第二章 变量和基本类型

2014年3月24日 星期一　　上午11:57

在计算机存储器中，数据以位序列的形式存在着。
在位这一级上，存储器是没有结构和意义的。
如何对位序列进行有意义的解释？
为此，我们引入了类型抽象（语言层面上），使得我们能够对位序列进行有意义的解释。

C++ 是一门静态类型（statically typed）语言，类型检查（type checking）发生在编译时。

基本内置类型（Primitive Built-in Types）

算术类型

整型（integral type，包括字符和布尔类型在内）

bool C99中也引入了bool类型，定义在stdbool.h头文件中，C++中，bool类型可参与重载决议。
char，wchar_t，char16_t，char32_t
short，int，long，long long

浮点数

float，double，long double

按照有无符号区分（unsigned, signed）

char类型包括三种类型：unsigned char，signed char 和 char
标准仅仅规定了它们的宽度是一致的，但是char究竟是unsigned还是signed则是implementation-defined，我们不能做任何假设。

切勿混用有符号类型和无符号类型

基本内置类型的宽度

基本内置类型的宽度是implementation-defined的，C++标准只规定了一个范围，
所以除了标准中保证的宽度，我们不能对内建型别的数据宽度做任何假设，否则会引入问题，同时代码也不具备可移植性。

C++标准保证：
对 char 类型或值为 char 类型的表达式做 sizeof 操作保证得 1；
wchar_t为两个字节宽；
short <= int <= long，int的宽度为机器字长，在32位机上，典型情况下，short为2字节，int为4字节，long为4字节；
float < double，在32位机器上，典型情况，float为4字节，double为8字节；
在标准中，bool类型的宽度只需要能容纳下0和1即可，很多编译器的实现选择了1个字节，但是标准不保证 sizeof(bool) == 1
在limits.h和climits头文件中，编译器按照标准定义了整数类型的大小（宏定义）

```
K & R C:
Exercise 2-1. Write a program to determine the ranges of char, short, int, and long variables, both signed
and unsigned, by printing appropriate values from standard headers and by direct computation.

    std::cout << "int : " << (1<<(sizeof(int)*8-1)) << " to " << (~(1<<(sizeof(int)*8-1))) << endl;

    std::cout << "unsigned int : " << 0 << " to " << ~((unsigned int)0) << endl;

    std::cout << "signed char : "
              << (int)(signed char)(1<<(sizeof(signed char)*8-1)) << " to "
              << (int)(signed char)(~(1<<(sizeof(signed char)*8-1))) << endl;

    std::cout << "unsigned char : " << 0 << " to " << (int)(unsigned char)(~0) << endl;

    std::cout << "long : " << (long)(((long)1)<<(sizeof(long)*8-1)) << " to "
              << (long)(~(((long)1)<<(sizeof(long)*8-1))) << endl;

    std::cout << "short : " << (short)(1<<(sizeof(short)*8-1)) << " to "
              << (short)(~(1<<(sizeof(short)*8-1))) << endl;

    int : -2147483648 to 2147483647
    unsigned int : 0 to 4294967295
    signed char : -128 to 127
    unsigned char : 0 to 255
    long : -9223372036854775808 to 9223372036854775807
    short : -32768 to 32767
```

在stdint.h头文件中，按照c99标准7.18节定义了一系列具有特定宽度的整数类型（typedef），例如：
typedef unsigned char uint8_t;

```
typedef unsigned short int uint16_t;
typedef unsigned int uint32_t;
......
```

在格式化输入/输出这些特定宽度的整数类型时，应使用特定的宏操作以确保代码的正确和移植性，具体参考文件inttypes.h。
在C++中包含inttypes.h头文件，必须在包含头文件前确保宏定义 __STDC_FORMAT_MACROS 的存在：

```
#define __STDC_FORMAT_MACROS
#include <inttypes.h>
#endif
```

对于这些内建型别，需要注意编译器的隐式转换行为：

```
#define VALUE_1 ((char)0xAA)
#define VALUE_2 ((char)0xAB)
int main()
{
    uint8_t x = VALUE_1;    // x = 0xAA
    if (x == VALUE_1)       // VALUE_1 和 x 被隐式整型提升为 int，VALUE_1 -> 0xFFFFAA
        printf("VALUE_1\n");
    else
        printf("VALUE_2\n");
    return 0;
}
```

标准中还定义了如下类型：

```
size_t:     implementation-defined unsigned integer type
ssize_t:    implementation-defined signed integer type
ptrdiff_t:  implementation-defined signed integer type of the result of subtracting two pointers.
std::size_type:
std::difference_type:
```

字面值常量 (Literal Constant)

称之为字面值是因为只能用它的值称呼它，称之为常量是因为它的值不能修改。

每个字面值都有相应的类型，只有内置类型存在字面值，可以通过增加前缀和后缀改变字面值常量的默认类型。

整型常量：默认为int或long，实际类型取决于其字面值，可有U后缀和L后缀；
浮点数常量：默认为double，可有F后缀和L后缀；
字符常量：可通过L前缀表示宽字符常量，如 L'a';
字符串常量：亦可通过L前缀表示宽字符串常量，如 L"hello";
bool常量：true和false
指针字面值：nullptr

字面值常量具有存储区域，但是是不可寻址的，可以作为右值。

某些字面值常量的类型在C和C++中是不同的，例如：

在C中，字符常量的类型是int：sizeof('a') == 4
在C++中，字符常量的类型是char：sizeof('a') == 1

变量（Variable）

具有右值属性（r-value）和左值属性（l-value），是可寻址的。

在此处需要强调一下对象和变量这两个概念：
所谓对象，在C++中指的是执行环境中的一块存储区域，该存储区域中的内容则代表了该对象的值；
所谓变量，指的是一种声明，通过声明，我们将一个名字和一个对象对应起来，通过名字可以操作对象，简而言之，即命名的存储区域。
注意区分：临时对象 和 临时变量。

内建型别变量的初始化：

1 显式：使用赋值操作符

```
int a = 100
```

2 隐式：使用括号

```
int a(100)
```

3 特殊的初始化为零的方式：使用括号

```
int a = int()
```

4 列表初始化：使用花括号

```
int a = {0}
int a{0}
```

如果使用列表初始化并且初始值存在丢失信息的风险，编译器将报错：

```
int x;
float f{x};  // error
```

变量定义后即立刻可见，因此如下语句在语法上是合法的：

```
int x = x;
```

注意区分初始化和赋值这两个概念：

```
int x = 1;  // copy-initialization
int x(1);   // direct-initialization
x = 10;     // assignment
```

值初始化（value initialization）

值初始化是一种初始化过程，内置类型初始化为0，类类型由类的默认构造函数初始化。
只有类包含默认构造函数时，该类的对象才能被值初始化。

复合类型（Compound type）

一条声明语句由一个基本数据类型（basic type）和紧随其后的一个声明符（declarator）列表组成。

指针（Pointer）

指针用于指向一个对象，从而我们可以通过指针去间接地操作该对象

指针的值是另外一个对象的地址，而指针的类型则用于如何去解析该地址

当我们仅关心地址值的时候（不带有指向对象的类型信息）
C/C++提供了void*类型供我们使用
void*可被**除了函数指针**以外所有类型的指针赋值
void*类型指针可用于传递地址值（函数参数，函数返回值，被赋值）以及比较地址值

```
using PF = void (*)(); // 等价于 typedef void (*PF)();
PF pf;
void *p1 = pf; // error
void *p2 = (void*)pf; // ok
void *p3 = static_cast<void*>(pf); // error
void *p4 = reinterpret_cast<void*>(pf); // ok
```

指针可用作右值或左值

对指针施加解引用操作符（Dereference Operator, *）会返回指针指向对象的左值，通过这种方式我们可以修改指针指向的对象值

```
int *p = 10;
*p = 90; // *p是左值
```

对指针施加取址操作符（Addressing Operator, &）返回的是指向指针的指针，其值为当前指针所在的地址值

```
int *p; // &p类型为int**
```

取址操作符作用于对象不会访址，只返回对象的地址：

```
struct Point3D
{
    int x;
    int y;
    int z;
};
int main()
{
    Point3D* pPoint = NULL;
    int offset = (int)(&(pPoint->z));  // 计算z的偏移，&(pPoint->z)不会发生访存操作
    printf("%d", offset);
    return 0;
}
```

指针与const限定符：

```
a. const int *ptr;
b. int const *ptr; // 和a一致，指针指向的对象为const的
c. int *const ptr; // 指针本身是const的，const指针必须被初始化
d. const int *const ptr;
e. int const *const ptr; // 和d一致，指针本身以及指向的对象都是const的
```

const指针必须被初始化：

```
int x = 100;
int *const p = &x;
```

指向非const类型的指针不能被const对象的地址初始化和赋值：

```
const int x = 100;
int *ptr = &x; // error, invalid conversion from 'const int*' to 'int*'
const int *ptr = &x; // ok
const int *const ptr = &x; // ok
```

指向const对象的指针可以被非const对象的地址或者const对象地址初始化和赋值
应该理解为"自以为指向const对象的指针"

```
int x = 100;
const int * p = &x; // ok, 自以为指向const对象的指针
x = 200; // ok
```

引用 (Reference)

引用相当于对象的别名（alias），通过引用我们可以间接地去操纵该对象
引用是对左值的引用（l-value reference）

右值引用(r-value reference)： TODO

引用不是对象，没有地址，指针不能指向引用，引用可以绑定到指针

```
int i = 42;
int *p;      // p is a pointer to int
int* &r = p; // r is a reference to the pointer p
r = &i;      // r refers to a pointer; assigning &i to r makes p point to i
*r = 0;      // dereferencing r yields i, the object to which p points; changes i to 0
```

非const引用必须被同类型的对象初始化，const对象不能绑定到非const引用上

const引用则可以绑定到不同但相关的类型的对象或者右值上，const引用可能会引入临时对象

```
double d = 9.0;
const int &rx = d;
    // 编译器会隐式地产生一个int类型的临时对象(左值)，用于rx的初始化，
    // 此处rx并非为d的引用，而是用d值初始化的int类型的临时对象的引用。
```

const引用可以被非const对象初始化，所以应该理解为”自以为是对const对象的引用”

```
const int &ry = 100; // 该引用实际绑定到用100作为右值初始化的临时对象（左值）上
int x = 10;
const int &rx = x; // rx是x的引用，两者为同一对象
x = 20; // Ok
const int x = 10;
const int &rx = x; // rx是x的引用，两者为同一对象
```

从汇编层面上，引用的实现：

```
int x = 99;   movl $99, -24(%ebp)
int &rx = x;  leal -24(%ebp), %eax // 实际就是通过被引用对象的地址进行操作
              movl %eax, -16(%ebp)
```

const限定符 （const qualifier）

const限定符修饰的对象是左值，但是不是可修改的左值
const对象必须初始化

```
const int i = get_size(); // ok: initialized at run time
const int j = 42; // ok: initialized at compile time
const int k; // error: k is uninitialized const
```

const对象默认为文件的局部变量，除非特别说明，在全局作用域声明的 const 变量是定义该对象的
文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。
通过指定 const 变更为 extern，就可以在整个程序中访问 const 对象。

```
const int x = 10; // 内部链接属性，仅当前文件可见
extern const int y = 10; // 外部链接属性，全局可见
```

常量折叠（const folding），编译器在可能的情况下，会将const对象由其值替代。

```
test.h: const int size = 100;
test.cpp: int array[size];  // int array[100]
```

顶层const（top-level const）表示任意的对象是常量

```
const int x = 100;
int *const px = &y;
```

底层const（low-level const）则与指针和引用等复合类型的基本类型部分有关

```
const int *x;
const int &y = d
```

对象的初始化和赋值时，顶层const常被忽略，底层const不能忽略

非引用类型的const对象初始化：

const对象可以由非const对象初始化，反之亦可。
普通的指针不能被const对象的地址初始化和赋值。

```
int x = 100;
const int y = x; // Okay

const int x = 100;
int y = x; // Okay

int x = 100;
const int *p = &x; // Okay

const int x = 100;
int *p = &x; // error, invalid conversion from 'const int*' to 'int*'
```

const引用类型对象初始化：
const引用可以被非const对象初始化；
const引用可以被绑定到相关类型（存在隐式转换）或者右值上，会引入临时对象。

```
double dval = 3.14;
const int &ri = dval;
const int temp = dval; // create a temporary const int from the double
const int &ri = temp; // bind ri to that temporary object
```

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时就能得到计算结果的表达式。
字面值属于常量表达式，用常量表达式初始化的const对象也是常量表达式。

```
const int max_files = 20;
const int limit = max_files+1;
```

constexpr变量
C++11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。
声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化。

```
constexpr int mf = 20;
constexpr int limit = mf+1;
constexpr int sz = size(); // 只有当size是一个constexpr函数时才是合法的声明语句
```

字面值类型（literal type）

常量表达式的值需要在编译时就得到计算，因此对声明constexpr时用到的类型必须有所限制，这类类型称为字面值类型
算术类型，引用和指针都属于字面值类型
指针和引用都能定义成constexpr，但它们的初始值却受到严格限制
一个constexpr指针的初始值必须是nullptr或者0，或者是存储于某个固定地址中的对象

```
constexpr int *p = nullptr; // ok

int x = 100; // global variable
constexpr int *p = &x; // ok
constexpr int &y = x; // ok
```

在constexpr声明中如果定义了一个指针，限定符constexpr仅对指针本身有效，与指针所指的对象无关

```
const int *p = nullptr; // p is a pointer to const int
Constexpr int *q = nullptr; // q is a const pointer to int
```

数组和指针

1. 数组是分配了一块连续的存储区域，数组名可代表整个数组，可以用sizeof取得真实大小；
   指针则只分配了指针大小的内存，并且可以指向某个有效的存储区域。

2. 数组名非左值，在此层面上，可以理解为常量；
   指针可作为左值。

3. 数组名在多数情况下会退化为指向数组第一个元素的指针（右值）。
   数组不发生退化的地方有几个场合，
   一是声明时
   一是用作sizeof的操作数时
   一是用作&取址运算的操作数时

```
int a[3] = {1,2,3};
int b[3];
b = a;       // error. b converted to int* (rvalue): array degration.
int *p = a;  // array degration: a converted to an rvalue of int*
sizeof(a);   // no degration.
&a;          // no degration.
```

