

Chapter 40: Implicit and Explicit Conversions

Section 40.1: Integer Conversions in Function Calls

Given that the function has a proper prototype, integers are widened for calls to functions according to the rules of integer conversion, C11 6.3.1.3.

6.3.1.3 Signed and unsigned integers

When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.

Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Usually you should not truncate a wide signed type to a narrower signed type, because obviously the values can't fit and there is no clear meaning that this should have. The C standard cited above defines these cases to be "implementation-defined", that is, they are not portable.

The following example supposes that `int` is 32 bit wide.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val); /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val); /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}
```

```

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64 = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8); /* param_u8 val is 127 */
    param_u16(u8); /* param_u16 val is 127 */
    param_u32(u8); /* param_u32 val is 127 */
    param_u64(u8); /* param_u64 val is 127 */
    param_s8(u8); /* param_s8 val is 127 */
    param_s16(u8); /* param_s16 val is 127 */
    param_s32(u8); /* param_s32 val is 127 */
    param_s64(u8); /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64); /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64); /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}

```

Section 40.2: Pointer Conversions in Function Calls

Pointer conversions to `void*` are implicit, but any other pointer conversion must be explicit. While the compiler allows an explicit conversion from any pointer-to-data type to any other pointer-to-data type, accessing an object through a wrongly typed pointer is erroneous and leads to undefined behavior. The only case that these are allowed are if the types are compatible or if the pointer with which you are looking at the object is a character type.

```

#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

```

```

int main(void) {

    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although they have identical definitions,
     * the types are not compatible, and that this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}

```