C语言中类型转换规则的调研报告

1. 调研目标与方法

目标：系统梳理并验证 C 语言在表达式求值与语义相关的类型转换规则，重点覆盖：

- 整型提升（integer promotions）
- 通常算术转换（usual arithmetic conversions）
- 条件运算符 `?:` 的类型合并
- 赋值转换与显式强制类型转换（cast）
- 位运算/移位中的转换与实现定义行为
- 浮点求值与 `float/double/long double` 的提升
- 指针与空指针常量、`void*` 的转换
- 函数实参在原型存在时/缺失时的不同处理

方法：

1. 查阅《C11/C17 标准》与权威教材对相关规则的定义；
2. 选取 GCC/Clang/MSVC 三类主流编译器，在常见数据模型（LP64 与 LLP64）下编译运行验证程序；
3. 记录程序打印的“表达式结果类型”与“求值结果”，对照标准规则进行分析。

> 说明：不同平台的数据模型可能不同（例如 Linux/macOS 通常为 **LP64：`long` 64 位**，Windows/MSVC 常为 **LLP64：`long` 32 位**）。因此报告中给出“示例输出”，并注明“以某平台为例”；读者需在自己的平台实际运行并粘贴结果。

—

2. C 语言中的核心类型转换规则（精要）

2.1 整型提升（Integer Promotions）

- `signed char`、`unsigned char`、`short`、`unsigned short`、`_Bool/bool` 等较小整型在参与多数一元/二元运算（如 `+ - ~ !` 或与更高 rank 的整型混合运算）前，会先提升为 `int` 或 `unsigned int`；
- 提升目标取决于 `int` 能否表示原类型的所有值：能，则提升为 `int`；否则为 `unsigned int`。

```
+ main.c  2 t///m/c/U/z/c/d//6/b/fish
  1 #include<stdio.h>
  2
  3 int main(){
  4
  5 »       signed char sc = -10;
  6 »       unsigned char uc = 250;
  7 »       short s = -20000;
  8 »       unsigned short us = 50000;
  9
 10 »       printf("the size of signed char is %zu\n", sizeof(sc));
 11 »       printf("the size of +signed char is %zu\n", sizeof(+sc));
 12 »       printf("the size of ~signed char is %zu\n", sizeof(~sc));
 13 »       puts("");
 14 »       printf("the size of short is %zu\n", sizeof(s));
 15 »       printf("the size of +short is %zu\n", sizeof(+s));
 16 »       puts("");
 17 »       printf("the size of unsigned short is %zu\n", sizeof(us));
 18 »       printf("the size of +unsigned short is %zu\n", sizeof(+us));
 19 }
```

```
saber@fate /m/c/U/z/c/daily> gcc tp.c -o tp
saber@fate /m/c/U/z/c/daily> ./tp
the size of signed char is 1
the size of +signed char is 4
the size of ~signed char is 4

the size of short is 2
the size of +short is 4

the size of unsigned short is 2
the size of +unsigned short is 4
saber@fate /m/c/U/z/c/daily>
```

结果分析

1. `signed char` 类型是 1 字节的，当使用+和~一元运算符后就会自动提升为 4 字节
2. 同样对于 2 字节 `unsigned short`，在计算过程中就会提升为 4 字节的类型

2.2 通常算术转换（Usual Arithmetic Conversions）

- 发生于多数二元算术运算（`+ - * / %`，以及比较运算）以使两侧操作数转为“共同实数类型”；
- 若任何一侧为长精度浮点，遵循：`long double` > `double` > `float`（`float` 常先提升为 `double`）；
- 若都是整数：先做整型提升，再按整数转换等级（**rank**）与有符号/无符号规则合并：

