C++程序设计精要教程

华中科技大学

第2章 类型、常量及变量

◆2.1 C++的单词

单词包括常量、变量名、函数名、参数名、类型名、运算符、关键字等。

关键字也被称为保留字，不能用作变量名。

予定义类型如int等也被当作保留字。

char16_t表示双字节字符类型，支持UTF-16。

char32_t表示四字节字符类型，支持UTF-32。

```
char16_t x = u'马'; char32_t y = U'马';
```

wchar_t表示char16_t，或char32_t。

nullptr表示空指针。

保留字表

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	double	int	signed	typename
break	do	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	

类型的作用

数据类型是任何一种编程语言的基础，它告诉我们数据在内存中的**表示方式**，以及我们能在类型上做的操作。

每个字节有地址

内存最小可寻址的单位为字节(byte)，8个bit

736424	0	0	0	0	0	0	0	0
736425	0	0	0	0	0	0	0	0
736426	0	0	0	0	0	0	0	0
736427	0	0	0	0	0	0	0	0

数据类型决定了数据所占的字节数以及如何解释这些字节的内容。

如果定义变量short i = 0，并且变量i的首地址为736424，那么CPU会读取736424开始的2个字节的内容（假设short为2个字节）

如果定义变量inti = 0，并且变量i的首地址为736424，那么CPU会读取736424开始的4个字节的内容（假设int为4个字节）

类型的作用

数据类型是任何一种编程语言的基础，它告诉我们数据在内存中的表示方式，以及我们能在类型上做的操作。

对于表达式i+j，如果i和j的类型是整型，那么这个表达式执行的是普通的的加法运算。

对于表达式i+j，如果i和j的类型是对象类型（比如Circle类型），那么这个表达式执行的是Circle类重载的+号运算符语义（Circle类必须要重载的+号运算符，否则表达式i+j会报编译时错误）

std::string s1 = “ab”,s2 = “12”;std::string s3 = s1+s2;

C++内置数据类型(C++11标准3.9节)

标准只规定了每种数据类型的最小尺寸，每种类型的具体字节数依赖于不同平台上的编译器实现

C++标准规定的内置数据类型

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

C++内置数据类型

需要特别注意的是：char可以显示地声明为带符号的和无符号的。因此C++11标准规定char，signed char和unsigned char是三种不同的类型。

但每个具体的编译器实现中，char会表现为signed char和unsigned char中的一种。

第2章 类型、常量及变量

◆2.2 预定义类型（内置数据类型）及值域和常量

- 类型的字节数与硬件、操作系统、编译有关。假定VS2019采用X86编译模式。
- void：字节数不定,void *p。常表示函数无参或无返回值。
- bool：单字节布尔类型，取值false和true。
- char：单字节有符号字符类型，取值-128～127。
- short：两字节有符号整数类型，取值-32768～32767。
- int：四字节有符号整数类型，取值-2^{31}～$2^{31}-1$。
- long：四字节有符号整数类型，取值-2^{31}～$2^{31}-1$。
- float：四字节有符号单精度浮点数类型，取值-10^{38}～10^{38}。
- double：八字节有符号双精度浮点数类型，取值-10^{308}～10^{308}。

注意：默认一般整数常量当作为int类型，浮点常量当作double类型。

第2章 类型、常量及变量

◆2.2 预定义类型及值域和常量

- char、short、int、long前可加unsigned表示无符号数。
- long int等价于long；long long占用八字节。
- 自动类型转换路径（数值表示范围从小到大）：char→unsigned char→short→unsigned short→ int→unsigned int→long→unsigned long→float→double→long double。
- 数值零自动转换为布尔值false，数值非零转换为布尔值true。
- 强制类型转换的格式为：(类型表达式) 数值表达式
- 字符常量：‘A’，‘a’，‘9’，‘\’’(单引号)，‘\\’(斜线)，‘\n’(换新行)，‘\t’(制表符)，‘\b’(退格)
- 整型常量：9,04,0xA(int); 9U,04U,0xAU(unsigned int); 9L,04L,0xAL(long); 9UL, 04UL,0xAUL(unsigned long), 9LL,04LL,0xALL(long long);

第2章 类型、常量及变量（课堂略）

预定义类型的数值输出格式化，如:#include<stdio.h>后printf(“%d”,4);

- double常量：0.9, 3., .3, 2E10, 2.E10, .2E10, -2.5E-10
- char: %c; short, int: %d; long：%ld; 其中%开始的输出格式符称为占位符。
- 输出无符号数用u代替d(十进制)，八进制数用o代替d，十六进制用x代替d
- 整数表示宽度如printf(“%5c”，‘A’)打印字符占5格(右对齐)。%-5d表示左对齐。
- float：%f; double：%lf。float, double：%e科学计数。%g自动选宽度小的e或f。
- 可对%f或%lf设定宽度和精度及对齐方式。“%-8.2f”表示左对齐、总宽度8(包括符号位和小数部分)，其中精度为2位小数。
- 字符串输出：%s。可设定宽度和对齐：printf(“%5s”,”abc”)。
- 字符串常量的类型：指向只读字符的指针即const char *, 上述”abc“的类型。
- 注意strlen(“abc”)=3，但要4个字节存储，最后存储字符‘\0’，表示串结束。

第2章 类型、常量及变量

◆2.3 变量及其类型解析

- 变量说明：描述变量的类型及名称，但没有初始化。可以说明多次。
- 变量定义：描述变量的类型及名称，同时进行初始化。只能定义一次。
- 说明例子：extern int x; extern int x; //变量可以说明多次
- 定义例子：int x=3; extern int y=4; int z; //全局变量z的初始值为0
- 模块静态变量：使用static在函数外部定义的变量，只在当前文件（模块）可用。可通过单目::访问。
- 局部静态变量：使用static在函数内部定义的变量。

```
static int x, y; //模块静态变量x、y定义，默认初始值均为0
int main( ){
   static int y;     //局部静态变量y定义，  初始值y=0
   return  ::y+x+y;//分别访问模块静态变量y,模块静态变量x,局部静态变量
}
```

