x);                //调用f(int &x)
    f(y);                //调用f(const int &)
    g(&x);               //调用g(int *x)
}
```

2.10.6 函数与**const**和**constexpr**

- 在C++中，函数参数可能会是大型对象，用**值传递方式**进行参数传递需要进行大量的数据复制，存储空间和运行时间的开销较大，**效率较低**。
- 用**const或constexpr**限定的指针或引用传递参数，则可以避免函数对参数对象进行修改，**既高效又安全**。

2.10.6 函数与const和constexpr

【例**】 用指针参数比较两个双精度数的大小。如果第1个数大于第2个数，函数值为1；如果第1个数等于第2个数，函数值为0；如果第1个数小于第2个数，函数值为-1。

```
//Eg2-15.cpp
#include<iostream.h>
int fcmp( double * d1, double *d2){
    if(*d1>*d2)
        return 1;
    else if(*d1=*d2) //错误
        return 0;
    else if(*d1<*d2)
        return -1;
}
void main(){
    double x,y;
    x=34.0;
    y=89.2;
    cout<<fcmp(&x,&y);
}
```

若将fcmp的参数限定为const，
本程序将不能通过编译！

```
int fcmp(const double &d1, const
double &d2){
    if(d1>d2)
        return 1;
    else if(d1=d2)
        //错误，d1是const型的
        return 0;
    else if(d1<d2)
        return -1;
}
```


2.10.6 函数与const和constexpr

1. 形参是顶层const

- 一方面，const限定的参数不可修改，另一方面，实参传递忽略顶层const。例如，

```
int f(int i1,const int i2){  
    i1++;  
    // i2++;           //错误，i2是const，不可修改  
    return i1+i2;  
}
```

- 对此函数的以下调用都是正确的

```
const int x=9;  
int y=100;  
f(100,x);           //x是常量实参  
f(x,y);             //y是非常量实参
```

2.10.6 函数与**const**和**constexpr**

2. 形参是底层**const**

- 底层**const**复制规则

- 同类型的底层 **const** 或者能够转换为相同的数据类型才能够复制，此外，非常量能够转换成常量，但常量不能转换为非常量。例如，

```
int i = 10, const j = 10;
```

```
const int *p1 = &i;    //正确
```

```
const int *p2 = &j;    //正确
```

```
const int &r1 = i;     //正确
```

```
const int &r2 = 10;    //正确
```

```
int *p3 = p1;         //错误
```

```
int &r3 = r1;          //错误
```

```
int &r4 = r2;          //错误
```

- 底层**const**复制规则适用于函数参数传递

2.10.6 函数与const和constexpr

【例2-31】 //Eg2-31.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
void fp1(const int *ap1){}
```

```
void fp2(int *ap2) {}
```

```
void fr1(const int & ar1)
```

```
{    cout << "fr1" << endl; }
```

```
void fr2(int &ar2)
```

```
{    cout << "fr2" << endl; }
```

```
void main() {
```

```
    int i = 10;
```

```
    const int j = 10;
```

```
    int *p1;
```

```
    const int *p2;
```

```
    const int *const p3=&i;
```

```
    fp1(p1);
```

```
    fp1(p2);
```

```
    fp1(p3);
```

```
    fp1(&i);
```

```
    fp1(&j);
```

```
    fp2(p1);
```

```
    fp2(p2);
```

//错误

```
    fp2(p3);
```

//错误

```
    fp2(&i);
```

```
    fp2(&j);
```

//错误

```
    fr1(i);
```

```
    fr1(j);
```

```
    fr2(i);
```

```
    fr2(j);
```

//错误,

```
}
```

2.10.6 函数与const和constexpr

3. 将函数返回值指定为引用值

– 返回值为引用时，则不能修改返回值。

【例2-32】 返回const引用的函数。

```
#include<iostream>
using namespace std;
const int& index(int x[],int n){
    return x[n];
}
void main(){
    int a[]={0,1,2,3,4,5,6,7,8,9};
    cout<<index(a,6)<<endl;
    index(a,2)=90; //错误
    cout<<a[2]<<endl;
}
```

2.10.6 函数与**const**和**constexpr**

4 . **constexpr**与函数

11C₊₊

- 用**constexpr**限定函数则与**const**有两点区别：
 - （1）如果需要在编译期间就确定常量值（如数组下标），最好使用**constexpr**函数而非**const**函数。只要传入的参数都能够在编译期确定，**constexpr**函数就能够在编译期计算出函数值。但是，如果有任何一个参数的值不能在编译期知道，就会产生调用错误。
 - （2）当使用了不能够在编译期间确定的参数调用**constexpr**函数时，该函数得就会像一个普通的函数一样，在运行期计算它的值。

