To:
Tenouk

C LAB WORKSHEET 5
C/C++ Variable, Operator And Type 1

Items in this page:

1. Variable, data type, operator and assignment.
2. Basic data type.
3. Standard and non-standard C libraries.
4. Tutorial references are: C/C++ intro & brief history, C/C++ data type 1, C/C++ data type 2, C/C++ data type 3 and C/C++ statement, expression & operator 1, C/C++ statement, expression & operator 2 and C/C++ statement, expression & operator 2.

- You have learnt about the basic data types: integer (whole number), floating point, character and string. In this practice we will play around with these data types. Together with data types you will also deal with mathematical operators such as +, * and -. Please give attention to the character and string data types.

C Operators

For the following C code:

```
int p, q, r;
p = q + r;
```

- q and r are operands and + is an operator. Also, in this case p, q and r are variables. The p = q + r; statement is evaluated from right to left.
- The C operators are a subset of the C++ operators thus in C++ you will find more operators. There are three types of operators based on the number of operand that they operate on: unary, binary and ternary.

    1. A **unary expression** consists of either a unary operator prepended to an operand such as ++x or p--, or the sizeof keyword followed by an expression. The expression can be either the name of a variable or a cast expression such as (float)x where x is an integer. If the expression is a cast expression, it must be enclosed in parentheses.
    2. A **binary expression** consists of two operands joined by a binary operator for example: x + y.
    3. A **ternary expression** consists of three operands joined by the conditional-expression operator for example: x = y ? 1 : 100;.

- C includes the following unary operators:

Symbol	Name
-, ~, !	Negation and complement operators

*, &	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator
++, --	Unary increment and decrement operators

- Binary operators associate from left to right. C provides the following binary operators:

Symbol	**Name**
*, /, %	Multiplicative operators
+, -	Additive operators
<<, >>	Shift operators
<, >, <=, >=, ==, !=	Relational operators
&, |, ^	Bitwise operators
&&, ||	Logical operators
,	Sequential-evaluation operator

- The conditional-expression operator has lower precedence than binary expressions and differs from them in being right associative. Expressions with operators also include assignment expressions, which use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple-assignment operator (=) and the compound-assignment operators. Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator.

Operator Precedence and Order of Evaluation

- The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. Precedence can also be described by the word "binding." Operators with a higher precedence are said to have tighter binding.
- The following table summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together, they have **equal precedence** and are evaluated according to their associativity. The operators in the table are described in the sections beginning with postfix operators. The rest of this section gives general information about precedence and associativity.

Precedence and Associativity of C Operators

- The following Table lists operator precedence and the associativity.

Symbol[1]	**Type of Operation**	**Associativity**
[ ] ( ) . –> postfix ++ and postfix --	Expression	Left to right
prefix ++ and prefix -- sizeof & * + – ~ !	Unary	Right to left
typecasts	Unary	Right to left

* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical-AND	Left to right
||	Logical-OR	Left to right
? :	Conditional-expression	Right to left
= *= /= %= += –= <<= >>= &= ^= |=	Simple and compound assignment[2]	Right to left
,	Sequential evaluation	Left to right

1
: Operators are listed in descending order of precedence. If several operators appear on the same line or in a group, they have equal precedence.

2
: All simple and compound-assignment operators have equal precedence.

- An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right. The direction of evaluation does not affect the results of expressions that include more than one multiplication (*), addition (+), or binary-bitwise (& | ^) operator at the same level. Order of operations is not defined by the language. The compiler is free to evaluate such expressions in any order, if the compiler can guarantee a consistent result.
- Only the sequential-evaluation (,), logical-AND (&&), logical-OR (||), conditional-expression (? :), and function-call operators constitute sequence points and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. Note that the comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.
- Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. This is called "short-circuit" evaluation. Thus, some operands of the expression may not be evaluated. For example, in the expression:

```
x && y++
```

- the second operand, y++, is evaluated only if x is true (nonzero). Thus, y is not incremented if x is false (0).
- The following list shows how the compiler automatically binds several sample expressions:

Expression	**Automatic Binding**
a & b || c	(a & b) || c
a = b || c	a = (b || c)
q && r || s--	(q && r) || s--