变量的声明和定义（C++标准3.1节）

为了允许把程序拆分成多个逻辑部分来编写，C++支持分离式编译(separation compilation),即将程序分成多个文件，每个文件独立编译。

为了支持分离式编译，必须将声明(Declaration)和定义(Definition)区分开来。声明是使得名字（Identifier，如变量名）为其它程序所知。而定义则负责创建与名字（Identifier）相关联的实体，如为一个变量分配内存单元。因此只有定义才会申请存储空间。

One definition rule（ODR）：只能定义一次，但可以多次声明

变量的声明和定义（C++11标准3.1节）

如果想要声明一个变量而非定义它，就在前面加关键字extern，而且不要显示地初始化变量：

```
extern int i;           //变量的声明
int i; //变量的定义（虽没有显示初始化，但会有初始值并分配内存单元存储，即使初始值随机）
```

任何包含显式地初始化的声明成为定义。如果对extern的变量显式初始化，则extern的作用被抵消。

```
extern int i = 0;       //变量的定义
```

声明和定义的区别非常重要，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，变量的定义必须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声明，而不能定义。例如，头文件里不要放定义，因为头文件可能会被到处include，导致定义多次出现。

变量的声明和定义（C++11标准3.1节）

Test.h

```
#pragma once

int i = 0; //在头文件里定义变量
```

Test.cpp

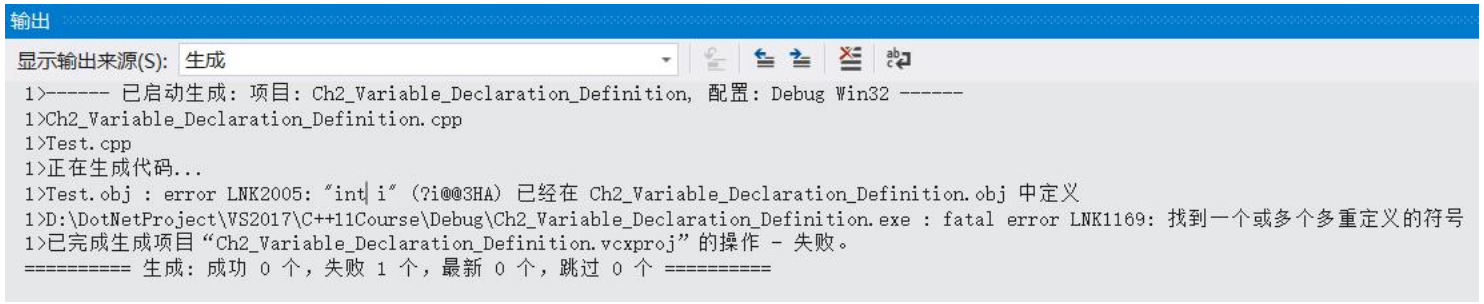
```
#include "pch.h"
#include <iostream>
#include "Test.h"      //Test.cpp包含了Test.h头文件
int j = ::i;
```

Ch2_Variable_Declaration_Definition.cpp

```
#include "pch.h"
#include <iostream>
#include "Test.h"  //Ch2_Variable_Declaration_Definition.cpp也包含了Test.h头文件

int main(){
    std::cout << "Hello World!\n";
}
```

由于头文件Test.h分别在二个cpp文件里二次被包含，因此头文件里定义 的变量在二个cpp文件里被定义二次，会报链接错误



变量的初始化（C++11标准8.5节）

变量在创建时获得一个值，我们说这个变量被初始化了(initialized)。最常见的形式为：

类型 变量名＝表达式；//表达式的求值结果为变量的初始值

用＝来初始化的形式让人误以为初始化是赋值的一种。其实完全不同：初始化的含义是创建变量时设定一个初始值；而赋值的含义是将变量的当前值擦除，而以一个新值来替代。

实际上，C++定义了多种初始化的形式：

```
int a = 0;
int b = { 0 };
int c(0);
int d{ 0 };
```

其中用{}来初始化作为C++11的新标准一部分得到了全面应用（而以前只在一些场合使用，例如初始化结构变量）。这种初始化形式称为列表初始化(list initialization)。

变量的初始化（C++11标准8.5节）

对于内置数据类型，如果存在丢失信息（损失精度）的危险，列表初始化会报编译错误,而其他形式只会报警告错误

```
long double ld = 3.1415926536;
int m = ld;     //警告
int n(ld);      //警告
int k = { ld }; //错误，更安全
int l{ld};      //错误，更安全
```

变量的初始化（C++11标准8.5节）

如果定义变量时没有指定初值，则变量被赋予默认值,称为默认初始化（default initialized），默认值取决于变量位置和变量类型。

- 内置数据类型的全局变量（定义在任何函数体之外）被默认初始化为0
- 内置类型的局部静态变量默认初始化为0
- 函数内部的内置类型的局部变量不被初始化（uninitialized）,其值是未确定的，试图访问这样的未初始化的局部变量会报编译错误
- 类对象的默认值取决于构造函数的定义

```
int ga;    //内置类型的全局变量默认初始化为0
void f() {
        std::cout << ga << std::endl;        //输出0
        int la;    //内置类型的局部变量未初始化，其值未定义
        //std::cout << la << std::endl;//错误，未初始化的局部变量不能访问
        static int sla;        //内置类型的局部静态变量默认初始化为0
        std::cout << sla << std::endl;        //输出0
}
```

第2章 类型、常量及变量

◆2.3 变量及其类型解析

- 只读变量：使用const或constexpr说明或定义的变量，定义时必须同时初始化。当前程序只能读不能修改其值。constexpr变量必须用编译时可计算表达式初始化。
- 易变变量：使用volatile说明或定义的变量，可以后初始化。当前程序没有修改其值，但是变量的值变了。不排出其它程序修改。
- const实例：extern const int x; const int x=3; //定义必须显式初始化x
- volatile例： extern volatile int y; volatile int y; //可不显式初始化y，全局y=0
- 若y=0，语句if(y==2)是有意义的，因为易变变量y可以变为任何值。

constexpr（自学）

常量表达式：是指值不会改变而且在编译时可以求值的表达式，如字面量就是常量表达式，用常量表达式初始化的const变量也是常量表达式。

```
const int max = 20;  // max是常量表达式
const int limit = max + 1;  // limit是常量表达式
int i = 0;                  //i不是常量表达式，非const
const int  min = get_min(); //min的值只有运行是才能确定，因此不是
```

