Expressions

Expressions

In C, expressions are used to calculate some value. The simplest expressions are constants and variables. There is practically no limit to how complex an expression can be.

Examples:

```
i
5
3.1415
a + b
rate * time
x * (a + b / 7.0) - value / y
x*(a+b/7.0)-value/y
x   *(a+   b/     7.0)-    value    / y
sqrt(25.8) + b * abs(c)
```

- All expressions consist of *operators* and *operands*, just like formulas/equations from high-school algebra.
- The *type* of an expression is based on the type of its operands.
- If there are operands of different types (mixed mode), then some of the operands will be converted to the other type. (The conversion rules can be quite involved.)
- The are *many* different operators in C. (Some will be used more than others.)
- The basic operators that all languages have are:
  - Arithmetic operators - for calculating values (e.g. add, subtract, multiply, etc.)
  - Relational operators - for comparing values (e.g. greater than, less than, equal to, etc.)
  - Logical operators - for combining operators (e.g. greater than 7 *and* less than 12)
- Note that there is no semi-colon after the expressions above, as they are not C *statements*. Sometimes, as we'll see, we may have a semi-colon after an expression (an *expression statement*).
- There are *unary* operators and *binary* operators. Most are binary operators. There is even a *ternary* operator (3 operands).

Examples:

```
-i
+5
a + b
rate * time
```

Simple arithmetic unary operators:

Unary operator	**Meaning**
+	Positive (redundant)
-	Negation

Some simple arithmetic binary operators:

Binary operator	**Meaning**
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (Remainder)

With the exception of the modulus operator, all other operators above can work with *integral* (whole numbers) and *floating point* types. The modulus operator can only be used with integral types.

Precedence and Associativity

Just as mathematics, all operators have a certain *precedence*. Simply put, when more than one operator is used in an expression, precedence determines which one gets evaluated first.

```
3 + 4 * 2 is 11 and is the same as 3 + (4 * 2)

(3 + 4) * 2 is 14

-4 + 7 is 3 and is the same as (-4) + 7

-(4 + 7) is -11
```

When two or more operators with the same precedence are used in an expression, you must look at the operator's *associativity* to determine the order of evaluation.

```
3 + 4 + 2 is 9 and is the same as (3 + 4) + 2

3 * 4 * 2 is 24 and is the same as (3 * 4) * 2

2 * 6 / 4 is 3 and is the same as (2 * 6) / 4

2 * (6 / 4) is 2
```

This precedence chart shows that there are quite a few different levels of precedence within the C operators. Each division (separated by -----) is a different precedence.

Assignment Operators

The assignment operator is very common. There are *simple* assignments and *compound* assignments.

Simple assignment statements:

```
a = 1;
a = b;
a = 3 * b;
a = 4 - 3 * b / 8;
```

Note that the `=` operator is *assignment*, not *equality* (which is `==`, by the way).

Examples:

```
int i;    /* i holds an undefined value  */
double d; /* d holds an undefined value  */

i = 10;    /* i now holds the value 10   */
i = 12.8;  /* i now holds the value 12   */
d = 10;    /* d now holds the value 10.0 */
d = 12.8;  /* d now holds the value 12.8 */
```

The assignment operator is unique compared to the arithmetic operators we've seen so far:

- Most operators do not modify their operands.
- Assignment operators *modify* the left operand.
- The left operand must be able to represent a memory location.
- Any expression that can represent a memory location is considered an *l-value* (or lvalue or lval).

Example:

```
int a, b, c; /* all are undefined */

b = 5;       /* b is now 5  */
c = 10;      /* c is now 10 */

a = b + c;   /* a is now 15, b and c are unchanged */
a = b * c;   /* a is now 50, b and c are unchanged */
```

Because of the *associativity* of the assignment operator, we can do this:

```
a = b = c = 5; /* all are now 5 */
```

This is the same as this:

```
a = (b = (c = 5)); /* all are now 5 */
```

Note that this is very different (and is illegal):

```
((a = b) = c) = 5; /* This is not legal C code */
```

This is because the assignment operator requires an l-value, so we can store a value. These are illegal as well:

```
10 = 5;     /* Illegal */
10 = a;     /* Illegal */
a + b = 8; /* Illegal */
10 = 10;    /* Illegal */
```

Remember, this is *assignment* **not** *equality*.

Compound Assignment

Often, we'd like to add or subtract a value from a variable, and assign the new value back to the variable. This is completely legal (and sane):

```
  /* get the current value of a, add 5 to it, */
  /* and put the new value back into a        */
a = a + 5;

  /* get the current value of b, subtract 6 from it, */
  /* and put the new value back into b               */
b = b - 6;
```

These can be done more succinctly with *compound assignment operators* or *arithmetic assignment operators*:

```
  /* get the current value of a, add 5 to it, */
  /* and put the new value back into a        */
a += 5;

  /* get the current value of b, subtract 6 from it, */
  /* and put the new value back into b               */
b -= 6;
```

Note that **+=** and **-=** are single tokens. You cannot insert a space. There is also a ***=** operator and a **/=** operator. (There are several more, which we'll see later.)

