C 2023/11/11

娄雨轩
WX: 15859007915

计算机世界没有所谓的黑魔法,一切都在规则下运行,如果有什么出错了,首先怀疑是自己错了----南京大学 蒋炎岩

目录

基本术语

1. 头文件, 源文件, C标准， 未定义行为
2. 关键字与标识符
3. 变量与地址
4. 字面量
5. 常量
6. auto, const关键字
7. 块, 生存期
8. 范围和可见性

头文件, 源文件, C标准， 未定义行为

头文件

以.h为结尾的文件, 通常内部书写了函数, 结构体的签名等. 我们熟悉的 `stdio.h`就是头文件

源文件

以.c, .cpp, .hpp, .cxx等后缀结尾的文件, 都是项目的编译单元. 通常内部书写了函数的实现等. 我们熟悉的 `源.c` 或` 源.cpp` 或 `main.c` 或` main.cpp`就是一种源文件

C语言标准

C语言的标准是由C语言委员会共同提出维护的, 自89年以来, 每3年更新一次。每一次的更新都包含C语言的新的特性的加入与部分旧特性的放弃或修改。 C标准库是C语言编译器实现厂商按照C标准的规定对C语言的一些基本函数功能的实现， 标准库包括：stdio.h， stdlib.h等等， 不同编译器厂商（如MSVC, GNU，Clang三巨头）对于标准库的实现可能不一样， 但是是可以确定他们的实现是符合C语言标准的并且安全性和正确性是值得信任的。

GlibC:

GlibC网站

未定义行为 (Undefined Behavior, UB)

简单地说， 未定义行为就是编译器或C语言标准为规范的行为， 如数组越界等行为皆为未定义行为，这类行为可

关键字与标识符

关键字

“关键字”是对 C 编译器具有特殊含义的单词。
C 语言使用下列关键字:

```
auto, break, case, char, const, continue, default, do, double, else , enum, extern,
float, for, goto, if, inline, int, long, register, restrict, return, short, signed, sizeof ......
```

标识符（Identifiers / Symbols）

“Identifiers”或“symbols”是您为程序中的变量、类型、函数和标签提供的名称。 标识符名称在拼写和大小写上必须与任何关键字都不同。 不能将关键词（C 或 Microsoft）用作标识符；将它们保留以用于特殊用途。 通过在变量、类型或函数的声明中指定标识符来创建标识符。

```
int case = 1; // 非法: case关键字不能作为变量
```

然而有的时候我们迫不得已使用一些具有特殊意义的变量, 采取的方法是:

```
// 添加前导 '_'
int _case = 1; // 合法: _case是一个变量
```

变量与地址

变量（Variable）

变量是程序可操作的存储区的名称

```
int a = 1; // a是一个变量
float b = 1;  // b也是一个变量
char c[] = "HelloWorld"; // c也是一个变量
```

地址（Address）

简单地来说，地址是变量在内存空间中的位置。变量一定有地址。这个地址我们可以用取地址符号&结合格式串中的占位符%p输出出来。

```
int a = 1; // a是一个变量
float b = 1;  // b也是一个变量
printf("%p\n", &a); // 打印a的地址
printf("%p\n", &b); // 打印b的地址
```

指针（Pointer）

指针是指值为内存地址的变量

字面量

字面量（literal）是用于表达源代码中一个固定值的表示法。

字面量	例子	默认类型
整数	8 或 8i	int
整数	8ui	unsigned int
整数	8ull	unsigned long long
整数	8ll	long long
小数	8.0	double
小数	8.0f	float
字符	'a'	char

常量

常量

常数是可以在程序中用作值的数字、字符或字符串。 使用常量可表示不能修改的浮点、整数、枚举或字符值。
因此, 常量即使用const修饰的变量

```
const char* b = "Hello World" // "Hello World"是一个字符串字面量 b是一个常量指针
const int a = 1; // 1是一个字面量, a是一个常量
b = "Hello"; // ERROR:
b[1] = 'H'; // ERROR:
a = 2; // ERROR;
```