- In the first expression, the bitwise-AND operator (&) has higher precedence than the logical-OR operator (||), so a & b forms the first operand of the logical-OR operation.
- In the second expression, the logical-OR operator (||) has higher precedence than the simple-assignment operator (=), so b || c is grouped as the right-hand operand in the assignment. Note that the value assigned to a is either 0 or 1.
- The third expression shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (&&) has higher precedence than the logical-OR operator (||), so q && r is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, q && r is evaluated before s--. However, if q && r evaluates to a nonzero value, s-- is not evaluated, and s is not decremented. If not decrementing s would cause a problem in your program, s-- should appear as the first operand of the expression, or s should be decremented in a separate operation. The following expression is illegal and produces a diagnostic message at compile time:

Illegal Expression	**Default Grouping**
p == 0 ? p += 1: p += 2	(p == 0 ? p += 1 : p) += 2

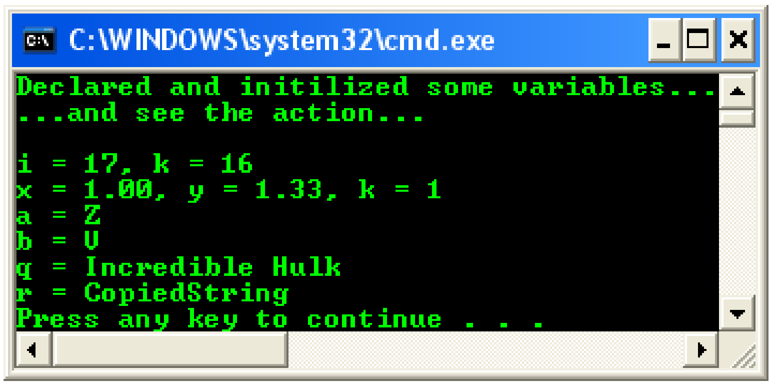
- In this expression, the equality operator (==) has the highest precedence, so p == 0 is grouped as an operand. The conditional-expression operator (? :) has the next-highest precedence. Its first operand is p == 0, and its second operand is p += 1. However, the last operand of the conditional-expression operator is considered to be p rather than p += 2, since this occurrence of p binds more closely to the conditional-expression operator than it does to the compound-assignment operator. A syntax error occurs because += 2 does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown below to correct and clarify the preceding example:

```
( p == 0 ) ? ( p += 1 ) : ( p += 2 )
```

- Create a new empty Win32 Console Application project named **variable** and add a C++ source file named **variablesrc** to the project. Make sure you set your project property to be compiled as C codes. Type the following code, build and run your program, show the output.
- When declaring a variables (the values that change during the program execution), a memory locations will be reserved for storing the values. Depending on the declared variables, these values may be number, name or any such item. An integer is a data type that stores only whole numbers. In the following code, n, m and k are declared as integers and n is initialized (given an initial value) to 40. Then m is assigned the value of 10 and k is assigned the value of 70 after adding m, n and 20.

```
// needed for printf()
#include <stdio.h>
// needed for strcpy(), strcpy_s() - a secure version and their family
#include <string.h>

int main()
{
    // declare variables and initialize some of them
    // some compiler ask you to initialize all the variables...
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    int i, j = 4, k;
    float x, y = 8.8; // warning, can change to double...
    char a, b ='V';
    char q[20], r[20] = "Incredible Hulk";
    printf("Declared and initialized some variables...\n"
        "...and see the action...\n\n");
    // READ AND EVALUATE C/C++ EXPRESSION FROM RIGHT TO LEFT
    i = 5;          // i becomes 5
    i = i + j;       // i becomes 9 (5 +4)
    i = i + y;       // i becomes 17 (9+8.8), warning...
    k = (5 + (6/2)) * (3-1);
    printf("i = %d, k = %d\n", i, k);
    i = 3;
    x = (j/i);       // integer divided by integer is an integer, warning
    y = (j*1.0) / i;  // warning...
    k = (j*1.0) / i;  // warning...
    printf("x = %.2f, y = %.2f, k = %d\n", x, y, k);
    a = 'Z';
    printf("a = %c\nb = %c\n", a, b);
    // using secure version instead of strcpy(), previously
    // these functions generate a buffer overflow problems...
    // we have strcpy(dest, source), wide character version
    // - wcscpy(dest, source), secure version - wcscpy_s(dest, dest size, source) and
    // strcpy_s(dest, dest size, source) another one is multibyte version - _mbscpy(...), find it
yourself...
    strcpy_s(q, 20, r);
    strcpy_s(r, 20, "CopiedString");
    printf("q = %s\nr = %s\n", q, r);
    return 0;
}
```

