**Expressions**

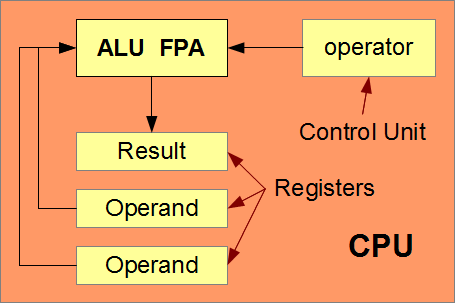
Programming languages support operators that combine variables and constants into expressions for transforming existing data into new data.  These operators take one or more operands.  The operands may be variables, constants or other expressions.  The C language supports a comprehensive set of these operators for arithmetic, relational and logical expressions.

This chapter describes the supported operators in detail, what happens when the operands are of different types, how to change the type of an operand and the order of evaluation of sub-expressions within expressions.  The introduction to this detailed description is a brief overview of the hardware components that evaluate expressions.  These are the ALU and FPA inside the CPU.

Evaluating Expressions

The ALU evaluates the simplest of instructions on integer values: for instance, additions where the operands are of the same type.  The FPA evaluates the simplest of instructions on floating-point values.  C compilers simplify C language expressions into sets of hardware instructions that either the ALU or the FPA can process.

The ALU receives the expression's operator from the Control Unit, applies that operator to integer values stored in the CPU's registers and places the result in one of the CPU's registers.  The FPA does the same but for floating-point values.



The expressions that the ALU can process on integer types are:

* arithmetic
* relational
* logical

The FPA can processes these same kinds of expressions on floating-point types.

Arithmetic Expressions

Arithmetic expressions consist of

* integral operands - destined for processing by the ALU
* floating-point operands - destined for processing by the FPA

Integral Operands

The C language supports 5 binary and 2 unary arithmetic operations on integral (int and char) operands.  Here, the term *binary* refers to *two* operands; *unary* refers to one operand.

Binary Operations

The binary arithmetic operations on integers are addition, subtraction, multiplication, division and remaindering.  Expressions take one of the forms listed below

|  |  |  |  |
| --- | --- | --- | --- |
| **Arithmetic Expression** | | | **Meaning** |
| operand | + | operand | add the operands |
| operand | - | operand | subtract the right from the left operand |
| operand | \* | operand | multiply the operands |
| operand | / | operand | divide the left by the right operand |
| operand | % | operand | remainder of the division of left by right |

Division of one integer by another yields a whole number.  If the division is not exact, the operation discards the remainder.  The expression evaluates to the truncated integer result; that is, the whole number without any remainder.  The expression with the modulus operator (%) evaluates to the remainder alone.

For example,

|  |
| --- |
| 34 / 10 // evaluates to 3 (3 groups of 10 people)  34 % 10 // evaluates to 4 (4 person left without a group) |

Unary Operations

The unary arithmetic operations are identity and negation.  Expressions take one of the forms listed below

|  |  |  |
| --- | --- | --- |
| **Arithmetic Expression** | | **Meaning** |
| + | operand | evaluates to the operand |
| - | operand | changes the sign of the operand |

The plus operator leaves the value unchanged and is present for language symmetry.

Floating-Point Operands

The C language supports 4 binary and 2 unary arithmetic operations on the floating-point (float and double) operands.

Binary

The binary arithmetic operations on floating-point values are addition, subtraction, multiplication and division.  Expressions take one of the forms listed below

|  |  |  |  |
| --- | --- | --- | --- |
| **Arithmetic Expression** | | | **Meaning** |
| operand | + | operand | add the operands |
| operand | - | operand | subtract the right from the left operand |
| operand | \* | operand | multiply the operands |
| operand | / | operand | divide the left by the right operand |

The division operator (/) evaluates to a floating-point result.  There is no remainder operator for floating-point operands.

Unary

The unary operations are identity and negation.  Expressions take the form listed below

|  |  |  |
| --- | --- | --- |
| **Arithmetic Expression** | | **Meaning** |
| + | operand | evaluates to the operand |
| - | operand | change the sign of the operand |

The plus operator leaves the value unchanged and is present for language symmetry.