2.10.6 函数与const和constexpr

【例2-33】 返回constexpr常量的函数。

```
//Eg2-33.cpp
#include<iostream>
using namespace std;
constexpr int inc(int i) { return i + 1;}
int main(){
    int x;
    cin >> x;
    double stu1[inc(x)]; //L1, 错误, 编译期x未知
    double stu2[inc(9)]; //L2, 正确
    cout << inc(x) << endl
    return 0;
    //L3, 正确
}
```

2.12 命名空间

Z1.cpp

```
Namespace Tom{  
int x,y;  
int f(...){..  
int g(...){...}
```

k1.cpp

```
Namespace Jack{  
int x,y;  
int f(...){..  
int g(...){...}
```

Z1.cpp

```
int x,y;  
int f(...){..  
int g(...){...}
```

k1.cpp

```
int x,y;  
int f(...){..  
int h(...){...}
```

Main1.cpp

```
#include "z1.cpp"  
#include "k1.cpp"  
int main()  
{  
.....  
}
```

```
#include "z1.cpp"  
#include "k1.cpp"  
int main(){  
    Tom::x=9;  
    Jack::x=20;  
    .....  
}
```

1、什么是命名空间？

- 就是将程序的构成要素（如变量名，数据类型，函数.....）局限在某个名字框定的范围内部，内部相互可见（可以直接引用），外部则要应用名字限定才能引用。
- 例如，某程序由 **Tom,Jack** 各编写一部分，分别为 **z1.cpp** 和 **k1.cpp**

Err:redefine x,y,f()

2.12 命名空间

1、C++引入命名空间的原因

- C++编程环境中，系统定义了大量的变量、函数和类的名称。
- 在编程中可能定义出系统已存在的变量、函数或类名称，产生冲突。
- 多人合作进行软件开发时，可能定义出相同的名称，产生冲突。
- 这些问题导致名字空间的运用：即程序员可以将自己定义的名字局限在一个自定义的名字空间中，就不会与其它人定义的名字冲突。

2.12 命名空间

2、命名空间的定义

```
namespace XX  
{  
    members;  
}
```

其中，**namespace**是定义名字空间的关键字；

XX是程序员指定的名字空间的名字；

members 是命名空间中包括的成员，可以是变量定义、函数声明、函数定义、结构声明，以及类的声明等。

2.12 命名空间

【例2-39】 设计命名空间**ABC**，其中包括计数器、学生结构、类型定义和简单函数

```
namespace ABC{  
    int count;  
    typedef float house_price;  
    struct student{  
        char *name;  
        int age;  
    };  
    double add(int a,int b)  { return (double)a+b;}  
    inline int min(int a,int b);  
};  
int ABC::min(int a,int b){  
    return a>b?a:b;  
}
```

2.12 命名空间

4、命名空间的应用

- 命名空间成员的作用域局限于命名空间内部，可以通过作用域限定符 (::) 访问它，语法如下：

namespace_name::identifier

命名空间**ABC**有5个成员：**count**、**student**、**house_price**、**add** 和 **min** 。对其引用如下：

```
void main(){
    ABC::count=1;           //访问ABC空间中的count
    int count=9; //main函数中的count与ABC中的count无关
    ABC::student s;        //用ABC空间中的student结构定义s
    s.age=9;
    int x=ABC::min(4,5);      //调用ABC中的min函数计算两数最小值
}
```

```
namespace ABC{
    int count;
    .....
    struct student{
        char *name;
        int age;
    };
    double add(int ,int );
    int min(int a,int b);
};
```

2.12 命名空间

5、用using namespace访问名字空间成员

① 引用名字空间的单个成员。用法如下：

using namespace_name::identifier

例如，用using简化ABC名字空间中count的使用：

```
void main(){  
    using ABC::count;           //L1  
    count =2;                   //L2  
    //int count=9;              //L3  
    .....  
    count=count+2;              //L4  
}
```

```
namespace ABC{  
    int count;  
    .....  
    struct student{  
        char *name;  
        int age;  
    };  
    double add(int ,int );  
    int min(int a,int b);  
};
```

