Chapter 4

Expressions

Operators

- Expressions are built from
  - variables
  - constants
  - operators
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - *assignment* operators
  - increment and decrement operators

  and many others

Arithmetic Operators

- C provides five binary ***arithmetic operators***
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `%` remainder

C has no power operator

- An operator is ***binary*** if it has two operands
- There are also two ***unary*** arithmetic operators
  - `+` unary plus
  - `-` unary minus

Unary Arithmetic Operators

- The unary operators require one operand

```
i = +1;
j = -i;
```

- The unary `+` operator does nothing

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`
  - `10 % 3` has the value 1, and `12 % 4` has the value 0
- All binary arithmetic operators (except `%`) allow either *integer* or *floating-point* operands, with mixing allowed
- When `int` and `float` operands are mixed, the result has type `float`
  - `9 + 2.5f` has the value 11.5, and
  - `6.7f / 2` has the value 3.35

The / and % Operators

- The `/` and `%` operators require special care
  - When both operands are integers,
    - `/` "*truncates*" the result (`3 / 4` is 0, not 0.75)
  - The `%` operator requires integer operands;
    - If either operand is not an integer, the program won't compile
  - Using *zero* as the right operand of either `/` or `%` causes undefined behavior
  - The behavior when `/` and `%` are used with *negative* operands is ***implementation-defined*** in C89
  - In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`

Operator Precedence

- Does `i + j * k` mean
  - "add `i` and `j`, then multiply the result by `k`" or
  - "multiply `j` and `k`, then add `i`"
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`
- If the parentheses are omitted, **C** uses ***operator precedence*** rules to determine the meaning of the expression

Operator Precedence

- The arithmetic operators have the following relative precedence:

  Highest: `+ -` (unary)
  `* / %`
  Lowest: `+ -` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`
  `-i * -j` is equivalent to `(-i) * (-j)`
  `+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence
- An operator is said to be ***left associative*** if it groups *from left to right*
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all ***left associative***, so

  `i - j – k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`

Program: Computing a UPC Check Digit

- Most goods sold in U.S. and Canadian stores are marked with a Universal Product Code (UPC):

0 13800 15173 5

- Meaning of the digits underneath the bar code
  First digit: Type of item
  First group of five digits: Manufacturer
  Second group of five digits: Product
  Final digit: Check digit, used to help identify an error in the preceding digits

Program: Computing a UPC Check Digit

- How to compute the check digit
  - Add the first, third, fifth, seventh, ninth, and eleventh digits
  - Add the second, fourth, sixth, eighth, and tenth digits
  - Multiply the first sum by 3 and add it to the second sum
  - The check digit is the digit which, when added to the above sum, produces a sum that is multiple of 10
    - Subtract 1 from the total
    - Compute the remainder when the adjusted total is divided by 10
    - Subtract the remainder from 9

Program: Computing a UPC Check Digit

- Example for UPC 0 13800 15173 5:

  First sum: 0 + 3 + 0 + 1 + 1 + 3 = 8
  Second sum: 1 + 8 + 0 + 5 + 7 = 21
  Multiplying the first sum by 3 and adding the second yields 45
  Subtracting 1 gives 44
  Remainder upon dividing by 10 is 4
  Remainder is subtracted from 9
  Result is 5

Program: Computing a UPC Check Digit

- The `upc.c` program asks the user to enter the first 11 digits of a UPC, then displays the corresponding check digit:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

- The program reads each digit group as five one-digit numbers
- To read single digits, we'll use `scanf` with the `%1d` conversion specification

upc.c

```c
/* Computes a Universal Product Code check digit */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

Simple Assignment

- The effect of the assignment *v* = *e* is to
  - evaluate the expression *e* and
  - copy its value into *v*
- *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;            /* i is now 5   */
j = i;            /* j is now 5   */
k = 10 * i + j;   /* k is now 55 */
```