C++中，数组还有其他场合不发生退化：
比如作为引用的initializer；
作为typeid/typeinfo的操作数和模板推导时；
作为decltype的参数时

4. 解引用操作符可作用于指针和数组名：
对指针和一维数组解引用，会发生访存操作

```
int x = 100;
int *p = &x;
*p = 200;  // x -> 200
int array[2] = { 10, 20 };
*array = 30;  // { 30, 20 }
```

对多维数组解引用，只是类型改变。

```
int array[3][4];  // *array的类型是 int[4]
```

5. 取址操作符可作用于数组名和指针：
对指针取址，取得的是指针所在的地址，即指向指针的指针；
对数组名取址，只是类型改变，得到的还是该数组首元素的地址。

```
int array[10];   // sizeof(array) = 10*sizeof(int)
printf("0x%x\n", &array);  // -> printf("0x%x\n", array);
                           // &array的类型:  int (*)[10]
```

6. 下标运算符 [] 可作用于数组名和指针。

7. 当指向同一数组中的两个元素的指针相减时，结果是这两个元素的下标差值。
标准引入了ptrdiff_t类型，定义在stddef.h(cstddef)头文件中，其类型为signed integer，用于表示这种差值。

8. 多维数组实际上是数组的数组，首元素的类型就是数组类型。

```
int array[3][4];                leal    -64(%ebp), %eax
int (*p1)[4] = array;           movl    %eax, -16(%ebp)
    // array的类型是 三个int[4]数组作为元素的数组

int (*p2)[4] = &array[2];     leal   -64(%ebp), %eax
                              addl   $32, %eax
                              movl   %eax, -12(%ebp)
    // array[2]的类型是 int[4]

int *p3 = array[2];
                          leal      -64(%ebp), %eax
                          addl     $16, %eax
                          movl    %eax, -8(%ebp)
```

9. C++11引入了名为begin()和end()的函数，这两个函数定义在iterator头文件
begin()函数返回指向数组首元素的地址，end()函数返回指向数组尾元素的下一位置的指针

10. 标准库类型限定使用的下标必须是无符号类型，而内置数组的下标运算无此要求

```
int *p = &ia[2];
int k = p[-2]; // p[-2]是ia[0]表示的那个元素
```

11. 使用数组初始化vector对象

```
int int_arr[] = { 0, 1, 2, 3, 4, 5 };
vector<int> ivec(begin(int_arr), end(int_arr));
vector<int> subVec(int_arr+1, int_arr+4);
```

12.

```
int ia[10];
auto ia2(ia); // 发生退化，ia2类型是int*，指向ia的首元素
```

13.

```
int ia[10];
decltype(ia) ia3; // ia3是数组
```

14. 使用范围for语句处理多维数组

```
int ia[3][4];
size_t cnt = 0;
for (auto &row : ia)  // 外层循环必须是reference，否则会退化为指针
    for (auto &col : row)
        col = cnt++;

for (const auto &row : ia) // 外层循环的控制变量必须是reference
    for (auto col : row)
        cout << col << endl;

vector<vector<int>> v; // vector和数组的区别
```

```
for (auto row : v)
    for (auto col : row)
        cout << col << endl;
```

枚举 (Enumerations)

枚举成员是常量，**必须用常量表达式初始化**，枚举成员值可以不唯一（值可以重复）。

```
enum E {
    E1 = 99,
    E2 = 99,  // Okay
    E3 = sizeof(int)  // Okay
};
```

枚举类型的对象的初始化和赋值只能通过其枚举成员或同一枚举类型的其它对象来进行。
枚举类型的宽度是implementation-defined的，只要规定了其宽度能容纳下最大的枚举成员值：
C中char和integer类型都可以，选择权在于实现；
C++中则是int->unsigned int->long->unsigned long，依次选择。

typedef

标识符或类型名并没有引入新的类型，而只是现有数据类型的同义词

为了隐藏特定类型的实现，强调使用类型的目的；

```
typedef unsigned int uint32_t;
```

简化复杂的类型定义，使其更易理解；

```
// 函数指针数组的不同写法
typedef void (*PF)(int);
PF pf_array[5]; // 等价于  void (*pf_array[5])(int);

// 返回函数指针的函数
typedef void (*PF)(int);
PF foo(int); // 等价于 void (*foo(int))(int);

// 指向函数指针数组的指针
typedef void (*PF)(int);
PF (*pf_array_p)[5]; // 等价于 void (*(*pf_array_p)[5])(int);
```

允许一种类型用于多个目的，同时使得每次使用该类型的目的明确

容易犯错的typedef用法

```
typedef char *cstring;
extern const cstring cstr; // cstr的类型：char *const cstr;
```

当链接指示符应用在一个声明上时，所有被它声明的函数都将受到链接指示符的影响

```
extern "C" void f1(void (*pfParm)(int)); // f1 和 pfParm 都具有C链接属性
extern "C" typedef void FC(int);
void f2(FC *pfParm); // f2：C++链接属性；pfParm：C链接属性
```

auto类型说明符

C++11引入了auto类型说明符
auto让编译器通过初始值来推断变量的类型，因此auto定义的变量必须有初始值

```
auto item = val1 + val2;
```

auto也能在一条语句中声明多个变量

```
auto i = 0, *p = &i;
```

复合类型和auto

编译器推断出来的auto类型有时候和初始值的类型并不完全一样

当引用被作为初始值的时候，编译器以引用对象的类型作为auto的类型：

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

auto一般会忽略掉顶层const，同时底层const则会保留下来

```
int i = 0;
const int ci = i, &cr = ci;
auto b = ci; // b is int
auto c = cr; // c is int
auto d = &i; // d is a pointer to int
auto e = &ci; // e is a pointer to const int
```

如果希望推断出的auto类型是一个顶层const，需要明确指出：

```
const auto f = ci; // f is const int
```

可以将引用类型设为auto，顶层const不会被忽略，除此之外原来的初始化规则仍然适用，

```
auto &g = ci; // g is a reference to const int
auto &h = 42; // error
const auto &j = 42; // ok
```

decltype类型指示符

C++11引入了decltype，它的作用是选择并返回操作数的数据类型，
在此过程中，编译器分析表达式并得到它的类型，但并不实际计算表达式的值

```
decltype(f()) sum = x;
```

decltype处理顶层const和引用的方式与auto不同，如果decltype使用的表达式是一个变量，
则decltype返回该变量的类型（包括顶层const和引用在内）

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x is const int
decltype(cj) y = x; // y is const int&
decltype(cj) z;  // error, z is a reference
```

引用从来都是作为其所指对象的同义词出现，只有在decltype处是一个例外

如果decltype使用的表达式不是一个变量，则decltype返回表达式结果对应的类型
有的表达式返回一个引用类型，因为着该表达式的结果对象是左值

```
int i = 42, *p = &i, &r = i;
decltype(r+0) b; // ok, b is int
decltype(*p) c; // error, c is int& (l-value)
```

delctype((variable))的结果永远是引用
decltype(variable)的结果只有当variable本身就是一个引用时才是引用

自定义数据类型

C++11规定，可以为数据成员提供一个类内初始值（in-class initializer）
创建对象时，类内初始值将用于初始化数据成员，没有初始值的成员将被默认初始化
对类内初始值的限制，或者放在花括号里，或者放在等号右边，不能使用圆括号

```
struct Sales_data
{
    std::string bookNo; // 默认初始化
    unsigned units_sold{0}; // ok
    double revenue = 0.0; // ok

    int x(10); // error
};
```

标量类型（Scalar types）

```
Arithmetic types (3.9.1),
enumeration types,
pointer types,
pointer to member types (3.9.2),
std::nullptr_t,
and cv-qualified versions of these types (3.9.3)
are collectively called scalar types.
```

标量类型是POD（Plain Old Data）类型

C风格字符串 (C-Style Character Strings)

以空字符null (‘\0’ 和 L’\0’)结尾的字符数组。

```
char s[] = {‘a’, ‘b’, ‘c’}; // 不是C风格字符串
char s[] = “abc”; // C风格字符串
```

字符串字面值的类型是const char类型的字符数组。

通过(const) char*类型的指针来操纵C风格字符串。

```
char *p = “hello”; // 指向只读数据块的指针，字符串可能被编译器放入字符串池。
p[1] = 'x'; // 可通过编译，但是会引起segmentation fault
char s[] = “hello”; // 可写数据块（全局或静态）或者堆栈（局部）
const char *p = “hello”;
const char s[] = “hello”;  // 注意这两种声明的区别
```

size_t是标准库中与机器相关的typedef类型定义，其类型定义在stddef.h中(cstddef)，标准规定其为unsigned integer类型

永远不要忘记字符串结束符 null

使用 strn.. 函数处理C风格字符串

string类提供了c_str()函数用于返回C风格字符串，返回值类型是const char*

```
string s("hello world");
const char *str = s.c_str();
```

POD(Plain Old Data) type

A type that consists of nothing but Plain Old Data.

A POD type is a C++ type that has an equivalent in C, and that uses the same rules as C uses for initialization, copying, layout, and addressing.

As an example, the C declaration struct Fred x; does not initialize the members of the Fred variable x. To make this same behavior happen in C++, Fred would need to not have any constructors. Similarly to make the C++ version of copying the same as the C version, the C++ Fred must not have overloaded the assignment operator. To make sure the other rules match, the C++ version must not have virtual functions, base classes, non-static members that are private or protected, or a destructor. It can, however, have static data members, static member functions, and non-static non-virtual member functions.

The actual definition of a POD type is recursive and gets a little gnarly. Here's a slightly simplified definition of POD: a POD type's non-static
data members must be public and can be of any of these types: bool, any numeric type including the various char variants, any enumeration type, any
data-pointer type (that is, any type convertible to void*), any pointer-to-function type, or any POD type, including arrays of any of these.

Note: data-pointers and pointers-to-function are okay, but pointers-to-member are not. Also note that references are not allowed. In addition, a POD type can't have constructors, virtual functions, base classes, or an overloaded assignment operator.

你可以将 POD 类型看作是一种来自外太空的用绿色保护层包装的数据类型，POD 意为“Plain Old Data”（译者：如果一定要译成中文，那就叫“彻头彻尾的老数据”怎么样！）这就是 POD 类型的含义。其确切定义相当粗糙（参见 C++ ISO 标准），其基本意思是 POD 类型包含与 C 兼容的原始数据。例如，结构和整型是 POD 类型，但带有构造函数或虚拟函数的类则不是。 POD 类型没有虚拟函数，基类，用户定义的构造函数，拷贝构造，赋值操作符或析构函数。

为了将 POD 类型概念化，你可以通过拷贝其比特来拷贝它们。
此外， POD 类型可以是非初始化的。例如：

```
struct RECT r; // value undefined
POINT *ppoints = new POINT[100]; // ditto
String s; // calls ctor ==> not POD
```

非 POD 类型通常需要初始化，不论是调用缺省的构造函数（编译器提供的）还是自己写的构造函数。

第三章 字符串 向量 和 数组

2014年3月24日 星期一　　下午8:13

标准库类型string

头文件不应该包含using声明

直接初始化和拷贝初始化

```
string s5 = "hiya"; // 拷贝初始化
string s6("hiya"); // 直接初始化
string s7(10, 'c'); // 直接初始化
string s8 = string(10, 'c'); // 拷贝初始化
```

size函数返回string 对象的长度（即string对象中字符的个数）

```
size_type size() const;
```

string::size_type类型

string类及其他大多数标准库类型都定义了几种配套的类型，这些配套类型体现了标准库类型与机器无关的特性，类型size_type即是其中的一种，它位于对应的类空间。

可以通过auto或decltype来推断size_type的类型：auto len = line.size();

由于历史原因，也为了和C兼容，C++中的字符串字面值并不是标准库类型string的对象。

使用C++版本的C标准库头文件

name.h -> cname （命名空间std）

使用基于范围的for语句

```
for (declaration : expression)
    statement
```

expression部分是一个对象，用于表示一个序列

declaration部分负责定义一个变量，该变量将被用于访问序列中的基础元素

每次迭代，declaration部分的变量会被初始化为expression部分的下一个元素值

```
string str("hello world");
for (auto c : str)
    std::cout << c << endl;

for (auto &c : str)
    c = toupper(c);

for (decltype(str.size()) i = 0;
    i < str.size();
    ++i)
    str[i] = toupper(str[i]);
```

标准库类型vector

vector是模版而非类型，由vector生成的类型必须包含vector中元素的类型，例如：vector<int>

vector能容纳绝大多数类型的对象作为其元素，但是因为引用不是对象，所以不存在包含引用的vector

定义和初始化vector对象

默认初始化

```
vector<T> v1; // 默认初始化，空vector
```

直接初始化

```
vector<T> v2(v1); // 直接初始化，v2中包含有v1所有元素的副本
```

拷贝初始化

```
vector<T> v2 = v1; // 拷贝初始化，等价于v2(v1)
```

创建指定数量的元素

```
vector<T> v3(n, val); // v3中包含了n个重复的元素，每个元素的值都是val
```

列表初始化vector对象

c++11标准提供了为vector对象的元素赋初值的方法，即列表初始化

```
vector<T> v5{a, b, c ...}; // 列表初始化，v5包含了初始值个数的元素，每个元素被赋予相应的初始值
vector<T> v5 = {a, b, c ...}; // 等价于v5{a, b, c ...}

vector<string> articles = {"a", "an", "the" };
vector<string> v1{"a", "an", "the"};
```

值初始化（value-initialized）

通常情况下，可以只提供vector对象容纳的元素数量而不用略去初始值
此时库会创建一个值初始化元素初值，并把它赋给容器中的所有元素，这个初值由vector对象中元素的类型决定
如果vector对象的元素是内置类型，则元素初始值自动设为0
如果是某种类类型，则元素由类默认初始化

```
vector<T> v4(n); // v4中包含了n个重复地执行了值初始化的对象

vector<int> ivec(10); // 10个元素，每个都初始化为0
vector<string> svec(10); // 10个元素，每个都是空string对象
```

列表初始化还是元素数量

确认无法执行列表初始化后，编译器会尝试用默认值初始化vector对象

