Chapter 7

Expressions and Operators

The precedence of expression operators is indicated by their syntax in this chapter; it usually follows the order of the major subsections, with earlier subsections having higher precedence. For example, since the multiplication operator * can have a *unary−expression* (which is a cast−expression) as well as an operand, the order of evaluation of the expression

```
~ i * z
```

gives ~ higher precedence than * and can be written

```
( ~ i ) * z
```

The text indicates this precedence by placing *unary−expressions* in "Unary Operators", and *multiplicative−expressions* in "Multiplicative Operators". This syntax-subsection correlation is violated in a few cases. For example, *cast−expressions* can be operands in *unary−expressions*, in which case the *cast−expression* has higher precedence. See "Cast Operators" and "Unary Operators" for more information.

Within each subsection, the operators have the same precedence. All operators group left to right, unless otherwise indicated in their discussion. Table 7−1 shows operators and indicates the priority ranking and grouping of each.

Table 7−1 Operator Precedence and Associativity

Operator (from high to low priority)	Grouping
`() [] -> .`	L−R
`! ~ ++ -- -` (type) `* & sizeof` (all unary)	R−L
`* / %`	L−R
`+ -`	L−R
`<< >>`	L−R
`< <= > >=`	L−R
`== !=`	L−R
`&`	L−R
`^`	L−R
`|`	L−R
`&&`	L−R
`||`	L−R
`? :`	L−R
`= += -= *= /= %= ^= &= |=`	R−L
`,`	L−R

The order of evaluation of expressions, as well as the order in which side−effects take place, is unspecified, except as indicated by the syntax, or specified explicitly in this chapter. The compiler can arbitrarily rearrange expressions involving a commutative and associative operator (***, +, &,** |, ^).

Integer divide−by−zero results in a trap. Other integer exception conditions are ignored. Silicon Graphics floating point conforms to the IEEE standard. Floating point exceptions are ignored by default, yielding the default IEEE results of infinity for divide−by−zero and overflow, not−a−number for invalid operations, and zero for underflow. You can gain control over these exceptions and their results most easily by using the Silicon Graphics IEEE floating point exception handler package (see handle_sigfpes(3c)). You can also control these exceptions by implementing your own handler and appropriately initializing the floating point unit (see fpc(3c)).

Primary Expressions

An identifier is a *primary−expression*, provided it has been declared as referring to an object, in which case it is an *lvalue*; or a function, in which case it is a function designator. *Lvalue*s and function designators are discussed in "Conversion of lvalues and Function Designators".

primary−expression:
    identifier
    constant
    string literal
    (expression)

A *constant* is a *primary−expression*. Its type is determined by its form and value, as described in "Constants".

A *string literal* is a *primary−expression*. Its type is *array of* **char**, subject to modification, as described in "Conversions of Characters and Integers".

A parenthesized *expression* is a *primary−expression* whose type and value are identical to those of the unparenthesized expression. The presence of parentheses does not affect whether the expression is an *lvalue*, *rvalue*, or function designator. For information on expressions, see "Constant Expressions".

Postfix Expressions

Postfix expressions involving ., ->, subscripting, and function calls group left to right.

postfix−expression:
    primary−expression
    postfix−expression [expression]
    postfix−expression (argument−expression−list$_{opt}$)
    postfix−expression. *identifier*
    postfix−expression -> *identifier*
    postfix−expression ++
    postfix−expression - -

argument−expression−list:
    argument−expression
    argument−expression−list, argument−expression

Subscripts

A *postfix−expression* followed by an expression in square brackets is a subscript. Usually, the

A *postfix−expression* followed by an expression in square brackets is a subscript. Usually, the postfix−expression has type *pointer to <type>*, the expression within the square brackets has type **int**, and the type of the result is <*type*>. However, it is equally valid if the types of the *postfix−expression* and the *expression* in brackets are reversed. This is because the expression postfix

E1[E2]

is identical (by definition) to

***((E1)+(E2))**

Since **+** is commutative, **E1** and **E2** can be interchanged.

You can find further information on this notation in the discussions on identifiers, and in the discussion of the operators * (in "Unary Operators") and + (in "Additive Operators").