2.12 命名空间

② 引入命名空间的全部成员。用法如下：

using namespace_name

例如，引有前述**ABC**命名空间的全体成员

```
void main(){
```

```
    using namespace ABC;    //L1
```

```
    int count=9; //错误，已有源于ABC中的count，重复定义
```

```
    student s;
```

```
    count=5;
```

```
    s.age=min(43,32);
```

```
}
```

```
namespace ABC{  
    int count;  
    .....  
    struct student{  
        char *name;  
        int age;  
    };  
    double add(int ,int );  
    int min(int a,int b);  
};
```

2.12 命名空间

【例】 名字空间的应用举例。

```
#include<iostream>
using namespace std;
namespace A{
    int n;
    void f(){ cout<<"namespace A::f()"<<endl; }
    void g(){ cout<<"namespace A::g()"<<endl;}
};
namespace B{
    int n;
    void f(){ cout<<"namespace B::f()"<<endl; }
    void t(){ cout<<"namespace B::t()"<<endl; }
};
void main(){
    using namespace A;
    using namespace B;
    A::n=0;
    A::f();
    B::f();
    g();
    t();
}
```

2.12 命名空间

6. std命名空间

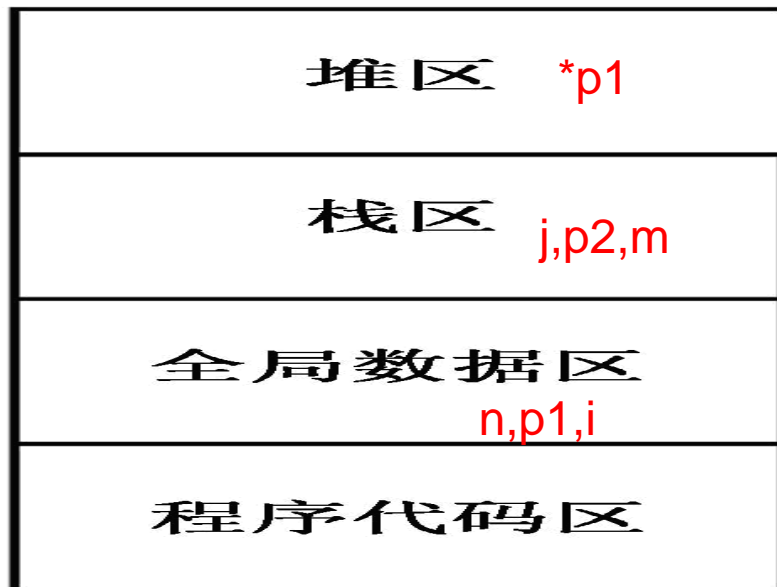
- 两个版本：一个是以Bjarne Stroustrup最初设计的C++为基础的版本，称为**传统C++**；另一个是1989年及之后以ANSI/ISO标准化委员会创建的C++，称为**标准C++**。
- 两种版本有大量相同的库和函数。为了将两者区分：传统C++采用与C语言同样风格的头文件；标准C++的新式头文件没有扩展名，即不需要.h之类的扩展名。例如，传统C++的头文件有*iostream.h*、*string.h*；标准C++对应的头文件有*iostream*、*string*。
- 程序要引用标准C++函数，可用下面语句将std名字空间中的名称引入到全局名字空间中。

using namespace std;

2.13.3 变量类型及生存期

1、内存的类型

一个程序在其运行期间，它的程序代码和数据会被分别存储在4个不同的内存区域，如图所示。



```
int n,int* p1;

void f() {
    static int i;    int j;
}

void main() {
    int* p2,  int m;
    p1=new int(1);
}
```


2.13.3 变量类型及生存期

2、内存类型与变量的关系

- 全局数据区中的数据由**C++**编译器建立，对于定义时没有初始化的变量，系统会自动将其初始化为**0**。这个区域中的数据一直保存，直到程序结束时才由系统负责回收。
- 堆区的数据由程序员管理，程序员可用**new**或**malloc**分配其中的存储单元给指针变量，用完之后，由程序员用**delete**或**free**将其归还系统，以便其他程序使用。

2.13.3 变量类型及生存期

3、变量类型

- 全局变量
 - 在程序文件的所有函数之外定义的变量，它存放在内存的全局数据区，可被程序中的所有函数所使用。
- 局部变量
 - 在局部使用域和函数作用域内定义的变量，有效范围在定义它的语句块或函数范围内。

2.13.3 变量类型及生存期

4、生存期

- 静态生存期

- 从程序开始直到程序结束.

