# Chapter 34: Generic selection

| Parameter | Details |
|---|---|
| generic-assoc-list | generic-association **OR** generic-assoc-list , generic-association |
| generic-association | type-name : assignment-expression **OR** default : assignment-expression |

## Section 34.1: Check whether a variable is of a certain qualified type

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Output:

```
i is a const int
j is a non-const int
k is of other type
```

However, if the type generic macro is implemented like this:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

The output is:

```
i is a non-const int
j is a non-const int
k is of other type
```

This is because all type qualifiers are dropped for the evaluation of the controlling expression of a `_Generic` primary expression.

## Section 34.2: Generic selection based on multiple arguments

If a selection on multiple arguments for a type generic expression is wanted, and all types in question are arithmetic types, an easy way to avoid nested `_Generic` expressions is to use addition of the parameters in the controlling expression:

```
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),               \
                           int:      max_int,      \
                           unsigned: max_unsigned, \
                           default:  max_double)   \
                     ((X), (Y))
```

Here, the controlling expression `(X)+(Y)` is only inspected according to its type and not evaluated. The usual conversions for arithmetic operands are performed to determine the selected type.

For more complex situation, a selection can be made based on more than one argument to the operator, by nesting them together.

This example selects between four externally implemented functions, that take combinations of two int and/or string arguments, and return their sum.

```
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                           \
   _Generic((y),          int: AddStrInt,   \
                         char*: AddStrStr,   \
                  const char*: AddStrStr )

#define AddInt(y)                           \
   _Generic((y),           int: AddIntInt,  \
                         char*: AddIntStr,  \
                  const char*: AddIntStr )

#define Add(x, y)                           \
   _Generic((x) ,         int: AddInt(y) ,  \
                        char*: AddStr(y) ,  \
                 const char*: AddStr(y))    \
                         ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Even though it appears as if argument `y` is evaluated more than once, it isn't 1. Both arguments are evaluated only once, at the end of macro Add: `( x , y )`, just like in an ordinary function call.

1 (Quoted from: ISO:IEC 9899:201X 6.5.1.1 Generic selection 3)
The controlling expression of a generic selection is not evaluated.

## Section 34.3: Type-generic printing macro

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Output:

```
int: 42
double: 3.14
unknown argument
```

Note that if the type is neither `int` nor `double`, a warning would be generated. To eliminate the warning, you can add that type to the `print(X)` macro.