Function Calls

The syntax of *postfix−expressions* that are function calls is

postfix−expression (argument−expression−list$_{opt}$)
argument−expression−list:
argument−expression
argument−expression−list, argument−expression

A *postfix-expression* followed by parentheses containing a possibly empty, comma−separated list of expressions (which constitute the actual arguments to the function) denotes a function call. The *postfix-expression* must be of type *function returning <type>*, and the result of the function call is of type <*type*>, and is not an *lvalue*. If the *postfix-expression* denoting the called function consists solely of a previously unseen identifier *foo*, the call produces an implicit declaration as if, in the innermost block containing the call, the declaration had appeared:

```
extern int foo();
```

If a corresponding function prototype that specifies a type for the argument being evaluated is in force, an attempt is made to convert the argument to that type. If the number of arguments does not agree with the number of parameters specified in the prototype, the behavior is undefined. If the type returned by the function as specified in the prototype does not agree with the type derived from the *postfix−expression* denoting the called function, the behavior is undefined. Such a scenario may occur for an external function declared with conflicting prototypes in different files. If no corresponding prototype is in scope or the argument is in the variable argument section of a prototype that ends in ellipses (…), the argument is converted according to the following default argument promotions:

- Type float is converted to double.
- Array and function names are converted to corresponding pointers.
- When using traditional C:
  - types **unsigned short** and **unsigned char** are converted to **unsigned int**.
  - types **signed short** and **signed char** are converted to **signed int**.

- When using ANSI C:
  - types **short** and **char**, whether **signed** or **unsigned**, are converted to **int**.

In preparing for the call to a function, a copy is made of each actual argument. Thus, all argument passing in C is strictly by value. A function can change the values of its parameters, but these changes cannot affect the values of the actual arguments. It is possible to pass a pointer on the understanding that the function can change the value of the object to which the pointer points. (Arguments that are array names can be changed as well, since these arguments are converted to pointer expressions.) Since the order of evaluation of arguments is unspecified, side effects may be delayed until the next sequence point, which occurs at the point of the actual call—after all arguments have been evaluated. (For example, the incrementation of *foo*, which is a side−effect of the evaluation of an argument *foo++*, may be delayed.) Recursive calls to any function are permitted.

Silicon Graphics recommends consistent use of prototypes for function declarations and definitions, as it is extremely dangerous to mix prototyped and nonprototyped function declarations/definitions. Never call functions before you declare them (although the language allows this). It results in an implicit nonprototyped declaration that may be incompatible with the function definition.

Structure and Union References

A *postfix−expression* followed by a dot followed by an identifier denotes a structure or union reference.

postfix−expression . identifier

The *postfix−expression* must be a structure or a union, and the *identifier* must name a member of the structure or union. The value is the named member of the structure or union, and it is an *lvalue* if the first expression is an *lvalue*. The result has the type of the indicated member and the qualifiers of the structure or union.

Indirect Structure and Union References

A postfix−expression followed by an arrow (built from **-** and **>**) followed by an *identifier* is an indirect structure or union reference.

postfix−expression **->** *identifier*

The *postfix−expression* must be a pointer to a structure or a union, and the *identifier* must name a member of that structure or union. The result is an *lvalue* referring to the named member of the structure or union to which the *postfix−expression* points. The result has the type of the selected member, and the qualifiers of the structure or union to which the *postfix−expression* points. Thus the expression

E1->MOS

is the same as

(*E1).MOS

Structures and unions are discussed in "Structure and Union Declarations".

Postfix ++ and - -

The syntax of **postfix ++** and **postfix −−** is:

postfix−expression **++**

postfix−expression **- -**

When postfix **++** is applied to a modifiable *lvalue*, the result is the value of the object referred to by the *lvalue.* After the result is noted, the object is incremented as if the constant 1 (one) were added to it. See the discussions in "Additive Operators" and "Assignment Operators" for information on conversions. The type of the result is the same as the type of the *lvalue* expression. The result is not an *lvalue.*

When postfix **- -** is applied to a modifiable *lvalue*, the result is the value of the object referred to by the *lvalue.* After the result is noted, the object is decremented as if the constant 1 (one) were subtracted from it. See the discussions in "Additive Operators" and "Assignment Operators" for information on conversions. The type of the result is the same as the type of the *lvalue* expression. The result is not an *lvalue.*

For both postfix **++** and **- -** operators, updating the stored value of the operand may be delayed until the next sequence point.

Unary Operators

Expressions with unary operators group from right to left.

unary−expression:
  postfix−expression
  ++ *unary−expression*
  - - *unary−expression*
  unary−operator cast−expression
  sizeof unary−expression
  sizeof (type−name)

unary−operator: one of
  ***** *&* **-** *!* **~ +**

Except as noted, the operand of a unary−operator must have arithmetic type.

Address−of and Indirection Operators

The unary ***** operator means "indirection"; the *cast−expression* must be a pointer, and the result is either an *lvalue* referring to the object to which the expression points, or a function designator. If the type of the expression is pointer to <type>, the type of the result is <type>.