宏

我们还经常使用预处理指令#define来定义一个量， 例如:

```
#define PI 3.1415926
#define X 3

int main () {
    int a = X; // X是变量, 常量, 还是字面量?
    return 0;
}
```

在定义宏的时候，最好将其全部写为大写字母

auto, const关键字

auto 关键字

在23前的C标准下auto关键字作用为修饰自动变量(变量的空间自动在块结束时(})释放):

```
  auto int a = 0; //完全等价于 int a = 0;
```

然而在C23 (C++11)以后: auto关键字可以用于变量类型的推断，我们可以在声明变量的时候根据变量初始值的类型自动为该变量选择合适的类型，而不需要我们显式指定类型, 如:

```
// AFTER C23 / AFTER C++11:
auto x = 1.0f // x被推导为float类型
auto y = 1.0 // y被推导为double类型
auto z = 1 // x被推导为int类型
auto uz = 1u //x被推导为unsigned int类型
```

const关键字

被**const**关键字修饰的变量不可以被修改

```
const int a = 0;
a = 1  // 不合法: a不可以被修改
cont int b; // ERROR: 常量必须被初始化
```

因为const类型一旦创建之后就不能再改变, 所以const对象必须初始化。

块, 生存期

块

用{ }包围的程序语句, 我们称之为(程序)块

```
{
  ...
  int a;
  ...
}
```

生存期

“生存期”是其中存在变量或函数的程序执行的时段。标识符的存储持续时间决定其生存期

```
{
  a = 1 // 不合法，a的生存期未开始
  ...
  int a = 1;
  ...
  a = 1; // 合法,在生存期内
}
a = 1; // 不合法，在a的生存期已结束
```

范围和可见性

范围和可见性

标识符的“可见性”确定其可以引用的程序部分，即其“范围”。标识符仅在其“范围”包含的程序部分中可见（即可使用），这可能仅限于（按限制增长的顺序）它显示在其中的文件、函数、块或函数原型。 标识符的范围是可使用名称的程序的一部分。 这有时被称为“词法范围”。有四种范围：函数、文件、块和函数原型。除标签之外，所有标识符的范围都由在其上进行声明的级别决定。 以下针对每种范围的规则将管理标识符在程序中的可见性 块范围 带块范围的标识符的声明符或类型说明符显示在块中或函数定义中的形参声明列表中。它仅从其声明或定义的点到包含其声明或定义的块的结尾可见。 其范围限制为该块以及嵌入该块中的任何块，并结束于封闭该关联块的大括号处。 此类标识符有时称为“局部变量”。

总结

可见性, 作用域, 生存期(一般情况下)高度绑定, 一个变量对于其所在的块中, 处于变量的声明语句之后的所有语句都是可见的, 对于作用域也是, 同样地, 一个(自动回收/栈上)变量的生存期从其声明开始, 到其所在的块的右中括号(})为结束, 因此,我们可以用块{}来维护一个局部变量的生存期/可见性/作用域。同时, 对于块内部与块外部同名的变量, 优先会使用块内部的变量, 我们称之为"内部屏蔽". 以上总结在99%的场景里是正确的, 除非遇到由static, global, extern等关键字修饰的变量时（这些以后会介绍）

样例:

```c
#include <stdio.h>

int x = 0; // 全局变量(global)

int main() { // block1:
  int a = x; // 合法，x是可见的
  { // block2:
    int b = a; // 合法，a在此是可见的
    b = x; // 合法，x可见
    int a = 1; // 合法，屏蔽外部a
    int c = a; // c = a = 1
    a = 2;
    // Flag1:
    // c，a(block2)，b 依次消亡
  }
  return 0;
  // a(block1)，x消亡
}
```

思考:

当函数执行到Flag1时, 块block2中的a为何值, 块block1的a值为何值。 如何验证这两个a不是同一个变量?