A sample output:

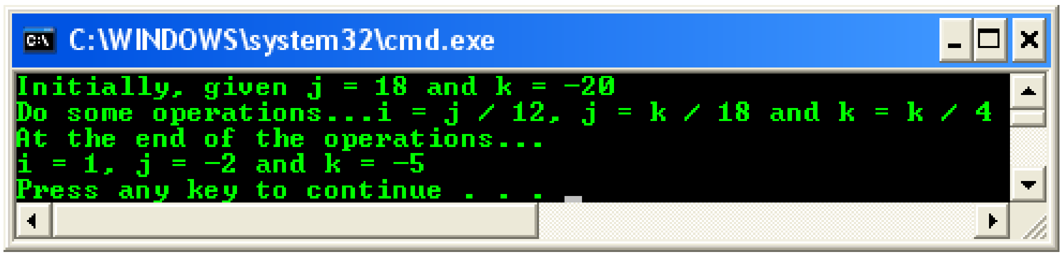


```
Declared and initilized some variables...
...and see the action...

i = 17, k = 16
x = 1.00, y = 1.33, k = 1
a = Z
b = U
q = Incredible Hulk
r = CopiedString
Press any key to continue . . .
```

- One must be careful when dividing two integers. The result will always be an integer, so that 14 divided by 3 is 4 else you need to use float or double. Show the output for the following program.

```
// needed for printf()
#include <stdio.h>

int main()
{
    // declare variables and initialize some of them
    // Some compiler ask you to initialize all the variables...
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    int i, j = 18, k = -20;
    printf("Initially, given j = 18 and k = -20\n");
    printf("Do some operations..."
           "i = j / 12, j = k / 18 and k = k / 4\n");
    i = j / 12;
    j = k / 8;
    k = k / 4;
    printf("At the end of the operations...\n");
    printf("i = %d, j = %d and k = %d\n", i, j, k);
    return 0;
}
```

A sample output:

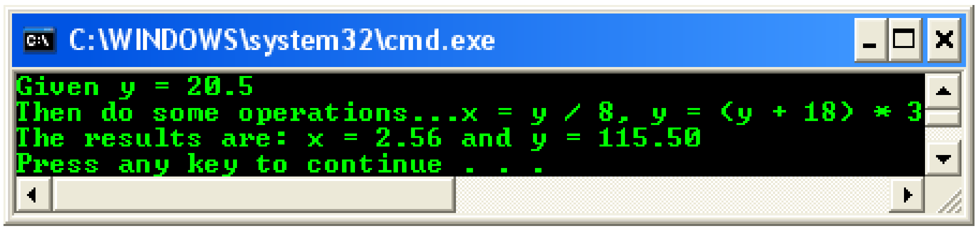


```
Initially, given j = 18 and k = -20
Do some operations...i = j / 12, j = k / 18 and k = k / 4
At the end of the operations...
i = 1, j = -2 and k = -5
Press any key to continue . . .
```

- To manipulate fractional numbers, we must declare variables of type float. Show the output for the following program.

```
// needed for printf()
#include <stdio.h>

int main()
{
    // declare variables and initialize some of them
    // some compiler ask you to initialize all the variables...and it is a good habit
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    float x, y = 20.5;
    printf("Given y = 20.5\n");
    printf("Then do some operations..."
        "x = y / 8, y = (y + 18) * 3\n");
    x = y / 8;
    y = (y + 18) * 3;
    printf("The results are: x = %.2f and y = %.2f\n", x, y);
    return 0;
}
```

A sample output:



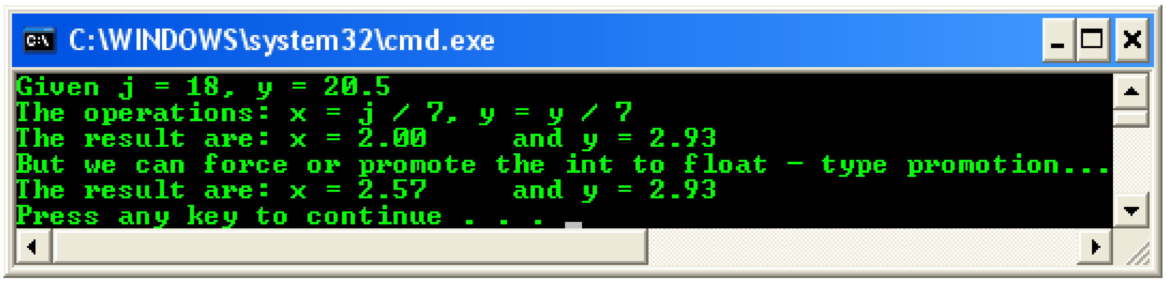
```
Given y = 20.5
Then do some operations...x = y / 8, y = (y + 18) * 3
The results are: x = 2.56 and y = 115.50
Press any key to continue . . .
```

- If an expression divides 10 by 4, the answer is 2 (integer) and not 2.5 (float). If 10 / 4 is assigned to a float, then 2 is converted to 2.0 before assign it to the float. In the following example we can force the int type to float but not vice versa. Show the output for the following program.

```
// needed for printf()
#include <stdio.h>

int main()
{
    // declare variables and initialize some of them
    // some compiler ask you to initialize all the variables...
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    int j = 18;
```

```
        float x, y = 20.5;
        printf("Given j = 18, y = 20.5\n");
        printf("The operations: x = j / 7, y = y / 7\n");
        x = j / 7;
        y = y / 7;
        printf("The result are: x = %.2f     and y = %.2f\n", x, y);
        printf("But we can force or promote the int to float - type promotion...\n");
        x = ((float) j / 7);
        printf("The result are: x = %.2f     and y = %.2f\n", x, y);
        return 0;
}
```

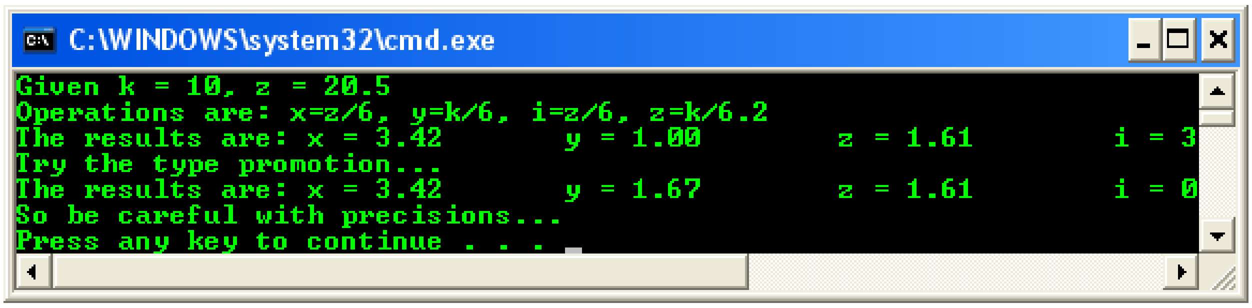
A sample output:


```
Given j = 18, y = 20.5
The operations: x = j / 7, y = y / 7
The result are: x = 2.00      and y = 2.93
But we can force or promote the int to float - type promotion...
The result are: x = 2.57      and y = 2.93
Press any key to continue . . .
```

- Try another program example.

```
// needed for printf()
#include <stdio.h>

int main()
{
        // declare variables and initialize some of them
        // some compiler ask you to initialize all the variables...
        // in this case just initialize them with dummy value..
        // for example: int x = 0, float y = 0.00 etc.
        int i, k = 10;
        float x, y, z = 20.5;
        printf("Given k = 10, z = 20.5\n");
        printf("Operations are: x=z/6, y=k/6, i=z/6, z=k/6.2\n");
        x = z / 6;       // Statement #1
        y = k / 6;      // Statement #2 - warning: int to float
        i = z / 6;      // Statement #3 - warning: float to int
        z = k / 6.2;  // Statement #4 - warning: int to float
        printf("The results are: x = %.2f\ty = %.2f\tz = %.2f\ti = %d\n", x, y, z, i);
        printf("Try the type promotion...\n");
        y = (float) k / 6;        // promote to float, OK
        i = (int) z / 6;            // promote to int, not OK...
        z = (float) k / 6.2;     // promote to float, OK
        printf("The results are: x = %.2f\ty = %.2f\tz = %.2f\ti = %d\n", x, y, z, i);
        printf("So be careful with precision...\n");
        return 0;
}
```