The operand of the unary & operator can be either a function designator or an *lvalue* that designates an object. If it is an *lvalue*, the object it designates cannot be a bitfield, and it cannot be declared with the storage−class register. The result of the unary **&** operator is a pointer to the object or function referred to by the *lvalue* or function designator. If the type of the *lvalue* is <*type*>, the type of the result is pointer to <type>.

Unary + and - Operators

The result of the unary **-** operator is the negative of its operand. The integral promotions are performed on the operand, and the result has the promoted type and the value of the negative of the operand. Negation of unsigned quantities is analogous to subtracting the value from 2^n, where *n* is the number of bits in the promoted type.

The unary **+** operator exists only in ANSI C. The integral promotions are used to convert the operand. The result has the promoted type and the value of the operand.

Unary *!* and ~ Operators

The result of the logical negation operator **!** is 1 if the value of its operand is zero, and 0 if the value of its operand is nonzero. The type of the result is **int**. The logical negation operator is applicable to any arithmetic type and to pointers.

The **~** operator yields the one’s complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

Prefix ++ and *- -* Operators

The prefix operators **++** and **- -** increment and decrement their operands.

++ unary−expression

- - unary−expression

The object referred to by the modifiable *lvalue* operand of prefix **++** is incremented. The value is the new value of the operand but is not an *lvalue*. The expression **++x** is equivalent to **x += 1**. See the discussions in "Additive Operators" and "Assignment Operators" for information on conversions.

The prefix **- -** decrements its *lvalue* operand in the same manner as prefix **++** increments it.

The *sizeof* Unary Operator

The **sizeof** operator yields the size in bytes of its operand. The size of a **char** is 1 (one). Its major use is in communication with routines like storage allocators and I/O systems.

sizeof unary−expression

sizeof (type−name)

The operand of **sizeof** can not have function or incomplete type, or be an *lvalue* that denotes a bitfield. It can be an object or a parenthesized type name. In traditional C, the type of the result is **unsigned**. In ANSI C, the type of the result is **size_t**, which is defined in <*stddef.h*> as **unsigned int** (in 32−bit mode) or as **unsigned long** (in 64−bit mode). The result is a constant and can be used anywhere a constant is required.

When applied to an array, **sizeof** returns the total number of bytes in the array. The size is determined from the declaration of the object in the *unary−expression*. The **sizeof** operator can also be applied to a parenthesized type−name. In that case it yields the size in bytes of an object of the indicated type.

When **sizeof** is applied to an aggregate, the result includes space used for padding, if any.

Cast Operators

A cast−expression preceded by a parenthesized type−name causes of the value the expression to convert to the indicated type. This construction is called a cast. Type names are discussed in "Type Names".

cast−expression:
    unary−expression
    (type−name) cast−expression

The type−name specifies scalar type or void, and the operand has scalar type. Since a cast does not yield an *lvalue*, the effect of qualifiers attached to the type name is inconsequential.

When an arithmetic value is cast to a pointer, and vice versa, the appropriate number of bits are simply copied unchanged from one type of value to the other. Be aware of the possible truncation of pointer values in 64−bit mode compilation, when a pointer value is converted to an (unsigned)**int**.

Multiplicative Operators

The multiplicative operators *****, **/**, and **%** group from left to right. The usual arithmetic conversions are performed.

multiplicative expression:
    cast−expression
    multiplicative−expression ***** *cast−expression*
    multiplicative−expression **/** *cast−expression*
    multiplicative−expression % cast−expression

Operands of ***** and **/** must have arithmetic type. Operands of % must have integral type.

The binary ***** operator indicates multiplication, and its result is the product of the operands.

The binary **/** operator indicates division of the first operator (dividend) by the second (divisor). If the operands are integral and the value of the divisor is 0, SIGTRAP is signalled. Integral division results in the integer quotient whose magnitude is less than or equal to that of the true quotient, and with the same sign.

The binary % operator yields the remainder from the division of the first expression (dividend) by the second (divisor).   The operands must be integral. The remainder has the same sign as the dividend, so that the equality is true when the divisor is nonzero:

```
(dividend / divisor) * divisor + dividend % divisor == dividend
```

If the value of the divisor is 0, SIGTRAP is signalled.

Additive Operators

The additive operators **+** and **-** group from left to right. The usual arithmetic conversions are performed.

additive−expression:
    multiplicative−expression

additive−expression **+** *multiplicative−expression*
additive−expression **-** *multiplicative−expression*

In addition to arithmetic types, the following type combinations are acceptable for additive−expressions:

- For addition, one operand is a pointer to an object type and the other operand is an integral type.
- For subtraction:
  - Both operands are pointers to qualified or unqualified versions of compatible object types.
  - The left operand is a pointer to an object type, and the right operand has integral type.

