Lecture 18: Type Conversion and Casting

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/18-type-conversion-casting.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we discuss type conversion and casting in C.

Implicit Type Conversions

In many situations the compiler performs **implicit conversions** between types. With arithmetic operations, it applies **integer promotion**, converting smaller types to larger ones. Consider this example:

```c
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;
```

We declare signed integers of various sizes. We have a `char`, note that we can store integers in chars, on x86-64 `char` size is one byte so it can store only 256 values. We also have a regular `int`, on x86-64 its size is 4 bytes so it can store about 4 billions values. And we also have a `long long int`, on x86-64 its size is 8 bytes, so it can store much more numbers.

Considering the operation computing the value of `result`, when the `char` is first added to the `int`, the `char` is automatically promoted to an `int`, and what is between parentheses evaluates to an `int`. This value is then promoted to a `long long` because it's added with a `long long`, and the resulting value is a `long long`. We can store it in a `long long` variable (`result`) without fear of loosing precision or data truncation due to the intermediary operations and types.

These implicit conversions, with an operation applied to integers of different types, are called **integer promotion**.

Integer Promotion

The key idea behind integer promotion is that no information is lost when going from smaller to larger types. Integer types are given ranks. In decreasing order of rank we have:

- `long long int`, `unsigned long long int` (highest rank).
- `long int`, `unsigned long int`.
- `int`, `unsigned int`.
- `short int`, `unsigned short int`.
- `signed char`, `char`, `unsigned char` (lowest rank).

The promotion rules for 2 operands of an operation are as follows, in order:

1. If the operands have the same type there is no need for promotion.
2. If both operands are signed or both operands are unsigned, the operand of lesser rank is promoted to the type of the operand of higher rank.
3. If the rank of the unsigned operand is superior or equal to the rank of the other operand, the signed operand is promoted to the type of the unsigned operand.
4. If the signed operand type can represent all the values of the unsigned operand type, the unsigned operand gets promoted to the signed type.
5. Otherwise, both operands are converted to the unsigned type corresponding to the signed operand's type.

It is important to keep these rules in mind, otherwise bugs may arise when mixing signed and unsigned integers. Consider this example:

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

We have a comparison between a signed `int` which is `-1` and an `unsigned int`. According to the rules the signed `int` gets converted to an `unsigned int`. And the binary representation of `-1` in memory when interpreted as an `unsigned int`, corresponds to the maximum value that can be stored in an unsigned int, which is about 4 billions So the expression evaluates to false, which is 0 in C. This is very counter-intuitive.

Integer Overflows

It is important to keep in mind the storage size of all types when mixing them in operations. Here's another example of behaviour that can be surprising:

```
int main(int argc, char **argv) {
    int i = -1000;
    unsigned int ui = 4294967295;
    printf("%d\n", i + ui); // prints -1001
    //      i:   11111111111111111111110000011000 (A)
    //     ui:   11111111111111111111111111111111 (B)
    // i + ui:  111111111111111111111110000010111 (C)
    //  final:   11111111111111111111110000010111 (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), loosing MSB
    // Solution: use %ld rather than %d to store the result on 64 bits
    return 0;
}
```

We add a signed `int`, `i`, equal to `-1000`, to an `unsigned int`, `ui`, that is actually the maximum value we can store with that type. `i` is promoted to an `unsigned int`, and because signed numbers are encoded with 2's complement we have the leading ones in the binary representation. `ui` being the maximum number that can be stored in an `unsigned int`, it is 32 bits full of ones. The addition overflows and the most significant bit gets truncated. What we obtain is the binary representation of -1001.

Integer to Floating-Point Conversion

Regarding floating point numbers, if an operand is `float/double`, the other gets converted to `float/double`. This is another implicit conversion realised by the compiler. Conversion spreads from left to right. See the following examples with explanation in comments:

```
//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);
```

Implicit Conversion When Passing Parameters

Conversion also happens implicitly when calling functions. Consider this example:

```
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 = 0x400000000000;

    f1(c);    // prints 97 (ascii code for 'a')
    f2(c);    // prints 97.0
    f3(ull);  // overflows int ... prints 0 (lower 32 bits of 0x400000000000)

    return 0;
}
```

Here we have a character variable `c` containing `'a'`, passed as parameter to functions expecting `int` (`f1`) and double (`f2`). Characters are encoded with ascii code, so when this data is interpreted as an `int` or a `double`, we get the ascii code for `'a'`: 97 or 97.0. We also have a `long long unsigned` variable, `ull`, that is passed to a function expecting an `unsigned int`, but `ull` is too large for that. So when printed within the function body we only see its lower 32 bits.

Note that this code, and all the faulty programs presented in this lecture, is perfectly legit from the compiler point of view: it will produce no warnings. Hence it is really important to be aware of the various implicit conversion rules to avoid bugs, some of which may be quite hard to investigate and fix.

Type Casting

Type casting lets the programmer force a conversion. It is achieved by writing the target type between parentheses in front of the expression one wish to covert. For example here we cast an int into a float:

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

Here we cast a floating point number into an integer:

```
// 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);
```

Another example:

```
// 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);
```

Recall the example above in which an `unsigned int` equal to `1` was evaluated as not inferior to an `int` equal to `-1` due to integer promotion. This can be fixed that with a cast:

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

We force the unsigned variable to be converted to a signed one, and we now compare two signed `int`.

Generic Pointers

In combination with the special type `void *`, casting also allows us to implement generic pointers in C. Generic pointers allow a pointer parameter or return value to point to data of different types. Consider the following code:

```
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");
    }
}
```

We have a function that takes a `void *`` generic pointer as parameter. According to a second parameter that is an`enum, the function prints the value of what is pointed by the pointer. It can be a`char, an`int, or a`double. When we call the function, we cast the pointers to these data types to`void *. Of course, we need to indicate the type somewhere, in this case we use the`enum‘. When printing the value, we use another cast to get the proper type.