3. Structure of a C Program

C Programming

Agenda

- Expression
- Precedence and Associativity
- Type Conversion
- Statement

Expressions and Statements

- Important elements in C language
  - Variables are used to store data

    Ex) int i = 5;

  - Expressions are mainly used to calculate values

    Ex) (i / j + 10) * 2

    Note! Expression can also specify action by side effect

  - Statements are used to specify actions

    Ex) i = j + 5;

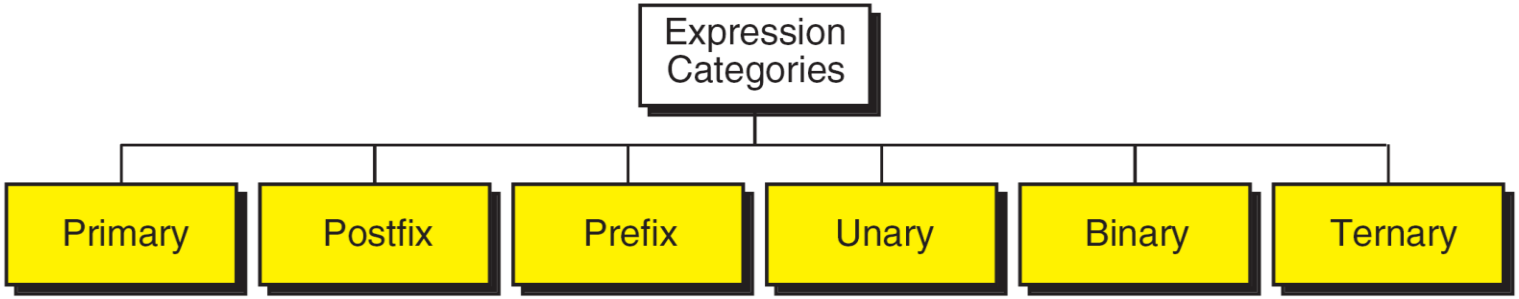
    printf(“Hello, World!₩n”);

Expressions

- Expression: a sequence of operands and operators that reduces to a single value

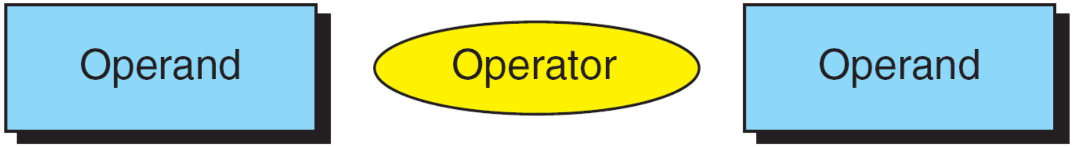
  Ex) 2 + 5, 2 + 5 * 7, …

- Categories of expressions



Binary Expressions

- Binary expression: operand–operator–operand combination



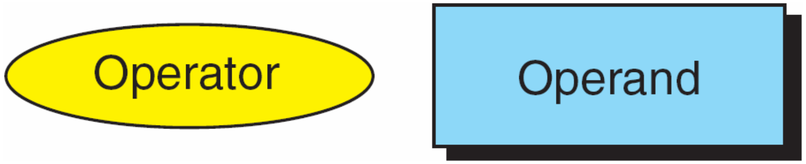
  - Multiplicative expressions(*, /, %)

    Ex) 10 * 3, true * 4, 'A' * 2, 22.3 * 2, …
  - Additive expressions(+, –)

    Ex) 3 + 7, 5 – 8, …

Unary Expression

- Unary expression: expression containing single operand



Operator

Operand

  - Unary plus/minus

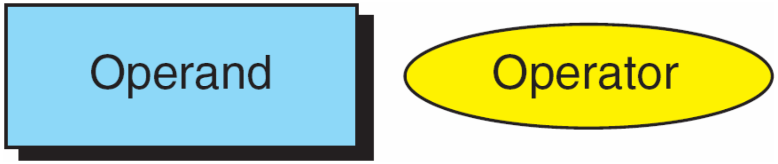
    Ex) +5, −3, −a, …
  - size of: size (in byte) of a type or primary expression
    - sizeof(int)
    - sizeof −345.23, sizeof x
  - Cast operator: type conversion