由于人有时很难判断一个表达式是否为常量表达式，因此costexper粉墨登场

constexpr变量：11标准可以将变量声明为constexpr类型以便由编译器检测变量的值是否为常量表达式（由编译器帮助我们来判断）。constexpr变量必须用常量表达式初始化，且一定是常量。

```
constexpr int mf = 0;          //mf是常量表达式
constexpr int limit = mf + 1;  //limit是常量表达式
//constexpr int sz2 = f1();//编译报错，sz2不是常量表达式，只有当函数f1是
```

constexpr函数时才正确

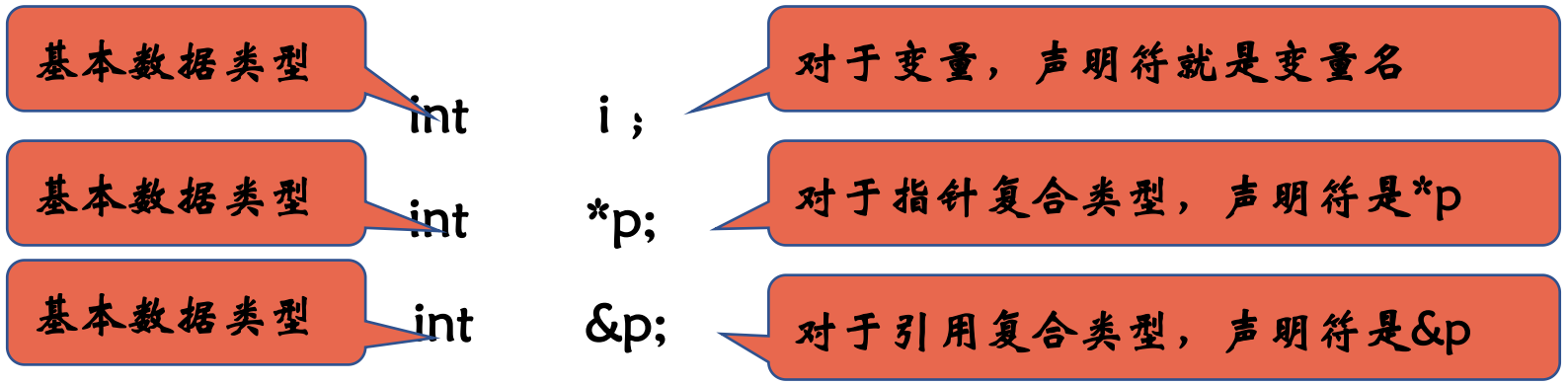
第2章 类型、常量及变量

◆2.3 变量及其类型解析

- 在多任务环境下，定义“const volatile int z=0;”是由意义的，不排除其它程序修改z使其值易变。
- 作为类型修饰符，const和volatile可以定义函数参数和返回值。
- 保留字inline用于定义函数外部变量或函数外部静态变量、类内部的静态数据成员。
- inline函数外部变量的作用域和inline函数外部静态变量一样,都是局限于当前代码文件的，相当于默认加了static。
- 用inline定义的变量可以使用任意表达式初始化。

复合类型(C++11标准3.9.2)

复合类型(Compound Type)是指基于其他类型定义的类型，例如指针、引用。和内置数据类型变量一样，复合类型也是通过声明语句(declaration)来声明。一条声明语句由基本数据类型和跟在后面的声明符列表(declarator list)组成。每个声明符命名一个变量并指定该变量为与基本数据类型有关的某种类型。



复合类型的声明符基于基本数据类型得到更复杂的类型，如*p，&p，分别代表指向基本数据类型变量的指针和指向基本数据类型变量的引用。*，&是类型修饰符。

复合类型(C++11标准3.9.2)

在同一条声明语句中，基本数据类型只有一个，但声明符可以有多个且形式可以不同，即一条声明语句中可以声明不同类型的变量：

```
int    i, *p, &r; //i是int变量，p是int指针，r是int引用
```

正确理解了上面的定义，就不会对下面的声明语句造成误解：

```
int *  p1，p2; //p1是int指针，p2不是int指针，而是int变量
```

为了避免类似的误解，一个好的书写习惯是把类型修饰符和变量名放连一起：

```
int  *p1，p2，&r1;
```

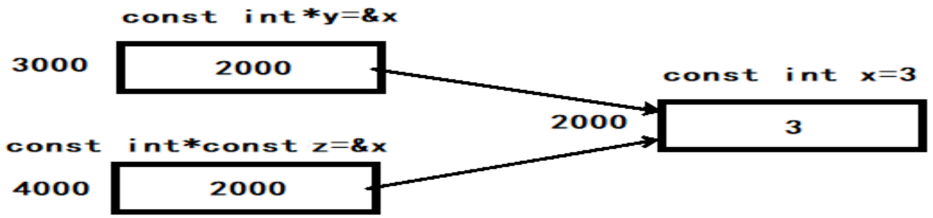
第2章 类型、常量及变量

指针及其类型理解

- 指针类型的变量使用*说明和定义，例如：int x=0; int *y=&x;。
- 指针变量y存放的是变量x的地址，&x表示获取x的地址运算，表示y指向x。
- 指针变量y涉及两个实体：变量y本身，y指向的变量x。
- 变量x、y的类型都可以使用const、volatile以及const volatile修饰。

```
const int x=3;              //不可修改x的值
const int *y=&x;            //可以修改y的值，但是y指向的const int实体不可修改
const int *const z=&x;      //不可修改z的值，且z指向的const int实体也不可改
```