变量类型

1. 整数类型
2. 浮点数类型
3. 字符串类型
4. 隐式类型转换
5. 强制类型转换

整数类型

整数类型

最后两栏在C标准头文件 <limits.h> 中定义

 "C整数类型"

浮点数类型

浮点数类型

float	4 Byte
double	8 Byte

浮点数的表达

```
float i = 1.f // 正确
float j = 1.0 // 正确 隐式类型转换到float类型
float k = 1e-1 // 正确 1 * 10^-1 等价于 1E-1 隐式转换到float类型

double x = 1.f //正确 隐式类型转换将1.f提升到double类型
double y = 1.0 //正确
double z = 1e-1 //正确 等价于1E-1
```

浮点数的输出

采用%f(float)或%lf(double)输出

完整的输出格式是%m.nlf，指定输出数据整数部分和小数部分共占m位，其中有n位是小数。如果数值长度小于m，则左端补空格，若数值长度大于m，则按实际位数输出。

采用%e (%E)输出

使用科学计数法进行输出

采用%g(%G)输出

用于输出 float 或 double 类型的浮点数，它会根据数值的大小自动选择 %f 或 %e\的格式。并且忽略尾部的0

```
float a = 123.456789;
double b = 123456789.123456789, c = 123456789.123456789;
printf("%.2f\n", a);  // 输出: 123.46
printf("%10.2f\n", a);  // 输出:     123.46
printf("%-10.2f\n", a);  // 输出: 123.46
printf("%.2lf\n", b);  // 输出: 123456789.12
printf("%10.2lf\n", b);  // 输出: 123456789.12
printf("%-10.2lf\n", b);  // 输出: 123456789.12
printf("%e\n", c);  // 输出: 1.234568e+08
printf("%.2e\n", c);  // 输出: 1.23e+08
printf("%10.2e\n", c);  // 输出:   1.23e+08
printf("%-10.2e\n", c);  // 输出: 1.23e+08
printf("%g\n", 5.0 / 2.0); // 输出: 2.5
printf("%lf\n", 5.0 / 2.0); // 输出: 2.500000
```

字符串类型

在C语言中，字符串实际上是使用 '\0' 结尾的字符数组。以下是一个简单的例子：

```
char str[6] = "Hello";
```

在这个例子中，str 是一个长度为6的字符数组，包含了5个字符 'H', 'e', 'l', 'l', 'o' 和一个终止符 '\0', 因此”Hello“长度为6。 C语言提供了一些处理字符串的函数，包括：

strlen(): 返回字符串的长度, strcpy(): 复制字符串, strcat(): 连接字符串, strcmp(): 比较两个字符串.

```
#include <string.h>
char str1[10] = "Hello";
char str2[10];
// 复制字符串
strcpy(str2, str1); //注意，你需要保证str2的容量大于str1，否则后果未知
// 连接字符串
strcat(str1, " World");
// 比较字符串
int cmp = strcmp(str1, str2);
// 获取字符串长度
int len = strlen(str1);
```

隐式类型转换

在C语言中，自动类型转换遵循以下规则：

1. 若参与运算量的类型不同，则先转换成同一类型，然后进行运算。
2. 转换按数据长度增加的方向进行，以保证精度不降低。如int型和long型运算时，先把int量转成long型后再进行运算。
3. a、若两种类型的字节数不同，转换成字节数高的类型
4. b、若两种类型的字节数相同，且一种有符号，一种无符号，则转换成无符号类型
5. 所有的浮点运算都是以双精度进行的，即使仅含float单精度量运算的表达式，也要先转换成double型，再作运算。
6. char型和short型参与运算时，必须先转换成int型。
7. 在赋值运算中，赋值号两边量的数据类型不同时，赋值号右边量的类型将转换为左边量的类型
8. 如果右边量的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度，丢失的部分按四舍五入向前舍入。

隐式类型转换

在何时发生隐式转换

1. 不同类型之间的数学运算中
2. 不同类型的相互赋值
3. 不同类型相互比较

思考

```
int main()
{
    char a = 0xb6;
      short b = 0xb600;
      int c = 0xb6000000;
      if (a == 0xb6)
          printf("a");
      if (b == 0xb600)
            printf("b");
      if (c == 0xb6000000)
            printf("c");
      return 0;
}
```