Simple Assignment

- If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;

i = 72.99f;   /* i is now 72 */
f = 136;      /* f is now 136.0 */
```

Simple Assignment

- In many programming languages,
  - assignment is a ***statement***
- In C, however,
  - assignment is an ***operator***, just like `+`, `-`, `*`, `/`, and `%`
- The value of an assignment operation *v* `=` *e* is
  - the value of *v after the assignment*
- Example:
  The value of `i = 72.99f` is 72 (*not* 72.99)

Side Effects

- An operator that modifies one of its operands is said to have a ***side effect***
- The simple assignment operator has a side effect
  - it modifies its left operand
- Evaluating the expression `i = 0`
  - produces the result 0 and
  - as a side effect, assigns 0 to `i`

Side Effects

- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The `=` operator is ***right associative***, so this assignment is equivalent to

```
i = (j = (k = 0));
```

- Watch out for unexpected results in chained assignments as a result of any type conversion

Side Effects

- What are the values of `i` and `j` below?

```
int i; float x;
i = x = 5.5;
x = i = 5.5;
```

- What are the values of `i, j` and `k` below?

```
i = 1;
k = 1 + (j = i);
```

- "Embedded assignments" can make programs hard to read
- They can also be a source of subtle bugs

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand
- An ***lvalue*** represents an object stored in computer memory, not a constant or the result of a computation
- Variables are ***lvalues***; expressions such as `10` or `2 * i` are *not*

```
52 = i;        /*** WRONG ***/
i + j = 0;     /*** WRONG ***/
-i = j;        /*** WRONG ***/
```

- The compiler will produce an error message such as "*invalid lvalue in assignment*"

Compound Assignment

- It is common in assignments to use the old value of a variable to compute its new value
- Example:

```
i = i + 2;
```

- In **C**, such increment can be simplified by using a *compound assignment operator* such as `+=`
- we simply write

```
i += 2;    /* same as i = i + 2; */
```

Compound Assignment

- Compound assignment operators include:

  `+=   -=   *=   /=   %=`
- Note that, there is ***NO*** space between the two characters
- All compound assignment operators work in much the same way:

  v `+=` *e* adds *v* to *e*, storing the result in *v* (i.e., *v* `=` *v* `+` *e*)

  v `-=` *e* subtracts *e* from *v*, storing the result in *v* (i.e., *v* `=` *v* `-` *e*)

  v `*=` *e* multiplies *v* by *e*, storing the result in *v* (i.e., *v* `=` *v* `*` *e*)

  v `/=` *e* divides *v* by *e*, storing the result in *v* (i.e., *v* `=` *v* `/` *e*)

  v `%=` *e* computes the remainder when *v* is divided by *e*, storing the result in *v* (i.e., *v* `=` *v* `%` *e*)

Compound Assignment

- With compound assignment we may face a problem, which is *operator precedence*

  `i *= j + k`
  - isn't the same as `i = i * j + k`
  - if fact, it is `i = i * (j + k)`

Increment and Decrement Operators

- Two of the most common operations on a variable are
  - "incrementing" (adding 1) and
  - "decrementing" (subtracting 1)

```
i = i + 1;
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators

```
i += 1;
j -= 1;
```


Increment and Decrement Operators

- C provides two special operators
  - The `++` operator adds 1 to its operand (***increment***)
  - The `--` operator subtracts 1from its operand (***decrement***)
- The increment and decrement operators
  - Can be used as
    - ***prefix*** operators (`++i` and `--i`) or
    - ***postfix*** operators (`i++` and `i--`)
  - Have ***side effects***
    - they modify the values of their operands

Increment and Decrement Operators

- Evaluating the expression `++i` (a "pre-increment")
  - as a side effect, `i` is incremented and then
  - produces the value of `i`, which is ***the incremented value***

```
i = 1;
printf("i is %d\n", ++i);   /* prints "i is 2" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

- Evaluating the expression `i++` (a "post-increment")
  - produces the result `i`, which is ***the original value*** and then
  - as a side effect, `i` is incremented