```
vector<int> v1(10); // 值初始化，10个元素，每个元素的值都是0
vector<int> v2{10}; // 列表初始化，1个元素，该元素的值为10

vector<int> v3(10, 1); // 10个元素，每个元素的值都是1
vector<int> v4{10, 1}; // 2个元素，值分别是10和1

vector<string> v5{"hi"}; // 列表初始化，一个元素
vector<string> v6("hi"); // error
vector<string> v7{10}; // 值初始化，10个空string元素
vector<string> v8{10, "hi"}; // 10个string元素，每个值为"hi"
```

vector对象能高效增长

C++标准要求vector对象应该能在运行时高效快速地添加元素
除非所有元素的值都一样，一旦元素的值有所不同，更有效的办法是先定义一个空的vector对象，再在运行时向其中添加元素

迭代器（Iterator）

迭代器提供了对对象的间接访问
迭代器分为有效和无效，有效的迭代器或者指向某个元素，或者只想容器中尾元素的下一位置；其他情况都属于无效

begin()返回只想第一个元素的迭代器，end()返回指向尾元素的下一位置（one past the end）的迭代器
如果容器为空，则begin()和end()返回的是同一个迭代器，都是尾后迭代器

```
vector<int> v;
auto b = v.begin(), e = v.end();

string s;
for (auto it = s.begin(); it != s.end(); ++it)
    *it = toupper(*it);
```

迭代器的类型是 iterator 和 const_iterator
iterator的对象可读可写，const_iterator能读取但是不能修改
const_iterator既可用于常量对象，也可用于非常量对象
iterator不可用于常量对象

```
const vector<int> cv;
vector<int>::const_iterator citer = cv.begin(); // ok
vector<int>::iterator iter = cv.begin(); // error
```

begin和end运算符返回的具体类型由对象是否是常量决定，如果对象是常量，返回const_iterator；如果不是常量，返回iterator

```
vector<int> v;
auto b1 = v.begin(), e1 = v.end(); // b1和e1的类型是vector<int>::iterator

const vector<int> cv;
auto b2 = cv.begin(), e2 = cv.end(); // b2和e2的类型是vector<int>::const_iterator
```

C++11引入了两个新函数，分别是cbegin()和cend()，返回值始终都是const_iterator

```
vector<int> v;
auto it = v.cbegin(); // it的类型是vector<int>::const_iterator
```

迭代器运算（iterator arithmetic）

```
template <typename T>
bool binary_search(vector<T> &v, const T &target)
{
    auto beg = v.begin();
    auto end = v.end();
    auto mid = beg + (end-beg)/2; // attention

    while (mid != end && *mid != target)
    {
        if (target < *mid)
            end = mid;
        else
            beg = mid+1;
        mid = beg + (end-beg)/2;
    }

    return mid!=end;
}

// iter1 - iter2：名为difference_type的有符号整型数
// 参与关系运算符运算的两个迭代器必须合法且指向同一个容器的元素（或者尾元素的下一位置）
```

第四章 表达式

2014年3月25日 星期二　　下午3:26

隐式类型转换（implicit cast）

编译器在必要的时候，隐式地将类型转换规则应用到内置类型和类类型的对象上。
在下列情况下，将会发生隐式类型转换：
在混合类型的表达式中，其操作数被转换为相同的类型；
用作条件的表达式被转换为bool类型；
用一个表达式初始化某个变量，或者将一个表达式赋值给某个变量，则该表达式被转换为该变量的类型；
函数调用发生时，实参被转换为形参的类型。

C++为内置类型定义了一组转换规则，包括：

算术转换：

算术转换规则将**二元操作符的操作数转换为同一类型，并且表达式的值也具有相同类型**。
算术转换规则定义了一个类型转换层次，规定了操作数应该按照什么规则转换为表达式中最宽的类型。

整型提升：
对于所有比 int 型小的类型，包括：char，unsigned/signed char，short，unsigned/signed short，
如果该类型的最大值可包含于int类型，则将会被提升到int类型，否则提升到unsigned int类型。

```
unsigned int x = 0xFF;
int y = -10;
if (x > y) // int converted to unsigned int

char flag = 0xAA; // signed or unsigned
if (flag != (unsigned int)0xAA) // char promoted to int, then converted to unsigned int
```

对于包含浮点类型和整型类型的混合运算表达式，则会**先施行整型提升，再将提升后的值转换为浮点数类型**。

浮点数类型转换也有转换层次，float -> double -> long double

```
bool      flag;
char      cval;
short     sval;
unsigned short usval;
int       ival;
unsigned int   uival;
long      lval;
unsigned long  ulval;
float     fval;
double    dval;
3.14159L + 'a'; // promote 'a' to int, then convert to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, then converted to float
sval + cval;    // sval and cval promoted to int
cval + lval;    // cval promoted to int, then converted to long
ival + ulval;   // ival converted to unsigned long
usval + ival;   // promotion depends on size of unsigned short and int
uival + lval;   // conversion depends on size of unsigned int and long
```

指针转换：

数组名在多数情况下会被隐式转换为指向数组元素类型的指针

```
extern void foo(int x[]);
int x[10];
foo(x); // int[10] converted to int*

extern void foo(int (*x)[5]);
int x[4][5];
foo(x);  // int[4][5] converted to int(*)[5]
```

例外情况如下：
数组用作decltype关键字的参数时，取址运算符作用于数组名，sizeof运算符，typeid运算符，
用数组名去初始化数组引用类型

整型常量 0 可以被转换为任何指针类型

nullptr能转换成任意指针类型

指向任意非常量的指针可以被转换为void*类型

指向任意对象的指针可以被转换为const void*类型

bool类型转换：

算术值和指针值均可被转换为bool类型，如果算术值和指针值为 0，则转换为 false；否则被转换为 true。

```
if (cp) /* ... */ // true if cp is not zero
while (*cp) /* ... */ // dereference cp and convert resulting char to bool
```

bool类型可以被转换为算术类型，false 转换为 0；true 转换为 1。

枚举类型转换：

枚举类型的对象和枚举类型的成员可被转换为整型类型，其转换结果可用于任何要求整型值的地方。
枚举类型转换为何种整型类型，取决于枚举类型的最大值。
C++中按照 int -> unsigned int -> long -> unsigned long依次选择。

const转换：

当用非const对象去初始化const对象的引用时，非const对象被转换为const对象；
当用非const对象的地址或指向非const对象的指针去初始化指向const对象的指针时，亦发生const转换。

```
int i;
const int ci = 0;
const int &j = i; // ok: convert non-const to reference to const int
const int *p = &ci; // ok: convert address of non-const to address of a const
```

C++标准库也定义了一系列转换规则

重要类型转换如下：

```
while(cin > s) // isstream -> bool
string s = "hello world"; // const char* -> string
```

显式转换

static_cast

任何具有明确定义的类型转换，只要不包含底层const，都可以使用static_cast
static_cast对于编译器无法自动执行的类型转换也非常有用
例如，可以使用static_cast找回存在于void*指针中的值

```
void *p = &d;
double *dp = static_cast<double*>(p);
```

dynamic_cast 运行时类型识别，TODO

const_cast

const_cast只能改变运行对象的**底层const**

```
const char *pc;
char *p = const_cast<char*>(pc); // ok，但是通过p写值是undefined behavior
```

只有const_cast能改变表达式的常量属性，使用其他形式的明明强制类型转换改变表达式的常量属性都将引发编译器错误
同样，也不能用const_cast改变表达式的类型

```
const char *cp;
char *q = static_cast<char*>(cp); // error
static_cast<string>(cp); // ok
const_cast<string>(cp); // error
```

reinterpret_cast

reinterpret_cast通常为运算对象的位模式提供较低层次上的重新解释

```
int *ip;
```

```
        char *pc = reinterpret_cast<char*>(ip);
```
reinterpret_cast本质上依赖于机器
```
        using PF = void (*)(); // 等价于 typedef void (*PF)();
        PF pf;
        void *p1 = pf; // error
        void *p2 = (void*)pf; // ok
        void *p3 = static_cast<void*>(pf); // error
        void *p4 = reinterpret_cast<void*>(pf); // ok
```

旧式的强制类型转换
```
    type (expr); // 函数形式
    (type) expr; // C语言风格
```

第六章 函数

2014年3月25日 星期二　　下午5:07

函数（Function）

函数体形成了一个作用域(scope)。

C++是静态强类型语言，每次函数调用，编译器都会进行参数检查（类型和个数），因此函数必须先被声明才能使用。

函数的形参表称为函数的符号特征(signature)，可用来区分函数的不同实例(重载函数)。
有了函数名和符号特征，就可以唯一的标识函数了。
对于C++，在编译器层面上，函数名+形参表 -> name mangling -> 生成编译器用于区分函数的符号；
对于C，在编译器层面上，函数名 -> 生成编译器用于区分函数的符号。

形参（Parameter）

位于函数声明和定义的形参表中。
生命期始于函数调用直到函数返回，作用域在函数体内。
形参必须在命名后才能被使用。
形参由实参初始化或者是缺省值初始化

```
void foo(); // C++中等价于 void foo(void)
            //  C 中可以接受任何参数，必须显式声明为void foo(void)
```

省略号（ellipsis）挂起了类型检查机制，它的出现告诉编译器，当函数调用发生时，可以有0个或者多个实参，而实参的类型未知。
省略号参与重载决议。

```
void foo(parm_list...);
void foo(); // 并不等价于 void foo(...);

void foo() {}
void foo(...) {}
foo(); // error: call of overloaded 'foo()' is ambiguous

// Use command "nm -a"
void foo(int, ...) {}   // 08048404 T _Z3fooiz
void foo(int) {}   //  0804840a T _Z3fooi
void foo(...) {}   //  08048410 T _Z3fooz
```

实参（Argument）

函数调用发生时，实参作为初始值初始化形参。
尽管实参和形参存在对应关系，但是并没有规定实参的求值顺序，编译器能以任意可行的顺序对实参求值。
实参必须和形参类型相同，或者可以隐式转换为形参类型。

参数传递

形参初始化的机理和变量初始化一样
当形参是是引用类型时，形参是它对应的实参的别名，称为**按引用传递**（passed by reference）
当实参的值被拷贝给形参时，形参和实参是两个互相独立的对象，称为**按值传递**（passed by value）

非引用类型的形参初始化（按值传递）

非指针类型

顶层const被忽略
可用const对象初始化非const对象，反之亦可

```
extern void foo1(const int x);
extern void foo2(int x);
int x = 100;
const int y = 90;
foo1(x);  // Okay
foo1(x);  // Okay
foo2(y);  // Okay
foo2(y);  // Okay
```

在C/C++中，按值传递时，const形参和非const形参的函数无区别，因为形参的顶层const被忽略

```
void foo(const int x);
void foo(int x); // error: redefinition of 'void foo(int)'
    > nm -a a.out | grep foo
    08048404 T _Z3fooi  // void foo(const int x);
```

指针类型

底层const不可忽略
指针形参指向的是const对象还是非const对象，将影响函数调用时可以使用的实参，因为形参的底层const不可被忽略
可以将指向const对象的指针初始化为指向非const对象，但是不能让指向非const对象的指针指向const对象。

```
int x = 100;
const int* p = &x; // Okay

const int x = 100;
int* p1 = &x; // Wrong: invalid conversion from 'const int*' to 'int*'
const int* p2 = &x; // Okay

extern void foo1(const int* p);
extern void foo2(int* p);
int x = 100;
const int y = 90;
foo1(&x); // Okay
foo2(&x); // Okay
foo1(&y); // Okay
foo2(&y); // error: invalid conversion from 'const int*' to 'int*'

void foo(const int* p) {}
void foo(int* p) {} // Okay, overloaded function

void foo(int* const p) {}
void foo(int* p) {} // error: redefinition of 'void foo(int*)'
```

引用类型的形参初始化

引用类型的形参是对应对象的别名，在初始化时直接绑定到对应的实参对象。

利用引用形参修改实参的值：

```
void swap(int& x, int& y);
```

使用引用形参传回额外的信息：

```
int foo(int& ret);
```

节省传递复杂庞大对象的开销：

```
void foo(const string& s);
```

在有效地实现重载操作符的同时，还能保证用法的直观性：

```
Matrix operator+(const Matrix& m1, const Matrix& m2);   // m = m1+m2;
```

引用类型的形参必须指向一个对象，因此如果一个参数可能在函数中指向不同的对象，或者不指向任何对象，那么就不能定义为引用类型。

非const引用形参只能绑定到相同类型的非const对象实参。

```
int incr(int &val)
{
    return ++val;
}
int main()
{
    short v1 = 0;
    const int v2 = 42;
    int v3 = incr(v1);   // error: v1 is not an int
    v3 = incr(v2);       // error: v2 is const
    v3 = incr(0);        // error: literals are not l-values
    v3 = incr(v1 + v2);  // error: addition doesn't yield an l-value
    int v4 = incr(v3);   // ok: v3 is a non const object type int
}
```

const引用，应该将不需要修改的引用形参声明为const引用。
普通的非const引用形参在使用时不太灵活，这样的形参既不能用const对象初始化，也不能用字面值或产生右值的表达式实参初始化。

```
void foo1(const string& s) {}
void foo2(string& s) {}
string s1;
const string s2;
foo1("hello world"); // Okay
foo1(s1); // Okay
foo1(s2); // Okay
foo2(s1); // Okay
foo2(s2); // Error:
```

数组形参

数组永远不会按值传递，数组长度不是参数类型的一部分（多维数组除第一维之外）

```
void foo(int arr[100]);  // 等价于void foo(int* arr);
```

如果形参是数组类型的引用，数组长度则称为参数的一部分

```
void foo(int (&arr)[10]);  // 必须接收 int [10] 类型的实参
```

数组形参的类型是指向数组元素类型的指针类型

```
extern void foo(int*);
extern void foo(int []);
extern void foo(int [10]);  // Three equivalent definitions of foo
```

多维数组实质是数组的数组，因此，对于多维数组参数，必须显式指明除了第一维之外所以维的长度。

