Chapter 6

Operator Conversions

A number of operators can, depending on the types of their operands, cause an implicit conversion of some operands from one type to another. The following discussion explains the results you can expect from these conversions. The conversions demanded by most operators are summarized in "Arithmetic Conversions". As necessary, a discussion of the individual operators supplements the summary.

Conversions of Characters and Integers

You can use a character or a short integer wherever you can use an integer. Characters are unsigned by default. In all cases, the value is converted to an integer. Conversion of a shorter integer to a longer integer preserves the sign. Traditional C uses "unsigned preserving integer promotion" (unsigned **short** to unsigned **int**), while ANSI C uses "value preserving integer promotion" (unsigned **short** to **int**).

A longer integer is truncated on the left when converted to a shorter integer or to a **char**. Excess bits are simply discarded.

Conversions of Float and Double

Historically in C, expressions containing floating point operands (either **float** or **double**) were calculated using double precision. This is also true of calculations in traditional C, unless you've specified the compiler option **-float**. With the **-float** option, calculations involving floating point operands and no **double** or **long double** operands take place in single precision. The **-float** option has no effect on argument promotion rules at function calls or on function prototypes.

ANSI C performs calculations involving floating point in the same precision as if **-float** had been specified in traditional C, except when floating point constants are involved.

In traditional C, specifying the **-float** option coerces floating point constants into type **float** if all the other subexpressions are of type **float**. This is not the case in ANSI C. ANSI C considers all floating point constants to be implicitly double precision, and arithmetics involving such constants therefore take place in double precision. To force single precision arithmetic in ANSI C, use the *f* or *F* suffix on floating point constants. To force long double precision on constants, use the *l* or *L* suffix. For example, `3.14l` is long double precision, `3.14` is double precision, and `3.14f` is single precision in ANSI C.

For a complete discussion with examples, see "Type Promotion and Floating–Point Constants".

Conversion of Floating and Integral Types

Conversions between floating and integral values are machine dependent. Silicon Graphics uses IEEE floating point, in which the default rounding mode is to nearest, or in case of a tie, to even. Floating point rounding modes can be controlled using the facilities of *fpc*(3c). Floating point exception conditions are discussed in the introductory paragraph of Chapter 7, "Expressions and Operators."

When a floating value is converted to an integral value, the rounded value is preserved as long as it does not overflow. When an integral value is converted to a floating value, the value is preserved unless a value of more than six significant digits is being converted to single precision, or fifteen significant digits is being converted to double precision.

Conversion of Pointers and Integers

An expression of integral type can be added to or subtracted from an object pointer. In such a case, the integer expression is converted as specified in the discussion of the addition operator in "Additive Operators". Two pointers to objects of the same type can be subtracted. In this case, the result is converted to an integer as specified in the discussion of the subtraction operator, in "Additive Operators".

Conversion of Unsigned Integers

When an **unsigned** integer is converted to a longer **unsigned** or signed integer, the value of the result is preserved. Thus, the conversion amounts to padding with zeros on the left.

When an **unsigned** integer is converted to a shorter **signed** or **unsigned** integer, the value is truncated on the left. This truncation may produce a negative value, if the result is **signed**.

Arithmetic Conversions

Many types of operations in C require two operands to be converted to a common type. Two sets of conversion rules are applied to accomplish this conversion. The first, referred to as the *integral promotions,* defines how integral types are promoted to one of several integral types that are at least as large as **int**. The second, called the *usual arithmetic conversions*, derives a common type in which the operation is performed.

ANSI C and traditional C follow different sets of these rules.

Integral Promotions

The difference between the ANSI C and traditional versions of the conversion rules is that the traditional C rules emphasize preservation of the (*un*)*signedness* of a quantity, while ANSI C rules emphasize preservation of its *value.*

In traditional C, operands of types **char, unsigned char,** and **unsigned short** are converted to **unsigned int.** Operands of types **signed char** and **short** are converted to **int**.

ANSI C converts all **char** and **short** operands, whether signed or unsigned, to **int**. Only operands of type **unsigned int**, **unsigned long**, and **unsigned long long** may remain unsigned.

Usual Arithmetic Conversions

Besides differing in emphasis on signedness and value preservation, the usual arithmetic conversion rules of ANSI C and traditional C also differ in the precision of the chosen floating point type.

Below are two sets of conversion rules, one for traditional C, and the other for ANSI C. Each set is ordered in decreasing precedence. In any particular case, the rule that applies is the first whose conditions are met.

Each rule specifies a type, referred to as the *result type*. Once a rule has been chosen, each operand is converted to the result type, the operation is performed in that type, and the result is of that type.

Traditional C Conversion Rules

The traditional C conversion rules are:

- If any operand is of type **double**, the result type is **double**.
- If an operand is of type **float**, the result type is **float** if you have specified the **-float** switch. Otherwise, the result type is double.
- The integral promotions are performed on each operand:
  - If one of the operands is of type **unsigned long long**, the result is of type **unsigned long long**
  - If one of the operands is of type **long long**, the result is of type **long long**
  - If one of the operands is of type **unsigned long**, the result is of type **unsigned long**
  - If one of the operands is of type **long**, the result is of type **long**
  - If one of the operands is of type **unsigned int,** the result type is **unsigned int**
  - Otherwise, the result is of type **int**

ANSI C Conversion Rules

The ANSI C rules are as follows:

- If any operand is of type **long double**, the result type is **long double**.
- If any operand is of type **double**, the result type is **double**.
- If an operand is of type **float**, the result type is **float**.
- The integral promotions are performed on each operand:
  - If one of the operands is of type **unsigned long long**, the result is of type **unsigned long long**
  - If one of the operands is of type **long long**, the result is of type **long long**
  - If one of the operands is of type **unsigned long**, the result is of type **unsigned long**
  - If one of the operands is of type **long**, the result is of type **long**
  - If one of the operands is of type **unsigned int,** the result type is **unsigned int**
  - Otherwise the result is of type **int**

Conversion of Other Operands

The following three sections discuss conversion of *lvalue*s, function designators, **void** objects, and pointers.

Conversion of *lvalue*s and Function Designators

Except as noted, if an *lvalue* that has type *array of <type>* appears as an operand, it is converted to an expression of the type *pointer to <type>*. The resultant pointer points to the initial element of the array. In this case, the resultant pointer ceases to be an *lvalue.* (For a discussion of *lvalues*, see "Objects and lvalues".)

A *function designator* is an expression that has function type. Except as noted, a function designator appearing as an operand is converted to an expression of type *pointer to function.*

Conversion of Void Objects

The (nonexistent) value of a **void** object cannot be used in any way, and neither explicit nor implicit conversion can be applied. Because a **void** expression denotes a nonexistent value, such an expression can be used only as an expression statement (see "Expression Statement"), or as the left operand of a comma expression (see "Comma Operator").

An expression can be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

Conversion of Pointers

A pointer to **void** can be converted to a pointer to any object type and back without change in the underlying value.

The NULL pointer constant can be specified either as the integral value zero, or the value zero cast to a *pointer to* **void**. If a NULL pointer constant is assigned or compared to a pointer to any type, it is appropriately converted.