

第二章 变量和基本类型

概述

- C++是一种静态数据类型语言，它的类型检查发生在编译时

一些语言，如Python，在程序运行时检查数据类型，所以不需要明确定义
所以C++要求声明数据类型

- 数据类型决定程序的数据和操作的意义；

对象的类型定义了对象能包含的数据和能参与的运算

```
1 i = i + j; // 其含义依赖于i、j的数据类型。
```

结合类来理解，所以的操作都是针对某个类来说的

基本数据类型

基本大小规则

- 一个 `char` 的大小和一个机器字节一样

char一定是1

- 大小关系一定要满足：`long long` \geq `long` \geq `int` \geq `short`

只限制了大小关系，没有明确说明int...等具体使用多少字节

- 一般来说，`float` 和 `double` 有效位数是7和16

有符号和无符号类型

- 字符型有三类：`char`、`signed char`、`unsigned char`

类型 `char` 实际上会表现为上述两种形式的一种，具体是哪种由编译器决定

建议：如何选择类型

经验准则：

- 当明确知晓数值不可能为负时，用无符号类型
- 优先使用 `int` 进行整数运算，超过范围用 `long long` (而不用 `long`)
- 在算数表达式中不要使用 `char` 或 `bool`，只有在存放字符或布尔值时才使用他们
- 执行浮点运算选用 `double`

类型转换

- 表示范围小的自动转换为表示范围大的
 - 有符号自动转换为无符号

无符号如果变成负值，最后会显示为 该负数+2^所有位数

后续章节做更详细的介绍

字面值

每个字面值常量都对应一种数据类型，字面值常量的**形式和值**决定了它的数据类型

字面值，表示的就是一个具体的值。

如，20，023（八进制），0x11(十六进制)，'h',"hello"

- **整型字面值**具体的数据类型由它的值和符号决定
 - 十进制字面值默认是带符号数，八进制、十六进制字面值既可能是带符号的也可能是无符号的

十进制字面值的类型是 `int`、`long`、`long long` 中能容纳该数的尺寸最小的那个；

八进制和十六进制的类型是 所有有符号（除`short`）和无符号的整数类型 中尺寸最小的那个

类型 `short` 没有对应的字面值

- 十进制字面值不会为负数，即-42的负号并不在字面值之内，仅仅是对字面值取负值
- **字符和字符串字面值**
 - 字符串字面值的类型实际上是有常量字符构成的数组；

直接写"abc"代表的是一个char*类型，而不是 `string`

编译器自动在每个字符串末尾添加一个空字符('\0')，因此实际长度比它的内容多1

通过在字面值添加前缀 或 后缀，可以指定其类型

L'a' --- 宽字符型字面值

333L ---long long型

用的不多

变量

- 变量的初始化和赋值是两个完全不同的操作

初始化不是赋值，初始化的含义是 **在创建变量时赋予其一个初始值**，而赋值的含义是把对象的当前值擦除（一般该对象已经创建），而以一个新值来代替

- 用 `=` 来初始化变量的方式容易让人认为初始化是赋值的一种

实际上可能是拷贝初始化

- 可以联想一下类的初始化方式存在 `string a(b)`，这种虽然没有用 `=` 赋值，但是是一种初始化操作 **初始化有多种形式**

- c++ 11** 列表初始化：利用 **花括号** 来初始化变量(可以用于**任何类型**对象的初始化)

```
1 int a{2};
2 int b{a};
```

当用于**内置类型的变量**时，如果我们使用列表初始化切初始值存在丢失的风险，则编译器会报错

```
1 double a = 3.13;
2 int b{a};           //编译器报错 不能通过
3 int c(a);           //编译通过，但丢失部分值
```

- 括号初始化实际上是调用了对应类型的初始化函数，里面是用赋值来实现的

并且，使用括号和花括号在一些特殊的情况下是有区别的，比如

vector

- 如果初始化提供的是初始元素的列表，则只能把初始化值放在花括号里进行初始化，不能放在圆括号里

- 绝大多数类都支持无须显示初始化而定义对象

如果没有显式的初始化，则其值取决于类内相应的函数

建议初始化每一个内置类型的变量

- **变量声明和定义**

- **声明使得名字（对象）为程序所知**，一个文件如果想要使用别处定义的名字必须包含对那个名字的声明；

变量声明规定变量的类型名字；

- 定义负责创建于名字关联的实体；

定义会给变量申请空间；

```
1 extern int i;    // 声明i而非定义i
2 int j;           // 声明并定义j
```

- 任何包含了显示初始化的声明即成为了定义

```
1 extern int i = 1;    //定义
```

- 在**函数体内部**，如果试图初始化一个有 `extern` 关键字标记的变量，将导致错误(但编译器不会检查出该错误) ??
- 变量能且仅能被定义一次，但可被多次声明
- **建议变量在头文件声明，在源文件定义**