请一定要小心隐式转换，它常常发生在你意想不到的地方并且有的时候会引发意料之外的结果

强制类型转换

一种更安全的方法是强制类型转换
强制类型转换的一般方式是(T)expression
强制类型转换类似于运算符, 因此他有优先级
有的时候expression很复杂比如: (a + b) * c, 我们相对其结果强制类型转换到T

```
T res = (T) ((a + b) * c);
```

有的时候强制类型转换的结果也是运算表达式的一部分, 比如这个运算

```
T res = (T) ((a + b) * c) * d;
```

我们这样解决:

```
T res = ((T) ((a + b) * c)) * d;
```

当你搞不清楚优先级时,请使用括号(即使你非常了解优先级,在复杂表达式面前也应当使用括号,程序开发不是一个人的事,为了你的团队,也为了一星期/一个月后的你review自己的代码时能够快速理解自己的思路)

强制类型转换原则

1. **整数转浮点数**：当一个整数被转换为浮点数时，它的值会被保持不变，但是它的表示方式会变为浮点数的表示方式。例如，整数 `3` 会被转换为浮点数 `3.0`。

```
float a = (int)3; // a = 3.0, double 相同
```

2. **浮点数转整数**：当一个浮点数被转换为整数时，它的小数部分会被丢弃。这种转换不是四舍五入，而是直接丢弃小数部分(或者说, 向0取整)。例如，浮点数 `3.9` 会被转换为整数 `3`。

```
int a = (int)3.9; // a = 3.0;
int b = (int)(-3.9) // a = -3.0;
```

3. **大类型转小类型**：当一个大的类型(例如`long long int`)被转换为一个小的类型(例如`int`)时，如果大的类型的值超出了小的类型的范围,那么结果是**未定义的(UB)** ,这解释了第19页的现象

```
long long a = 0xffffffffffffffff;
int b = a; // b的值未知，不同编译器可能不同
```

4. **小类型转大类型**：当一个小的类型（例如`char`）被转换为一个大的类型（例如`int`）时，如果小的类型是有符号的，那么符号会被保持（这被称为符号扩展）；如果小的类型是无符号的，那么结果会是一个正数。

```
char a = 'a';
int b = a; // Correct, b = 97
unsigned char c = 11;
unsigned int d = c // Correct
```

5. **有符号类型转无符号类型**：当一个有符号类型的值被转换为无符号类型时，如果原来的值是负数，那么结果会是一个很大的正数。这是因为无符号类型不能表示负数，所以负数会被转换为无符号类型的最大值加上一再加上原来的值。

```
char a = -1; // char ~ [-128, 127], a = -1;
unsigned char b = a; // unsigned char ~ [0, 255], b = -1 + 255 + 1 = 255;
```

小练习(3min)

在...处添加代码, 使其输出answer为输入x的四舍五入之后的值, 例如: x = 3.5, answer = 4; x = 3.4, answer = 3; x = -3.5, answer = -4; x = -3.4, answer = -3;

```c
#include <stdio.h>
int main() {
  double x = 0.0;
  scanf("%lf", &x);
  ...
  ...
  int answer = (int)x;
  printf("%d", answer);
  return 0;
}
```

```c
#include <stdio.h>
int main() {
  double x = 0.0;
  scanf("%lf", &x);
  if (x >= 0) {
    x += 0.5;
  } else {
    x -= 0.5;
  }
  int answer = (int) x;
  printf("%d", answer);
  return 0;
}
```