```
void foo(int array[][10]);   // Okay,  等价于 void foo(int (*array)[10]);
```

含有可变参数的函数 TODO

initializer_list形参
省略符形参

默认实参

默认实参有利于函数接口的设计细节。调用函数时，可以省略有默认值的实参，编译器会为省略的实参提供默认值。
默认实参必须从函数形参表的尾部向前指定。
既可以在函数定义也可以在函数声明中指定默认实参，但是，在一个编译单元中，只能为一个形参指定默认实参一次。

```
// ff.h
int foo(int = 0);
// ff.cpp
#include "ff.h"
int foo(int i = 0) {} // error

// ff.h
int foo(int, int, int = 9);
// ff.cpp
int foo(int x, int y = 8, int z) {} // Ok
int foo(int x, int y = 8, int z = 9) {} // error
```

默认实参只对包含了默认值的文件有效，因此通常在函数声明中指定默认实参。

默认实参可以是任意形式的表达式，即可以是运行时才可决定的结果：

```
int foo(int value = calc()) {}
```

返回类型和return语句

返回非引用类型：返回值用于初始化函数调用处的对象（可能是临时对象）

返回引用类型：返回的是左值，我们可以为返回类型是非const引用的函数的结果赋值

```
char& get_val(string &str, string::size_type ix) { return str[ix]; }
get_val(s, 0) = 'A'; // ok

const string& shorterString(const string &s1, const string &s2);
shorterString("hi", "bye") = "X"; // error
```

列表初始化返回值
C++11规定，函数可以返回花括号包围的值的列表，用于对表示函数返回的临时量进行初始化
如果列表为空，临时量执行值初始化，否则，返回的值由函数的返回类型决定

```
vector<string> process()
{
    if (expected.empty()) return {}; // 返回一个空vector列表
    else if (expected == actual)
        return {"functionX", "Okay"};
    else
        return {"functionX", expected, actual};
}
```

主函数main()的返回值
我们允许main()函数没有return语句直接返回，此时编译器会隐式地插入一条返回0的return语句

返回数组指针

```
typedef int arrT[10];
using arrT = int[10]; // 等价于typedef int arrT[10]
arrT* func(int i);
int (*func(int i))[10]; // 等价于arrT* func(int i)
```

使用尾置返回类型（Trailing return type）
C++11引入了尾置返回类型，任意函数的定义都可以使用尾置返回类型，但是这种形式对于返回类型复杂的函数最为有效

```
auto func(int i) -> int (*)[10];
```

使用decltype

```
int array[10];
```

```
decltype(array) *foo(); // 返回值类型是int(*)[10]，数组作为decltype关键字的参数不发生退化
```

函数指针

函数的类型由函数的返回值类型和形参类型决定，省略号是函数类型的一部分。

```
int printf(const char*, ...);
int strlen(const char*);  // 和printf不是同一类型
```

当一个函数名没有被调用操作符修饰时，会被解释为指向该类型函数的指针。
取址运算符作用在函数名上也会产生指向该函数类型的指针。

```
int foo(int);
foo; // int (*)(int)
&foo; // int (*)(int)
```

在函数指针之间不存在隐式类型转换

```
typedef int (*PF1)(int);
typedef void (*PF2)(void);
PF1 pf1;
PF2 pf2;
pf1 = pf2; // error: invalid conversion
pf1 = (PF1)pf2; // ok: explicit cast
pf1 = static_cast<PF1>(pf2); // error
pf1 = reinterpret_cast<PF1>(pf2); // ok
```

通过函数指针调用函数，有如下两种形式：

```
(*pf)(); // 显式
pf(); // 缩写
```

使用typedef或using可以简化函数指针类型的声明和定义。

```
typedef int (*PF)(const string&, const string&);
using PF = int (*)(const string&, const string&); // 等价于typedef
PF pfs[2] = {foo1, foo2}; // 函数指针数组
PF (*pf)[2] = &pfs; // 指向函数指针数组的指针
pfs[0](s1, s2);
(*pf)[0](s1, s2); // 显式调用
```

函数指针也可作为函数的返回值类型

```
typedef int (*PF)(int*, int);
PF ff(int); // 等价于 int (*ff(int))(int*, int)
```

将auto和decltype用于函数指针类型

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
decltype(sumLength) *getFcn(const string&);
// 当用函数名作为decltype的参数时，返回的是函数类型而非指针，因此需要显式加上*
```

指向C链接属性的函数指针 和 指向C++链接属性的函数指针类型不同

```
extern "C" int (*PF1)(int); // C链接属性
int (*PF2)(int); // C++链接属性
```

链接指示符作用在函数以及其形参上

```
extern "C" int (*PF)(int (*)(int)); // 函数参数也具有C链接属性
extern "C" int (*PF)(int);
int foo(PF pf); // pf具有C链接属性，foo具有C++链接属性
```

内联函数（in-line function）

inline 函数可以避免函数调用的开销，它被编译器在函数调用点上内联的展开
inline 仅仅是对编译器的建议，编译器可以选择忽略它
inline 函数定义必须对编译器可见，以便编译器在函数调用点上将其展开

constexpr函数

constexpr函数是指能用于常量表达式的函数
constexpr函数的返回类型及所有形参的类型都必须是**字面值类型**，而且函数体中必须有且只有一条return语句

```
constexpr int new_sz() { return 100; }
constexpr int foo = new_sz(); // ok, foo是一个常量表达式

class T {};
constexpr T* foo() { return nullptr; }
constexpr T* p = foo();
```

执行初始化任务时，编译器把对constexpr函数的调用替换为其结果值，为了能在编译过程中随时展开，constexpr函数被隐式地指定为内联函数

constexpr函数体内也可以包含其他语句，只要这些语句在运行时不执行任何操作就可，例如，空语句，using声明和类型别名。

constexpr函数的返回值可以不是一个常量

```
constexpr size_t scale(size_t cnt) { return new_sz()*cnt; } // 如果arg是常量，则scale(arg)也是常量表达式
int arr[scale(2)]; // ok, scale(2)是常量表达式
int i = 2;
int arr2[scale(i)]; // error, i不是常量，因此scale(i)不是常量表达式
```

把内联函数和constexpr函数放在头文件中
和其它函数不同，内联函数和constexpr函数可以被多次定义，通常将它们的定义置于头文件中

assert预处理宏

assert宏定义在cassert头文件中，用于检查“不能发生”的条件

重载函数（Overloaded Function）

在同一作用域中的同名函数，但是形参表不同。
函数重载将在同一作用域中为函数取名的词汇复杂性中解放出来，如果不同的函数名对于程序的理解性可读性更有益处的话，则无需使用重载函数。

main()函数不能重载

当一个函数名在一个作用域中被声明多次的时候，编译器按照如下步骤解释后续的声明：
如果函数形参表不同，则是函数重载
如果形参表和返回类型精确匹配(不包括形参名)，则是重复声明（Okay）
如果形参表完全匹配，返回类型不同，则标识后续声明为重复声明（Error）
如果只有缺省实参不同，则为重复声明。

```
int foo(int, double) { return 0; }
int foo(int, double); // ok, declaration
void foo(int, double); // error, redefintion
void foo(int); // ok, overloaded
```

省略号(...)参与函数重载决议

bool类型参与函数重载决议

const和volatile限定符作用于形参：
顶层const不参与重载决议
底层const参与重载决议
修饰引用类型，参与重载决议：

```
int foo(int&);
int foo(volatile int&); // ok
int foo(const int&); // ok
```

修饰指针指向的类型（底层const），参与重载决议：

```
int foo(int *);
int foo(volatile int*); // ok
int foo(const int*); // ok
```

修饰指针本身，不参与重载决议：

```
int foo(int *);
int foo(int *volatile); // error, redefintion
int foo(int *const); // error, redefinition
```

修饰值传递类型，不参与重载：

```
int foo(int);
int foo(volatile int); // error, redefintion
int foo(const int);  // error, redefinition
```

函数重载和作用域
函数重载是针对一个作用域而言的，using指示符会对函数重载产生影响
using指示符引入的函数声明会参与对应作用域中的函数重载决议。

```
using namespace xxx;
int foo(int); // xxx名字域中的foo函数参与当前域中的函数重载
```

★ **C++中的名字查找发生在类型检查之前**

```
void print(const string &);
void print(double); // overloaded
void foo()
{
    void print(int); // 新作用域，隐藏了外部作用域的print函数
```

```
        print("hello"); // error，外部作用域的print(const string&)函数被隐藏
        print(3.14); // ok，调用的是print(int)，外部作用域的print(double)函数被隐藏
        print(100); // ok，调用的是print(int)
    }
```

C链接修饰符（extern "C"）只能修饰重载函数集中的一个（**C/C++链接性会决定编译器内部生成的名字**）

```
    extern "C" void foo(void); // foo
    extern "C" int foo(int); // error, redefintion, 还是 foo

    extern "C" void foo(void); // foo
    int foo(int); // ok，overloaded
```

给指向重载函数的指针初始化或赋值时，将会精确匹配返回值类型和形参表

```
    int foo(void);
    int foo(int);
    int (*FP)(int) = &foo; // 精确匹配
```

const_cast和重载函数

```
    const string& shorterString(const string &s1, const string &s2)
    {
        return s1.size() <= s2.size() ? s1 : s2;
    }

    string& shorterString(string &s1, string &s2)
    {
        auto &r = shorterString(const_cast<const string&>(s1), const_cast<const string&>(s2));
        return const_cast<string&>(r);
    }
```

重载决议（Function Overload Resolution）的步骤： TODO

第七章 类

2014年3月26日 星期三　　下午5:23

类的基本思想是数据抽象（data abstraction）和封装（encapsulation）。
数据抽象是一种依赖于接口（interface）和实现（implementation）分离的编程以及设计技术。
封装实现了类的接口和实现的分离，封装后的类隐藏了它的实现细节。

类提供了用户自定义自己的数据类型，类常被称为用户定义的类型（User-Defined Type, UDT）。
简而言之，类就是定义了一个新的类型和新的作用域。

类的声明和定义

类定义形成了一个独立的类域，类域可以嵌套。

```
class A { ... };
class A a1; // ok, C++沿用了C的声明方式
A a2; // ok, C++的声明方式
```

只有当类的类体被完整定义时，它才被视为已完全定义，所以一个类不能有其自身类型的数据成员。
当类头出现后，即可视为该类已经被声明，因此可以在类体内定义指向类类型的指针或者类引用类型的数据成员。

```
class A {
    A a_; // error
    A* a_; // ok
    A& a_; // ok
};
```

类声明（前向声明）引入了一个不完全定义。
不完全定义只可使用于定义指向该类型的指针和引用
或者用于声明（而不是定义）使用该类型作为形参或返回值类型的函数

```
class A; // forward declaration，前向声明
A* pa; // ok
A& a; // ok
A foo(A a); // ok
A foo(A a) {} // error
```

类定义不会引起内存的分配，只有类的实例出现时，才会分配内存

名字查找（name lookup）和类的作用域

对于定义在类内部的成员函数来说，解析名字的方式与传统的名字查找规则有所区别
类的定义分两步处理：

a. 首先，编译成员的声明
b. 直到类全部可见后才编译函数体

按照这种两阶段的方式处理类可以简化类代码的组织结构，因为成员函数体直到整个类可见后才会被处理
因此它能使用类中定义的任何名字

这种两阶段的处理方式只适用于成员函数体中使用的名字
声明中使用的名字，包括返回类型或者参数列表中使用的名字，都必须在使用前确保可见
如果某个成员的声明使用了类中尚未出现的名字，则编译器则会在定义该类的外部作用域中继续查找

```
typedef double Money;
string bal;
class Account {
public:
    Money balance() { return bal; } // Money是类外声明的类型别名，bal是类的数据成员
private:
    Money bal; // Money是类外声明的类型别名
}
```

类型名要特殊处理
一般来说，内层作用域可以重新定义外层作用域中的名字，即使该名字已经在内层作用域中使用过
然而在类中，如果成员使用了外层作用域中的某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字
因此类型名的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类型名的定义之后

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }
private:
    typedef double Money; // error, 不能重新定义Money, 用clang可以通过编译？？？
    Money bal;
    ...
};
```

成员函数中使用的名字按照如下方式解析：

a. 首先在成员函数内查找该名字的声明，只有在该名字出现之前的声明才被考虑
b. 如果在成员函数内没有找到，则在类内继续查找，这时类的所有成员都可以被考虑
c. 如果类内也没有找到该名字的声明，则在**成员函数定义之前的作用域内**继续查找

```
int height;
class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height)
    {
        cursor = width * height; // 哪个height? 形参
    }
private:
    pos height = 0, width = 0;
    pos cursor = 0;
}
```

尽管类的成员被隐藏了，但我们仍然可以通过加上类名或显式地使用**this**指针来强制访问成员

```
cursor = width * this->height; // 数据成员
cursor = width * Screen::height; // 数据成员
```

尽管外部作用域的对象被隐藏了，但我们可以使用作用域运算符访问到它

```
cursor = width * ::height; // 外部作用域定义的height
```

在外部作用域的名字查找不仅仅是类定义之前的外部作用域，还包括成员函数定义之前的外部作用域中的声明

```
int height;
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0;
};
Screen::pos verify(Screen::pos); // 在setHeight成员函数定义之前
void Screen::setHeight(pos var) {
    height = verify(var); // ok，verify可以被正常调用
}
```

成员函数（member function）

在类内部定义的函数是隐式的**inline**函数，也可以加上**inline**关键字做为显式声明。

```
class A
{
    int foo() {}  // inline函数
};
```

在类体外定义的函数，可以加上**inline**关键字显式地声明**inline**函数。

成员函数可以被重载

对非**static**成员函数，编译器隐式地增加了一个**this**指针参数，用于区分类的不同对象实例。

非**const**成员函数的**this**指针类型是指向类类型的**const**指针。

```
A::a(); // this的类型: A* const
```

const成员函数的**this**指针类型是指向**const**类类型的**const**指针。

```
A::a() const; // this的类型: const A* const
```

非常量对象，以及非常量对象的指针和引用可以调用**const**成员函数和非**const**成员函数
常量对象，以及常量对象的指针和引用只能调用**const**成员函数

this指针是被隐式地定义的，但是可以在成员函数中显式地使用**this**指针。
对**this**指针解引用得到了类对象。

```
class A {
public:
    A& test() { return *this; }
}
A a; // this: &a; *this: a
```

每个类的实例共享类成员函数的代码段。