- x的值为3，地址为2000。
- y和z的值均为2000，表示y和z都指向x。
- y可被修改指向别的变量，但z不行。



第2章 类型、常量及变量

指针及其类型理解

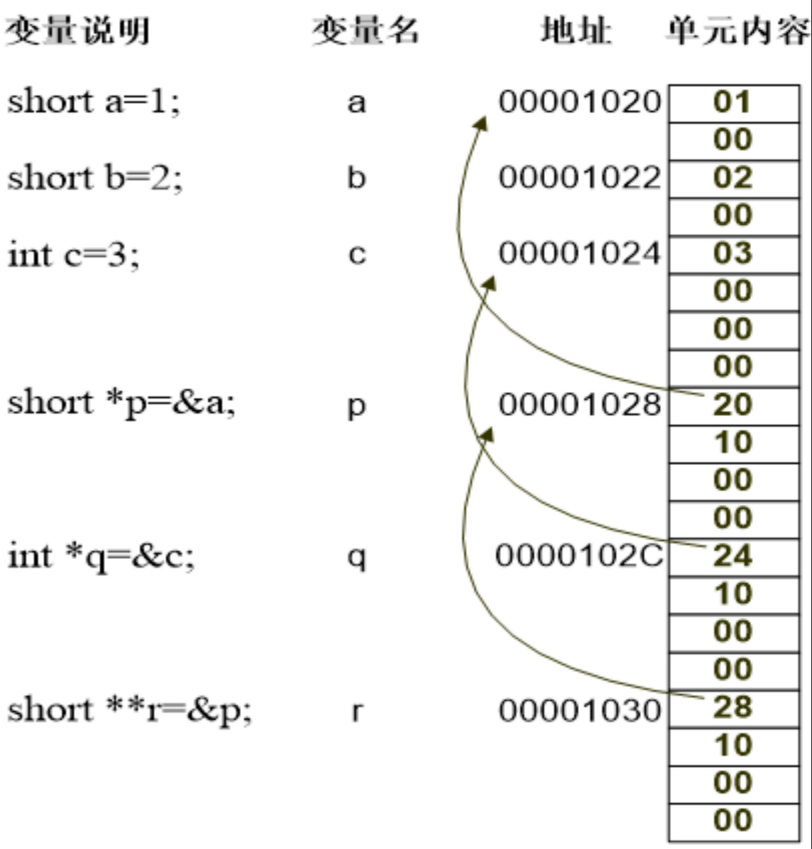
```
//p1是指针，p2是int变量
//和引用不同，指针不用马上初始化，但很危险，不建议这样
int *p1, p2;
int i=0, j= 0;
p1 = &i; //p1这时指向（绑定到）i，注意&i是右值
p1 = &j ; //p1这时可重新指向（绑定到）j，这点和引用不同
*p1 = 10; //改变被绑定对象的值，这时*为解引用操作符
```

第2章 类型、常量及变量(课堂略)

指针及其类型理解

- 指针变量还可以指向指针变量，从而形成多重指针。
- 例如：　short *p=&a定义p是一个指针变量
  short **r=&p定义r是一个双重指针变量
  变量r存储的是p的地址00001028
- VS2019在X86编译模式下，使用4个字节表示地址
- 根据表2.7有关*的结合性“自右向左”，故先解释右边的指针，再向左解释左边的指针，如下图所示：

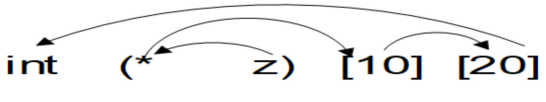
- *r取r指向的单元的值00001020(a的地址)，故**r才能取a的值1，而**r=7修改a



第2章 类型、常量及变量

指针及其类型理解

- 在一个类型表达式中，先解释优先级高的，若优先级相同，则按结合性解释。
- 如：int *y[10][20]; 在y的左边是*，右边是[10]，据表2.7知[ ]的优先级更高。



(b)定义int (*z)[10][20]

- 解释: (1) y是一个10元素数组；(2)每个数组元素均为20元素数组
  (3) 20个元素中的每个元素均为指针int *
- 但括号()可提高运算符的优先级，如：int (*z)[10][20];
- (…)、[10]、[20]的运算符优先级相同，按照结合性，应依次从左向右解释。
- 因此z是一个指针，指向一个int型的二维数组，注意z与y的解释的不同。
- 指针移动：y[m][n]+1移动到int *指针指向的下一整数，z+1移动到下一10*20整数数组。

第2章 类型、常量及变量

指针使用注意事项

- 只读单元的指针(地址)不能赋给指向可写单元的指针变量。
- 例如：const int x=3; const int *y=&x; //x是只读单元，y是x的地址

  int *z=y; //错：y是指向只读单元的指针

  z=&x; //错：&x是是只读单元的地址

  证明：(1)假设int *z=&x正确（应用反正法证明）

  (2)由于int *z表示z指向的单元可写，故*z=5是正确的

  (3)而*z修改的实际是变量x的值，const int x规定x是不可写的。矛盾。
- 可写单元的指针(地址)能赋给指向只读单元的指针变量：y=z;
- 前例的const换成volatile或者const volatile，结论一样。
- int *可以赋值给const int *，const int *不能赋值给int *

第2章 类型、常量及变量

指针使用注意事项

- 除了二种例外情况，指针类型类型都要与指向(绑定)的对象严格匹配：二种例外是：
  - 指向常量的指针（如const int *）可以指向同类型非常量
  - 父类指针指向子类对象

```
double d = 0.0;
int i = 0;

double *pd = &d;          //正确，类型一致
//int *pi = &d;           //错误，类型不一致

const int *pc = &i;       //正确，const int类型的指针可以指向int（例外）
//const int *pc2 = &d;    //错误，const int类型的指针不能指向double
```

第2章 类型、常量及变量

指针使用注意事项

- 一个没有初始化是非常危险的。当不清楚指针应该指向何处时，应该将指针初始化为空指针。可以用三种办法：
  - 用字面量nullptr来初始化，这是C++11新标准引入的,推荐
  - 用字面量0初始化
  - 用预定义符号NULL来初始，但必须先include <cstdlib>,因此不推荐

```
int *p1 = nullptr;  //等价于int *p1 = 0;

int *p2 = 0;

int *p3 = NULL;
```

第2章 类型、常量及变量

指针使用注意事项

void用于表示函数无参或者无返回值。void *p所指向的实体单元字节数不定（泛型指针）。

- 因此，p可以指向任何实体，即可以将任何指针和任意类型变量的地址赋给p，或者调用时通过值参传递给参数p (如果p是函数参数) 。

```
void *p; int i = 0; int * q = &i; p = q;
void f(void *p); f(q); //实参传递给形参等价于void *p = q;
```

