Variables, Operators and Expressions

Overview

- Variables
- Operators
- Expressions

2

Objectives

This section is devoted to one of the most important parts of programming in C. This section focuses on using and creating values and expressions. A C source file consists of operations and expressions, so therefore this is a very important section of the book.

The topics discussed in this unit

- Declarations
- Expressions
- Types of operators
- Arithmetic operators (e.g. +, -, *).
- Bit manipulation operators (e.g. &, |, ^).
- Logical operators (AND, OR, XOR).

Variables

- Variables are used to work with values
- Variables consist of
  - type
  - name
- Variables need to be declared before usage
- In C, variables are declared in the beginning of blocks
- In C++, not needed to declare variables at the beginning of a block

3

Identifiers

Variables are named storage locations for C data types. A variable declaration consists of an identifier (name) and a type. A declaration introduces one or more identifiers into a program. In general terms, we say that declarations introduce identifiers into programs without actually specifying a body for those names. Identifiers can be used for variables or functions.

Rules for identifiers:

1) First token must be an alphabetic character
2) No special tokens like & % / * “
3) Identifier can have underscores for readability
4) None of the following words can be used:

```
asm, auto , break, case, char
const, continue, default, do
double, else, enum, extern, float, for
goto, if, int, long, register
return, short, signed, sizeof, static, struct
switch, typedef
union, unsigned, void, volatile, while
```

It is thus not allowed to use the above as identifiers in programs; doing so will result in a compiler error. C has in principle no limit on the maximum length of an identifier (this is however, compiler specific). It is advised not to use names beginning with an underscore (_). These are reserved by C implementations and standard libraries.

Some examples:

```
counter
Variable
Variable2
reset_counter
```

Forbidden identifiers:

```
main
2variable
#number
```

Declaring variables

In a lot of languages you can just use a variable when you need it, you do not have to specify which variables you are going to use in the calculation or operation. In C it is mandatory to declare a variable before using it. When declaring a variable you specify what data type the variable is. This type can be any of the fundamental types in conjunction with a specifier or qualifier.

```
int counter;
int position;
float deposit_amount;
long AccountNumber;
```

After declaring a variable it can be used only for the data type specified. The `counter` in the above example should only be used in operations with integer numbers. It is possible to use it in calculation where float values are used but this causes the value to be converted to another type causing compiler warnings.

Expressions

An expression is defined as a sequence of operators and operands which, taken together specify a computation. The evaluation of sub expressions within a given expression obey rules which are determined by the precedence and grouping of the operators. It is possible to force expressions to be evaluated in a certain order by the use of parentheses.

Initialising Variables

- After declaring a variable the value is unknown
- Initialise variable before usage
  - `int a;`
    `a = 10;`
- Can declare and initialise variable at once
  - `int  a = 10;`

4

Initialising variables

After declaring a variable, its value is undefined. Using the variable without initialising it can result in an unpredictable outcome. A variable always needs to be initialised before it is used to prevent unpredictable errors.

The straight forward way to initialise a variable is to declare it and assign it a value.

```
int number;
number = 10;
```

It is also possible to initialise a variable while declaring it by using the equal sign and a value before the semicolon.

```
int Number_with_initialisation = 10;
```

The variable is initialised with the value 10.

Concatenating Declarations

- Normally declare variable on each line
  - ```
    int counter;
    double double;
    int some_number;
    ```
- Possible to declare more of same type on one line
  - ```
    int counter, some_number;
    double first, second, amount;
    ```

5

Concatenating Declarations

It is possible to declare a number of variables of the same type in one expression. The variable identifiers need to be separated by a comma. This is however not recommended. Specifying a number of variables on one line makes it more difficult to specify what they are going to be used for.

The example shows how to comment each variable at declaration.

```
/* One variable on each line */
int counter;            /* Counter used in loops */
int position;           /* Position on the screen */
float deposit_amount;   /* Value deposited on the account */
long AccountNumber;     /* Account number of transaction */
```

One line comment is placed after each variable to make the program more readable.

If we place several variables on one line it becomes more difficult to comment these.

```
int counter, position;
float deposit_amount;
long AccountNumber;
```

It is more difficult to specify the usage of each variable when multiple variables are declared on the same line.