```
class A
{
public:
    A(int value) { m_value = value; }
    void Print1() { printf("hello world"); }
    void Print2() { printf("%d", m_value); } // 相当于printf("%d", this->m_value);

private:
    int m_value;
```

```
};

int main()
{
    A *pA = nullptr;
    pA->Print1(); // 非虚函数
    pA->Print2(); // 通过this指针访问对象的数据成员 m_value，segmentation fault
    return 0;
}
```

类成员访问

关键字public，private 和 protected 称为访问限定符（access specifier）
公有成员（public）可以在任何地方被访问
私有成员（private）可以被类成员函数和类友元访问
保护成员（protected）可以被类成员函数和派生类访问，亦可被友元访问

使用class和struct定义类的唯一区别就是默认的访问权限
在缺省情况下，未指定访问限定符，class中的成员为private的，struct中的成员为public的

友元

友元机制允许一个类授权其它的函数和类访问其非公有成员
友元可以是一个命名空间函数，也可以是类成员函数，或者是一个类
友元声明只可出现在类体中，友元不是类的成员也不受访问限定符的约束
友元声明仅仅指定了访问的权限，而不是一个通常意义上的函数声明

```
class A
{
    friend int foo(int);  // 友元声明，普通函数

    // B::foo必须在A类之前被声明
    friend int B::foo(int); // 类成员函数

    friend class B; // 类
};

int foo(int); // 函数声明，不是必须声明在友元声明之前
```

每个类负责控制自己的友元类或友元函数，友元关系不存在传递性。

尽管重载函数的名字相同，但是它们仍然是不同的函数，如果一个类想把一组重载函数声明成它的友元，需要对这组重载函数的每一个分别声明

类和非成员函数的声明不是必须在它们的友元声明之前，当一个名字第一次出现在一个友元声明中时，我们隐式地假定该名字在当前作用域中是可见的。

友元函数可以定义在函数的内部，但是即使在类的内部定义了该友元函数，我们也必须在类的外部提供相应的声明从而使得函数可见
当然有些编译器并不强制执行关于友元的限定规则

```
struct X {
    friend void f() {} // 定义在类内部
    ...

    void foo() { f(); } // error, f还没有被声明
};
void X::g() { f(); } // error, f还没有被声明
void f();
void X::h() { f(); } // ok
```

类成员

类型成员

除了数据成员和函数成员外，类还可以自定义某种类型在类中的别名
用来定义类型的成员必须先定义后使用

```
class Screen
{
public:
    typedef std::string::size_type pos;
    ...
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};
class Screen
{
public:
    using pos = std::string::size_type; // 等价于typedef
    ...
```

```
};
```

在类的作用域之外，使用作用域运算符访问类类型成员

```
Screen::pos ht = 10, wd = 10;

Screen::pos Screen::size() const  // 类外定义的成员函数的返回类型需要用作用域运算符显式地指明类
{
    return height * width;
}
```

可变数据成员（mutable data member）

将数据成员声明为mutable，则允许在const成员函数中修改该数据成员

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr; // 可变数据成员
};
void Screen::some_member() const
{
    ++access_ctr;
}
```

类数据成员的初始值

当我们提供一个类内初始值时，必须以符号＝或者花括号表示

```
class Window_mgr {
private:
    std::size_t window_cnt = 0; // 类内初始值
    std::vector<Screen> screens{Screen(24, 80, '')}; // 类内初始值
};
```

const成员函数

为了尊重类的实例的常量性，编译器必须区分安全的和不安全的成员函数
语义上，将成员函数声明为const，表示函数不会修改类的对象
const在成员函数的声明和定义处都需要显式声明

```
// a.h
class A
{
int foo(int) const;
};
// a.cpp
int foo(int) const { ... }
```

指向const成员函数的指针类型需要显式地声明const限定符

```
int (A::*)(int) const;
```

const成员函数对应的this指针类型是指向const类对象的const指针

将一个成员函数声明为const并不能完全阻止可能做到的修改动作，例如通过指针成员去间接修改指针所指的对象

```
class Text
{
public:
    void foo(const string& s) const;
private:
    char* text_;
};

void Text::foo(const string& s) const
{
    text_ = s.cstr();  // error：编译器可以监测到
    text_[0] = s[0];  // ok：通过指针间接的修改，编译器无法监测到这种行为
}
```

类的const对象只能调用类的const成员函数，除了类的构造函数和析构函数

const限定符修饰类成员函数参与类函数重载决议

const成员函数所绑定的this指针类型为指向const类对象的const指针
因此，const成员函数返回值类型如果是类的引用类型的话，只能是const引用

```
class A {
const A& GetThis() const { return *this; }
    // this指针类型是 const A* const，解引用后*this的类型是const A，因此只能绑定到const A&上

A& GetThis() const { return *this; }
```

```
    // error: invalid initialization of reference of type 'A&' from expression of type 'const A'
};
```

volatile成员函数
volatile类实例只能调用volatile类成员函数，构造函数和析构函数除外。
volatile限定符参与函数重载决议。
volatile成员函数的this指针类型是指向const类类型对象的volatile指针。

构造函数

构造函数是特殊的成员函数，在创建类的对象的时候用于初始化对象

构造函数的执行可以看成两个阶段，初始化阶段，然后是执行函数内的语句
无论是否在初始化列表中显式初始化，类类型的数据成员总是在初始化阶段被初始化

```
class A
{
public:
    A(const string& s);
private:
    string name_;
};
A(const string& s) : name_(s) {} // 调用string的构造函数初始化name_
A(const string& s) // 调用string的缺省构造函数初始化name_
{
    name_ = s; // 赋值运算
}
```

构造函数和类名同名，没有返回值，可以有形参

构造函数可以被重载

构造函数不能声明为const的，当创建类的一个const对象时，直到构造函数完成初始化过程，对象才真正取得其const属性

```
class A {
public:
    A() const;  // error
};
```

类通过一个特殊的构造函数来控制默认初始化过程，这个函数称之为默认构造函数（default constructor）

默认构造函数无需任何实参
如果一个构造函数为所有参数都提供了默认实参，那它实际上也定义了默认构造函数

```
struct Sales_data
{
    Sales_data() {}
    Sales_data(int i = 100) {}
};
Sales_data s; // error, ambigious
```

合成的默认构造函数（Synthesized Default Constructor）
如果我们的类没有显式地定义构造函数，那么编译器会隐式地定义一个默认构造函数。
编译器创建的构造函数称为合成的默认构造函数（synthesized default constructor）
它按照如下规则初始化类的数据成员：

a. 如果存在类内初始值，用它来初始化数据成员
b. 否则，默认初始化该数据成员

只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数

定义在块中的内置类型和复合类型（数组和指针）的对象被默认初始化时，它们的值是未定义的
因此如果类包含有内置类型或复合类型的数据成员，则只有当这些成员全都被赋予了类内的初始值时，这个类才适合于使用合成的默认构造函数

有的时候编译器不能为某些类合成默认构造函数，例如，如果类中某个数据成员的类型没有默认构造函数，那么编译器将无法初始化该成员

C++11中，如果我们需要默认的行为，可以通过在参数列表后面加上=default来要求编译器生成构造函数
如果=default在类内部，则默认构造函数是内联的；如果在类外部，则默认情况下不是内联的

```
struct Sales_data {
    Sales_data() = default;
    Sales_data(const std::string &s) : bookNo(s) {}
    ...
};
或
struct Sales_data
{
    Sales_data();
};
```

```
Sales_data::Sales_data() = default;
```

构造函数初始值列表（constructor initialize list）
通常情况下，构造函数使用类内初始值不失为一个好的选择
当某个数据成员没有被构造函数初始值列表显式地初始化，它将以与合成默认构造函数相同的方式隐式初始化

```
Sales_data(const std::string &s, unsigned n, double p) :
    bookNo(s), units_sold(n), revenue(p*n) {}
```

如果数据成员是const，引用或属于某种没有提供默认构造函数的类类型的时候，必须通过构造函数初始值列表为这些数据成员提供初值

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
ConstRef::ConstRef(int ii)
{
    i = ii;
    ci = ii; // error, 未初始化
    ri = i; // error, 未初始化
}

ConstRef::ConstRef(int ii) : ci(ii), ri(i) {} // ok
```

构造函数初始值列表只用于初始化数据成员的值，而不限定初始化的具体执行顺序
数据成员的初始化顺序与它们在类定义中的出现顺序一致

委托构造函数（Delegating Constructor）
一个类的委托构造函数使用它所属类的其它构造函数执行他自己的初始化过程，或者说他把它自己的一些（或者全部）职责委托给了其它构造函数
当一个构造函数委托给另一个构造函数时，受委托的构造函数的初始值列表和函数体被依次执行，最后才会执行委托者的函数体

```
class Sales_data {
public:
    Sales_data(std::string s, unsigned cnt, double price) :
        bookNo(s), units_sold(cnt), revenue(cnt*price) {}

    // delegating constructor
    Sales_data() : Sales_data("", 0, 0) {}
    Sales_data(std::string s) : Sales_data(s, 0, 0) {}
    Sales_data(std::istream &is) : Sales_data() { ... }
};
```

隐式的类类型转换
可以用单个实参调用的构造函数实际定义了从构造函数的参数类型转换为此类类型的隐式转换机制
我们称这种构造函数为转换构造函数（converting constructor）

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = "") : isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
};
string null_book = "9-999-99999-9";
item.same_isbn(null_book); // ok
item.same_isbn(cin);       // ok
```

只允许一步类类型转换，编译器只会自动地执行**一步类型转换**

可以通过将构造函数声明为explicit，来抑制由构造函数定义的隐式转换
explicit关键字只能出现在类内的构造函数声明处

```
class Sales_item {
public:
    // default argument for book is the empty string
    explicit Sales_item(const std::string &book = "") : isbn(book), units_sold(0), revenue(0.0) { }
    explicit Sales_item(std::istream &is);
};
item.same_isbn(null_book); // error: string constructor is explicit
item.same_isbn(cin);       // error: istream constructor is explicit
```

通常而言，除非有明显的理由需要定义隐式转换，否则，单个实参可以调用的构造函数应该声明为explicit。

当使用explicit关键字声明构造函数时，它只能以直接初始化的形式使用

```
Sales_data item1(null_book); // ok, 直接初始化
Sales_data item2 = null_book; // error, 不能用于拷贝初始化
```

尽管编译器不会将explicit构造函数用于隐式转换过程，但我们可以使用这样的构造函数显式地进行强制转换

```
item.combine(Sales_data(null_book)); // ok
item.combine(static_cast<Sales_data>(cin)); // ok
```

标准库中含有显式构造函数的类

接受一个单参数const char*的string构造函数不是explicit的

```
string str = "hello world"; // ok, copy initialization
string str("hello world"); // ok, direct initialization
```

接受一个容量参数的vector构造函数是explicit的

```
vector<int> v(n); // ok, direct initialization
vector<int> v = n; // error, explicit constructor
```

聚合类（Aggregate class）

聚合类使得用户可以直接访问其成员，并且具有特殊的初始化语法

当一个类满足如下条件时，我们说它是聚合的

a. 所有成员都是public的
b. 没有定义任何构造函数
c. 没有类内初始值
d. 没有基类，也没有virtual函数

和初始化数组元素的规则一样，如果初始化列表中的元素个数少于类的数据成员个数，则靠后的成员被值初始化

```
struct Data {
    int ival;
    string s;
};
Data val1 = {0, "Anna"};
```

字面值常量类

TODO

类的静态成员

类的static成员的名字位于类域中

```
int foo(); // 全局域
class A
{
    static int foo(); // 类域
};
```

类的static成员可以是public的或private的

类的static成员可以通过类的对象，引用或指针访问，也可以用类名加上限定修饰符（::）访问

```
class A
{
    static int foo();
};
A::foo();
A a;
A &aa = a;
A *pa = &a;
a.foo();
aa.foo();
pa->foo();
```

类的static成员独立于类的所有实例，类的static成员是类的一部分，但不是类对象的组成部分。

```
class A
{
    static int x; // sizeof(A) = 1
};
int A::x = 10;

class B
{
    int x; // sizeof(B) = 4
};
```

static成员函数没有this指针

指向static成员函数的指针和普通指针一样。

```
class A
{
    static int foo();
};
int (*pf)() = &A::foo;
```

static成员函数只能访问静态数据成员，因为static成员函数没有this指针，所以无法区分类的对象。

static成员函数不能被声明为const和volatile
在语义上，const成员函数不会修改类的对象，类的对象是通过this指针加以区分，而static成员函数并没有this指针。

static成员函数不能被声明为virtual

static只可出现在类体内，而不能出现在类体外的函数定义处
```
class A {
    static int foo(int);
};
int A::foo(int) {} // 不需要static
```

static数据成员不是通过构造函数初始化的，而是在类定义的时候初始化

static数据成员必须在类体外定义：
```
class A {
    static int x_;   // 声明
};
int A::x_ = 100;  // 定义+初始化
```

ODR（One Definition Rule）要求static数据成员的定义不应该放在头文件中

static数据成员的类型可以是不完全类型
```
class A {
    static A a; // ok
    A* a; // ok
    A& a; // ok
    A a; // error
};
```

static数据成员可以作为类成员函数的缺省实参。
```
class Screen {
public:
    Screen& clear(char = bkground);
private:
    static const char bkground = '#';
};
```

通常情况下，类的static成员不应该在类的内部初始化。
然而，我们可以为static成员提供const类型的类内初始值，不过要求static成员必须是字面值常量类型的constexpr，初始值必须是常量表达式
```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);
private:
    static const int period = 30; // ok，等价于static constexpr int period = 30;
    double daily_tbl[period]; // ok, period is constant expression
};

struct A
{
     static constexpr double a_ = 1.00; // ok
};
```

const的static成员在类体内初始化时，仍然需要类体外的定义（无须指定初始化值）。
```
constexpr int Account::period; // ok, definition
```

指向类的非static成员的指针

```
class Screen {
public:
    typedef std::string::size_type index;
    char get() const; // char (Screen::*)() const
    char get(index ht, index wd) const;  // char (Screen::*)(index, index) const
private:
    std::string contents;   // 指向Screen类中的std::string类型的指针类型为：std::string Screen::*
    index cursor;
    index height, width;
};
std::string Screen::*ps = &Screen::contents;  // 和普通指针类型相比增加了类类型
```