- 嵌套的作用域中，内部作用域的定义的变量会覆盖外部定义的变量。

```
1 // 不好的使用
2 int a = 1;
3 int main{
4     {   int a = 2;
5         cout<<a;    // 输出 2
6         cout<<::a;  // 输出 1    可以通过符号::变量名来访问全局变量
7     }
8     cout<<a;    // 输出 1
9 }
10
```

扩展阅读

- **【C++】用花括号初始化和用括号初始化有什么区别？**

复合类型

理解复合类型(引用、指针、数组)

不能认为 `int*` or `int&` 是两种类型。

复合类型 = 基本 **数据类型** + 声明符；声明符命名了该变量并指定该变量为与基本类型有关的某种类型

因此，可以知道为什么声明两个指针要用 `int *p,*q` 而不是 `int * p,q`

左值引用

引用的实现是通过指针实现的

- 引用就是为对象起了个别名,绑定在一起，并没有创建新的对象；

对引用的操作均相当于直接对原对象操作。

- 引用必须初始化

因为引用并没有创建新的对象，所以如果声明一个引用而不初始化，实际上是不存在这个对象的。

- 不能定义引用的引用

```
1 int a=1;
2 int &b=a;
3 int &c=b;    // 正确，c绑定到b绑定的对象上
4 int &(&b)=a;    // 引用的引用，错误，编译不过
```

因为引用不是一个对象

- 引用的类型要与之绑定的对象 **严格匹配**；

引用只能绑定在某个对象上，而不能与某个字面值或表达式绑定在一起

例外：

- 常量引用：只要引用的对象可以转换为相应的引用类型即可（注意是作左值）

```

1 int b = 2;
2 const int &r = 2;    // 正确, 可以为常量引用绑定字面值
3 const int &r2 = b;   // 正确
4 int &r3 = r;         // 错误, 类型不匹配
5
6 const int a=1;
7 int &b=a;            // 错误, 类型不匹配

```

解释

- c++认为, 使用普通引用绑定一个对象, 就是为了能通过引用对这个对象做改变。

如果普通引用绑定的是一个临时量而不是对象本身, 那么改变的是临时量而不是希望改变的那个对象, 这种改变是无意义的。

所以规定普通引用不能绑定到临时量上

- 那么为什么常量引用就可以呢?

因为常量是不能改变的。也就是说, 不能通过常量引用去改变对象, 那么绑定的是临时量还是对象都无所谓了, 反正都不能做改变也就不存在改变无意义的情况。

所以常量引用可以绑定临时量, 也就可以绑定非常量的对象、字面值, 甚至是一般表达式, 并且不用必须类型一致

指针

- 指针本身就是一个对象, 允许赋值和拷贝, 无须在定义时赋初值,
 - 引用不是一个对象, 没有实际地址, 所以不能定义指向引用的指针
- 除了一些特殊情况外, 指针类型也要和它所指向的对象 **严格匹配**

特殊情况:

- 指向常量的指针指向一个非常量对象 (注意是作左值)

```

1 double dval = 3.14;
2 const double *cptr = &dval // 正确, 就和对const引用一样;
3
4 const int a=1;
5 int *p=a;                 // 错误

```

- **P534**

- **c++11** 推荐使用 `nullptr` 来初始化指针, 代表指向为空, 而不用 `null`

在C语言中，NULL通常被定义为：`#define NULL ((void *)0)`，所以说NULL实际上是一个空指针，所以说NULL实际上是一个空指针，如果在C语言中写入以下代码，编译是没有问题的，因为在C语言中把空指针赋给int和char指针的时候，发生了**隐式类型转换**，把void指针转换成了相应类型的指针

```
1 int *pi = NULL;
2 char *pc = NULL;
```

但是问题来了，以上代码如果使用C++编译器来编译则是会出错的，因为**C++是强类型语言¹**，**void*是不能隐式转换成其他类型的指针的**，所以实际上编译器提供的头文件做了相应的处理：

```
1 #ifdef __cplusplus
2 #define NULL 0
3 #else
4 #define NULL ((void *)0)
5 #endif
```