Limits

The result of any operation is an expression of related type.  Arithmetic operations can produce values that are outside the range of the expression's type.  Consider the following program, which multiplies two ints and then two doubles

|  |
| --- |
| // Limits on Arithmetic Expressions  // limits.c  int main(void)  {  int i, j, ij;  double x, y, xy;  printf("Enter an integer : ");  scanf("%d", &i);  printf("Enter an integer : ");  scanf("%d", &j);  printf("Enter a floating-point number : ");  scanf("%lf", &x);  printf("Enter a floating-point number : ");  scanf("%lf", &y);  ij = i \* j;  xy = x \* y;  printf("%d \* %d = %d\n", i, j, ij);  printf("%le \* %le = %le\n", x, y, xy);  return 0;  } |

Compile this program and execute it inputting different values.  Try some very small numbers.  Try some very large numbers.  When does this program give incorrect results?  When it does, explain why?

Relational Expressions

The C language supports 6 relational operations.  A relational expression evaluates a condition.  It compares two values and yields 1 if the condition is true and 0 if the conditon is false.  The value of a relational expression is of type int.  Relational expressions take one of the forms listed below

|  |  |  |  |
| --- | --- | --- | --- |
| **Relational Expression** | | | **Meaning** |
| operand | == | operand | operands are equal in value |
| operand | > | operand | left is greater than the right |
| operand | >= | operand | left is greater than or equal to the right |
| operand | < | operand | left is less than the right |
| operand | <= | operand | left is less than or equal to the right |
| operand | != | operand | left is not equal to the right |

The operands may be integral types or floating-point types.

Example

The following program, accepts two ints and outputs 1 if they are equal; 0 otherwise

|  |
| --- |
| // Relational Expressions  // relational.c  int main(void)  {  int i, j, k;  printf("Enter an integer : ");  scanf("%d", &i);  printf("Enter an integer : ");  scanf("%d", &j);  k = i == j; // compare i to j and assign result to k  printf("%d == %d yields %d\n", i, j, k);  return 0;  } |

The first conversion specifier in the format string of the last printf() corresponds to i, the second corresponds to j and the third corresponds to k.

Logical Expressions

The C langauge does not have reserved words for true or false.  It interprets the value 0 as false and any other value as true.  C supports 3 logical operators.  Logical expressions yield 1 if the result is true and 0 if the result is false.  The value of a logical expression is of type int.  Logical expressions take one of the forms listed below

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | | | **Meaning** |
| operand | && | operand | both operands are true |
| operand | || | operand | one of the operands is true |
|  | ! | operand | the operand is not true |

The operands may be integral types or floating-point types.

Example

The following program, accepts three ints and outputs 1 if the second is greater than or equal to the first and less than or equal to the third; 0 otherwise:

|  |
| --- |
| // Logical Expressions  // logical.c  int main(void)  {  int i, j, k, m;  printf("Enter an integer : ");  scanf("%d", &i);  printf("Enter an integer : ");  scanf("%d", &j);  printf("Enter an integer : ");  scanf("%d", &k);  m = j >= i && j <= k; //store the value of this expression in m  printf("%d >= %d and %d <= %d yields %d\n", j, i, j, k, m);  return 0;  } |

The conversion specifiers in the last printf() correspond to the arguments in the same order (first to j, second to i, etc.).

Shorthand Assignments

The C language also supports shorthand operators that combine an arithmetic expression with an assignment expression.  These operators store the result of the arithmetic expression in the left operand.

Integral Operands

C has 5 binary and 2 unary shorthand assignment operators for integral (int and char) operands.

Binary Operands

The binary operators yield the same result as shown in the longhand expressions listed alongside:

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Shorthand** | **Longhand** | **Meaning** |
| operand += operand | i += 4 | i = i + 4 | add 4 to i and assign to i |
| operand -= operand | i -= 4 | i = i - 4 | subtract 4 from i and assign to i |
| operand \*= operand | i \*= 4 | i = i \* 4 | multiply i by 4 and assign to i |
| operand /= operand | i /= 4 | i = i / 4 | divide i by 4 and assign to i |
| operand %= operand | i %= 4 | i = i % 4 | remainder after i/4 and assign to i |

Unary Operands

The unary operators yield the same result as shown in the longhand expressions listed alongside:

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Shorthand** | **Longhand** | **Meaning** |
| ++operand | ++i | i = i + 1 | increment i by 1 |
| operand++ | i++ | i = i + 1 | increment i by 1 |
| --operand | --i | i = i - 1 | decrement i by 1 |
| operand-- | i-- | i = i - 1 | decrement i by 1 |

We call the unary operator that precedes its operand a *prefix* operator and the unary operator that succeeds its operand a *postfix* operator.