- 全局变量、静态全局变量、静态局部变量

- 局部生存期

- 局部变量，生存期从其声明点始，至声明它的块结束。

- 系统不会自动初始化此类变量，其初值不定。

- 动态生存期

- **Malloc**，**new**创建的变量，结束于**free**、**delete**操作。

堆区
栈区
全局数据区
程序代码区

2.13.3 变量类型及生存期

【例2-42】 静态变量的生存期长于其作用域的例子。

// Eg2-42.cpp

#include <iostream.h>

static int n; //n被初始化为0

void f(){
 static int i; //i被初始化为0

int j=0;

i+=2;

j+=2;

cout<<"i="<<i<<" ";

cout<<"j="<<j<<endl;

}

void main(){

n+=5;

f(); //输出i=2, j=2;

i=2; //错误, i虽然为static, 但其作用域为函数f()内部

f(); //输出i=4, j=2;

} //i, n的生命期到此才结束

2.13.4 变量初始化

1. C++变量初始化概述

- ① 未初始化变量的值不确定，是导致许多程序错误的根源。
- ② C++强调：常量、引用、类对象必须初始化。
- ③ C++的全局变量初始化时可以使用任意表达式，不再局限于C的常量表达式。
- ④ C++提供一种函数风格的初始化方式，便于对象初始化，因为对象初始化参数可以不止一个。

2.13.4 变量初始化

2. 初始化的方式

① `int x=0;`

② `int x(0);`

③ `int x={0};`

11C₊₊

④ `int x{0};`

11C₊₊

- 第3、4种是C++11规范中新设的变量初始化和赋值，称为初始化列表（统一初始化）。花括号除了用于变量初始化，还可用于赋值。而在C++11标准之前，仅部分场合才允许使用这种初始化方式，如数组初始化。

2.13.4 变量初始化

3. 使用列表初始化或赋值的注意事项

- C++11中，可以用{}对变量进行列表初始化或赋值。它是防窄化的。
即：

- 如果{}中的初始值存在丢失信息的风险，将出现编译错误；而只用花括号中的（类型不匹配）值只会出现编译警告。例如

```
double d = 92.221, d1{ 92.221 }, d3{ d };
```

```
int x = d;
```

```
x = { 32 };
```

11C₊₊

```
d = { 32 };
```

11C₊₊

```
d = x;
```

```
d = { x };
```

//错误

11C₊₊

```
int y = { d };
```

//错误

11C₊₊

```
int z={1.3};
```

//错误

11C₊₊

2.13.4 变量初始化

4. 变量初始化的默认规则

- ① 如果定义变量时提供了初始值表达式，系统就用这个表达式的值作为变量的初值；
- ② 如果定义变量时没有为它提供初值，则全局数据区中的变量将被系统自动初始化为0，栈和堆中的变量不被初始化。
- ③ 全局变量、命名空间的变量、静态变量会被保存在全局数据区中，所以它们会被系统自动初始化为0；
- ④ 局部变量（也叫自动变量）被存储在栈区中，动态分配的变量（用malloc和new建立）被存储在堆区中，它们都不会被系统用默认值初始化。

2.13.4 变量初始化

【例2-44】 全局变量、静态变量、局部变量的初始化。

//CH2-44.cpp

#include <iostream.h>

int n;

//初始化为0

void f(){

static int i;

//初始化为0

int j;

//不被初始化, j值未知

cout<<"i="<<i<<" ";

cout<<"j="<<j<<endl;

}

int *p1;

//p1被初始为0

void main(){

int *p2;

//p2不被初始化, 值未知

int m;

//m不被初始化, 值未知

f();

//输出i=0, j=?, ?表示不确定值

cout<<"n="<<n<<endl;

//输出n=0

cout<<"m="<<m<<endl;

//输出m=?, ?表示不确定值

if(p1) cout<<"p1="<<p1<<endl;

//p1=0, 无输出

if(p2) cout<<"p2="<<p2<<endl;

//输出p2=?, ?表示不确定地址

}

作业

- 本章作业2周内完成
- 课后作业（教学立方提交）
 - 2道单选，3道多选，4道简答(课本)
 - 简单题：课本P104 2.2、2.4、2.5、2.11(1-4)
注意：2.11(1)最后一行`print(l)`修改为`print(int(l))`;
- 实验作业（单独发布在QQ群）