引用指向类的非static数据成员的指针需要通过类的对象
```
Screen screen, *pscreen;
screen.*ps;
```

```
pscreen->*ps;
```

除了普通函数指针要求的参数表类型和返回类型外（const和volatile限定符也需匹配），还需显式地加上对应的类类型

```
char (Screen::*pf)() const = &Screen::get;
```

引用指向类的非static成员函数的指针需要通过类的对象

```
Screen screen, *pscreen;
(screen.*pf)();
(pscreen->*pf)();
```

第十三章 拷贝控制

2014年3月28日 星期五　　下午1:17

拷贝构造函数（Copy Constructor）

如果一个构造函数的第一个参数是**自身类类型的引用**，且任何额外参数都有默认值，则此构造函数是拷贝构造函数
虽然可以定义一个非const引用的拷贝构造函数，但是此参数几乎总是一个const的引用
拷贝构造函数在几种情况下都会被隐式调用，因此拷贝构造函数通常不应该是explicit的

拷贝构造函数的第一个参数类型必须是引用类型
如果不是引用类型则调用永远不会成功，为了调用拷贝构造函数，我们必须拷贝它的实参
为了拷贝实参，我们又必须调用拷贝构造函数，由此循环下去

合成拷贝构造函数（synthesized copy constructor）
如果编译器没有为类定义一个拷贝构造函数，编译器会为我们定义一个拷贝构造函数
编译器从给定对象中依次将每个非static成员拷贝到正在创建的对象中
对于类类型的成员，会调用其拷贝构造函数来拷贝
对于内置类型的成员则直接拷贝
对于数组类型，则逐个元素的拷贝，如果数组元素是类类型，则用元素类型的拷贝构造函数进行拷贝

拷贝初始化（copy initialization）
当使用直接初始化时，编译器使用普通的函数匹配来选择参数最匹配的构造函数
当使用拷贝初始化时，编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还会进行隐式类型转换

```
string dots(10, '.'); // 直接初始化
string s(dots); // 直接初始化
string s2 = dots; // 拷贝初始化
string null_book = "9-999-99999-9"; // 拷贝初始化
string nines = string(100, '9'); // 拷贝初始化
```

拷贝初始化在如下情况下发生

a. 使用＝定义变量时
b. 将一个对象作为实参传递给一个非引用类型的形参
c. 将一个返回类型为非引用类型的函数返回一个对象
d. 用花括号列表初始化一个数组中的元素或一个聚合类中的成员
e. 某些类类型还会对它们所分配的对象使用拷贝初始化，例如初始化标准库容器或是调用insert或push成员，与之相反，用emplace则相反

拷贝初始化的限制
如果我们使用的初始化值需要调用一个explicit的构造函数来进行类型转换
那么只能使用直接初始化或者显式地调用explicit构造函数

```
vector<int> v1(10); // 直接初始化
vector<int> v2 = 10; // error, 接受大小参数的构造函数是explicit的
void f(vector<int>);
f(10); // error
f(vector<int>(10)); // ok, 显式地调用构造函数

class A
{
public:
    A() {}
    explicit A(const A&) {}
};

A a1; // ok, default constructor
A a2 = a1; // error, direct initialization, copy constructor is explicit
A a3(a1); // ok, explicityly called copy constructor
```

编译器可以绕过拷贝构造函数进行优化
在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象

```
string null_book = "9-999-99999-9"; // 1)
string null_book("9-999-99999-9"); // 2) 编译器可能将1)转换为，直接略过拷贝/移动构造函数
```

但是即使编译器略过了拷贝/移动构造函数，拷贝/移动构造函数必须是存在而且可以访问的（不能是private的）

拷贝赋值运算符（Copy Assignment Operator）

如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个
拷贝赋值运算符接受一个与其所在类相同类型的参数

```
class Foo {
public:
```

```
        Foo& operator=(const Foo&);
    };
```

为了和内置类型的赋值保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用
通常标准库要求保存在容器中的类型要有赋值运算符，且其返回值是左侧运算对象的引用

合成拷贝赋值运算符（synthesized copy-assignment operator）
将右侧运算对象的每个**非static成员**赋予左侧运算对象的对应成员
这一工作是通过成员类型的拷贝赋值运算符来完成的
对于数组类型的成员，逐个赋值数组元素

析构函数（Destructor）

析构函数执行与构造函数相反的操作
构造函数初始化对象的非static数据成员以及一些其他工作
析构函数释放对象使用的资源，并销毁对象的非static数据成员

析构函数没有返回值，也没有参数，不能被重载

在析构函数中，首先执行函数体，然后销毁成员
成员按照初始化顺序的逆序销毁

析构函数的析构部分是隐式的，成员销毁时发生什么完全依赖于成员的类型
销毁类类型的成员需要调用成员自己的析构函数，内置类型没有析构函数，因此销毁内置类型成员什么也不需要做

无论何时一个对象被销毁，就会自动调用其析构函数
a. 变量在离开其作用域时被销毁
b. 当一个对象被销毁时，其成员被销毁
c. 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁
d. 对于动态分配的对象，当对指向它的指针应用delete运算符时被销毁
e. 对于临时对象，当创建它的表达式结束时被销毁

合成析构函数（synthesized destructor）
当一个类未定义自己的析构函数时，编译器会为它定义一个合成析构函数

三/五法则

如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数

需要拷贝操作的类也需要赋值操作，反之亦然

使用＝default

```
class Sales_data {
public:
    Sales_data() = default; // inline
    Sales_data(const Sales_data&) = default; // inline
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default; // inline
};

Sales_data& Sales_data::operator=(const Sales_data &) = default; // 非内联
```

阻止拷贝

定义删除的函数（=delete）
C++11允许将拷贝构造函数和拷贝赋值运算符定义为删除的函数（deleted function）
删除的函数时这样一种函数，虽然我们声明了它们，但是不能以任何方式使用它们
和=default不同，=delete必须出现在第一次声明的地方

```
    struct NoCopy {
        NoCopy() = default;
        NoCopy(const NoCopy&) = delete; // 阻止拷贝
        NoCopy& operator=(const NoCopy&) = delete; // 阻止赋值
    };
```

对于析构函数已删除的类型，不能定义该类型的对象或者释放指向该类型动态分配对象的指针

```
    struct NoDtor {
        NoDtor() = default;
        ~NoDtor() = delete;
    };
    NoDtor nd; // error
    NoDtor *p = new NoDtor(); // ok
```

```
    delete p; // error
```

private拷贝控制
在C++11标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为private来阻止拷贝
我们将这些拷贝控制成员声明为private的，并且不定义它们，声明但不定义一个成员函数是合法的

```
    class PrivateCopy {
    public:
        PrivateCopy() = default;
        ~PrivateCopy();

    private:
        PrivateCopy(const PrivateCopy&);
        PrivateCopy& operator=(const PrivateCopy&);
    };
```

合成的拷贝控制成员可能是删除的
本质上，当不可能拷贝，赋值或者销毁类的成员时，类的合成的拷贝控制成员就被定义为删除的
对某些类来说，编译器将会把合成的拷贝控制成员定义为删除的函数

a. 如果类的某个成员的析构函数是删除的或不可访问的，则类的合成析构函数被定义为删除的
b. 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的
   如果类的某个成员的析构函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的
c. 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个const的或引用成员，则类的合成拷贝赋值运算符被定义为删除的
d. 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化值，或是类有一个const成员，它没有类内初始化值且其类型未显式地定义默认构造函数，则该类的默认构造函数被定义为删除的

行为像值的类

```
class HasPtr
{
public:
    HasPtr(const string &s = string()) :
        ps(new string(s)), i(0) {}

    HasPtr(const HasPtr &p) :
        ps(new string(*p.ps)), i(p.i) {}

    ~HasPtr() { delete ps; }

    HasPtr& operator=(const HasPtr &rhs) // 异常安全，可以处理自赋值情况
    {
        auto newp = new string(*rhs.ps);
        delete ps;
        ps = newp;
        i = rhs.i;
        return *this;
    }

private:
    std::string *ps;
    int i;
};
```

行为像指针的类

```
class HasPtr
{
public:
    HasPtr(const string &s = string()) :
        ps(new string(s)), i(0), use(new size_t(1)) {}

    HasPtr(const HasPtr &p) :
        ps(p.ps), i(p.i), use(p.use) { ++*use; }

    HasPtr& operator=(const HasPtr &rhs)
    {
        ++*rhs.use;
        if (--*use == 0)
        {
            delete ps;
            delete use;
        }
        ps = rhs.ps;
        i = rhs.i;
        use = rhs.use;
```

```
        return *this;
    }

    ~HasPtr()
    {
        if (--*use == 0)
        {
            delete ps;
            delete use;
        }
    }

private:
    string *ps;
    int i;
    size_t *use;
};
```

对象移动

C++11一个最主要的特性是可以移动而非拷贝对象的能力。
在某些情况下，对象拷贝后就立即被销毁了。在这些情况下，移动而非拷贝对象会大幅提升性能。

右值引用

必须绑定到右值的引用，通过&&来获得右值引用
右值引用只能绑定到一个将要销毁的对象，因此我们可以自由的将一个右值引用的资源移动到另一个对象中
右值引用不能绑定到一个左值上

```
int i = 42;
int &r = i; // 左值引用
int &&rr = i; // error
int &&rr = 42; // ok, 绑定到字面常量
int &&rr2 = i*42; // ok, 绑定到右值
const int &r2 = 42; // ok, const的左值引用
```

左值持久，右值短暂

变量是左值，因此不能将一个右值引用绑定到一个变量上，即使这个变量是右值引用类型也不行

```
int &&rr1 = 42;
int &&rr2 = rr1; // error
```

可以显式地将一个左值转换为对应的右值引用类型
可以通过std::move()函数获得绑定到左值上的右值引用

```
#include <utility>
int &&rr3 = std::move(rr1); // ok
```

调用std::move()意味着承诺：**我们可以销毁一个移后源对象，也可以赋予它新值，但不是使用一个移后源对象的值**

移动构造函数（move constructor）和移动赋值运算符（move-assignment operator）

移动构造函数

移动构造函数的第一个参数是该类类型的一个右值引用，其它任何额外的参数都有默认实参
除了完成资源移动，移动构造函数还必须确保移后源对象处于一个状态，即销毁它是无害的
一旦资源完成移动，源对象必须不在指向被移动的资源，这些资源的所有权已经归属新创建的对象

```
StrVec::StrVec(StrVec &&s) noexcept : // 移动操作不应抛出异常
    elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 另s进入这样的状态，即对它析构是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

不抛出异常的移动构造函数和移动赋值运算符必须标记为noexcept
虽然移动操作通常不抛出异常，但是抛出异常也是允许的，同时，标准库容器对异常发生时其自身的行为提供保障
例如，vector保证，如果我们调用push_back()时发生异常，vector自身不会发生改变，
因此除非vector知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数

移动赋值运算符

```
StrVec& StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值，此右值rhs可能是move调用返回的结果
    if (this != &rhs)
    {
        free();
        elements = rhs.elements;
```

```
            first_free = rhs.first_free;
            cap = rhs.cap;

            rhs.elements = rhs.first_free = rhs.cap = nullptr;
        }

        return *this;
    }
```

移后源对象必须可析构

在移动操作之后，移后源对象必须保持有效的，可析构的状态，但是用户不能对其值进行任何假设

合成的移动操作

只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，
编译器才会为它合成移动构造函数或移动赋值运算符

```
struct X {
    int i; // 内置类型可以移动
    std::string s; // string定义了自己的移动操作
};

struct hasX {
    X mem; // X有合成的移动操作
};

X x, x2 = std::move(x); // 使用合成的移动构造函数
hasX hx, hx2 = std::move(hx); // 使用合成的移动构造函数
```

定义了一个移动构造函数或移动赋值运算符的类也必须定义自己的拷贝操作，否则，这些成员默认地被定义为删除的

与拷贝操作不同，移动操作永远不会隐式地定义为删除的函数，但是如果我们显式地要求编译器生成=default的移动操作且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数 TODO

移动右值，拷贝左值

如果一个类既有移动构造函数，也有拷贝构造函数，编译器使用普通的函数匹配规则来去定使用哪个构造函数

```
StrVec v1, v2;
v1 = v2; // v2是左值，使用拷贝赋值
StrVec getVec(istream &);
v2 = getVec(cin); // getVec返回的是右值，StrVec&&是精确匹配，使用移动赋值
```

如果没有移动构造函数，右值也被拷贝

如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来移动的
拷贝赋值运算符和移动赋值运算符的情况类似
用拷贝构造函数代替移动构造函数几乎肯定是安全的（赋值运算符的情况类似）

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&);
    ...
};
Foo x;
Foo y(x); // 拷贝构造函数，x是左值
Foo z(std::move(x)); // 拷贝构造函数，因为没有定义移动构造函数
```

拷贝并交换赋值运算符和移动操作

```
class HasPtr {
public:
    HasPtr(HasPtr &&p) noexcept :
        ps(p.ps), i(p.i)
    {
        p.ps = nullptr;
    }

    HasPtr& operator=(HasPtr rhs)
    {
        swap(*this, rhs);
        return *this;
    }
};

hp = hp2; // hp2是一个左值，rhs由拷贝构造函数初始化
```

```
hp = std::move(hp2); // rhs由移动构造函数初始化
```

第十五章 面向对象程序设计

2014年4月9日 星期三　　上午8:12

面向对象程序设计（Object-oriented programming）的核心思想是数据抽象，继承和动态绑定
通过**数据抽象**我们可以将类的接口和实现分离
使用继承（inheritance）可以定义类似的类型并对其相似的关系建模
使用动态绑定（dynamic binding）可以在一定程度上忽略相似类型的区别，而已统一的方式使用它们的对象

OOP的核心思想是多态性（polymorphism），我们把具有继承关系的多个类型称为多态类型，
因为我们能使用这些类型的多种形式而无须在意它们的差异
引用或指针的静态类型和动态类型可能不一致这一事实是C++语言支持多态性的根本所在