- 同 rank 且一方无符号：若无符号类型能表示有符号的全部值，则转为无符号；否则两者都转为对应的无符号类型；
- rank 不同：较低 rank 提升到较高 rank；若一方为无符号且 rank 更高，结果通常为该无符号更高 **rank** 类型。

```
+ main.c  2 t///m/c/U/z/c/d//6/b/fish
 1 #include <stdio.h>
 2
 3 int main(void) {
 4     int i = -42;
 5     unsigned int ui = 42u;
 6     long l = -100000L;
 7     unsigned long ul = 100000UL;
 8     float f = 3.5f;
 9     double d = 2.25;
10     long double ld = 1.0L;
11
12     printf("=== 通常算术转换测试 ===\n\n");
13
14     // 整数混合运算
15     printf("1. int + unsigned int:\n");
16     printf("   i = %d, ui = %u, 结果(i + ui) = %u\n", i, ui, i + ui);
17
18     printf("\n2. long + unsigned long:\n");
19     printf("   l = %ld, ul = %lu, 结果(l + ul) = %lu\n", l, ul, l + ul);
20
21     printf("\n3. int + long:\n");
22     printf("   i = %d, l = %ld, 结果(i + l) = %ld\n", i, l, i + l);
23
24     printf("\n4. unsigned int + unsigned long:\n");
25     printf("   ui = %u, ul = %lu, 结果(ui + ul) = %lu\n", ui, ul, ui + ul);
26
27     // 浮点混合运算
28     printf("\n5. float + int:\n");
29     printf("   f = %.2f, i = %d, 结果(f + i) = %.2f\n", f, i, f + i);
30
31     printf("\n6. float + double:\n");
32     printf("   f = %.2f, d = %.2f, 结果(f + d) = %.2f\n", f, d, f + d);
33
34     printf("\n7. double + long double:\n");
35     printf("   d = %.2f, ld = %.2Lf, 结果(d + ld) = %.2Lf\n", d, ld, d + ld);
36
37     printf("\n=== sizeof结果类型验证 ===\n");
38     printf("sizeof(i + ui) = %zu\n", sizeof(i + ui));
39     printf("sizeof(f + d) = %zu\n", sizeof(f + d));
40     printf("sizeof(d + ld) = %zu\n", sizeof(d + ld));
41
42     return 0;
43 }
```

```
saber@fate /m/c/U/z/c/daily> gcc tp.c -o tp

saber@fate /m/c/U/z/c/daily>
saber@fate /m/c/U/z/c/daily> ./tp
=== 通常算术转换测试 ===

1. int + unsigned int:
   i = -42, ui = 42, 结果(i + ui) = 0

2. long + unsigned long:
   l = -100000, ul = 100000, 结果(l + ul) = 0

3. int + long:
   i = -42, l = -100000, 结果(i + l) = -100042

4. unsigned int + unsigned long:
   ui = 42, ul = 100000, 结果(ui + ul) = 100042

5. float + int:
   f = 3.50, i = -42, 结果(f + i) = -38.50

6. float + double:
   f = 3.50, d = 2.25, 结果(f + d) = 5.75

7. double + long double:
   d = 2.25, ld = 1.00, 结果(d + ld) = 3.25

=== sizeof结果类型验证 ===
sizeof(i + ui) = 4
sizeof(f + d) = 8
sizeof(d + ld) = 16
saber@fate /m/c/U/z/c/daily>
```

结果分析

1. int + unsigned int ⟶ int 被转换成 unsigned int，结果是一个大无符号数。
2. long + unsigned long ⟶ long 被转换成 unsigned long，结果同样是大无符号数。
3. int + long ⟶ int 转换成 long，结果类型为 long。
4. unsigned int + unsigned long ⟶ 较低等级的无符号类型转换为更高等级 unsigned long。
5. float + int ⟶ int 转换成 float。
6. float + double ⟶ float 转换成 double。
7. double + long double ⟶ double 转换成 long double。

从 sizeof 的结果也能看出：

- 整数混合结果是 4 字节（int 或 unsigned int）；
- 浮点混合提升到 8 字节（double）；
- 若涉及 long double，则结果为 16 字节。