Arithmetic Operators

- Used with variables
- Normal arithmetic behaviour
  - +, -, *, /, % (mod)
- Special cases
  - Division
  - Modulo

6

Arithmetic operators

The arithmetic operators are the most used operators. The arithmetic operators are used the most because you would use them in any common calculation.

Below is a list of the arithmetic operators.

+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo

Special Cases

- Division
  - Integer division results in chopped result
    - `5 / 2 = 2 and not 2.5 or 3`
      `1 / 2 = 0 and not 0.0 or 1`
- Modulo
  - Remainder after division
    - `9  % 2 = 1 -> (9  / 2) = 4 + 1(remainder)`
      `11 % 3 = 2 -> (11 / 3) = 3 + 2(remainder)`

7

Division

The division operator can give unwanted results when using it with integer values. The division operator is often referred to as an integer divisor. When you divide two integer numbers the result will be an integer as well. Dividing five by two will result in 2 and not 2.5.

```
int a = 10;
int b = 20;
int c;

c = b / a;        /* c is 2, no problems */

int a = 10;
int b = 20;
int c;

c = a / b;        /* c equals 0 and not 0.5 */
```

When dividing integers the part behind the decimal point is discarded and only the whole part of the number will be the result. The result is not rounded but cut-off.

If we do want a precise division we have to divide floating point numbers.

```
double f1 = 10.0;
double f2 = 20.0;
double f3;

f3 = f1 / f2;     /* f3 equals 0.5 */
```

Modulo

The modulo operator can be very useful in some situations. This operator returns the remainder after the division.

```
int a = 21;
int b = 10;
int c;

c = a % b;        /* equals 1 */
```

In the above example 21 is divided by 10 resulting in 2. The remainder of this division is 1.

```
11 % 10 = 1       ->     11 - (1 * 10) = 1

12 % 10 = 2       ->     12 - (1 * 10) = 2
10 % 11 = 10      ->     10 - (0 * 11) = 10
22 % 11 = 0       ->     22 - (2 * 11) = 0
```

This operator can only be used for integer values.

Relational Operators

- Used for Boolean expressions in loops and decisions
  - A==B, A!=B
  - A<B, A>B
  - A<=B , A>=B
- These operators return true or false
  - false == 0
  - true =! 0

8

Relational and logical operators

The relational and logical operators are mostly used in Boolean expressions and they return true or false. The C language however has no true or false. These Boolean results are implemented as being 0 or not 0.

True
Everything but 0
False
0

A Boolean expression is defined to always return 0 (false) or 1 (true)

The relational operators are:

- Equality operators: == and !=
- Relational operators: < ,> , <= , and >=
- Logical operators: && (AND) and || (OR)

True and false conditions can be used in calculations as well:

```
void main()
{
    int a = 10;
    int b = 10;
    int c;

    c = (a == b);/* c equals 1 (true) */
}
```

Watch out when using the equality operator. The equality operator is not a single '=' but a double '==''. The single '=' is an assignment. If in the above example we would have used the single '=' instead of '==', the value of c would be 10 and not 1.

Logical Operators

- Used for concatenating Boolean expressions
  - && (AND)
  - || (OR)
  - ! (NOT)
- Returns
  - false -> 0
  - true -> 1

9

Logical Operators

The logical operators are used for combining Boolean expressions into one Boolean expression.

! logical not
&& logical and
|| logical or

These operators can also be used in calculations because they also return true(1) or false(0).

```
void main()
{
    int a = 10;
    int b = 11;
    int c = 12;
    int d = 13;
    int e;

    e = ((a > b) && (c < d));  /* e equals 0 */
}
```

A	!A
false	true
true	false

A	B	A&&B	A||B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Assignment Operators

- To assign two values use =
  - a = b;
- Can be combined with other binary operators
  - +=, -=, *=, /=, %=, <<=, >>=

10

Assignment operators

In the preceding examples we used the assignment operators for assigning values to variables. There are more ways of using the assignment operator. We can use it more than one time in the same expression and we can combine it with other operators.

```
void main()
{
    int a = 10;   /* a equals 10 */
    int b = 20;   /* b equals 20 */
    int c = 30;   /* c equals 30 */

    c = b = a;    /* c equals 10, b equals 10 en a equals 10 */
}
```

The above assignment expression is a concatenation of the assignment operator. There are more cryptically usages of the assignment operator as well. Before we describe these usages, first a little warning: **Watch out when using the following technique. It can make your code very unreadable**.