基类

基类通常都应该定义一个虚析构函数，即使该函数不执行任何操作也是如此
在C++语言中，基类必须将它的两种成员函数区分开来，一种是基类希望其派生类进行**覆盖（override）**的函数，
另一种是基类希望派生类直接继承而不要改变的函数，对于前者，基类通常将其定义为虚函数（virual）

任何构造函数之外的非静态函数都可以是虚函数

如果基类把一个函数定义为虚函数，那么该函数在派生类中隐式地也是虚函数

成员函数如果没有被声明为虚函数，则其解析过程发生在编译时而非运行时

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员
派生类能访问继承而来的公有成员，而不能访问继承而来的私有成员
如果基类希望它的派生类有权访问某一类成员，同时禁止其他用户访问，则可以用protected访问说明符说明这样的成员

派生类

派生类必须将其继承而来的成员函数需要覆盖的那些重新声明
派生类经常但不总是覆盖继承的虚函数，如果派生类没有覆盖基类中的某个虚函数，则派生类会直接继承其在基类中的版本

一个派生类对象包含多个组成部分，一个含有派生类自己定义的（非静态）成员的子对象，以及一个与该派生类继承的基类对应的子对象
C++标准并没有明确规定派生类的对象在内存中如何分布，在一个对象中，继承自基类的部分和派生类自定义的部分不一定是连续存储的

派生类到基类（derived-to-base）类型转换
因为派生类对象中含有与其基类对应的组成部分，
所以我们能把派生类对象当成基类对象来使用，而且也能将基类的指针或引用绑定到派生类对象中的基类部分

```
class Derived : public Base {};
Base b;
Derived d;
Base *pb1 = &b;
Base *pb2 = &d;
Base &rb = d;
```

编译器会隐式地执行派生类到基类的转换，这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引用的地方
同样也可以把派生类对象的指针用在需要基类指针的地方

在派生类对象中含有其与基类对应的组成部分，这一事实是继承的关键所在

派生类构造函数
每个类控制它自己的成员初始化过程，派生类也必须使用基类的构造函数来初始化它的基类部分
除非我们显式地指出，否则派生类对象的基类部分会像数据成员一样执行默认初始化
如果想使用其它的基类构造函数，则需要以类名加圆括号内的实参列表的形式为构造函数提供初始值

派生类可以访问基类的公有成员和受保护成员

派生类的作用域嵌套在基类的作用域之内

如果想将某个类用做基类，则该类必须已经定义而非仅仅声明；一个类不能派生自它自身

```
class Base;
class Derived : Base { ... }; // error
```

C++11引入了关键字final用于防止继承的发生

```
class Noderived final {};
class Base {};
```

```
class Last final : Base {};
class Bad : Noderived {}; // error
class Bad2 : Last {}; // error
```

静态成员
如果基类定义了一个静态成员，则在整个继承体系中只存在该成员的唯一定义
静态成员遵循通用的访问规则，如果基类中的成员是private的，则派生类无权访问

类型转换和继承
理解基类和派生类之间的类型转换是理解C++语言面向对象编程的关键所在

通常情况下，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致，
或者对象的类型含有一个可接受的const类型转换规则
存在继承关系的类是一个重要的例外：**我们可以将基类的指针或引用绑定到派生类对象上**

和内置指针一样，智能指针类也支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内

静态类型（static type）和动态类型（dynamic type）
静态类型是编译时已知的，动态类型是运行时才可知
基类的指针或引用的静态类型可能与其动态类型不一致

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致

不存在从基类向派生类的隐式类型转换
因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换
```
Base b;
Derived *d = &b; // error
Derived &d = b; // error
```

即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类向派生类的转换
```
Derived d;
Base *pb = &d;
Derived *pd = pb; // error
```
编译器在编译时无法确定某个特定的转换在运行时是否安全，这是因为编译器智能通过检查指针或引用的静态类型来推断该转换是否合法
如果在基类中含有一个或多个虚函数，我们可以使用dynamic_cast请求一个类型转换，该转换的安全检查将在运行时执行
如果我们已知某个基类向派生类的转换是安全的，则我们可以使用static_cast来强制覆盖掉编译器的检查工作

在对象之间不存在类型转换
派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换
当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝，移动或赋值，它的派生类部分被忽略掉
```
Derived d;
Base b(d); // 派生类部分被切割（sliced down）
b = d;  // 派生类部分被切割（sliced down）
```

虚函数
通常情况下，如果我们不使用某个函数，则无需为该函数提供定义
但是我们必须为每一个虚函数提供定义，而不管它是否被用到了，这是因为编译器也无法确定到底会使用哪个函数

对虚函数的调用可能在运行时才被解析

派生类中的虚函数
如果一个派生类的函数覆盖了继承而来的虚函数，则它的形参类型必须与它覆盖的基类虚函数完全一致

同样，派生类中的虚函数的返回类型也必须与基类函数匹配
该规则存在一个例外，当类的虚函数返回类型是类本身的指针或引用时，上述规则无效
也就是说，如果D由B派生而来，则基类的虚函数可以返回B*而派生类的对应函数可以返回D*，
只不过这样的返回类型要求从D到B的类型转换是可访问的

final和override说明符
派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这仍然是合法的行为，编译器认为新定义的函数与基类中原有的函数是相互独立的，派生类的函数并没有覆盖掉基类中的版本，但是这种情况经常意味着发生了错误

C++11引入了override关键字来说明派生类中的虚函数，这样使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误
如果用override标记了某个虚函数，但该函数没有覆盖已存在的虚函数，此时编译器将报错

```
struct B
{
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

struct D1 : B
{
    void f1(int) const override;
    void f1(int) override; // error, 没有覆盖基类的虚函数
    void f3() override; // error, 不是虚函数
    void f4() override; // error, 新声明的函数
};
```

如果我们将某个函数标记为`final`，则之后任何尝试覆盖该函数的操作都将引发编译器报错

```
struct D2 : B {
    void f1(int) const final;
};

struct D3 : D2 {
    void f2();
    void f1(int) const; // error
};
```

虚函数和默认实参

如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定
因此如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致

回避虚函数的机制

可以使用作用域运算符来实现回避虚函数

```
double undiscounted = baseP->Quote::net_price(42); // 强制调用基类定义的版本
```

如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，
则在运行时该调用会被解析为对派生类版本自身的调用，从而导致无限递归

抽象基类（abstract base class）

纯虚函数（`pure virtual`），在类的内部声明虚函数时，在分号之前使用了`=0`。
一个纯虚函数不需要（但是可以）被定义，不过函数体必须定义在类的外部

含有纯虚函数（或者没有覆盖直接继承）的类是抽象基类
抽象基类负责定义接口，而后续的类可以覆盖该接口

我们不能创建抽象基类的对象

派生类构造函数只初始化它的直接基类

```
class Derived {
public:
    Derived(...) : Base(...), ... {}
    ...
};
```

访问控制和继承

每个类分别控制着其成员对于派生类是否可访问（`accessible`）

受保护的成员

和私有成员类似，受保护的成员对于类的用户来说是不可访问的
和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的
派生类的成员和友元只能通过派生类对象来访问基类的受保护成员，派生类对于一个基类对象中的受保护成员没有任何访问特权

```
class Base {
protected:
    int prot_mem;
};
class Sneaky : public Base {
    friend void clobber(Sneaky &s) { s.j = s.prot_mem = 0; } // ok
    friend void clobber(Base &b) { b.prot_mem = 0; } // error
    int j;
```

```
};
```

公有，私有和受保护继承

某个类对其继承而来的成员的访问权限受两个因素影响：一是基类中该成员的访问说明符，二是派生类的派生列表中的访问说明符

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响，对基类成员的访问权限只与基类中的访问说明符有关

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限

派生访问说明符还可以控制继承自派生类的新类的访问权限

```
class Base {
public:
    void pub_mem();
protected:
    int prot_mem;
private:
    char priv_mem;
};
struct Pub_Derv : public Base {
    int f() { return prot_mem; } // ok
    char g() { return priv_mem; } // error
};
struct Prot_Derv : protected Base {
    int foo() { return prot_mem; } // ok
};
struct Priv_Derv : private Base {
    int f1() const { return prot_mem; } // ok, private不影响派生类的访问权限
};
struct Derived_from_Public : public Pub_Derv {
    int use_base() { return prot_mem; }
};
struct Derived_from_Private : public Priv_Derv {
    int use_base() { return prot_mem; } // error
};
Pub_Derv d1;
Priv_Derv d2;
Prot_Derv d3;
d1.pub_mem(); // ok
d2.pub_mem(); // error
d3.pub_mem(); // error
```

派生类向基类转换的可访问性

派生类向基类的转换是否可访问由该转换的代码决定，同时派生类的派生访问说明符也会有影响

- 只有当D公有地继承B时，用户代码才能使用派生类向基类的转换；如果D继承B的方式是受保护的或私有的，则用户代码不能使用该转换
- 不论D以什么方式继承B，D的成员函数和友元都能使用派生类向基类的转换
- 如果D继承B的方式是公有的或受保护的，则D的派生类的成员和友元可以使用D向B的类型转换；如果D继承B的方式是私有的，则不能使用

友元和继承

友元关系不能传递，也不能继承

每个类负责控制各自成员的访问权限，对基类的访问权限由基类本身控制，即使对于派生类的基类部分也是如此

```
class Base {
    friend class Pal;
};
class Pal {
public:
    int f(Base b) { return b.prot_mem; }
    int f2(Sneaky s) { return s.j; } // error, Pal不是Sneaky的友元
    int f3(Sneaky s) { return s.prot_mem; } // ok, Pal是Base的友元，对派生类的基类部分的访问权限由基类控制
};
```

改变个别成员的可访问性

如果需要改变派生类继承的某个名字的访问级别，可以使用using声明

派生类只能为那些它可以访问的名字提供using声明

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
private:
```

```
    int x;
};

class Derived : private Base {
public:
    using Base::size; // ok, size()在Derived是public的
protected:
    using Base::n; // ok, n在Derived是protected的
    using Base::x; // error, x在Base中是private的
};
Derived d;
d.size(); // ok
```

默认的继承保护级别

struct默认public继承；class默认private继承

继承中的类作用域

派生类的作用域嵌套在其基类的作用域之内

在编译时进行名字查找

一个对象，指针或引用的静态类型决定了该对象的哪些成员是可见的，
即使静态类型和动态类型不一致，我们能使用哪些成员依旧是由静态类型决定的

派生类的成员将隐藏同名的基类成员
派生类也能重用定义在其直接基类或间接基类中的名字，此时定义在内层作用域（即派生类）的名字将隐藏定义在外层作用域（即基类）的名字

通过作用域运算符来使用隐藏的成员

★ 名字查找和继承

理解函数调用的解析过程对于理解C++的继承至关重要，假定我们调用p->mem()或obj.mem()，则依次执行一下四个步骤：

- 首先确定p或obj的静态类型，因为我们调用的是一个成员，所以该类型必然是类类型
- 在p或obj的静态类型对应的类中查找mem，如果找不到，则依次在直接基类中不断查找直至到达继承链的顶端，如果仍然找不到，则编译器报错
- 一旦找到了mem，就进行常规的类型检查以确认调用是否合法
- 假设调用合法，则编译器根据调用的是否是虚函数而产生不同的代码，如果是虚函数而且我们通过引用或指针进行调用，则编译器生成的代码将在运行时确定到底运行该函数的哪个版本，依据的是对象的动态类型；如果不是虚函数或者通过对象进行的调用，则编译器产生一个常规的函数调用

```
struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int); // 隐藏基类的memfcn
};
Derived d;
Base b;
b.memfcn();
d.memfcn(10);
d.memfcn(); // error
d.Base::memfcn(); // ok
```

假如基类与派生类的虚函数接受的实参不同，则我们无法通过基类的引用或指针调用派生类的虚函数了

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    int fcn(int); // 隐藏了基类的fcn，这个fcn不是虚函数，D1继承了Base::fcn()的定义
    virtual void f2();
};
class D2 : public D1 {
public:
    int fcn(int); // 隐藏了D1::fcn(int)
    int fcn(); // 覆盖了Base::fcn()的虚函数
    void f2(); // 覆盖了D1::f2()的虚函数
};
Base bobj;
D1 d1obj;
D2 d2obj;
```

```
Base *bp1 = &bobj;
Base *bp2 = &d1obj;
Base *bp3 = &d2obj;
bp1->fcn(); // Base::fcn()
bp2->fcn(); // Base::fcn()
bp3->fcn(); // D2::fcn()

D1 *d1p = &d1obj;
D2 *d2p = &d2obj;
bp2->f2(); // error
d1p->f2(); // D1::f2()
d2p->f2(); // D2::f2()

Base *p1 = &d2obj;
D1 *p2 = &d2obj;
D2 *p3 = &d2obj;
p1->fcn(42); // error
p2->fcn(42); // 静态绑定，D1::fcn(int)
p3->fcn(42); // 动态绑定，D2::fcn(int)
```

覆盖重载的函数

成员函数无论是否是虚函数都能被重载，派生类可以覆盖重载函数的0个或多个实例
如果派生类希望所有的重载版本对于它来说都是可见的，那么他就需要覆盖所有的版本，或者一个也不覆盖

使用using声明语句可以把基类中成员函数的所有重载实例添加到派生类作用域中，此时派生类只需要定义其特有的函数就可以了，
而无需为继承而来的其他函数重新定义

构造函数和拷贝控制

虚析构函数

如果基类的析构函数不是虚函数，则delete一个指向派生类对象的基类指针将产生未定义的行为

Virtual destructors are useful when you can delete an instance of a derived class through a pointer to base class:

```
class Base
{
    // some virtual methods
};

class Derived : public Base
{
    ~Derived()
    {
        // Do some important cleanup
    }
}
```

Here, you'll notice that I didn't declare Base's destructor to be virtual.
Now, let's have a look at the following snippet:

```
Base *b = new Derived();
// use b
delete b; // Here's the problem!
```

Since Base's destructor is not virtual and b is a Base pointing to a Derived object, delete b has undefined behaviour. In most implementations, the call to the destructor will be resolved like any non-virtual code, meaning that the destructor of the base class will be called but not the one of the derived class, resulting in resources leak.*

To sum up, always make base classes' destructors virtual when they're meant to be manipulated polymorphically.

If you want to prevent the deletion of an instance through a base class pointer, you can make the base class destuctor protected and nonvirtual; by doing so, the compiler won't let you call delete on a base class pointer.