The result of the **+** operator is the sum of the operands. The result of the
- operator is the difference of the operands. When an operand of integral type is added to or subtracted from a pointer to an object type, the integral operand is first converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer.

Suppose *a* has type *array of <object>*, and *p* has type *pointer to <object>* and points to the initial element of *a*. Then the result of *p n*, where *n* is an integral operand, is the same as *&a [\xb1 n].*

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an integral quantity representing the number of objects separating them. Unless the pointers point to objects in the same array, the result is undefined. The actual type of the result is **int** in traditional C, and **ptrdiff_t** (defined in <*stddef.h*> as **int** in 32−bit mode and as **long** in 64−bit mode) in ANSI C.

Shift Operators

The shift operators **<<** and **>>** group from left to right. Each operand must be of an integral type. The integral promotions are performed on each operand. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative or greater than or equal to the length in bits of the promoted left operand.

shift−expression:
additive−expression
shift−expression << additive−expression
shift−expression >> additive−expression

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left−shifted **E2** bits. Vacated bits are filled with zeros.

The value of **E1>>E2** is **E1** right−shifted **E2** bit positions. Vacated bits are filled with zeros if E1 is unsigned, or if it’s signed and its value is nonnegative. If **E1** is signed and its value is negative, vacated bits are filled with ones.

Relational Operators

The relational operators group from left to right.

relational−expression:
  shift−expression
  relational−expression < *shift−expression*
  relational−expression > *shift−expression*
  relational−expression <= *shift−expression*
  relational−expression >= *shift−expression*

The operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) all yield a result of type **int** with the value 0 if the specified relation is false and 1 if it is true.

The operands must be one of the following:

- both arithmetic, in which case the usual arithmetic conversions are performed on them
- both pointers to qualified or unqualified versions of compatible object types
- both pointers to qualified or unqualified versions of compatible incomplete types

When two pointers are compared, the result depends on the relative locations in the address space of the pointed−to objects. Pointer comparison is portable only when the pointers point to objects in the same aggregate. In particular, no correlation is guaranteed between the order in which objects are declared and their resulting addresses.

Equality Operators

The **==** (equal to) and the **!=** (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value.)

equality−expression:
  relational−expression
  equality−expression == relational−expression
  equality−expression != relational−expression

The operands must be one of the following:

- both arithmetic, in which case the usual arithmetic conversions are performed on them
- both pointers to qualified or unqualified versions of compatible types
- a pointer to an object or incomplete type, and a pointer to qualified or unqualified void type
- a pointer and a null pointer constant

The semantics detailed in "Relational Operators" apply if the operands have types suitable for those operators. Combinations of other operands have the behavior detailed below:

- Two null pointers to object or incomplete types are equal. If two pointers to such types are equal, they either are null, point to the same object, or point to one object beyond the end of an array of such objects.

- Two pointers to the same function are equal, as are two null function pointers. Two function pointers that are equal are either both null or both point to the same function.

Bitwise *AND* Operator

Each operand must have integral type. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands, that is, each bit in the result is 0 unless the corresponding bit in each of the two operands is 1.

AND−expression:
    equality−expression
    AND−expression **&** *equality−expression*

Bitwise Exclusive *OR* Operator

Each operand must have integral type. The usual arithmetic conversions are performed. The result has type int, **long**, or **long long**, and the value is the bitwise exclusive OR function of the operands. That is, each bit in the result is 0 unless the corresponding bit in one of the operands is 1, and the corresponding bit in the other operand is 0.

exclusive−OR−expression:
    AND−expression
    exclusive−OR−expression ^ *AND−expression*

Bitwise Inclusive *OR* Operator

Each operand must have integral type. The usual arithmetic conversions are performed.

inclusive−OR−expression:
    exclusive−OR−expression
    inclusive−OR−expression | *exclusive−OR−expression*

The result has type int, **long**, or **long long**, and the value is the bitwise inclusive OR function of the operands. That is, each bit in the result is 0 unless the corresponding bit in at least one of the operands is 1.

Logical *AND* Operator

The && operator groups left to right.

logical−AND−expression:
    inclusive−OR−expression
    logical−AND−expression && inclusive−OR−expression

Each of the operands must have scalar type. The result has type int and value 1 if neither of its operands evaluates to 0. Otherwise it has value 0.

Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to zero. There is a sequence point after the evaluation of the first operand.

