Expressions and assignments

A more pathological situation occurs when unsigned and signed 32-bit integers are mixed in an expression and assigned to a `signed long`. The C and C++ language standards effectively promote the expression to `unsigned` in the case where `int` and `unsigned int` are mixed. The following code fragment shows a case where a signed 32-bit integer with a value of -2 is added to an unsigned 32-bit integer whose value is 1. Arithmetically the result should be -1, but in a 64-bit system the right hand value expression becomes `unsigned` according to the C promotion rules, and then, upon assignment to a 64-bit integer, the sign is not extended. One solution is to cast one of the operands to its 64-bit type. This will cause the other operands to be promoted to 64-bits as is the resulting expression, and no further conversion is needed when the expression is assigned. Another solution is to cast the entire expression such that sign extension occurs on the assignment.

```
long n;
int i = -2;
unsigned k = 1;
```

Expression	Result on 32-bit system	Result on 64-bit system	Explanation
`n = i + k;`	-1	4294967295	32-bit `int` **i** is added to 32-bit `unsigned int` **k**. The expression becomes a 32-bit `unsigned int`. This is then assigned to `long` **n**. Because the expression is unsigned, no sign extension occurs.
`n = (long)i + k;`	-1	-1	32-bit `int` **i** is promoted to a `long` and **k** is promoted to an `unsigned long` resulting in a 64-bit unsigned expression. The sign of **i** has been extended when it was promoted. No conversion is performed when the result is assigned to **n**.
`n = (int)(i + k);`	-1	-1	This expression achieves the result in a different way. The `unsigned int` expression is then cast to an `int`. Since the resulting expression is signed, the sign is properly extended when assigned to **n**.

Passing parameters into functions

In C and C++, parameters are normally passed to functions by value, and C++ also has a call by reference feature. In all cases, the parameters are fully evaluated first, whether single variables, constants or expressions. The order of the evaluation of the parameters is unspecified and may be different, not only on different systems, but also on the same system.

The C language standard defines function prototypes where parameters passed into a function must be typed, as in:

```
double AMathFunction(double, int);
```

In this case, all the parameters are fully defined. There is another case where C allows a variable number of parameters. In this case, a function may take an unknown number of parameters:

```
int printf(const char *, ...);
```

The ellipsis (`. . .`) tells the compiler that the caller of the function may provide more than the single parameter. There is no type checking on the additional parameters.

A third case is provided as compatibility to legacy C applications where a function prototype is either not included at all or a function declaration is included. The function declaration contains no parameter list, and, similar to the variable parameter list above, there is no type checking. This compatibility is frequently termed “K&R” for Brian Kernighan and Dennis Ritchie, the inventors of C. In the case where the data type is not defined, the parameter is promoted according to the usual promotion rules defined by the standard. In the case where the data type is defined by a function prototype, the parameter is converted to that type according to the standard rules. When the type of a parameter is not specified, the parameter is promoted to the larger type. In a 64-bit system, integral types are converted to 64-bit integral types and single precision floating-point types are promoted to double precision. If a return value is not otherwise specified, the default return value for a function is `int`.

While the C++ language requires fully prototyped functions, function prototypes should always be used in a C program because of their strong data typing and error reduction properties. Also, the use of function prototypes improves performance in reducing the additional code used in the promotion and demotion of the data. The use of function prototypes can also expose latent bugs that might exist in a program and will significantly aid porting applications to 64-bit platforms.

Parameters behave as expressions, and are evaluated before being promoted. In the following case:

```
long testparm(long j)
{
      return j;
}

int main()

{
int i = -2;
unsigned k = 1U;
long n = testparm(i + k);
      …

}
```

On a 64-bit system, `testparm` returns `4294967295` because the expression `i + k` is an `unsigned` 32-bit expression, and when promoted to a `long`, the sign does not extend.

Additionally, many systems now use registers to pass parameters rather than the stack. While this should be transparent to most programs, there is one common programming trick that can cause incorrect results. We want the program to print the hexadecimal value of a float:

```
float f = 1.25;

printf(“The hex value of %f is %x\n”, f, f);
```

On a stack-based system, the appropriate hexadecimal value is printed, but in a register-based system, the hexadecimal value is read from an integer register, not the floating-point register. One solution is to use a pointer:

```
printf(“The hex value of %f is %x\n”, f, *(int *)&f);
```

In this case, the address of the floating-point variable `f` is cast to a pointer to an `int` which is then dereferenced.

`Double`s are generally 64-bits wide and comply with the IEEE-754 floating-point standard on both 32-bit and 64-bit systems.

Passing parameters by reference in C++

The C++ language also permits passing parameters by reference. The compiler will not permit a 64-bit variable to be passed by reference to a function unless the parameter is `const`, where the low order 4 bytes are passed into the function with the same effect as when passing parameters by value. The example below demonstrates this concept.

```
  int64_t var64;
  int noconst32(int &);
  int const32(const int &);
  int rv = noconst32(var64); // compiler will reject this
noconst.cc:13 error: invalid initialization of reference of type 'int&'
from expression of type 'int64_t'
  rv = const32(var64); // compiler will accept this.
```

Numeric constants

Integer constants are generally taken as signed 32-bit integers, such as `1234`. The suffix **L** is used to indicate a `long` constant, such as `1234L`. The suffix **U** is used to indicate an `unsigned` constant, and may be used either alone or with **L**. Hexadecimal constants are commonly used as masks or specific bit values. Hexadecimal constants without a suffix are defined as `unsigned int` if it will fit into 32 bits and if the high order bit is turned on.

On a 32-bit system, the constant that might be used to set all the bits in a value might be:

- `0xFFFFFFFF` This is a 32-bit `unsigned int`.
- `0xFFFFFFFFL` This is a `signed long` on a 64-bit system and an `unsigned long` on a 32-bit system. On a 32-bit system this sets all the bits, but on a 64-bit system only the low order 32-bits are set, resulting in the value `0x00000000FFFFFFFF`.
- `0x7FFFFFFF` This is a `signed int`.

If the developer wants to turn all the bits on, a portable way to do this is to define a `signed long` constant with a value of -1. This turns all the bits on since twos complement arithmetic is used.

```
long x = -1L;
```

Another common construct is the setting of the most significant bit. Typically, the constant `0x80000000` is used on a 32-bit system. A more portable method of doing this is to use a shift expression using the compile-time constant expression ((sizeof(long)*8) - 1). This expression will be 31 on a 32-bit system and 63 on a 64-bit system.

```
1L << ((sizeof(long)*8) - 1);
```

Since this is a constant expression, the compiler will fold this expression into the appropriate constant so that this will work on a 16-bit, 32-bit or 64-bit system.

C/C++ integer promotions

The standard C and C++ language integer promotion rules can cause some problems when porting existing code from 32-bits to 64-bits. Every integer type is assigned a rank. One key statement is "No two signed integer types shall have the same rank, even if they have the same representation".[6] Signed integer types have higher rank than signed integer types of lesser precision. Thus the ranking of signed integer types are from the highest to lowest rank: `long long int`, `long int`, `int`, `short int`, `signed char`. The rank of an unsigned integer type is equal to the rank of the corresponding signed integer type. When evaluating an expression consisting of two operands of different rank, the standard dictates that the operand of lesser rank be promoted to the type of the operand with greater rank. In the case of a signed type, when it is promoted, its sign bit is propagated. In the case of promoting an unsigned 32-bit value to a 64-bit value, the sign bit does not

[6]ANSI/ISO/IES-9899:1999 C International standard Section 6.3.1.1