请大家查阅释放内存的函数free的原型，参数为 void *p

- 对void *p指向的实体单元赋值时，类型或字节数必须确定，所以必须进行强制类型转换

```
int x = 0;
void *p = &x;//可以将任意类型的指针或变量地址赋给p
*p = 345; //错误，p指向的单元字节数不确定
*(int *)p = 345; //OK          *(double *)p = 12.3; //OK
```

第2章 类型、常量及变量

引用

引用(reference)为变量起了一个别名，引用类型变量的声明符用&来修饰变量名：

```
int i = 10; //i为一个整型变量
```

基本数据类型

引用声明符

```
int &r = i; //定义一个引用变量r，引用了变量i，r是i的别名
```

定义引用变量时必须马上初始化，即马上指明被引用的变量。

```
int &r2; //编译错误，引用必须初始化，即指定被引用变量
```

C++11增加了一种新的引用：右值引用(rvalue reference)，课本上叫无址引用，用&&定义. 当采用术语引用时，我们约定都是指左值引用(lvalue reference)，课本上叫有址引用，用&定义。关于右值引用，会在后续介绍。

左值和右值(C++11标准3.10)

C++表达式的求值结果要不是左值(lvaue)，要不是右值(rvalue)。在C语言里，左值可以出现在赋值语句的左侧(当然也可以在右侧)，右值只能出现在赋值语句的右侧。

但是在C++中，情况就不是这样。C++中的左值与右值的区别在于是否可以寻址：可以寻址的对象（变量）是左值，不可以寻址的对象（变量）是右值。这里的可以寻址就是指是否可以用&运算符取对象（变量）的地址。

```
int i = 1;              //i可以取地址，是左值；1不可以取地址，是右值
//3 = 4;                //错误：3是右值，不能出现在赋值语句左边
const int j = 10;       //j是左值,j可以取地址
const int *p = &j;
// j = 20;              //错误：const左值（常量左值）不能出现在赋值语句左边
```

非常量左值可以出现在赋值运算符的左边，其余的只能出现在右边。右值出现的地方都可以用左值替代。

左值和右值(C++11标准3.10)

区分左值和右值的另一个原则就是：**左值持久、右值短暂**。左值具有持久的状态（取决于对象的生命周期），而右值要么是字面量，要么是在表达式求值过程中创建的临时对象。

```
int i = 1;          //i的状态持久，而字面量1的生命周期随着语句的结束而结束
// i++ = 1;         //错误：i++是右值。解释见下面
++i = 1;            //正确：++i是左值
```

```
//i++等价于用i作为实参调用下列函数
//第一个参数为引用x，引用实参，因此x = x + 1就是将实参+1；第二个int参数只是告诉编译器是后置++
int operator++(int &x, int)  {
   int tmp= i; //先取i的值赋给temp
   x = x + 1;
   return tmp;
}
```

因此i++ = 1不成立，因为1是要赋值给函数的返回值，而函数返回后，tmp生命周期已经结束，不能赋值给tmp

i++等价于operator++(i, int),实参i传递给形参x等价于int &x = i;

左值和右值(C++11标准3.10)

区分左值和右值的另一个原则就是：**左值持久、右值短暂**。左值具有持久的状态（取决于对象的生命周期），而右值要么是字面量，要么是在表达式求值过程中创建的临时对象。

```
int i = 1;          //i的状态持久，而字面量1的生命周期随着语句的结束而结束
// i++ = 1;         //错误：i++是右值，因为i++等价于下列语句
++i = 1;            //正确：++i是左值
```

++i等价于用i作为实参调用下列函数

```
//第一个参数为引用x，引用实参，因此x = x + 1就是将实参+1
int& operator++(int &x)
{
   x = x + 1;
   return x;
}
```

实参i传递形参x，等价于int &x = i; x引用了i，所以x = x + 1就是将i+1，最后返回x就是返回i的引用

++i = 1就是将1赋值给i的引用，也就是把1赋值给i，因此最后i= 1

左值和右值(C++11标准3.10)

区分左值和右值的另一个原则就是：**左值持久、右值短暂**。左值具有持久的状态（取决于对象的生命周期），而右值要么是字面量，要么是在表达式求值过程中创建的临时对象。

```
int i = 1;
int f( ){
   int tmp = i;
   return tmp;
}
//f( ) = 10; //错误
```

由于返回的是值类型，因此当f结束后，tmp生命周期结束。所以函数f返回的是右值，因此f()只能在=右边

```
int i = 1;
int &f( ){
   return i;
}
f( ) = 10; //正确 。最后i= 10
```

由于返回的是引用类型，当{}结束后，i的生命周期没有结束，是持久的，函数f返回的是左值

因此返回**值类型**的函数调用结果是**右值**，不能出现在赋值语句的左边。

返回**非const引用**的函数调用结果是**左值**，可以出现在赋值语句的左边。

引用的本质

引用的本质还是指针，考查以下C++代码及其对应的汇编代码

```
int i = 10;
int &ri = i;
ri = 20;
```

从汇编代码可以看到，引用变量ri里存放的就是i的地址

```
            int i = 10;
mov         dword ptr [i],0Ah     //将文字常量10送入变量i

            int &ri = i;
lea         eax,[i]               //将变量i的地址送入寄存器eax
mov         dword ptr [ri],eax    //将寄存器的内容（也就是变量i的地址）送入变量ri

            ri = 20;
mov         eax,dword ptr [ri]    //将变量ri的值送入寄存器eax
mov         dword ptr [eax],14h //将数值20送入以eax的内容为地址的单元中
```

引用的本质

引用的本质还是指针，考查以下C++代码及其对应的汇编代码

```
int j = 10;
int  *const pj = &j;
*pj = 20;
```

从汇编代码可以看到，指针pj里存放的就是j的地址。二段汇编代码一样，引用变量在功能上等于一个常量指针

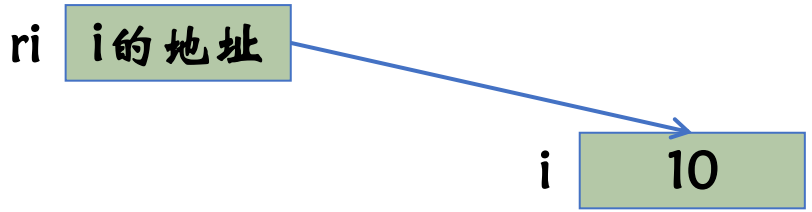
```
            int j = 10;
mov         dword ptr [j],0Ah     //将文字常量10送入变量j

            int *const pj = &j;
lea         eax,[j]               //将变量j的地址送入寄存器eax
mov         dword ptr [pj],eax    //将寄存器的内容（也就是变量j的地址）送入变量pj

            *pj = 20;
mov         eax,dword ptr [pj]    //将变量pj的值送入寄存器eax
mov         dword ptr [eax],14h //将数值20送入以eax的内容为地址的单元中
```