值类型

左值与右值

浅显地来说，在内存中占据一定空间， 有地址的变量为左值， 否则为右值。

非常量左值可以在赋值运算中处于等号左边， 也可以处于等号右边。右值不可以在等号左边。能够置于等号左右并非是左值右值的判断依据（由const修饰的变量是左值， 但是不能置于等号左侧）， **能否被取地址运算符作用才是左值右值的判断依据**.

```
int a = 1; // a为左值，1为右值
char b = 'b'; // b为左值， 'b'为右值
scanf("%d %c", &a, &b); // 合法， a， b均为左值， 具有地址空间， 可以被取地址运算符&作用。
scanf("%c", &'c'; // 非法，'c'为右值， 不能被取地址运算符作用
'c' = 1; // 非法， 'c'为右值， 不能在等号左边
```

值类型会在C++中作为最重要的概念之一出现

思考：

1. 字符串字面值(例如：”Hello World!“)是左值还是右值？如何验证
2. 对于i而言， 表达式’++i‘是左值还是右值， 表达式’i++‘是左值还是右值?

```
int i = 0;
```

变量的声明与初始化

变量定义

用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。

变量声明

用于向程序表明变量的类型和名字。

变量初始化

指在变量使用前给予变量一个初始状态。

在未介绍extern关键词前，你可以把定义与声明看作是一件事，所以在不使用这个关键字时他们是一样的，extern关键字在未来将介绍

下面是一些示例：

```
extern int x; // 声明， 但不是定义（因为定义在其他文件中）
int y; // 声明也是定义， 但未初始化。（y的初始值是未知，视编译器，系统不同可能出现不同的情况）
int z = 0; // 声明， 定义， 并且初始化为 0;
```

WARNING:使用未初始化变量是UB(Undefined Behavior)会导致未知行为(可能崩溃退出/或继续携带错误运行)因此我建议你总是在变量声明时立即给予一个初值

指针类型初探

指针是指值为内存地址的变量

声明一个指针

对于一个类型T,我们声明**保存这个类型的变量的地址的变量**的方式为T*

```
int* p_i; // p_i可以保存某个int类型的变量的地址
float* p_f; // p_f可以保存某个float类型的变量的地址
```

初始化一个指针

```
int a = 0;
int* p = &a; // p 保存了a的地址 或 称 p指向a
// 如果暂时没有需要保存的地址，将其置为NULL
int* q = NULL; // NULL是C的一个宏,其值为((void*)(0)) 或 ((long long)(0))
```

我们还可以通过解引用运算符+指针来访问到一个值

```
*p = 1 // 此时修改a为1
```

也可以修改p的指向(即改变p保存的地址)

```
int b = 2, c = 3;
p = &b; // *p = 2
p = &c; // *p = 3
```

指针与生存期

```
int* p = NULL;
{
  int a = 1;
  p = &a;
}
*p = 1;
```

经典的UB, 指针指向了一个已经消亡的对象, 并且试图对这个地址写入数据

解决方法1

使指针变量的生存期和可见性与被指向的变量一致(或小于)

```
{
  int a = 1;
  int* p = &a;
  ...
  // p消亡, a消亡
}
*p = 1; // 错误: 未定义标识符(此处p不可见)
```

解决方法2

在临时变量消亡或即将消亡时改变p的指向

```
int* p = NULL;
{
  int a = 0;
  p = &a;
  // do something...
  ...
  p = NULL;
}
if (p != NULL) {
  // do something with p
}
```

指针与const

指向常量的指针(pointer to const) 不能用于修改所指对象的值,**想要存放常量对象的地址,只能使用指向常量的指针**, 但常量指针可以指向非常量, 只是你不能通过这个指针修改它, 并且与其他常量一样, **你必须在声明时初始化它**.

```
int a = 0;
const int* p = &a; // p是指向a的常量指针
*p = 1; // ERROR:
int b = *p // OK, 只读
const int c = 1;
int* q = c; // ERROR:
const int* q = c; // OK
const int* _q; // ERROR: 必须初始化
```

顶层const

如前所述, 指针本身就是一个变量, 它又可以指向另一个变量. 因此, 指针本身是不是常量以及指针所指的是不是常量就是两个相互独立的问题.

所以我们用**顶层const(top-level const)**表示指针本身是一个常量, 用**底层const(low-level const)**来表示指针所指的变量是一个常量.