To increase a variable we have to use the variable on the RHS as well as the LHS of the assignment operator:

```
counter = counter + 2;
```

There is a shorter way. We can combine the assignment and the addition operator to create the '+=' operator.

```
counter += 2;
```

It is a shorter notation but can therefore be more cryptically than the full notation. So know what you are doing before using this technique.

When we combine the operators, the compiler uses parentheses for the expression.

```
int a = 2;
a *= 2 + 3;
```

Is translated into:

```
a = a * (2 + 3);    /* a equals 10 */
```

and not into:

```
a = a * 2 + 3/* a equals 7 */
```

We can use this combining of operators for all operators.

```
+= -= *= /= %= <<= >>= &= |= ^= ~=
```

Increment and Decrement

- Short way for increasing or decreasing integer values

```
int val, res;
val++;          /* val = val + 1 */
val--;          /* val = val - 1 */
```

- Can be used in post and prefix form

```
res = ++val; /* res = (val = val + 1) */
res = val++; /* res = val; val = val + 1 */
```

11

Increment and Decrement

Lots of C structures use counters which are incremented or decremented in each iteration.

The straight forward way of doing so is by adding or subtracting 1 from the counter.

```
int counter;

counter = counter + 1;
counter += 1;
```

However the C language gives us another way of easily incrementing or decrementing an integer value using an operator. These operators are called the increment and decrement operators. For incrementing we use the operator '++' and for decrementing '--'.

```
int counter;

counter++;        /* increment counter by 1 */
counter--;        /* decrement counter by 1 */
```

These operators can be used in two different ways, the post and prefix. Postfix increases the counter after it is evaluated for the expression. The prefix increases the counter before it is evaluated for the expression.

Below a simple example to show the difference between post en prefix:

```
void main()
{
    int a = 10;   /* a equals 10 */
    int b = 20;   /* b equals 20 */
    int c;

    /* Postfix operator */
    c = a + b++; /* c equals 30 and b equals 21
                    equivalent code:
                    c = a + b;
                    b = b + 1;
                 */

    /* Prefix operator */
    b = 20;      /* b equals 20 */
    c = a + ++b; /* c equals 31 and b equals 21
                    equivalent code:
                    b = b + 1;
                    c = a + b;
                 */
}
```

Conditional Operator

- The only ternary operator
- Use for short conditional expressions
  - *value = (expr) ? exprt : exprf*
- if *expr* evaluates true value = *exprt* else value = *exprf*

12

Conditional Operators

The C language has one conditional operator which is also the only ternary operator the language has. This operator is mostly used in one line choice constructs. Its syntax is as follows.

```
(expr) ? expt : expf;
```

If *expr* (a Boolean expression) evaluates to true the *expt* is performed and otherwise the *expf* is performed. For example you can use this operator in cases of determining the maximum or minimum of two numbers:

```
void main()
{
    int a = 10;
    int b = 20;
    int c;

    c = (a > b) ? a : b;
}
```

In plain English: if variable ‘a’ is greater than ‘b’ (a>b) then the expression evaluates to ‘a’ which is then assigned to ‘c’. Otherwise ‘b’ would be assigned to ‘c’. This short line assigns the maximum of two numbers (a and b) to a third one (c).

Do not use this operator for larger conditional expressions than the one shown otherwise the code becomes less readable.

The Comma Operator

- Used for concatenating expressions
- Left side is discarded
- Right side is value of expression
  - `x = (z = 6, z + 1)   /* x = 7 */`

13

The comma is an operator as well

In a previous chapter the comma was used to declare multiple variables on the same line. Besides this use the comma can be used as an operator as well. The comma operator has two arguments the first expression and the second expression.