引用的本质

因此，引用变量ri的内存布局如图所示。



但是，为了消除指针操作的风险（例如指针可以++，--），引用变量ri的地址不能由程序员获取，更不允许改变ri的内容。

由于引用本质上是常量指针，因此凡是指针能使用的地方，都可以用引用来替代，而且使用引用比指针更安全。例如Java、C#里面就取消了指针，全部用引用替代。

引用的本质

引用与指针的区别是：

- 引用在逻辑上是“幽灵”，是不分配物理内存的，因此无法取得引用的地址，也不能定义引用的引用，也不能定义引用类型的数组。引用定义时必须初始化，一旦绑定到一个变量，绑定关系再也不变（常量指针一样）。
- 指针是分配物理内存的，可以取指针的地址，可以定义指针的指针（多级指针），指针在定义时无需初始化（但很危险）。对于非常量指针，可以被重新赋值（指向不同的对象，改变绑定关系），可以++，--（有越界风险）

对引用的操作

定义了引用后，对引用进行的所有操作实际上都是作用在与之绑定的对象之上。被引用的实体必须是分配内存的实体（能按字节寻址）

- 寄存器变量可被引用，因其可被编译为分配内存的自动变量。
- 位段成员不能被引用，计算机没有按位编址，而是按字节编址。注意有址引用被编译为指针，存放被引用实体内存地址。
- 引用变量不能被引用。对于int x; int &y=x; int &z=y; 并非表示z引用y，int &z=y表示z引用了y所引用的变量i。

对引用操作示例

```
struct A {
        int j : 4;    //j为位段成员
        int k;
} a;
void f() {
        int i = 10;
        int &ri = i;          //引用定义必须初始化，绑定被引用的变量
        ri = 20;              //实际是对i赋值20
        int *p = &ri;         //实际是取i的地址，p指向i，注意这不是取引用ri的地址
        //int &*p = &ri;      //错误：不能声明指向引用的指针
        //int & &rri = ri;    //错误：不能定义引用的引用
        //int  &s[4];         //错误：数组元素不能为引用类型，否则数组空间逻辑为0

        register int i = 0, &j = i;     //正确：i、j都编译为(基于栈的)自动变量
        int  t[6], (&u)[6] = t;         //正确：有址引用u可引用分配内存的数组t
        int  &v = t[0];       //正确：有址引用变量v可引用分配内存的数组元素
        //int  &w = a.j;      //错误：位段不能被有址引用，按字节编址才算有内存
        int  &x = a.k;        //正确：a.k不是位段有内存
}
```

引用初始化

引用初始化时，除了二种例外情况，引用类型都要与绑定的对象严格匹配：即必须是用求值结果类型相同的左值表达式来初始化。二种例外是：

- const引用
- 父类引用绑定到子类对象

```
int j = 0;
const int c = 100;
double d = 3.14;
int &rj1 = j;                    //用求值结果类型相同的左值表达式来初始化
//int &rj2 = j + 10;             //错误：j + 10是右值表达式
//int &rj3 = c;                  //错误：c是左值，但类型是const int，类型不一致
//int &rj4 = j++;                //错误：j++是右值表达式
//int &rd = d;                   //错误：d是double类型左值，类型不一致
```

引用初始化

而const引用则是万金油，可以用类型相同（如类型不同，看编译器）的左值表达式和右值表达式来初始化。

```
int j = 0;
const int c = 100;
double d = 3.14;
const int &cr1 = j;          //常量引用可以绑定非const左值
const int &cr2 = c;                  //常量引用可以绑定const左值
const int &cr3 = j + 10;     //常量引用可以绑定右值
const int &cr4 = d;          //类型不一致，报警告错误 (VS2017)

int &&rr = 1;                //rr为右值引用
const int &cr5 = rr;         //常量引用可以绑定同类型右值引用
```

引用指针(自学）

如何引用指针

```
int ival = 1024;
int * &pi_ref = &ival;
```

//错误， &ival为右值,不能初始化非const引用

因为只有const类型引用才能用右值初始化，因此必须要把pi_ref声明为const引用

```
const int * &pi_ref = &ival; // 错误，为什么？
```

//pi_ref还是非const引用，引用了一个const int *指针，该指针指向了const对象

```
int *  const &pi_ref = &ival; //OK，这时pi_ref才是const 引用
```

引用初始化（自学）

为什么非const引用不能用右值或不同类型的对象初始化？

对不可寻址的右值或不同类型对象，编译器为了实现引用，必须生成一个临时（如常量）或不同类型的值，对象，引用实际上指向该临时对象，但用户不能通过引用访问。如当我们写

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

编译器将其转换成

```
int temp = dval;  //注意将dval转换成int类型
int &ri = temp;
```

如果我们给ri赋给新值，改变的是temp而不是dval。对用户来说，感觉赋值没有生效（这不是好事）。

const引用不会暴露这个问题，因为它本来就是只读的。

干脆禁止用右值或不同类型的变量来初始化非const引用比“允许这样做，但实际上不会生效”的方案好得多。

第2章 类型、常量及变量

&定义的有址引用（左值引用）

- const和volatile有关指针的用法可推广至&定义的(左值）引用变量
- 例如：“只读单元的指针(地址)不能赋给指向可写单元值的指针变量”推广至引用为“只读单元的引用不能初始化引用可写单元的引用变量”。如前所述，反之是成立的。
- int &可以赋值给const int &，const int &不能赋值给int &

```
const int &u=3;  //u是只读单元的引用
int &v=u;        //错：u不能初始化引用可写单元的引用变量v
int x=3; int &y=x;//对：可进行y=4，则x=4。
const int &z=y;  //对：不可进行z=4。但若y=5，则x=5, z=5。
volatile int &m=y;//对，m引用x。
```