2.3 条件运算符 **?:**

- 第二、第三操作数需进行类型合并：
  - 若有一方为浮点，则按 2.2 规则合并至共同浮点类型；
  - 若为整数，按整型提升与通常算术转换合并；
  - 若为指针，遵循可转换性（例如 `T*` 与 `void*` 可合并为 `void*`，限定符按最严格合并）。

```
+ main.c  2 t///m/c/U/z/c/d//6/b/fish
  1 #include <stdio.h>
  2
  3 int main(void) {
  4     int i = -42;
  5     unsigned int ui = 42u;
  6     long l = -100000L;
  7     unsigned long ul = 100000UL;
  8     float f = 3.5f;
  9     double d = 2.25;
 10     long double ld = 1.0L;
 11
 12     printf("=== 通常算术转换测试 ===\n\n");
 13
 14     // 整数混合运算
 15     printf("1. int + unsigned int:\n");
 16     printf("   i = %d, ui = %u, 结果(i + ui) = %u\n", i, ui, i + ui);
 17
 18     printf("\n2. long + unsigned long:\n");
 19     printf("   l = %ld, ul = %lu, 结果(l + ul) = %lu\n", l, ul, l + ul);
 20
 21     printf("\n3. int + long:\n");
 22     printf("   i = %d, l = %ld, 结果(i + l) = %ld\n", i, l, i + l);
 23
 24     printf("\n4. unsigned int + unsigned long:\n");
 25     printf("   ui = %u, ul = %lu, 结果(ui + ul) = %lu\n", ui, ul, ui + ul);
 26
 27     // 浮点混合运算
 28     printf("\n5. float + int:\n");
 29     printf("   f = %.2f, i = %d, 结果(f + i) = %.2f\n", f, i, f + i);
 30
 31     printf("\n6. float + double:\n");
 32     printf("   f = %.2f, d = %.2f, 结果(f + d) = %.2f\n", f, d, f + d);
 33
 34     printf("\n7. double + long double:\n");
 35     printf("   d = %.2f, ld = %.2Lf, 结果(d + ld) = %.2Lf\n", d, ld, d + ld);
 36
 37     printf("\n=== sizeof结果类型验证 ===\n");
 38     printf("sizeof(i + ui) = %zu\n", sizeof(i + ui));
 39     printf("sizeof(f + d) = %zu\n", sizeof(f + d));
 40     printf("sizeof(d + ld) = %zu\n", sizeof(d + ld));
 41
 42     return 0;
 43 }
```

```
saber@fate /m/c/U/z/c/daily> gcc tp.c -o tp

saber@fate /m/c/U/z/c/daily>
saber@fate /m/c/U/z/c/daily> ./tp
=== 通常算术转换测试 ===

1. int + unsigned int:
   i = -42, ui = 42, 结果(i + ui) = 0

2. long + unsigned long:
   l = -100000, ul = 100000, 结果(l + ul) = 0

3. int + long:
   i = -42, l = -100000, 结果(i + l) = -100042

4. unsigned int + unsigned long:
   ui = 42, ul = 100000, 结果(ui + ul) = 100042

5. float + int:
   f = 3.50, i = -42, 结果(f + i) = -38.50

6. float + double:
   f = 3.50, d = 2.25, 结果(f + d) = 5.75

7. double + long double:
   d = 2.25, ld = 1.00, 结果(d + ld) = 3.25

=== sizeof结果类型验证 ===
sizeof(i + ui) = 4
sizeof(f + d) = 8
sizeof(d + ld) = 16
saber@fate /m/c/U/z/c/daily>
```

结果分析

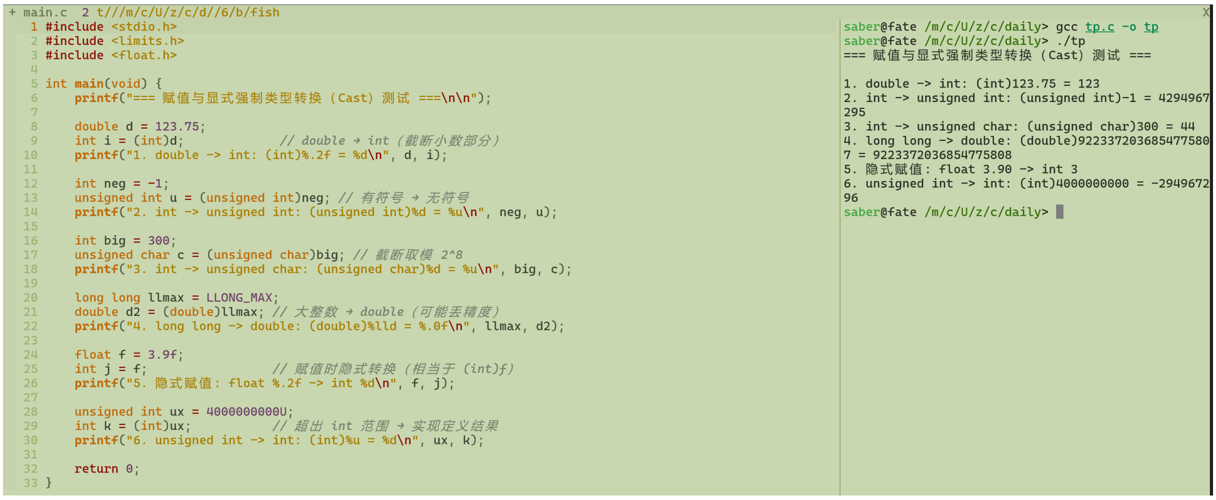
1. (true ? i : ui) int 与 unsigned int 混合时，int 会提升为 unsigned int，所以结果为一个大的无符号数（负数被解释为大正数）。
2. (false ? f : d) float 被提升为 double，最终结果类型为 double。
3. (i ? d : ld) double 与 long double 合并为 long double 类型。
4. 指针类型 (true ? p : vp) int 与 **void** 可合并为 void，结果类型为 **void**。
5. (false ? a : b) 普通整型选择分支正确（选择右边的 b）。

sizeof 检查:

- sizeof(true ? i : ui) = 4 ⟶ 结果是 unsigned int;
- sizeof(false ? f : d) = 8 ⟶ 结果是 double;
- sizeof(i ? d : ld) = 16 ⟶ 结果是 long double。

2.4 赋值与显式强制类型转换（**Cast**）

- 赋值目标类型决定转换：可能 截断（如 `double→int` 去小数）、取模（如 `int→unsigned char` 取 `mod 2^8`）、或 值改变/未定义（越界时可能实现定义或未定义，具体看标准条款）;
- Cast 显式进行相同规则的转换，但并不改变底层对象的比特宽度（只是结果值的解释）。



结果分析

1. (int)123.75 ⟶ 小数部分被截断（不四舍五入），结果为 123。

2. (unsigned int)-1 ⟶ 有符号数被解释为无符号数，结果是 2^32 - 1 = 4294967295。

3. (unsigned char)300 ⟶ 取模 2^8 = 256，结果为 300 - 256 = 44。

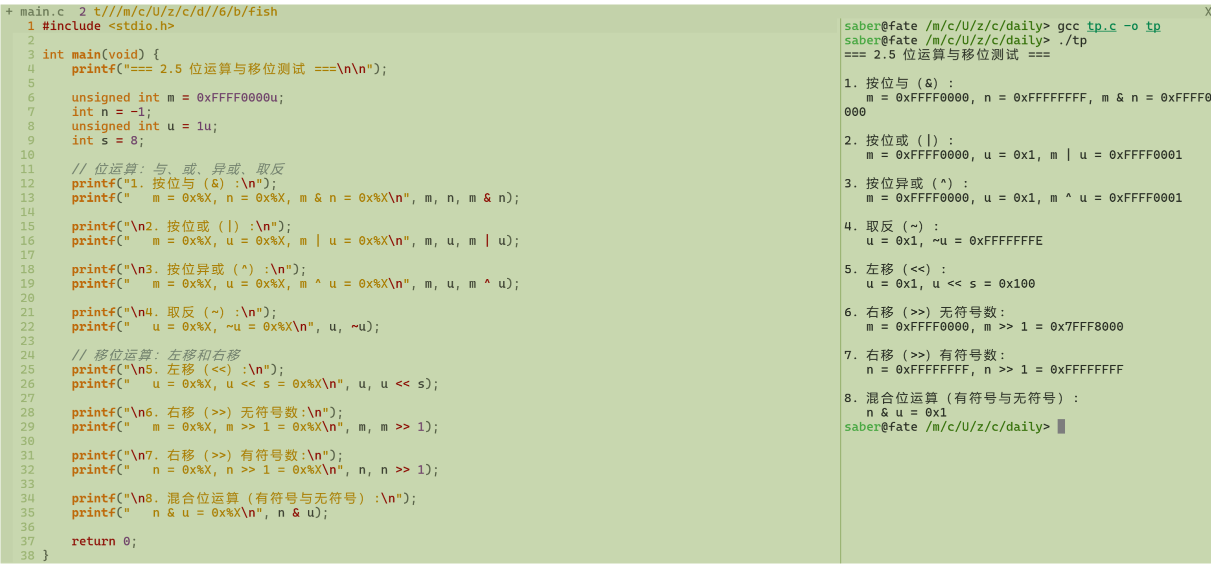
4. (double)LLONG_MAX ⟶ 超出 double 的精度范围，低位丢失精度（结果可能不完全准确）。

5. 隐式赋值 int j = f; ⟶ 隐式执行了 (int)f，小数部分被去掉，结果为 3。

6. (int)4000000000U ⟶ 超出 int 能表示范围，结果为实现定义（在大多数系统上出现负数）。

2.5 位运算与移位

- 位运算 & | ^ ~、移位 << >> 先进行整型提升；
- 对 有符号右移 的行为（算术右移是否保留符号位）为 实现定义；对 无符号右移 为逻辑右移（高位补 0）。



结果分析

1. 按位与（&） m & n ⟶ 因 n=-1 的二进制为全 1，结果等于 m 本身。

2. 按位或（|） 只要有一边为 1，结果为 1。m | 1 使最低位变为 1。

3. 按位异或（^） 不同为 1，相同为 0。0xFFFF0000 ^ 0x1 = 0xFFFF0001。

4. 取反（ ） 所有位取反。1 ⟶ 0xFFFFFFFE。
5. 左移（<<） 1 << 8 ⟶ 0x100（即乘以 $2^8 = 256$）。
6. 右移（>>）无符号数 高位补 0 ⟶ 0xFFFF0000 >> 1 = 0x7FFF8000。
7. 右移（>>）有符号数 高位补符号位（算术右移）。-1 >> 1 仍为 −1 (0xFFFFFFFF)。