    Ex) int x = 10;

    (float)x example

Postfix Expression

- Postfix expression: operator follows operands



Operand

Operator

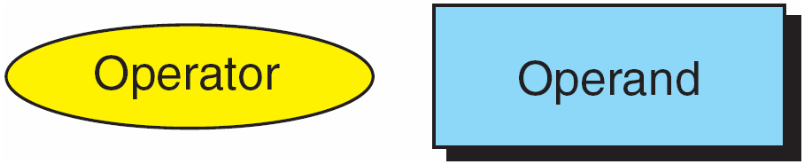
  - Postfix increment/decrement
    - □ a++/a-- (equivalent to a = a + 1 / a = a – 1)

```
Ex) x = a++; is equivalent to …
      x = a;
      a = a + 1;
```

```
Ex) int a = 4;
      printf("value of a =          %2d₩n", a);
      printf("value of a++ =        %2d₩n", a++);
      printf("new value of a =   %2d₩n", a);
```

Prefix Expressions

- Prefix expressions: operator precedes operand



- Prefix increment/decrement
  - ++a/−−a (equivalent to a = a + 1 / a = a − 1)

    Ex) x = ++a; is equivalent to …

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

Ex) int a = 4;

```
    printf(“value of a =        %2d₩n”, a);
    printf(“value of ++a =      %2d₩n”, ++a);
    printf(“new value of a =    %2d₩n”, a);
```

Assignment Expressions

- Assignment expression(=): evaluates operand on right side and places its value in variable on left side

  Ex) a = 5, b = x + 1, i = i + 1

  - Value of total expression: the assigned value

  Ex) printf(“Value of ₩“a = 5 ₩” = %d₩n”, a = 5);

- Compound assignment (*=, /=, %=, +=, −=): binary operator + assignment

  Ex) x *= y + 3;   // equivalent to x = x * (y + 3)

Demonstration of Compound Assignment

- Source code

```
#include <stdio.h>

int main (void)
{
      int x = 10, y = 5;

      printf("x: %2d  |  y: %2d ", x, y);
      printf("  |  x *= y + 2: %2d ", x *= y + 2);
      printf("  |  x is now: %2d₩n",  x);

      x = 10;
      printf("x: %2d  |  y: %2d ",     x, y);
      printf("  |  x /= y + 1: %2d ", x /= y + 1);
      printf("  |  x is now: %2d₩n",  x);

      x = 10;
      printf("x: %2d  |  y: %2d ",     x, y);
      printf("  |  x %%= y − 3: %2d ", x %= y − 3);
      printf("  |  x is now: %2d₩n", x);

      return 0;
}     // main
```

Review

- What is the result of the following program?

```
#include <stdio.h>

int main()
{
    int x = 4;
    int y = 0;

    printf("₩"x = 4₩" = %d₩n", x = 4);
    printf("₩"y = ++x₩" = %d₩n",y = ++x);

    printf("₩n");
    printf("₩"x = 4₩" = %d₩n", x = 4);
    printf("₩"y = x++₩" = %d₩n",y = x++);

    return 0;
}
```

Side Effects

- Side effect: action that results from evaluation of an expression

  Ex) Assignment, increment, decrement, …

```
x = 4;          // evaluation result: 4
x = x + 3;      // evaluation result: 7
y = ++x;        // evaluation result: 8
```

Side Effects

- Evaluation of expressions with side effect

  Ex) --a * (3 + b) / 2 – c++ * b, given a = 3, b = 4, c = 5

  1. Parenthesis

     --a * 7 / 2 – c++ * b

  2. Postfix expression

     --a * 7 / 2 – 5 * b // c is increased after evaluation

  3. Prefix expression

     2 * 7 / 2 – 5 * b // a is increased at this point

  4. Multiplication and division

     7 – 20

  5. Subtraction

     -13

  - Side effects after evaluation: a = 2, b = 4, c = 6

- Warning: in C, if an expression variable is modified more than once during its evaluation, the result is undefined.