(exp1) , (exp2)

The first expression is performed and the result is discarded. The second expression is performed as well and its resulting value is the resulting value of the comma expression. This operator is not used so much because it makes your code more unreadable.

```
void main()
{
    int y = 5;
    int c;

    c = (y +=4, y +5);  /* c equals 14 */
}
```

Type Conversion

- Values automatically converted to bigger or more precise type

```
int i;
double d;
d = i;  /* i converted to a double */
```

- Values converted to smaller or less precise types with compiler warning

```
i = d;  /* d converted to an int */
```

14

Standard Conversions between Data Types

This section discusses the topic of type conversion, by which we mean how objects of a given type are transformed into other types. Rules are given for promotion and truncation. Only the fundamental data types are dealt with here.

Conversion from one data type to another type can be a source of programming errors. For example, the receiving data type may not have enough bytes (determined by the sizeof() operator) to hold all the necessary information. This will result in loss of information and subtle programming errors.

The following topics concern us here:

- Integral promotions and conversions.
- Floating point conversions.
- Conversions between floating point numbers and integers.

Integral Promotions and Conversions

We speak of integral promotion in the case where objects of type *char*, *short int*, *enumerator type* and *int bit fields* (both signed and unsigned varieties) are used in place of an *int*. If the *int* can hold all the values of the original type, the resulting value is converted to an *int*. If an *int* cannot hold the value, the result is converted to an *unsigned int*.

The following example shows how certain types are promoted to *int*.

```
// Program to show integral promotion.
// Objects of char and short int are studied.
// (C) Datasim BV 1995

#include <stdio.h>
#include <stdlib.h>

void print_type(int i)
{
    printf("%d",i);     // Prints argument on screen
}

void main()
{
    char c = 'a';
    short int i = 127;

    print_type(c);      // Value output  == 97 (ASCII value)
    print_type(i);      // Value output  == 127
}
```

We thus see that we can use a number of types when an int is expected. This has its advantages and its disadvantages. The main advantage is that we can use certain data types as arguments to functions without having first to convert them to the type which is expected in the declaration of the function. The main disadvantage is that conversions take place which are beyond the control of the programmer; in fact, the compiler has taken over and unexpected results can take place. For example, printing the character "a" in the above program results in the value 97. We can thus see integral promotion as a type of "widening" of the shorter type to the longer type.

Floating Point and Integral Conversions

There are three basic float types, namely float (single precision), double and long double (double precision). The set of values as represented by float is a subset of the set of values as represented by double which in turn is a subset of those values represented by long double.

The general conversion rules are:

- Conversion from a less precise type to a more precise type suffers no change in value.
- Conversion from a more precise type to a less precise type results in the next higher or next lower representable value (assuming that the value is within the representable range). If the value is not within range, the result is undefined.
- Conversion from a float to an integral type results in truncation (the fractional part is thrown away).
- Conversion between an integral type to a float type is machine dependent.

A number of conversions are machine dependent. Consequently, care should be taken in programs when converting from one type to another type.
The following program gives some examples of arithmetic conversions.

```
// Floatingpoint and integral conversions (some examples).
// (C) Datasim BV 1995

#include <stdio.h>
#include <stdlib.h>
#include <float.h>

void main()
{
    double d1 = 1.004;
    double d2 = 1.00004e+3;
    float f1, f2;
    int i1, i2;

    f1 = d1;
    f2 = d2;
    i1 = d1;
    i2 = d2;
    printf("Assigning float to 1.004 is %f\n", f1);
    printf("Assigning float to 1.00004e+3 is %f\n", f2);

    printf("Assigning int to 1.004 is %d\n", i1);
    printf("Assigning int to 1.00004e+3 is %d\n", i2);
}
```

You can run this program to see what the effects of the conversions are.

The sizeof Operator

- Yields the size in bytes of the expression
  `sizeof(expr);`
- No cryptic symbol
- Unary operator, not a function

15

Sizeof Operator

The *sizeof* operator is a simple operator that is used to determine the size of a certain object. This object can be a variable or a type.

```
void main()
{
    float f;
    printf("%d\n", sizeof(int));      /* size of int in bytes */
    printf("%d\n", sizeof(f));        /* size of f in bytes */
}
```

Bitwise Operators

- Bitwise operators work on bit level
  - &, |, ^, <<, >>, ~
- Some of them are machine dependent
- First examine how your program uses these operators

16

Bitwise Operators

C has a number of operators which work on bit-level. Some of these operators are machine dependent so be careful with applying them.