Logical *OR* Operator

The || operator groups left to right.

logical−OR−expression:
 logical−AND−expression
 logical−OR−expression || *logical−AND−expression*

Each of the operands must have scalar type. The result has type int and value 1 if either of its operands evaluates to one. Otherwise it has value 0.

Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated unless the first operand evaluates to zero. A sequence point occurs after the evaluation of the first operand.

Conditional Operator

Conditional expressions group from right to left.

conditional−expression:
 logical−OR−expression
 logical−OR−expression **?** *expression* **:** *conditional−expression*

The type of the first operand must be scalar. Only certain combinations of types are allowed for the second and third operands. These combinations are listed below, along with the type of result the combination yields.

- Both can be arithmetic types. In this case, the usual arithmetic conversions are performed on them to derive a common type, which is the type of the result.
- Both are compatible structure or union objects. The result has that type.
- Both are void. The type of the result is void.
- One is a pointer, and the other a null pointer constant. The type of the result is the type of the nonconstant pointer.
- One is a pointer to void, and the other is a pointer to an object or incomplete type. The second operand is converted to a pointer to void, and this is the type of the result.
- Both are pointers to qualified or unqualified versions of compatible types. The result has a type compatible with each, qualified with all the qualifiers of the types pointed to by both operands.

Evaluation of the conditional operator proceeds as follows. The first expression is evaluated, after which a sequence point occurs. If the value of the first expression is nonzero, the result is the value of the second operand; otherwise it is that of the third operand. Only one of the second and third operands is evaluated.

Assignment Operators

All assignment operators group from right to left.

assignment−expression:
    conditional−expression
    unary−expression assignment−operator assignment−expression

assignment operator: one of
    = *= /= %= += −= <<= >>= &= ^= |=

Assignment operators require a modifiable *lvalue* as their left operand. The type of an assignment expression is that of its unqualified left operand. The result is not an *lvalue.* Its value is the value stored in the left operand after the assignment, but the actual update of the stored value may be delayed until the next sequence point.

The order of evaluation of the operands is unspecified.

Assignment Using = (Simple Assignment)

The operands permissible in simple assignment must obey one of the following:

- Both have arithmetic type or are compatible structure or union types.
- Both are pointers, and the type pointed to by the left has all of the qualifiers of the type pointed to by the right.
- One is a pointer to an object or incomplete type, and the other is a pointer to void. The type pointed to by the left must have all of the qualifiers of the type pointed to by the right.
- The left operand is a pointer, and the right is a null pointer constant.

In simple assignment, the value of the right operand is converted to the type of the assignment expression and replaces the value of the object referred to by the left operand. If the value being stored is accessed by another object that overlaps it, the behavior is undefined unless the overlap is exact and the types of the two objects are compatible.

Compound Assignment

For the operators **+=** and **−=**, either both have arithmetic types, or the left operand is a pointer and the right is an operand integral. In the latter case, the right operand is converted as explained in"Additive Operators". For all other operators, each operand must have arithmetic type consistent with those allowed for the corresponding binary operator.

The expression **E1 op = E2** is equivalent to the expression **E1 = E1 op E2**, except that in the former, **E1** is evaluated only once.

Comma Operator

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded.

expression:
    assignment−expression

expression, *assignment−expression*

The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, the comma operator as described in this section can appear only in parentheses. Two such contexts are lists of actual arguments to functions (described in "Primary Expressions") and lists of initializers (see "Initialization"). For example, the following code has three arguments, the second of which has the value 5.

```
f(a, (t=3, t+2), c)
```

Constant Expressions

A constant expression can be used any place a constant can be used.

constant−expression:
  conditional−expression

It cannot contain assignment, increment, decrement, function−call, or comma operators. It must evaluate to a constant that is in the range of representable values for its type. Otherwise, the semantic rules for the evaluation of nonconstant expressions apply.

Constant expressions are separated into three classes:

- An integral constant expression has integral type and is restricted to operands that are integral constants, **sizeof** expressions, and floating constants that are the immediate operands of integral casts.
- An arithmetic constant expression has arithmetic type and is restricted to operands that are arithmetic constants, and **sizeof** expressions. Cast expressions in arithmetic constant expressions can convert only between arithmetic types.
- An address constant is a pointer to an *lvalue* designating an object of static storage duration, or a pointer to a function designator. It can be created explicitly or implicitly, as long as no attempt is made to access an object value.

Either address or arithmetic constant expressions can be used in initializers. In addition, initializers can contain null pointer constants and address constants (for object types), and plus or minus integral constant expressions.