The difference between the prefix and postfix expressions is in the value of the expression itself.  The prefix operator changes the value of its operand and sets the expression's value to be the changed value.  The postfix operator sets the expression's value to the operand's original value and then changes the operand's value.  In other words, the prefix operator changes the value before using it, while the postfix operator changes the value after using it.

|  |  |
| --- | --- |
| // Prefix and Postfix Operators  // pre\_post.c  int main(void)  {  int age = 19;  printf("Prefix: %d\n", ++age);  printf(" %d\n", age);  printf("Postfix: %d\n", age++);  printf(" %d\n", ++age);  return 0;  } | Prefix: 20  20  Postfix: 20  22 |

Floating-Point Operands

C has 4 binary and 2 unary shorthand assignment operators for floating-point (float and double) operands.

Binary Operands

The binary operators yield the same result as in the longhand expressions listed alongside:

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Shorthand** | **Longhand** | **Meaning** |
| operand += operand | x += 4.1 | x = x + 4.1 | add 4.1 to x and assign to x |
| operand -= operand | x -= 4.1 | x = x - 4.1 | subtract 4.1 from x and assign to x |
| operand \*= operand | x \*= 4.1 | x = x \* 4.1 | multiply x by 4.1 and assign to x |
| operand /= operand | x /= 4.1 | x = x / 4.1 | divide x by 4.1 and assign to x |

Unary Operands

The unary operators yield the same result as in the longhand expressions listed alongside:

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Shorthand** | **Longhand** | **Meaning** |
| ++operand | ++x | x = x + 1 | increment x by 1.0 |
| operand++ | x++ | x = x + 1 | increment x by 1.0 |
| --operand | --x | x = x - 1 | decrement x by 1.0 |
| operand-- | x-- | x = x - 1 | decrement x by 1.0 |

The prefix and postfix operators operate on floating-point operands in the same way on integral operands.

Ambiguities

Compact use of shorthand operators can yield ambiguous results across different platforms.  Consider the following longhand statements

|  |
| --- |
| int i = 5;  int j = i++ + i; // \*\*\* AMBIGUOUS \*\*\* |

One compiler may increment the first i before the addition, while another compiler may increment i after the addition.  The C language does not address this ambiguity and only stipulates that the value must be incremented before the semi-colon.  To avoid ambiguity, we re-write this code to make our intent explicit

|  |  |
| --- | --- |
| int i = 5;  i++; // ++ before  int j = i + i; // j is 12 | int i = 5;  int j = i + i; // j is 10  i++; // ++ after |

Casting

The C language supports conversions from one type to another.  To convert the type of an operand, we precede the operand with the target type enclosed within parentheses.  We call such an expression a *cast*.  Casting expressions take one of the forms listed below

|  |  |  |
| --- | --- | --- |
| **Cast Expression** | | **Meaning** |
| (long double) | operand | long double version of operand |
| (double) | operand | double version of operand |
| (float) | operand | float version of operand |
| (long long) | operand | long long version of operand |
| (long) | operand | long version of operand |
| (int) | operand | int version of operand |
| (short) | operand | short version of operand |
| (char) | operand | char version of operand |

Consider the example below.  To obtain the number of hours in fractional form, we cast minutes to a float type and then divide it by 60.  The input and output are listed on the right

|  |  |
| --- | --- |
| // From minutes to hours  // cast.c  int main(void)  {  int minutes;  float hours;  printf("Minutes ? ");  scanf("%d", &minutes);  hours = (float)minutes / 60;  printf("= %.2lf hours\n", hours);  return 0;  } | //Minutes ? 45  //= 0.75 hours |

Without the type cast, the output for the same input would have been 0.00 hours.