Side Effects

- Evaluation of expressions without side effect

  Ex) a * 3 − (3 + b) / 2 + c * 2, given a = 3, b = 4, c = 5

  1. Parenthesis

     −−a * 7 / 2 − c++ * b

  2. Multiplication and division

     6 − 3 + 10

  5. Addition and Subtraction

     13

  - After evaluation: a = 3, b = 4, c = 5

Agenda

- Expression
- Precedence and Associativity
- Type Conversion
- Statement

Precedence and Associativity

- Precedence: order of different operators in a complex expression

  Ex) 2 + 3 * 4 = 2 + (3 * 4) = 14

  -b++ = -(b++)

- Associativity: order of operators with the same precedence

  Ex) 5 – 3 + 2 = (5 – 3) + 2 = 4

  - Left-to-right associativity: *, /, %, +, -

    Ex) 3 * 8 / 4 % 4 * 5

  - Right-to-left associativity: assignment operator

    Ex) a += b *= c -= 5 : (a += (b *= (c -= 5)))

Precedence and Associativity

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Agenda

- Expression
- Precedence and Associativity
- Type Conversion
- Statement

Type Conversion

- What happens when we write an expression that involves different data types?

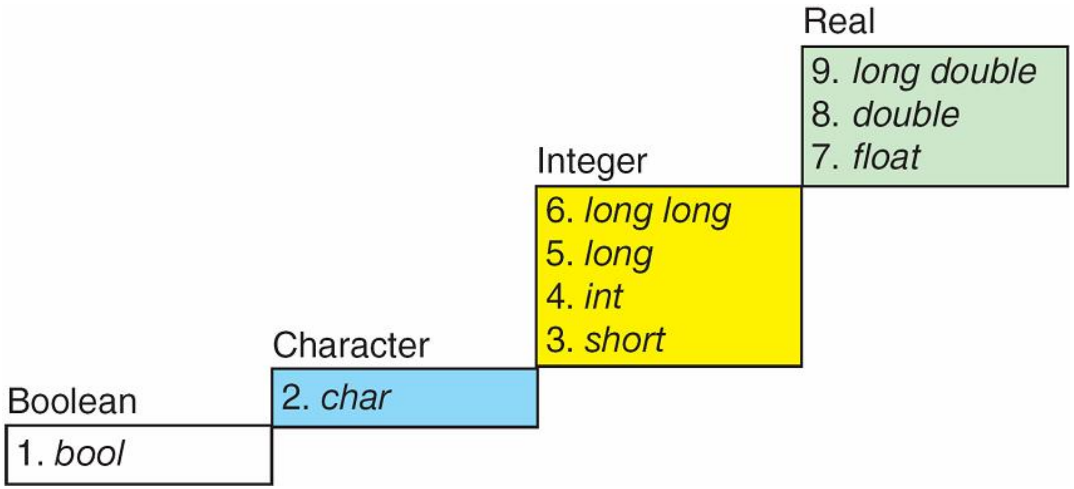
  Ex) 2 * 3.141592

  → Integer 2 is converted to floating-point type (2.0)

- Type conversion: changing an entity of one data type into another
  - Implicit type conversion (coercion)
  - Explicit type conversion (casting)

Implicit Type Conversion

- Implicit type conversion: when two operands in a binary expression are of different types, C automatically converts one type to another
  - The conversion is decided by conversion rank.
    (The actual conversion rule is more complex.)