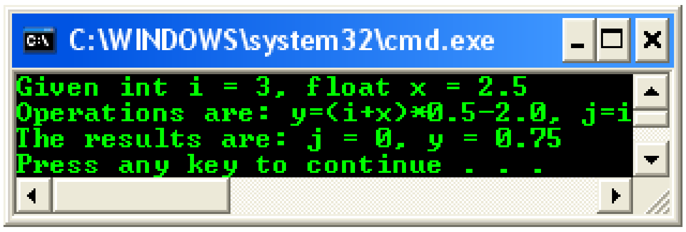
A sample output:


```
Given k = 10, z = 20.5
Operations are: x=z/6, y=k/6, i=z/6, z=k/6.2
The results are: x = 3.42        y = 1.00        z = 1.61        i = 3
Try the type promotion...
The results are: x = 3.42        y = 1.67        z = 1.61        i = 0
So be careful with precisions...
Press any key to continue . . .
```

- Statements 1, 3 and 4 have at least one float in their divisions, so the expressions evaluate to floats. x and z are float variables so they can store floats. i isn't a float variable and the whole number 3 is stored in it. Statement 2 uses an integer division, so the fraction portion is truncated.
- Try another program and show the output.

```
// needed for printf()
#include <stdio.h>

int main()
{
    // declare variables and initialize some of them
    // some compiler ask you to initialize all the variables...
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    int i = 3, j;
    float x = 2.5, y;
    printf("Given int i = 3, float x = 2.5\n");
    printf("Operations are: y=(i+x)*0.5-2.0, j=i\n");
    y = ((i + x) * 0.5) - 2.0; // warning
    j = y; // float to integer...failed!!!
    printf("The results are: j = %d, y = %.2f\n", j, y);
    return 0;
}
```

A sample output:

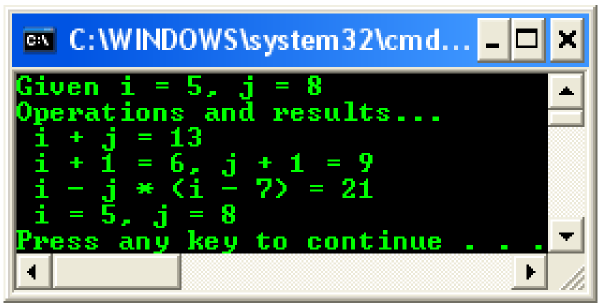

```
Given int i = 3, float x = 2.5
Operations are: y=(i+x)*0.5-2.0, j=i
The results are: j = 0, y = 0.75
Press any key to continue . . .
```

- Try next example, show the output.

```
// needed for printf()
#include <stdio.h>

int main()
{
    // declare variables and initialize some of them
    // some compiler ask you to initialize all the variables...
    // in this case just initialize them with dummy value..
    // for example: int x = 0, float y = 0.00 etc.
    int i = 5, j = 8;
    printf("Given i = 5, j = 8\n");
    printf("Operations and results...\n");
    printf(" i + j = %d\n", i + j);
```

```
        printf(" i + 1 = %d, j + 1 = %d\n", i + 1, j + 1);
        printf(" i - j * (i - 7) = %d\n", i - j * (i - 7));
        // operator precedence...which operator that compiler execute first?
        // 5 - 8 * (5  - 7) = 5 - 8 * (-2) = 5 - (-16) = 21
        // better to use parentheses: i - (j * (i - 7))
        // start from the innermost then to the outermost of the parentheses
        printf(" i = %d, j = %d\n", i, j);
        return 0;
}
```

A sample output:



```
Given i = 5, j = 8
Operations and results...
 i + j = 13
 i + 1 = 6, j + 1 = 9
 i - j * (i - 7) = 21
 i = 5, j = 8
Press any key to continue . . .
```

| Main |< C main() and printf() functions 2 | C/C++ Variable, Operator & Type 2 >| Site Index | Download |

The C Variables, Operators and Data Types: Part 1 | Part 2 | Part 3