可见，在C++中，**NULL实际上是0**

```
1 void test(void *p)
2 {
3     cout<<"p is pointer "<<p<<endl;
4 }
5 void test(int num)
6 {
7     cout<<"num is int "<<num<<endl;
8 }
9 int main(void)
10 {
11     test(NULL);
12     return 0;
13 }
14 /*
15 编译报错了，提示我们有二义性，按照重载函数匹配规则，两个都可以匹配，因此最终报错。
16 */
```

C++11加入了nullptr，**可以保证在任何情况下都代表空指针**。如果你想表示空指针，那么使用nullptr，而不是NULL。

- **void*** 可以存放任意对象的地址;

不能直接操作 `void*` 所指向的对象，因为我们不知道其指向对象的类型，无法确定能在该对象上执行什么操作

指针的指针

一般来说，声明符中修饰符的个数并没有限制。

指向指针的引用

```
1 int i = 3;
2 int *p;
3 int * &r = p;    // r是一个对指针p的引用
4 r = &i;
5 cout<<*p;      // 输出3
```

如何阅读 `int *& r` ?

要理解r的类型到底是什么，最简单的方式是**从右向左**阅读r的定义。**离变量最近的符号对变量类型有最直接的影响**

扩展阅读

- [nullptr与NULL的区别](#)
- [C++中NULL和nullptr的区别](#)

const限定符

- `const` 对象一旦创建后其值就不能改变，所以必须初始化
- 如果想在多个文件中共享 `const`对象，必须在变量声明的定义前添加 `extern`关键字

```
1 extern const int bufSize = 100;
2 extern const int bufSzie;    //在其他文件中使用bufSize的声明
```

- 对 `const` 的引用可能不是一个 `const`对象

```
1 int i = 1;
2 const int &r = i;
3 i = 2;
4 cout<<r;    // 输出2
5 r = 2;      //错误
```


const限制引用，只有一种类型 `const int &r = j`，没有 `int & const r = j`
因为&只是一个声明符

• 指向常量的指针

要想存放常量对象的地址，只能使用指向常量的指针

```
1 const double dval = 3.14;
2 double *p2 = &dval;      // 错误
3 const double *cptr = &dval // 正确，就是对const引用一样;
```

• const 指针

- 顶层const: **指针**本身是一个常量
- 底层const: **指针**所指的对象是一个常量

- 这两类其实不仅是用在const指针上，实际上给出一个**层次**的概念，如果const作用于对象本身，那么就是顶层const,与位置无关；但是一般来说，这两个层次都是针对指针才显示区别的

引用实际上只有底层引用，直接作用于变量

```
1 const int ci = 32;      // ci不能变，顶层const
2 const int &r = ci;      // 对引用来说，引用的对象 ci,ci是可以改变的，所以是 底层const
3 const int *p = &j;      // 底层const,因为指针的对象是p,而不是*p
```

- **当对象进行拷贝操作时，顶层const不影响（即不要求匹配），而底层const会限制，要求类型匹配**
 - 要么就是都是底层const拷贝，要么右值的数据类型可以转换为常量

```
1 //注意const的位置
2 int i = 1;
3 int *const p = &i;      // 指针常量，p的值不能（即指针的地址不能变，当地址内的值可以变）
4 const int * p2 = &i;    // 常量指针，不允许改变指针指向的值，但是指针实际的地址可以变
5 const int * const p3 = &i
```

常量表达式

指 **值不会改变** 且 **编译过程就能得到结果** 的表达式

- 字面值是常量表达式；

用常量表达式初始化的 `const` 对象 是常量表达式

- 在一个复杂的系统中，很难（几乎不能）分辨与一个初始化是否是常量表达式
- `const` 是 **运行期常量**，实际是“只读”的意思，并不代表是常量表达式。
(如果 `const` 对象所赋初值在编译阶段就能确定，这个 `const` 才是常量表达式)

`constexpr` 是编译期常量，是值类型。

- **C++11** `constexpr` 类型：这个类型会 **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，且必须用常量表达式初始化

```
1 constexpr int mf = 20;
2 constexpr int mf2 = mf + 1;
3 constexpr int sz = size(); //只有size是一个constexpr函数时才是一条正确的声明语句
```

如果将指针或引用定义为 `constexpr`

- 一个 `constexpr` 指针 的初始化必须是 `nullptr`，或者是存储在某个 **固定地址** 中的对象
 - 函数体内的变量的地址都不固定

- **字面值类型**：编译时就能得到结果的类型

- 算术类型、引用和指针都属于字面值类型，还有字面值类

- 指针和 `constexpr`

- `constexpr` 把它所定义的对象均致为顶层 `const`

```
1 const int *p = nullptr; // *p不能变
2 constexpr int *q = nullptr; // q不能变
```