```
i = 1;
printf("i is %d\n", i++);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 2" */
```


Increment and Decrement Operators

- `++i` means increment `i` immediately
- `i++` means use the old value of `i` for now, but increment `i` later
- The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);   /* prints "i is 0" */
printf("i is %d\n", i);     /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 0" */
```

Increment and Decrement Operators

- ```
  i = 1;
  j = 2;
  k = ++i + j++;
  ```
  The last statement is equivalent to
  ```
  i = i + 1;
  k = i + j;
  j = j + 1;
  ```
  The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively
- ```
  i = 1;
  j = 2;
  k = i++ + j++;
  ```
  will give `i`, `j`, and `k` the values 2, 3, and 3, respectively

Chapter 4: Expressions

Increment and Decrement Operators

- How about
  ```
  i = 1;
  j = ++i++;
  /*The previous statement is WRONG*/
  j = ++(i++);
  /*The previous statement is WRONG*/
  ```
- The operant of the increment and decrement operators must be an ***lvalue***

Expression Evaluation

- Table of operators discussed so far:

Precedence	*Name*	*Symbol(s)*		*Associativity*
1	increment (postfix)	`++`	(i.e., `i++`)	left
	decrement (postfix)	`--`	(i.e., `i--`)	left
2	increment (prefix)	`++`	(i.e., `++i`)	right
	decrement (prefix)	`--`	(i.e., `--i`)	right
	unary plus	`+`		right
	unary minus	`-`		right
3	multiplicative	`* / %`		left
4	additive	`+ -`		left
5	assignment	`= *= /= %= += -=`		right

- Example:

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


Expression Evaluation

- Starting with the operator with highest precedence, put parentheses around the operator and its operands
- Example:

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

	Precedence level
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += ((c++) - d) + ((--e) / (-f))`	4
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`a = (b += (((c++) - d) + ((--e) / (-f))))`	5
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

Order of Subexpression Evaluation

- The value of an expression may depend on the order in which its subexpressions are evaluated
- `C` doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical *and*, logical *or*, conditional, and *comma* operators)
- In the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`
- *Will it make a difference?*
  Who cares?

Order of Subexpression Evaluation

- Most expressions have the same value regardless of the order in which their subexpressions are evaluated
- However, this may not be true when a subexpression modifies one of its operands:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- The effect of executing the second statement is undefined

Order of Subexpression Evaluation

- Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression
- Some compilers, when encounter such an expression, may produce a warning message such as *"operation on 'a' may be undefined"*

Order of Subexpression Evaluation

- To prevent problems, it's a good idea to avoid using assignment operators in subexpressions
- Instead, use a series of separate assignments:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6

Order of Subexpression Evaluation

- Besides the *assignment operators* (including the *compound assignment operators*), the only operators that modify their operands are *increment* and *decrement*
- When using these operators, be careful that an expression doesn't depend on a particular order of evaluation

Order of Subexpression Evaluation

- Example:

```
i = 2;
j = i * i++;
```

- What is the value of `i` and `j`?
- It's natural to assume that `j` is assigned 4
- However, `j` could just as well be assigned 6 instead
  1. The second operand (the original value of `i`) is fetched, then `i` is incremented
  2. The first operand (the new value of `i`) is fetched
  3. The new and old values of `i` are multiplied, yielding 6

Undefined Behavior

- Statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` might cause ***undefined behavior***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers
  - The program may not compile in the first place
  - If it compiles it may not run
  - If it does run, the program may crash, behave erratically, or produce meaningless results
- *Undefined behavior should be avoided*

Expression Statements

- C has the unusual rule that

  any expression can be used as a statement
- Example:

```
++i;
```

`i` is first incremented, then the new value of `i` is fetched but then discarded

Expression Statements

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* not useful */
```


Expression Statements

- A slip of the finger can easily create a "*do-nothing*" expression statement
- For example, instead of entering

  ```
  i = j;
  ```

  we might accidentally type

  ```
  i + j;
  ```

- Some compilers can detect *meaningless* expression statements
  - If this is the case, you may get a warning such as "*statement with no effect*"