第2章 类型、常量及变量

&&定义的无址引用（右值引用）

- 右值引用：就是必须绑定到右值的引用。
- 右值引用的重要性质：只能绑定到即将销毁的对象，包括字面量，表达式求值过程中创建的临时对象。
- 返回非引用类型的函数、算术运算、布尔运算、位运算、后置++，后置--都生成右值，右值引用和const左值引用可以绑定到这些运算的结果上。
- c++ 11中的右值引用使用的修饰符是&&，如：
- int &&aa = 1; //实质上就是将不具名(匿名)变量取了个别名
- aa = 2; //可以。匿名变量1的生命周期本来应该在语句结束后马上结束，但是由于被右值引用变量引用，其生命期将与右值引用类型变量aa的生命期一样。这里aa的类型是右值引用类型(int &&)，但是如果从左值和右值的角度区分它，它实际上是个左值

第2章 类型、常量及变量

&&定义的无址引用（右值引用）

- &&定义右值引用变量，必须引用右值。如int &&x=2;
- 注意，以上x是右值引用（引用了右值），但它本身是左值，即可进行赋值：x=3;
- 但：const int &&y=2; //不可赋值: y=3;
- 同理：右值引用共享被引用对象的“缓存”，本身不分配内存。

```
int && *p;          //错：p不能指向没有内存的无址引用
int && &q;          //错：int &&没有内存，不能被q引用
int & &&r;          //错：int &没有内存，不能被r引用。
int && &&s;         //错：int &&没有内存，不能被s引用
int &&t[4];         //错：数组的元素不能为int &&：数组内存空间为0。
const int a[3]={1,2,3};  int(&& t)[3]=a; //错：a是有址的，有名的均是有址的。&&不能引用有址的
int(&& u)[3]= {1,2,3};   //正确，{1，2，3}是无址右值
```

右值引用

```
int b = 1;
//int && c = b; //编译错误！ 右值引用不能引用左值

A getTemp() { return A(); }
A o = getTemp();  // o是左值 getTemp()的返回值是右值（临时变量），被拷贝给o，会引起对象的拷贝

// getTemp()返回的右值本来在表达式语句结束后，其生命也就该终结了，而通过右值引用，该右值又重获新生，其生命期将与右值引用类型变量refO的生命期一样，只要refO还活着，该右值临时变量将会一直存活下去。
A && refO = getTemp();  //getTemp()的返回值是右值（临时变量）,可以用右值引用,但不会引起对象的拷贝

//注意：这里refO的类型是右值引用类型(A &&)，但是如果从左值和右值的角度区分它，它实际上是个左值(其生命周期取决于refO）。因为可以对它取地址，而且它还有名字，是一个已经命名的左值。因此
A *p = &refO;
//不能将一个右值引用绑定到一个右值引用类型的变量上
//A &&refOther = refO; //编译错误，refO是左值
```

第2章 类型、常量及变量

&&定义的无址引用（右值引用）

- 若函数不返回（左值）引用类型，则该函数调用的返回值是无址（右值）的；

```
int &&x=printf( “abcdefg” ); //对：printf( )返回无址右值
int &&a=2;          //对：引用无址右值
int &&b=a;          //错：a是有名有址的，a是左值
int&& f( ) { return 2; }
int &&c=f( );       //对：f返回的是无址引用，是无址的
```

- 位段成员是无址的。

```
struct A { int a;   /*普通成员：有址*/   int b : 3; /*位段成员：无址*/ }p = { 1,2 };
int &&q=p.a;        //错：不能引用有址的变量，p.a是左值
int &&r=p.b;        //对：引用无址左值
```

第2章 类型、常量及变量

枚举

- 枚举一般被编译为整型，而枚举元素有相应的整型常量值；
- 第一个枚举元素的值默认为0，后一个元素的值默认在前一个基础上加1

```
enum  WEEKDAY {Sun, Mon, Tue, Wed, Thu, Fri, Sat};  //Sun=0, mon=1
WEEKDAY  w1=Sun, w2(Mon);        //可用限定名WEEKDAY::Sun,
```

- 也可以为枚举元素指定值，哪怕是重复的整数值。

```
enum  E{e=1,  s,  w= -1,  n,  p};  //正确, s=2, p= 1和e相等
```

- 如果使用“enum class”或者“enum struct”定义枚举类型，则其元素必须使用类型名限定元素名

```
enum struct RND{e=2, f=0, g, h};     //正确：e=2，f=0，g=1，h= 2
RND m= RND::h;                       //必须用限定名RND::h
int  n=sizeof(RND::h);               //n=4, 枚举元素实现为整数
```

第2章 类型、常量及变量

元素、下标及数组

- 数组元素按行存储，对于“int a[2][3]={{1,2,3},{4,5,6}};”，先存第1行再存第2行

  a: 1, 2, 3, 4, 5, 6 //第1个元素为a[0][0]，第2个为a[0][1]，第4个为a[1][0]
- 若上述a为全局变量，则a在数据段分配内存，1,2…6等初始值存放于该内存。
- 若上述a为静态变量，则a的内存分配及初始化值存放情况同上。
- 若上述a函数内定义的局部非静态变量，则a的内存在栈段分配
- C++数组并不存放每维的长度信息，因此也没有办法自动实现下标越界判断。每维下标的起始值默认为0。
- 数组名a代表数组的首地址，其代表的类型为int [2][3]或int(*)[3]。

第2章 类型、常量及变量

元素、下标及数组

- 一维数组可看作单重指针，反之也成立。例如：

```
int b[3];                    //*(b+1)等价于访问b[1]
int *p=&b[0];                //*(p+2)等价访问p[2]，也即访问b[2]
```

- 字符串常量可看做以’\0’结束存储的字符数组。例如“abc”的存储为

‘a’	‘b’	‘c’	‘\0’

