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字节**

- 大小关系一定要满足： `long long` >= `long` >= `int` >= `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

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；
>
>   所以，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**

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**；

  > "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，占两个字节
> - 333L — long型; 333LL — long long; 333U—unsigned； 1.2F—float; 1.2L—long double;

变量

> 变量提供一个具名的、可供程序操作的存储空间

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函数的功能；
> - 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)

```
int a{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
>
> ```
> double a  = 3.13;
> int b{a};       //编译器报错 不能通过
> 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

【C++】 用花括号初始化和用括号初始化有什么区别?

- 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；

> 个人总结：C++中声明语句不多，而且多于类有关

- 定义负责创建于与关联的实体；

  定义一个变量会为该变量分配内存空间；

```
extern int i;   // 声明i而非定义i
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
>
> ```
> extern int i = 1;   //定义，但是用法不对，会导致重复定义
> ```
>
> - **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。

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 = 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，可以知道为什们声明两个指针要用 `int *p,*q` 而不是 `int * p,q`，* 只是**类型修饰符**术语声明符的一部分，所以建议和变量名写在一起

左值引用

> 1. 引用只是c++语法糖，可以看作编译器自动完成取地址、解引用的常量指针
> 2. 引用区别于指针的特性都是编译器约束完成的，一旦编译成汇编就和指针一样

- 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

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

  因为引用不是一个对象

  - 引用的类型要与初始化绑定的对象**严格匹配**；
    - 引用不能再更改绑定的对象；
    - **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）
  >
  >   ```
  >   int b = 2;
  >   const int &r = 2;    // 正确,可以为常量引用绑定字面值
  >   const int &r2 = b;   // 正确
  >   int &r3 = r;         // 错误，类型不匹配
  >
  >   const int a=1;
  >   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，**并且不用保证类型一致**

- 基类引用：可以绑定到派生类上

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，指针也要和它每次所指向的对象**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）
  >
  > ```
  > double dval = 3.14;
  > const double *cptr = &dval  //正确，就和对const引用一样;
  >
  > const int a=1;
  > int *p=a;       //  错误
  > ```
  >
  > - 基类的指针可以指向派生类

- `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，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

```
int  *pi = NULL;
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：

```
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

可见，**在C++中，NULL实际上是0**

```
void test(void *p)
{
 cout<<"p is pointer "<<p<<endl;
}
void test(int num)
{
 cout<<"num is int "<<num<<endl;
}
int main(void)
{
 test(NULL);
 return 0;
}
/*
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。
*/
```

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*`可以暂时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。

指针的引用

```
int i = 3;
int *p;
int *&r = p;     // r是一个对指针p的引用
r = &i;
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(无论是变量或指针）一旦创建后其值就不能改变，所以必须显式初始化
- 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`

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

> 一般变量不需要，只要求在其他文件声明即可

- `const`引用的可能不是一个`const对象`

```
int i = 1;
const int &r = i;
i = 2;
cout<<r;     // 输出2
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`
>
> 后一种理解为：不能修改引用去重新绑定另一个对象。而引用一经创建本来就不能再改变其引用的绑定对象，所以这种写法是没有意义的

- **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

```
const double dval = 3.14;
double *p2 = &dval;      // 错误
const double *cptr = &dval  //正确，就和对const引用一样;
```

> 个人总结：保证指向的对象的限制不会放松;
>
> 这也是为什们常量指针可以指向非常量变量，通过该指针访问该对象相当于提高了该对象的限制

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

> - 这种分类只针对指针，对引用没用（引用const只能放在一个位置，是底层const）
>
> ```
> const int ci = 32;       // ci不能变，顶层const
> 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
> const int *p =&j;        // 底层conost,因为指针的对象时p,而不是*p
> ```
>
> - **当进行拷贝操作时，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**

```
//注意const的位置
int i = 1;
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）
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可以变
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。
  >
  > ```
  > const int a = 1;     // a是常量表达式
  > const int a = get_size();    // a不是常量表达式
  > ```

- **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

```
constexpr int mf = 20;
constexpr int mf2 = mf + 1;
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，必须使用定义与所有函数体之外的变量

- **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

    > 字面值类：字面值常量类[2]； enum类[3]

- 指针和`constexpr`
  - `constexpr`把它所定义的对象均致为顶层const

```
const int *p = nullptr;     // *p不能变
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，顶层const也就没有必要

处理类型

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）

类型别名

`typedef`——传统方法

using

C++11

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

- **typedef 和 using 区别**
  - 定义一般类型时没区别；
  - 定义模版的别名只能使用using

```
template <typename T> using Vec = MyVector<T, MyAlloc<T>>;
Vec<int> vec;
```

> **在C++中，请使用using，而非typedef**

> **理解：**
>
> ```
> typedef char* pstring;  // 使用using也是一样的
> const pstring p=0;      // const的作用?
> ```
>
> 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。

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

`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

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

- `auto` 一般会忽略掉顶层 `const`；底层const会保留

```
const int ci = i, &cr = ci;
auto b = ci;    // b是一个整数，忽略了const
auto c = cr;    // c是一个整数
auto d = &i;    // d是一个整型指针
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）
```

> 设置一个类型为auto的引用时，初始值中的顶层const会被保留
>
> ```
> auto &g = ci;       //  const int &g = ci
> const auto &g2 = ci;    //   const int &g2 = ci;
> auto &h = 3;        // 错误
> const auto & j = 3; // 正确，常量引用可以绑定字面值
> ```

`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**表达式是单个变量**，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;
  - 如果 `expr` 是一个 lvalue，那么 `decltype` 将返回 `T &`；

- 如果 `expr` 是一个 xvalue，那么 `decltype` 将返回 `T &&`；
- 如果 `expr` 是一个 prvalue，那么 `decltype` 将返回 `T`

```
int i =42,*p = &i,&r = i;
decltype(r) b;       // 错误，因为r是引用类型
decltype(r+0) a;     //正确，r+0的结果是int
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

```
int i =1;
decltype(i) e;
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

> 类内初始值将用来初始化数据成员

- 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 `#inndef`
    - 定义结束 `#endif`

```
#ifndef STUDENT
#define STUDNET
#include<string>
struct Student{
    ...
}
#endif
```

> 预处理可以无视c++语言中关于作用域的规则

术语表

声明(*declaration*)：声称存在一个变量、函数或别处定义的类型

> 名字必须在定义或声明后才能使用

声明符(*declarator*)：声明的一部分，包括被定义的名字和类型修饰符

> 类型修饰符是可选的

定义(*definition*)：为某一个特定类型的变量申请存储空间

列表初始化(*list initialization*)：利用花括号把初值放在一起的初始化形式

字面值(*literal*)：不能改变的值

> 数字、字符、字符串

对象(*object*)：内存的一块区域，具有某种类型

> **变量是命名了的对象**

临时值(*temporary*)：编译器在计算表达式结果时创建的无名对象。

> 为某表达式创建的一个临时值，则此临时值将一直存在直到包含该表达式的最大的表达式计算完成为止

变量(*varialbe*)：命名的对象或引用

---

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，如果不经过强制转换，则它永远就是该数据类型 ↩

2. 第7章-再探构造函数-字面值常量类 ↩

3. 第19章-枚举类型 ↩