虚析构函数将阻止合成移动操作

合成拷贝控制和继承

基类或派生类的合成拷贝控制成员的行为和其他合成的构造函数，赋值运算符或析构函数类似
它们对类本身的成员依次进行初始化，赋值或销毁的操作，
此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化，赋值或销毁的操作

无论基类成员是合成的版本还是自定义的版本都没有太大影响，唯一的要求就是相应的成员应该可访问并且不是一个被删除的函数

派生类中删除的拷贝控制与基类的关系

基类或派生类出于同样的原因将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数 TODO

移动操作和继承

因为基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以我们确实需要移动操作时应该首先在基类中进行定义

```
class Quote {
public:
    Quote() = default;
    Quote(const Quote&) = default;
    Quote(Quote&&) = default;
    Quote& operator=(const Quote&) = default;
    Quote& operator=(Quote&&) = default;
    virtual ~Quote() = default;
};
```

派生类的拷贝控制成员

当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象

在默认情况下，基类默认构造函数初始化派生类对象的基类部分
如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式地使用基类的拷贝（或移动）构造函数

```
class Base {};
class D : public Base {
public:
    D(const D& d) : Base(d) ... {}
    D(D &&d) : Base(std::move(d)) ... {}
};
```

同样，派生类的复制运算符也必须显式地为其基类部分赋值

```
D& operator=(const D &rhs) {
    Base::operator=(rhs);
    ......

    return *this;
}
```

派生类析构函数只负责销毁由派生类自己分配的资源，基类部分是隐式销毁的
对象销毁的顺序与其创建的顺序相反

如果在构造函数或析构函数中调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚函数版本

继承的构造函数

。 。 。 。 。

第十六章 模板与泛型编程

2014年4月9日 星期三　　下午3:13

所谓泛型编程就是以独立于任何特定类型的方式编写代码（代码与对象类型彼此独立）
模板是泛型编程的基础

泛型编程（Generic Programming）：编译时多态
面向对象（OO）：运行时多态

函数模板

模板定义以template开始，后接模板形参表（template parameter list）
模板形参表是以 <> 括住的一个或多个模板形参的列表，模板形参表不能为空

模板形参可以是**表示类型的类型形参**（type parameter），也可以是**表示常量表达式的非类型形参**

类型形参在关键字class或typename之后，class 和 typename 在此处具有相同的含义
非类型形参在类型说明符之后，非类型模板形参的模板实参必须是常量表达式

```
template <typename T, int x>  // T为类型形参，x为非类型形参
inline void foo() // 函数模板可以声明为inline
{
    T array[x]; // x为常量表达式
...
}

foo<int, 100>();
```

使用类模板时，必须显式指定实参

```
Queue<int> q;
```

使用函数模板时，可以显式指定实参，也可以让编译器自己推断

模板形参的作用域从声明开始一直到模板声明或定义的结束

模板形参遵循常规的名字屏蔽（hide）规则，与外层作用域中声明的对象，函数或类型同名的模板形参会屏蔽外层作用域中的名字。

```
typedef double T;
template <typename T>
void foo()
{
    T a; // T为模板形参
    ...
}
```

用作模板形参的名字不能在模板内部重用。

```
template <typename T>
void foo(T a)
{
    typedef double T; // error: redefinition of formal parameter
    ...
}
```

同一模板的声明和定义中，模板形参的名字可以不同。

```
template <typename T> void foo();
template <typename U> void foo(); // 相同的函数声明
```

typename关键字

在模板定义的内部指定类型，通过在成员名前加上关键字typename来显式地告诉编译器将成员当做类型。

```
template <typename T, typename U>
T foo(T *array, U value)
{
    T::sizetype *p; // error，编译器无法确定究竟是*运算还是指针的声明
    typename T::sizetype *p; // ok
    ...
}

template <typename T>
```

```
    typename T::value_type bop(const T& c)
    {
        if (!c.empty())
            return c.back();
        else
            return typename T::value_type(); // 值初始化的元素
    }
```

typename不得在base class lists或member initialization list内以它做为base class的修饰符。

编写独立于类型的代码（writing type-independent code）

在模板内部完成的操作限制了可用于实例化该模板的类型，因此，应该保证用作函数实参的类型实际上支持所有的操作。
编写模板代码时，对实参类型的要求尽可能少是有益的，两个重要的原则：

1. 模板形参是const引用
2. 函数体中的测试只用 < 比较

实例化（instantiation）

模板本身并不是函数或类，编译器用模板产生指定的函数或类的特定类型版本。
"实例化"这个术语反映了创建函数模板和类模板的新"实例"的概念。

当编译器遇到一个模板定义时，它并不生成代码。

类模版在引用实际的类模板类型的时候被实例化，必须被显式的指定模板实参

```
    Queue<int> qi;
    Queue<string> qs; // 类模板每次实例化都会产生独立的类类型，qs 和 qi 没有关系
```

函数模板在调用它 或 用它对函数指针进行初始化或赋值的时候 被实例化，可以显式的指定模版实参或者让编译器推断

```
    template <typename T> int compare(const T& v1, const T& v2);
    compare(10, 15); // 函数模板形参被绑定到int类型
    compare(2.14, 3.15); // 函数模板形参被绑定到double类型
    compare<int>(10, 15); // 显式指定实参类型
```

```
C++ Primer 5th Ex16.27
    template <typename T> class Stack {
        typedef typename T::ThisDoesntExist StaticAssert; // T::NotExisting doesn't exist at all!
    };

    void f1(Stack<char>); // No instantiation, compiles

    class Exercise {
        Stack<double> &rsd; // No instantiation, compiles (references don't need instantiation, are similar to
    pointers in this)

        Stack<int> si; // Instantiation! Doesn't compile!!
    };

    int main(){

        Stack<char> *sc; // No Instantiation, this compiles successfully since a pointer doesn't need
    instantiation

        f1(*sc); // Instantiation of Stack<char>! Doesn't compile!!

        int iObj = sizeof(Stack< std::string >); // Instantiation of Stack<std::string>, doesn't compile!!
    }
```

控制实例化

当模板被使用时才会进行实例化这一特性意味着，相同的实例可能出现在多个编译生成的对象文件中。
当多个独立编译的源文件使用了相同的模板，并提供了相同的模版参数时，每个文件中都会有该模板的一个实例。

在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。
在C++11中，我们可以通过显式实例化（explicit instantiation）来避免这种开销。

```
    extern template declaration; // 实例化声明
    template declaration; // 实例化定义
```

类模板的实例化定义会实例化所有成员
因此，在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

```
class NoDefault
{
    NoDefault() = delete;
};
template vector<NoDefault> v; // error
```

模板实参推演（Template Argument Deduction）

从函数调用的实参确定模板实参的类型和值的过程叫 模板实参推演。

与函数模板不同之处是，编译器不能为类模板推断模板参数类型。

多个类型形参的实参必须完全匹配

```
template <typename T> int compare(const T& v1, const T& v2);
short si;
compare(si, 1024);
    // error, cannot instantiate compare(short, int), must be compare(short, short) or compare(int, int)
```

一般而论，不会转换实参以匹配已有的实例化，相反，会产生新的实例。
除了产生新的实例化之外，编译器只会执行两种转换： const转换 和 数组或函数到指针的转换

const转换

A function that takes a reference or pointer to a const can be called with a reference or pointer to non-const object, respectively, without generating a new instantiation.
If the function takes a non-reference type, then const is ignored on either the parameter type or the argument. That is, the same instantiation will be used whether we pass a const or non-const object to a function defined to take a non-reference type.

```
template <typename T>
void foo(const T&, const T&)
{
}

string s;
foo(s, s); // foo(const string&, const string&)
foo(10, 100); // foo(const int&, const int&)

short si = 100;
foo(si, si); // foo(const short&, const short&)

const int x = 1000;
foo(x, x); // 同一个foo(const int&, const int&)实例

const string cs;
foo(cs, s); // 同一个foo(const string&, const string&)实例

// 从nm可以看出生成了三个函数实例
nm -a a.out
0000000100000e50 T __Z3fooINSt3__112basic_stringIcNS0_11char_traitsIcEENS0_9allocatorIcEEEEEvRKT_S9_
0000000100000e60 T __Z3fooIiEvRKT_S2_
0000000100000e70 T __Z3fooIsEvRKT_S2_
```

数组或函数到指针的转换

如果模板形参不是引用类型，则对数组或函数类型的实参执行到指针的转换

```
template <typename T> T fobj(T, T);
template <typename T> T fref(const T&, const T&);
string s1;
const string s2;
fobj(s1, s2); // ok, calls fobj(string, string), const忽略
fref(s1, s2); // ok, s1 -> const string
```

函数模板的显式实参

在某些情况下，不可能推断出模板实参的类型，例如：当函数的返回值类型必须与形参表中所有的类型都不同时。
因此，在返回类型中使用类型形参，或者为了消除二义性，可以显式指定实参。

```
template <typename T> int compare(const T&, const T&);
```

```
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));

func(compare<int>); // ok, explicitly specify which version of compare
```

模板默认实参

在C++11中，我们可以为函数和类模板提供默认实参

```
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

无论何时使用一个类模板，都必须在模板名之后接上尖括号

```
template <class T = int>
class Numbers
{
public:
    Numbers(T v = 0) : val(v) {}
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // 空<>表示我们希望使用默认实参
```

类模板成员

类模板成员函数的形式

```
template <typename T>
void A<T>::member_name()
{
    ......
}
```

类成员模板不能是虚函数。

类模板的成员函数只有在被使用的时候才进行实例化。

```
Queue<int> qi; // 实例化 class Queue<int>
short s = 42;
int i = 42;
qi.push(s); // 实例化Queue<int>::push(const int&)，允许常规转换
qi.push(i); // 使用Queue<int>::push(const int&)
        // 因为对象的模板形参可以确定其成员函数的模板形参，所以调用类模板成员函数相比函数模板
        // 用模板形参定义的函数形参的实参允许进行常规转换。
```

模板特化（Template Specializations）

模板特化是这样一个定义，该定义中一个或多个模板形参的实际类型或实际值是指定的

函数模板的特化

```
template <typename T>
int compare(const T&, const T&);

template <>
int compare<const char*>(const char* const &v1, const char* const &v2)
    // special version of compare to handle C-style character strings
{
    return strcmp(v1, v2);
}

const char* cp1 = "world", *cp2 = "hi";
char* p1, *p2;

compare(cp1, cp2); // 特化版本
compare(p1, p2); // 泛型版本
```

函数模板特化可以只声明而无定义，特化声明和定义很像，只是忽略了函数体。

不能总是检测到重复定义，因此和其它函数声明一样，应在一个头文件中包含模板特化的声明，
然后使用该特化版本的每个源文件包含该文件。
如果程序由多个文件构成，模板特化的声明必须在使用该特化的每个文件中出现。
不能在一些文件中从泛型模板定义实例化一个函数模板，而在其他文件中为同一模板实参集合特化该函数模板。

类模板的特化 Specializing a class template

类模板特化可以定义与模板本身完全不同的成员。
在类模板特化外部定义成员时，成员之前不能加 `template<>` 标志。

类模板可以部分特化

类模板特化可以特化成员而不特化类

```
template <typename T>
class FOO
{
public:
    void B(T t) {}
    void B(int i) {}
    void B(int i) const {}
};

template <>
class FOO<bool>
{
public:
    void B(int i) {}
    void B(int i) const {}
};

int main()
{
    FOO<bool> f;
    const FOO<bool> g = f;
    g.B(10);

    return 0;
}
```

重载和函数模版

函数模板可以重载，可以定义在相同名字但形参数目或类型不同的多个函数模板，也可以定义与函数模板有相同名字的普通非模板函数。

1. 建立候选函数集合
   a. 同名的普通函数
   b. 相匹配的函数模板实例化（模板实参推演）
2. 确定可行函数
   a. 模板实例都是可行的
3. 如果需要转换，则根据转换的种类排列可行函数（调用函数模板所允许的转换是有限的）
   a. 如果仅有一个函数，则调用该函数
   b. 如果有二义性，从可行函数中去掉所有函数模板实例
4. 重新排列去掉了模板实例的可行函数
   a. 只有一个函数，调用它
   b. 存在二义性

```
template <typename T> int compare(const T&, const T&); // (1)
template <> int compare<const char*>(const char * const&, const char * const&); // (2)
int compare(const char*, const char*); // (3)

char *p1, *p2;
const char *cp1, *cp2;
compare(cp1, cp2); // (3)
compare(p1, p2); // (3)

如果没有(3)声明呢：
compare(cp1, cp2); // (2)
compare(p1, p2);  // (1)
```

引用折叠和右值引用参数

通常我们不能定义一个引用的引用，但是，通过类型别名或通过模板类型参数间接定义是可以的。

```
typedef int& INT_REF;
int i = 10;
INT_REF &r = i; // ok
```

引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

```
Exercise 16.45: Given the following template, explain what happens if we call g on a literal value such as
42. What if we call g on a variable of type int?
template <typename T>
void g(T&& val)
{
    std::vector<T> v;
}

int main()
{
    g(42); // compiles

    int i;
    g(i); // error
}

The first of the remaining two rules for rvalue references affects old-style lvalue references as well.
Recall that in pre-11 C++, it was not allowed to take a reference to a reference: something like A& & would
cause a compile error. C++11, by contrast, introduces the following reference collapsing rules:

A& & becomes A&
A& && becomes A&
A&& & becomes A&
A&& && becomes A&&
Secondly, there is a special template argument deduction rule for function templates that take an argument by
rvalue reference to a template argument:

template<typename T>
void foo(T&&);
Here, the following apply:

When foo is called on an lvalue of type A, then T resolves to A& and hence, by the reference collapsing rules
above, the argument type effectively becomes A&.
When foo is called on an rvalue of type A, then T resolves to A, and hence the argument type becomes A&&.
So case 1, when passing 42, you are calling g with a rvalue, so T is resolved to int thus g's parameter is
int&& and std::vector is legal.

In case 2, when passing i, you are calling g with a lvalue, so T is resolved to int& thus g's parameter is
int& and std::vector<int&> is NOT legal.

Remove the line with the vector and it will work fine in both cases.
```