//字符串长度即strlen(“abc”)=3，但需要4个字节存储。

```
char c[6]=“abc”;    //sizeof(c)=6，strlen(c)=3，“abc”可看作字符数组
char d[ ]=“abc”;    //sizeof(d)=4，编译自动计算数组的大小，strlen(d)=3
const char*p=“abc”;//sieof(p)=4, p[0]=‘a’，“abc”看作const char指针，
```

注意必须加const

- 故可以写： cout << “abc”[1]; //输出b

第2章 类型、常量及变量

◆2.4 运算符及表达式

- C++运算符、优先级、结合性见表2.7。优先级高的先计算，相同时按结合性规定的计算顺序计算。可分如下几类：
  - 位运算：按位与&、按位或|、按位异或^、左移、右移。左移1位相当于乘于2，右移1位相当于除于2。
  - 算数运算：加+、减-、乘*、除/、模%。
  - 关系运算：大于、大等于、等于、小于、小等于
  - 逻辑运算：逻辑与&&、逻辑或||
- 由于C++逻辑值可以自动转换为整数0或1，因此，数学表达式的关系运算在转换为C++表达式容易混淆整数值和逻辑值。假如x=3，则数学表达式“1<x<2”的结果为假，但若C++计算则1<x<2⇔1<3<2⇔1<2⇔真，
- 数学表达式实际上是两个关系运算的逻辑与，相当于C++的“1<x&&x<2”。

第2章 类型、常量及变量

赋值、选择与自增和自减运算

- 赋值表达式也是C++一种表达式。对于int x(2); x=x+3; 赋值语句中的表达式：
  - x+3是加法运算表达式，其计算结果为传统右值5。
  - x=5是赋值运算表达式，其计算结果为传统左值x(x的值为5)。
  - 由于计算结果为传统左值x，故还可对x赋值7，相当于运算：(x=x+3)=7；结果为左值x
- 选择运算使用”?:”构成，例如：y=(x>0)?1:0; 翻译成等价的C++语句如下。

```
if(x>0) y=1;
else    y=0;
```

- 前置运算“++c”、后置运算“c++”为自增运算；相当于c=c+1，前置运算“—c”、后置运算“c—”为自减运算，相当于c=c-1。前置运算先运算后取值，结果为传统左值；后置运算先取值后运算，结果为传统右值。

第2章 类型、常量及变量

◆2.5 结构与联合

- 结构是使用struct定义的一组数据成员，每个成员都要分配相应的内存。
  - 数据成员可以是基本数据类型如char、int等。
  - 数据成员也可以是复杂的结构或联合成员。
  - 所有成员都可被任意函数访问。
  - 类型不大于int的成员可以定义为使用若干位二进制的位段类型。
- 联合是使用union定义的一组数据成员，所有成员共用最大成员分配的内存。
  - 数据成员可以是基本数据类型、结构或联合类型
  - 所有成员都可被任意函数访问
  - 类型不大于int的成员可以定义为使用若干位二进制的位段类型
- 结构和联合还可以包含函数成员。

第2章 类型、常量及变量

◆2.5 结构与联合

- 在vs2019的x86编译模式下，定义如下结构。

```
struct Person{                  //sizeof(Person)=sizeof(const char*)+sizeof(int)
    const char *name;           //定义实例数据成员name，不允许修改name指向的姓名
    int birthYear;              //定义实例数据成员birthYear
}liShiZhen;                     //定义Person类型的同时定义该类型的变量liShiZhen
struct Person huaTuo;           //定义Person类型后，struct可省略
```

- 在vs2019的x86编译模式下，定义如下联合。

```
union Long { //自定义union类型Long。成员c、s、x共享内存
    char   c;                   //c共享x的内存
    short  s;                   //s共享x的内存
    long   x;                   //sizeof(Long) = max(sizeof(char), sizeof(short), sizeof(long))=sizeof(long)
}a;    Long b;                  //修改任意成员，都会同时修改其它成员
```

constexpr（自学）

字面值类型：对声明constexpr用到的类型必须有限制，这样的类型称为字面值类型（literal type）。

- 算术类型（字符、布尔值、整型数、浮点数）、引用、指针都是字面值类型
- 自定义类型（类）都不是字面值类型，因此不能被定义成constexpr
- 其他的字面值类型包括字面值常量类、枚举

constexpr类型的指针的初始值必须是

- nullptr
- 0
- 指向具有固定地址的对象（全局、局部静态）。注意局部变量在堆栈里，地址不固定，因此不能被constexpr类型的指针指向

当用constexpr声明或定义一个指针时，constexpr仅对指针有效，即指针是const的，

constexpr（自学）

```
int i=0;
//constexpr指针
const int *p = nullptr;                    //p是一个指向整型常量的指针
constexpr int *q = nullptr;                //q是一个指向整数的constexpr指针
constexpr int *q2 = 0;
constexpr int *q3 = &i; //constexpr指针初始值可以指向全局变量i

int f2() {
    static int ii = 0;
    int jj = 0;
    constexpr int *q4 = &ii;//constexpr指针初始值可以指向局部静态变量
//constexpr int *q5 = &jj;//错误：constexpr指针初始值不可以指向局部变量，局部变量在堆栈，非固定地址
  return ++i;
}
```

constexpr函数（自学）

constexpr函数：是指能用于常量表达式的函数，规定：

- 函数返回类型和形参类型都必须是字面值类型
- 函数体有且只有一条return语句
- constexpr函数被隐式地指定为内联函数，因此函数定义可放头文件
- constexpr函数体内也可以包括其他非可执行语句，包括空语句，类型别名，using声明
- 在编译时，编译器会将函数调用替换成其结果值，即这种函数在编译时就求值
- constexpr函数返回的不一定是常量表达式

constexpr函数

```
//constexpr函数
constexpr int new_size() { return 42; }
constexpr int size = new_size() * 2; //函数new_size是constexpr函数，因此size是常量表达式

//允许constexpr返回的是非常量
constexpr int scale(int cnt) { return new_size() * cnt;}

//这时scale是否为constexpr取决于实参
//当实参是常量表达式时，scale返回的也是constexpr
constexpr int rtn1 = scale(sizeof(int)); //实参sizeof(int)是常量表达式，因此rtn1也是

int i = 2;
//constexpr int rtn2 = scale(i); //编译错：实参i不是是常量表达式，因此scale返回的不是常量表达式
```