The bitwise operators are:

&	Bit-and
|	Bit-or
^	Exclusive bit-or
<<	Shift left
>>	Shift right
~	One complement (unary)

All of these operators work on bit level with their operands. The following example combines two values with a bit level *and*.

```
void main()
{
    int i = 10;
    int j = 7;
    int result;

    result = i & j;   /* Bit wise and */
}
```

The example shows the *and* operation on the values 10 and 7. The resulting value will be calculated as follows:

```
variable  binary            decimal
i         0000000000001010  10
j         0000000000000111  7

result    0000000000000010  2
```

When using the bitwise operators the following tables are used:

Complement (~)	
0	1
1	0

Bitwise AND (&)		
	0	1
0	0	0
1	0	1

Bitwise OR (|)		
	0	1
0	0	1
1	1	1

Bitwise XOR (^)		
	0	1
0	0	1
1	1	0

The shifting operators are a story on their own. The shifting operators are machine dependent. Shifting operators shift bits in or out a bit pattern. The shift left operator shifts bits to the left while inserting 0 at the right. The shift right operator shifts bits to the right but what is shifted in at the left depends on what type it is and what type of machine you have. The best thing is to not use it or first check how your machine reacts to shifting right. The following example shows how to determine what is inserted at the left when shifting right. This example uses conditional statements like *if* which are explained in the next module.

```
void main()
{
    int i;

    i = -3;             /* Left bit (sign bit) equals 1 */
    printf("i=%d\n");
    i = i >> 1;
    printf("i >> 1 evaluates to: %d" , i);

    if (i < 0)
    {
          printf("Negative, a 1 was inserted left\n");
    }
    else
    {
          printf("Positive, a 0 was inserted left\n");
    }
}
```

Operator Precedence

- Statement can have multiple operators
- Order of evaluation is defined by rules
  - Similar to mathematical order rules
- Can force evaluation order by using ()

17

Priority of Operators

When creating expressions which contain operators, the order in which is evaluated relies on a few rules. Take for example: *a + b * c*. This expression is evaluated to: *a + (b * c)*. '*' has a higher priority than '+'. The expression: *a / b * c* is evaluated as: *(a / b) * c*. '/' and '*' have the same priority thus they are evaluated from left to right.

To examine how an expression is evaluated you have to know the priority of the different operators. Operators with a higher priority are evaluated first. If there are operators with the same priority they are evaluated in their evaluation direction. In the above example from left to right but with some operators it is from right to left. Most of the operators evaluate from left to right. In the table below only line 2, 13 and 14 are evaluated from right to left.

In the next table all of the operators of the C language are represented. The operators on the same line have the same priority.

```
1   () [] ->
2   ! ~ ++ -- + - * & (type) sizeof   (unary +- and pointer *)
3   * / %
4   + -
5   << >>
6   < <= > >=
7   == !=
8   &
9   ^
10  |
11  &&
12  ||
13  ?:
14  = += -= *= /= %= &= ^= |= <<= >>=
15  ,
```

The operators on line 2 are all unary operators. Notice that () has the highest priority thus use it wisely. If you are in doubt in which order an expression is evaluated then use these parenthesis to make sure the expression is evaluated in the right order.

When using operators with the same priority the evaluation order can be misinterpreted. The operators * and / have the same priority and evaluate from left to right. Left to right has nothing to do with the order of the operators on the same line but with the place inside the expression.

```
a * b / c
```

is the same as

```
(a * b) / c
```

and

```
a / b * c
```

is the same as

```
(a / b) * c
```

For example:

```
/*
   Program to show the use of expressions and operator precedence.
   When in doubt about the order of evaluation, use parentheses.
   (C) Datasim BV 1995
*/

#include <stdio.h>
#include <stdlib.h>

void main()
{
    int j = 2 + 3 * 4;                    /* j == 14 */
    int m = 2 + (3 * 4);                  /* m == 14 */
    int k = (2 + 3) * 4;                  /* k == 20 */

    double d = 20.0 / 5.0 * 2.0;          /* d == 8.0 */
    double s = (20.0 / 5.0) * 2.0;        /* s == 8.0 */

    printf("%d\n", j);
    printf("%d\n", m);
    printf("%d\n", k);
    printf("%f\n", d);
    printf("%f\n", s);
}
```