`constexpr` 仅对指针有效，与指针所指的对象无关；

其实，因为 `constexpr` 必须要用常量赋值，也不存在 `*q` 会发生变化的情况

处理类型

背景

随着程序越来越复杂，程序中用到的类型也越来越复杂（类型难拼写；不清楚需要哪种类型）

处理

- 类型别名

- `typedef`
- **C++ 11** `using`

```
1 using wages = double; // wages是double的同义词
```

using包含类typedef的所有功能以及其他功能，而且容易理解，建议优先考虑使用

理解 `typedef char* pstring; const pstring p=0` ?

不能直接替换成 `const char *p =0` ,这个形式声明的是一个指向 `const char` 的指针；

而原形式中，因为pstring的基本类型就是一个指针，那么可以理解为 `const 指针型 p =0` 。安装从右向左分析，那么p就是一个常量指针。

```
1 // 测试
2 typedef int* pstring;
3 const int i = 1;
4 int j = 2;
5 const pstring p1 = &i; // 编译器会提示错误，类型不匹配
6 const pstring p2 = &j; // 正确
```

- **C++11** `auto` 类型说明符：让编译器替我们去分析表达式所属的类型

- `auto` 定义的变量必须有初始值
- 使用 `auto` 在一条语句中定义多个变量，初始类型必须相同

```
1 auto i = 0, *p = &i; // 正确，都是整型(*只是声明符不是类型)
2 auto sz = 0, pi = 3.14; // 错误，类型不一致
```

- `auto` 会忽略掉顶层 `const`

```

1 const int ci = i, &cr = ci;
2 auto b = ci;    // b是一个整数, 忽略了const
3 auto c = cr;    // c是一个整数
4 auto d = &i;    // d是一个整型指针
5 auto e = &ci;   // e是一个指向整数常量的指针 (对常量对象取地址是一种底层const)

```

可以通过显示声明

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

- **C++11** `decltype` : 返回操作数的数据类型 (只想从表达式中推断类型, 但是不想用该表达式初始化变量)
 - 如果表达式是变量, 则返回该变量的**所有类型** (包括顶层const 和 引用)
 - 如果表达式不是一个变量, 返回表达式结果对应的类型

特殊情况: 一般而言, 如果**表达式**的求值结果是**左值**, 那么**decltype**作用于该表达式 (不是变量) 得到一个引用类

```

1 int i =42, *p = &i, &r = i;
2 decltype(r) b;    // 错误, 因为r是引用类型
3 decltype(r+0) a;  // 正确, r+0的结果是int
4 decltype(*p) c;   // 错误, 因为*p是解引用指针操作, 所以类型为int&,

```

- 表达式加括号和不加括号有时结果不同。加了括号会认为是一个计算式, 而没加括号会被认为是个变量

```

1 int i =1;
2 decltype(i) e;
3 decltype((i)) d;  // 错误, 返回的是int&, 需要初始化

```

实际上牵扯了lvalue,rvalue的知识。这里我大略说一下 (不一定正确)

两个对应"="两边, 所以实际上作为左值的一定有自己的内存空间, 即地址, 那么可能赋值的时候是通过解变量地址来给内部赋值的, 所以(i)被认为是一个表达式且可以作为左值, 那么就会返回引用类型

自定义数据类型

- **C++ 11** 可以为类内数据成员提供一个 **类内初始值**，进行默认初始化
- 类一般不定义在函数体内，通常定义在头文件中
类所在的头文件名字应与类的名字一样

头文件

- 头文件通常包含哪些只能被定义一次的实体：类、const和constexpr变量等
- 头文件保护符：依赖于预处理变量，可以防止重复包含头文件
 - 预处理器：在编译之前就执行的一段程序，可以部分的改变程序所写的程序
 - 预处理变量：一般全部大小
 - 已定义 **#ifdef**
 - 未定义 **#ifndef**
 - 定义结束 **#endif**

```
1 #ifndef STUDENT
2 #define STUDNET
3 #include<string>
4 struct Student{
5     ...
6 }
7 #endif
```

预处理可以无视c++语言中关于作用域的规则

1. 强类型语言是一种强制类型定义的语言，一旦某一个变量被定义类型，如果不经强制转换，则它永远就是该数据类型

↩