Mixed-Type Expressions

Since CPUs process integral expressions and floating-point expressions differently (using the ALU and the FPA respectively), they only handle expressions with operands of the same type.  For expressions with operands of different types, we need rules for converting operands of one type to another type.

The C language using the following ranking:

|  |  |
| --- | --- |
| long double | higher |
| double | ... |
| float | ... |
| long long | ... |
| long | ... |
| int | ... |
| short | ... |
| char | lower |

There are two distinct kinds of expressions to consider with respect to type coercion:

* assignment expressions
* arithmetic and relational expressions

Assignment Expressions

Promotion

If the left operand in an assignment expression is of a higher type than the right operand, the compiler promotes the right operand to the type of the left operand.  For the example below, the compiler promotes the right operand (loonies) to a double before completing the assignment:

|  |  |
| --- | --- |
| // Promotion with Assignment Operators  // promotion.c  int main(void)  {  int loonies;  double cash;  printf("Loonies ? ");  scanf("%d", &loonies);  cash = loonies; // promotion  printf("Cash is $%.2lf\n", cash);  return 0;  } | //Loonies ? 23  //Cash is $23.00 |

Narrowing

If the left operand in an assignment expression is of a lower type than the right operand, the compiler truncates the right operand to the type of the left operand.  For the example below, the compiler truncates the type of the right operand (cash) to an int:

|  |  |
| --- | --- |
| // Truncation with Assignment Operators  // truncation.c  int main(void)  {  double cash;  int loonies;  printf("Cash ? ");  scanf("%lf", &cash);  loonies = cash; // truncation  printf("%d loonies.\n", loonies);  return 0;  } | //How much cash ? 23.45  //23 loonies. |

Arithmetic and Relational Expressions

C compilers promote the operand of lower type in an arithmetic or relational expression to an operand of the higher type before evaluating the expression.  The table below lists the type of the promoted operand.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Right Operand | | | | | | | |
| Left Operand | long double | double | float | long long | long | int | short | char |
| long double | long  double | long  double | long  double | long  double | long  double | long  double | long  double | long  double |
| double | long  double | double | double | double | double | double | double | double |
| float | long  double | double | float | float | float | float | float | float |
| long long | long  double | double | float | long  long | long  long | long  long | long  long | long  long |
| long | long  double | double | float | long  long | long | long | long | long |
| int | long  double | double | float | long  long | long | int | int | int |
| short | long  double | double | float | long  long | long | int | short | short |
| char | long  double | double | float | long  long | long | int | short | char |

For example,

|  |
| --- |
| 1034 \* 10 evaluates to 10340 // an int result  1034 \* 10.0 evaluates to 10340.0 // a double result  1034 \* 10L evaluates to 10340L // a long result  1034 \* 10.f evaluates to 10340.0f // a float result |

Compound Expressions

A compound expression is an expression that contains an expression as one of its operands.  C compilers evaluate compound expressions according to specific rules called *rules of precedence*.  These rules define the order of evaluation of expressions based on the operators involved.  C compilers evaluate the expression with the operator that has the highest precedence first.

The order of precedence, from highest to lowest, and the direction of evaluation are listed in the table below.

|  |  |
| --- | --- |
| **Operator** | **Evaluate From** |
| ++ -- (postfix) | left to right |
| ++ -- (prefix) + - & ! (all unary) | right to left |
| (type) | right to left |
| \* / % | left to right |
| + - | left to right |
| < <= > >= | left to right |
| == != | left to right |
| && | left to right |
| || | left to right |
| = += -= \*= /= %= | right to left |

To change the order of evaluation, we introduce parentheses.  C compilers evaluate the expressions within parentheses (( )) before applying the rules of precedence.  For example,

|  |
| --- |
| 2 + 3 \* 5 evaluates to 2 + 15, which evaluates to 17  ( 2 + 3 ) \* 5 evaluates to 5 \* 5, which evaluates to 25  3 / (double)2 evaluates to 3 / 2.0, which evaluates to 1.5  (double)(3 / 2) evaluates to (double)1, which evaluates to 1.0 |