Increment and Decrement Operators

Adding one or subtracting one from a variable is a very common occurrence. Because of this, there are a few operators that are dedicated to this.

Pre-increment	Post-increment	Pre-decrement	Post-decrement
`++i`	`i++`	`--i`	`i--`

These three assignment expressions are similar:

Assignment	Compound Assignment	Increment/ Decrement
`a = a + 1`	`a += 1`	`a++` `++a`
`a = a - 1`	`a -= 1`	`a--` `--a`

There is an important but subtle difference between the *prefix* and *postfix* versions of the increment/decrement operators which causes the above to be not quite true. In other words, the value of these *expressions* are the same:

```
a = a + 1        a += 1       ++a
```

Notice the missing `a++` expression. This means that if you displayed these expressions using `printf`, you'd see:

```
a = 5;
printf("value is %i\n", a = a + 1); /* value is 6 */

a = 5;
printf("value is %i\n", a += 1); /* value is 6 */

a = 5;
printf("value is %i\n", ++a); /* value is 6 */

a = 5;
printf("value is %i\n", a++); /* value is 5 */
```

However, as *statements*, these are all equivalent:

```
a = a + 1;
a += 1;
++a;
a++;
```

More examples: Assuming that `a` is an integer:

Statements	Output

`a = 5;`
`printf("The value of a is %i\n", ++a);`
`printf("The value of a is %i\n", a);`
	`The value of a is 6`
	`The value of a is 6`

`a = 5;`
`printf("The value of a is %i\n", a++);`
`printf("The value of a is %i\n", a);`
	`The value of a is 5`
	`The value of a is 6`

`a = 5;`
`printf("The value of a is %i\n", --a);`
`printf("The value of a is %i\n", a);`
	`The value of a is 4`
	`The value of a is 4`

`a = 5;`
`printf("The value of a is %i\n", a--);`
`printf("The value of a is %i\n", a);`
	`The value of a is 5`
	`The value of a is 4`

Looking closer, this statement:

```
c = a++ + ++b;
```

is equivalent to these statements:

```
b = b + 1;
c = a + b;
a = a + 1;
```

Look closely at the expressions below to determine the output:

Statements	Output

`a = 5;`
`b = 3;`
`c = a++ + b++;`
`printf("a = %i, b = %i, c = %i\n", a, b, c);`
	`a = 6, b = 4, c = 8`

`a = 5;`
`b = 3;`
`c = ++a + b++;`
`printf("a = %i, b = %i, c = %i\n", a, b, c);`	

`a = 5;`
`b = 3;`
`c = a++ + ++b;`
`printf("a = %i, b = %i, c = %i\n", a, b, c);`	

`a = 5;`
`b = 3;`
`c = ++a + ++b;`
`printf("a = %i, b = %i, c = %i\n", a, b, c);`	

The statement below modifies the values of `a, b, and c`:

```
c = a++ + ++b;
```

- Which one is modified first?
- Which one is modified last?

Remember, to *modify* a variable means to change the value that is stored at the memory location represented by the variable. Graphically, `a++` would look (simplified) something like this:

Fetch value from memory	Increment the value by 1	Store the new value in memory
□	□	□

Notice that there is a time when the old value and the new value both exist. This is key to understanding the increment/decrement operators.

Order of Evaluation

Example expressions:

```
int w = 1;
int x = 2;
int y = 3;
int z = 4;
int r;

r = w * x + y * z;      /* 1. same as: (w * x) + (y * z) */
r = w + x * y + z;      /* 2. same as: w + (x * y) + z    */
r = (w + x) * (y + z); /* 3. only way to write this      */
```

In the *compound expressions* above, there are actually several *subexpressions* in each. In other words, the assignment expression

```
r = w * x + y * z      /* 1. same as: (w * x) + (y * z) */
```

must perform each of these evaluations: (The registers used here are completely arbitrary.)

1. The value stored at **w** must be fetched from memory. (Put in register A)
2. The value stored at **x** must be fetched from memory. (Put in register B)
3. The value stored at **y** must be fetched from memory. (Put in register C)
4. The value stored at **z** must be fetched from memory. (Put in register D)
5. The value in register A must be multiplied by the value in register B. (Put result in register E)
6. The value in register C must be multiplied by the value in register D. (Put result in register F)
7. The value in register E is added to the value in register F. (Put result in register G)
8. The value in register G is stored in memory location **r**.