换言之, 顶层const不允许我们修改指针的指向(所保存的地址), 而可以修改指针指向的对象的值, 底层const不允许我们修改指针指向的对象的值, 而可以修改指针的指向(所保存的地址).

一个指针可以即带有顶层const又带有底层const

```
int i = 0;
int * const p1 = &i; // 不能改变p1的指向，这是一个顶层const
const int ci = 42; // 不能改变ci的值，这是一个常量
const int* p2 = &c; // 允许改变p2的指向，这是一个底层const
p2 = &i // OK
const int* const p3 = p2 // 靠右的const是顶层const，靠左的const是底层const
```

快速记忆const

唯一原则: const的左结合律

const关键字默认和优先与左侧的关键字结合, 与表示类型的关键字结合时为底层const, 意为不可修改其指向的变量的值, 与*结合时为顶层const, 意为不可修改其本身的值(即指针的指向)

```
const int i = 0; // const左侧没有关键字，故与右侧关键字int结合，不能修改i的值
const int* p = &i; // const左侧没有关键字，故与右侧关键字int结合，底层const，不能修改p所指变量即i的值
int const* q = &i; // const优先与左侧关键字int结合，与const int* q完全相同
int* const _p = &i; // const优先与左侧*结合，顶层const，不能修改_p的指向
_p = NULL; // ERROR: 不能修改_p的值(指向)
// 注意: 没有const* T这种写法
```

选择语句

选择语句的基本结构：

```
if ( ... ) {
  ...
} else if ( ... ) {
  ...
} else {
  ...
}
```

在if后面的括号中可以写入一个**可以被显式或隐式转换为bool或int值的变量或表达式**
另外， 不论在if和else后有多少条语句， （即使只有一条）写上{}是非常好的习惯。

什么是可以被显式或隐式转换为bool或int值的变量或表达式：

1. 一切整数值
2. 比较表达式， 结果为整数的算数表达式， 逻辑表达式

逻辑运算符的运算顺序

在 C 语言中，逻辑运算符的优先级从高到低如下：

1. ！（逻辑非）
2. &&（逻辑与）
3. ||（逻辑或） 这意味着在一个复杂的逻辑表达式中，！运算符首先被计算，然后是 &&，最后是 ||。 此外，C 语言中的逻辑运算符还具有短路特性。对于 && 运算符，如果左边的表达式为假，那么整个逻辑表达式就为假，右边的表达式将不会被执行。对于 || 运算符，如果左边的表达式为真，那么整个逻辑表达式就为真，右边的表达式将不会被执行。 以下是一个示例：

```
int a = 0, b = 1, c = 2, d = 3;
if (a && b || c && d) {
    // ...
}
```

在这个例子中，首先计算 a && b 和 c && d，然后计算它们的 || 运算。 对于各种运算符的优先级顺序，为了考试你应该全部记下来，但是在工程中请你经可能地用括号表示你所期望的执行顺序，不仅可以防止自己出错， 也方便阅读，特别是在面对长表达式的时候。

短路性质的练习

```c
#include <stdio.h>

int main() {
    int a = 0, b = 0;
    if (a++ && b++) {
        printf("Both variables are incremented.\n");
    } else {
        printf("Not all variables are incremented.\n");
    }

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

以上代码的结果是什么?

短路性质的练习

```c
#include <stdio.h>
int main() {
    int a = 0, b = 0, c = 1;
    if (a++ && b++ || c++) {
        printf("At least one variable is incremented.\n");
    } else {
        printf("No variables are incremented.\n");
    }

    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

以上代码的结果是什么?

保持良好的代码风格

变量声明即初始化

同层次语句保持相同缩进 (像python一样)

良好的命名规范

尽量用代码本身表现意图, 而非注释

阅读良好的代码,学习实践中的Best Practice

K&R

SICP

课堂练习

探究自动变量的生存期/作用域
上一题的进阶: 简单探究C程序的内存模型(即 变量在内存中是如何排布的)