Ex) <int value: 4> + <float value: 7>
→ <int value> is converted into <float value>

Implicit Type Conversion

- Conversions in assignment
  - For an assignment expression, C makes right expression the same rank with left variable

    ```
    Ex) char c = ‘A’;           // ‘A’ == 0x41 == 65
        int i = 1234;
        double d = 3458.0004;
        i = c;                  // char -> int (promotion)
        i = d;                  // double -> int (demotion)
    ```

  - Promotion: lower rank -> higher rank

    ```
    Ex) float f = 10;
    ```

  - Demotion: higher rank -> lower rank

    ```
    Ex) int i = 10.5;
    ```

    - A problem can occur, if value of right expression is too large to be accommodated in left variable

      char c = INT_MAX;   // INT_MAX is usually $2^{31}-1$

Explicit Type Conversion

- Explicit type conversion: type conversion through cast operator

```
Ex) int -> float
     int a = 10;
     (float) a              // result: 10.F

Ex) int totalScores = 250;
      int numScores = 3;
     float average = 0.;
     average = totalScores / numScores;                // 83.000000
     average = (float) totalScores / numScores;       // 83.333333
     average = (float) (totalScores / numScores);   // 83.000000
```

Agenda

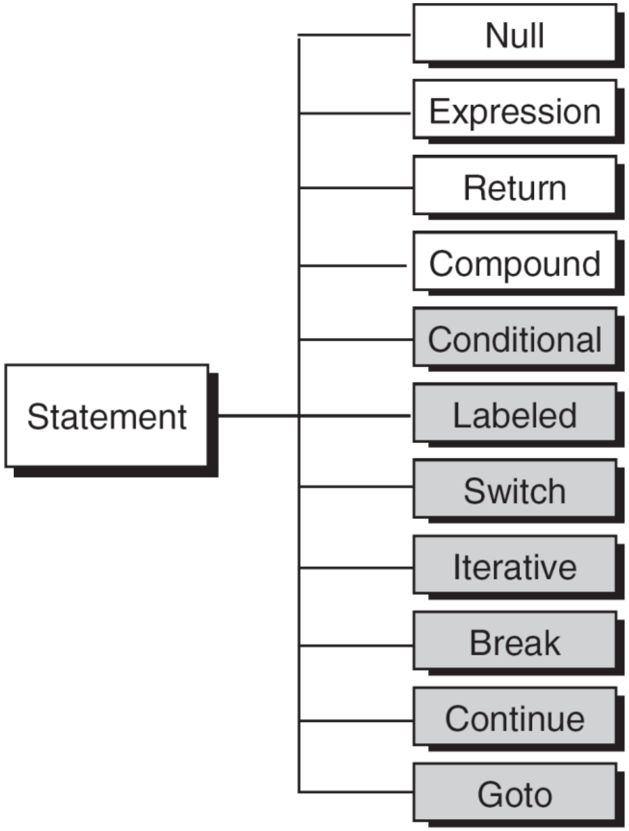
- Expression
- Precedence and Associativity
- Type Conversion
- <u>Statement</u>

Statements

- Statement: an instruction to execute something that will not return a value.
  - Most C statements are terminated by semicolon

  Ex) printf("Hello₩n");

- Types of statements
  - Null/expression/return/compound
  - Control statements
    - Explained in later chapters.



Statements

- Null statement: a semicolon

```
Ex) ;
```

- Expression statement: expression + semicolon

```
Ex) a = 2;
    a = b = 3;        // equivalent to a = (b = 3);
    ioResult = scanf(“%d”, &x);
    a++;
```

- Return statement: termination of a function

```
Ex) return expression;
```

Statements

- Compound statement (block): a unit of code consisting of zero or more statements, enclosed by braces

Ex)

```
{
    // local declarations
    int x, y, z;

    // statements
    x = 1;
    y = 2;
}  //          semicolon is not needed for compound statement
```

Use of Semicolon

- Every declaration in C is terminated by semicolon
- Most statements in C are terminated by a semicolon.
- A semicolon should not be used with a preprocessor directives

```
Ex 1) #include <stdio.h>
      #define MY_SALARY 2000000

Ex 2) #define SALES_TAX_RATE 0.825;
      salesTax = SALES_TAX_RATE * salesAmount;
```