Interestingly, the order of some of these operations is *undefined*. For example, the first four instructions could actually be these:

1. The value stored at **z** is fetched from memory. (Put in register A)
2. The value stored at **w** is fetched from memory. (Put in register B)
3. The value stored at **x** is fetched from memory. (Put in register C)
4. The value stored at **y** is fetched from memory. (Put in register D)

The instructions could be ordered like this:

1. The value stored at **w** is fetched from memory. (Put in register A)
2. The value stored at **x** is fetched from memory. (Put in register B)
3. The value in register A is multiplied by the value in register B. (Put result in register C)
4. The value stored at **y** is fetched from memory. (Put in register A)
5. The value stored at **z** is fetched from memory. (Put in register B)
6. The value in register A is multiplied by the value in register B. (Put result in register D)
7. The value in register C is added to the value in register D. (Put result in register E)
8. The value in register E is stored in memory location **r**.

Or, the instructions could even be like this:

1. The value stored at **w** is fetched from memory. (Put in register A)
2. The value stored at **x** is fetched from memory. (Put in register B)
3. The value in register A is multiplied by the value in register B. (Put result in register C)
4. The value stored at **y** is fetched from memory. (Put in register A)
5. The value stored at **z** is fetched from memory. (Put in register B)
6. The value in register A is multiplied by the value in register B. (Put result in register A)
7. The value in register A is added to the value in register C. (Put result in register B)
8. The value in register B is stored in memory location **r**.

There are many other orderings that are equally valid. However, some subexpressions must be in a particular order. For example:

- **w** and **x** must be fetched before they can be multiplied.
- **y** and **z** must be fetched before they can be multiplied.
- **w** and **x** must be multiplied before adding to the product of **y** and **z**.
- All fetching and arithmetic operations must be done before storing the result into **r**.

So, even though the statement:

```
r = w * x + y * z;      /* 1. same as: (w * x) + (y * z) */
```

can be evaluated in a multitude of ways, the result will always be the same: 14

Side-effects in Expressions

Anytime an operator causes a value in memory to change, it is called a *side-effect* operator. The most obvious side-effect operator is the assignment operator:

```
e = a * b + c * d;  /* Changes the value stored at e */
```

However, this assignment statement is actually performing three assignments:

```
e = a++ * b++;  /* 3 modifications */
```

After the statement completely executes, **e**, **a**, and **b** will have different values.

This is problematic, though:

```
e = a++ * a;  /* dangerous code! */
```

Since **a** is used twice, it will be evaluated twice. And, depending on when the increment to **a** occurs, the result will be different:

```
a = 2;
e = a++ * a;  /* e is undefined (either 4 or 6)  */
```

In fact, the GNU gcc compiler will actually warn you about it:

```
warning: operation on 'a' may be undefined
```

Here's an example of undefined code:

```
#include <stdio.h>

int main(void)
{
  int a = 5;
  a = a-- - --a * (a = -3) * a++ + ++a;
  printf("a = %i\n", a);

  return 0;
}
```

Depending on which compiler you use, you may get different results:

Compiler	Output
GNU gcc	`a = 22`
Microsoft	`a = 4`
Borland	`a = 21`

Fortunately, the GNU compiler will warn you about this:

```
main.c: In function `main':
main.c:6: warning: operation on `a' may be undefined
main.c:6: warning: operation on `a' may be undefined
main.c:6: warning: operation on `a' may be undefined
main.c:6: warning: operation on `a' may be undefined
main.c:6: warning: operation on `a' may be undefined
```

Some side-effect operators:

```
=  +=  -=  *=  /=
++ (pre/post increment)
-- (pre/post decrement)
```

There are 4 side-effect operators in this expression:

```
a = b += c++ - d + --e / -f
```

What will be printed by this code? The first thing you should do is identify which variables are going to have their values changed. You'll definitely want to refer to the precedence chart.

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = 5;
int f = 6;
```

```
a = b += c++ - d + --e / -f;

printf("a = %i, b = %i, c = %i, d = %i, e = %i, f = %i\n", a, b, c, d, e, f);
```

To better understand the expression above, you should add parentheses to explicitly show the order of evaluation.

There are rules that dictate precedence and associativity, but there is still ambiguity: Example:

```
int x = a * b + c * d + e * f;
```

More complex example:

```
int PrintAndReturn(int value)
{
  printf("%i\n", value);
  return value;
}

int main(void)
{
  int x, y;

  x = PrintAndReturn(1) + PrintAndReturn(2) + PrintAndReturn(3);
  printf("x = %i\n", x);

  y = PrintAndReturn(1) + PrintAndReturn(2) * PrintAndReturn(3);
  printf("y = %i\n", y);

  return 0;
}
```

The order in which expressions are evalutated is more relevant when the expressions have side effects. All compilers will not generate the same output.

```
GNU                 MS                Borland
1                   1                 1
2                   2                 2
3                   3                 3
x = 6               x = 6             x = 6
1                   1                 2
2                   2                 3
3                   3                 1
y = 7               y = 7             y = 7
```