

COMP26020 Programming Languages and Paradigms Part 1: C Programming

Type Conversion and Casting

Implicit Type Conversions

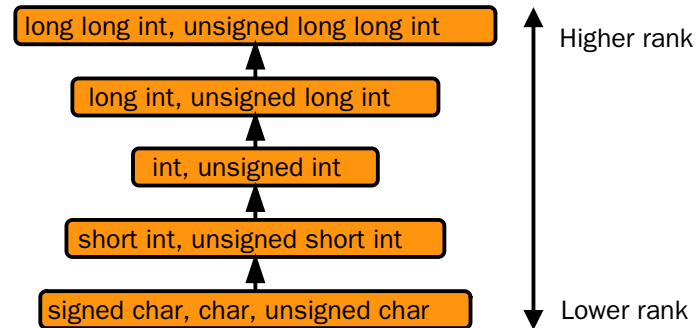
```
#include <stdio.h>

int main(int argc, char **argv) {
    char char_var = 12;    // 1 byte, -128 to 127
    int int_var = 1000;    // 4 bytes, -2*10^9 to 2*10^9
    long long ll_var = 0x840A1231AC154; // 8 bytes, -9*10^18 to 9*10^18

    // here, char_var is first promoted to int, then the result of the first
    // addition is promoted to long long
    long long result = (char_var + int_var) + ll_var;
    printf("%lld\n", result);
    return 0;
}
```

[15-preprocessor/implicit-type-conversion.c](#) 


Integer Promotion



1. Same type? no promotion
2. Same signed/unsigned attribute? lesser rank promoted to greater rank
3. Unsigned rank \geq other operand rank? signed promoted to the unsigned rank
4. Signed type can represent all the values of the unsigned type? unsigned operand promoted to signed type
5. Otherwise, both operands converted to the unsigned type corresponding to the signed operand's type

Integer Promotion

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < ui); // prints 0! si converted to unsigned int
```

[15-preprocessor/integer-promotion.c](#) 

- Keep these rules in mind!

Integer Promotion


- Always remember the storage sizes of types

```
int main(int argc, char **argv) {  
    int i = -1000;  
    unsigned int ui = 4294967295;  
    printf("%d\n", i + ui); // prints -1001  
  
    //      i:   1111111111111111111111110000011000 (A)  
    //     ui:   1111111111111111111111111111111111 (B)  
    //  i + ui:  1111111111111111111111110000010111 (C)  
    // final:   1111111111111111111111110000010111 (D)  
  
    // (A) originally 2's complement promoted to unsigned  
    // (B) standard unsigned representation, max number an unsigned int can store  
    // (C) addition result  
    // (D) result overflows 32 bits as an int (expected by %d), losing MSB  
    // Solution: use %ld rather than %d to store the result on 64 bits  
  
    return 0;  
}
```

[15-preprocessor/integer-overflow.c](#)  


Integer to Floating-Point Conversion

```
int main(int argc, char **argv) {  
    //prints 7: 25/10 rounds to 2; 2 * 15 = 30; 30/4 rounds to 7  
    printf("%d\n", 25 / 10 * 15 / 4);  
  
    // prints 7.5: 25/10 rounds to 2; 2*15 = 30; 30 gets converted to  
    // 30.0 (double) and divided by 4.0 (double) giving result 7.5 (double)  
    printf("%lf\n", 25 / 10 * 15 / 4.0);  
  
    // prints 9.375: 25.0 / 10.0 (converted from 10) is 2.5, multiplied by 15.0  
    // (converted from 15) gives 37.5, divided by 4.0 (converted from 4) gives  
    // 9.375  
    printf("%lf\n", 25.0 / 10 * 15 / 4);  
  
    // prints garbage, don't try to interpret a double as an int!  
    printf("%d\n", 25.0 / 10 * 15 / 4);  
  
    return 0;  
}
```

[15-preprocessor/int-float-conversion.c](#) 

Parameter Passing

```
void f1(int i) {  
    printf("%d\n", i);  
}  
  
void f2(double d) {  
    printf("%lf\n", d);  
}  
  
void f3(unsigned int ui) {  
    printf("%u\n", ui);  
}  
  
int main(int argc, char **argv) {  
    char c = 'a';  
    unsigned long long ull = 0x4000000000000;  
  
    f1(c);    // prints 97 (ascii code for 'a')  
    f2(c);    // prints 97.0  
    f3(ull);  // overflows int ... prints 0 (lower 32 bits of 0x4000000000000)  
  
    return 0;  
}
```

[15-preprocessor/parameter-passing.c](#) 

Type Casting

- **Casting** allows to force the type of an expression
- syntax: `(type)expression`


```
// prints 3.75: 4 gets converted to 4.0
printf("%lf\n", (float)15/4);

// prints 4: 2.5 converted to 2 (int), multiplied by 12 gives 24, divided
// by 5 gives 4
printf("%d\n", ((int)2.5 * 12)/5);

// prints 4.8: 2*12 = 24, converted to 24.0, divided by 5.0 gives 4.8
printf("%lf\n", ((int)2.5 * 12)/(double)5);
return 0;
```

[15-preprocessor/type-casting-1.c](#) 

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < (int)ui); // prints 1
```

[15-preprocessor/type-casting-2.c](#) 

Generic Pointers

- `void *` can be used as a generic pointer to pass data of different types

```
typedef enum {
    CHAR, INT, DOUBLE
} type_enum;

void print(void *data, type_enum t) {
    switch(t) {
        case CHAR:
            printf("character: %c\n",
                *(char *)data);
            break;
        case INT:
            printf("integer: %d\n",
                *(int *)data);
            break;
        case DOUBLE:
            printf("double: %lf\n",
                *(double *)data);
            break;
        default:
            printf("Unknown type ...\n");
    }
}
```

```
int main(int argc, char **argv) {
    char c = 'x';
    int i = 42;
    double d = 11.5;

    print((void *)&c, CHAR);
    print((void *)&i, INT);
    print((void *)&d, DOUBLE);
    return 0;
}
```

[15-preprocessor/generic-pointer.c](#) 

Summary

- Type conversion
 - Implicit: integer promotion, integer to floating point types, parameter passing
 - Explicit: type casting
-

Feedback form: <https://bit.ly/2VDzVgG>

