02. Operations

02.1 Arithmetic Operators

C supports the following arithmetic operators for numbers: `short`, `int`, `long`, `long long`, `char` (treated as 8-bit signed integer), `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `unsigned char`, `float`, `double` and `long double`.

Operator	Description	Usage	Examples
*	Multiplication	*expr1 * expr2*	2 * 3 → 6; 3.3 * 1.0 → 3.3
/	Division	*expr1 / expr2*	1 / 2 → 0; 1.0 / 2.0 → 0.5
%	Remainder (Modulus)	*expr1 % expr2*	5 % 2 → 1; -5 % 2 → -1
+	Addition	*expr1 + expr2*	1 + 2 → 3; 1.1 + 2.2 → 3.3
-	Subtraction	*expr1 - expr2*	1 - 2 → -1; 1.1 - 2.2 → -1.1

All the above operators are *binary* operators, i.e., they take two operands. The multiplication, division and remainder take *precedence* over addition and subtraction. Within the same precedence level (e.g., addition and subtraction), the expression is evaluated from left to right. For example, `1+2+3-4` is evaluated as `((1+2)+3)-4`.

It is important to take note that `int/int` produces an `int`, with the result *truncated*, e.g., `1/2 → 0` (instead of `0.5`).

Take note that C does not have an exponent (power) operator (`'^'` is exclusive-or, not exponent).

02.2 Arithmetic Expressions

In programming, the following arithmetic expression:

$$\frac{1+2a}{3} + \frac{4(b+c)(5-d-e)}{f} - 6\left(\frac{7}{g}+h\right)$$

must be written as `(1+2*a)/3 + (4*(b+c)*(5-d-e))/f - 6*(7/g+h)`. You cannot omit the multiplication symbol `'*'` (as in Mathematics).

Like Mathematics, the multiplication `'*'` and division `'/'` take precedence over addition `'+'` and subtraction `'-'`. Parentheses `()` have higher precedence. The operators `'+'`, `'-'`, `'*'`, and `'/'` are *left-associative*. That is, `1 + 2 + 3 + 4` is treated as `(((1+2) + 3) + 4)`.

02.3 Mixed-Type Operations

If both the operands of an arithmetic operation belong to the *same type*, the operation is carried out in that type, and the result belongs to that type. For example, `int/int → int; double/double → double`.

However, if the two operands belong to *different types*, the compiler promotes the value of the *smaller* type to the *larger* type (known as *implicit type-casting*). The operation is then carried out in the *larger* type. For example, `int/double → double/double → double`. Hence, `1/2 → 0, 1.0/2.0 → 0.5, 1.0/2 → 0.5, 1/2.0 → 0.5`.

For example,

Type	Example	Operation
int	2 + 3	int 2 + int 3 → int 5

double	2.2 + 3.3	double 2.2 + double 3.3 ⟶ double 5.5
mix	2 + 3.3	int 2 + double 3.3 ⟶ double 2.0 + double 3.3 ⟶ double 5.3
int	1 / 2	int 1 / int 2 ⟶ int 0
double	1.0 / 2.0	double 1.0 / double 2.0 ⟶ double 0.5
mix	1 / 2.0	int 1 / double 2.0 ⟶ double 1.0 + double 2.0 ⟶ double 0.5

Example

```
/* Testing mix-type arithmetic operations (TestMixTypeOp.c) */
#include <stdio.h>

int main() {
   int i1 = 2, i2 = 4;
   double d1 = 2.5, d2 = 5.2;

   printf("%d + %d = %d\n", i1, i2, i1+i2);          // 2 + 4 = 6
   printf("%.1lf + %.1lf = %.1lf\n", d1, d2, d1+d2); // 2.5 + 5.2 = 7.7
   printf("%d + %.1lf = %.1lf\n", i1, d2, i1+d2);    // 2 + 5.2 = 7.2  <== mix
type

   printf("%d / %d = %d\n", i1, i2, i1/i2);          // 2 / 4 = 0  <== NOTE:
truncate
   printf("%.1lf / %.1lf = %.2lf\n", d1, d2, d1/d2); // 2.5 / 5.2 = 0.48
   printf("%d / %.1lf = %.2lf\n", i1, d2, i1/d2);    // 2 / 5.2 = 0.38  <== mix
type
   return 0;
}
```

02.4 Overflow/UnderFlow

Study the output of the following program:

```
/* Test Arithmetic Overflow/Underflow (TestOverflow.c) */
#include <stdio.h>

int main() {
  // Range of int is [-2147483648, 2147483647]
   int i1 = 2147483647;      // max int
   printf("%d\n", i1 + 1);   // -2147483648 (overflow)
   printf("%d\n", i1 + 2);   // -2147483647
   printf("%d\n", i1 * i1);  // 1

   int i2 = -2147483648;     // min int
   printf("%d\n", i2 - 1);   // 2147483647 (underflow)
   printf("%d\n", i2 - 2);   // 2147483646
```

```
   printf("%d\n", i2 * i2); // 0
   return 0;
}
```

In arithmetic operations, the resultant value *wraps around* if it exceeds its range (i.e., overflow or underflow). C runtime does not issue an error/warning message but produces *incorrect* result.

It is important to take note that *checking of overflow/underflow is the programmer's responsibility*, i.e., your job!

This feature is an legacy design, where processors were slow. Checking for overflow/underflow consumes computation power and reduces performance.

To check for arithmetic overflow (known as *secure coding*) is tedious. Google for "INT32-C. Ensure that operations on signed integers do not result in overflow" @ www.securecoding.cert.org.

02.5 Compound Assignment Operators

Besides the usual simple assignment operator `'='` described earlier, C also provides the so-called *compound assignment operators* as listed:

Operator	Usage	Description	Example
=	*var = expr*	Assign the value of the LHS to the variable at the RHS	x = 5;
+=	*var += expr*	same as *var = var + expr*	x += 5; same as x = x + 5
-=	*var -= expr*	same as *var = var - expr*	x -= 5; same as x = x - 5
*=	*var *= expr*	same as *var = var * expr*	x *= 5; same as x = x * 5
/=	*var /= expr*	same as *var = var / expr*	x /= 5; same as x = x / 5
%=	*var %= expr*	same as *var = var % expr*	x %= 5; same as x = x % 5

02.6 Increment/Decrement Operators

C supports these *unary* arithmetic operators: increment `'++'` and decrement `'--'`.

Operator	Example	Result
++	x++; ++x	Increment by 1, same as x += 1
--	x--; --x	Decrement by 1, same as x -= 1

Example

```
/* Test on increment (++) and decrement (--) Operator (TestIncDec.cpp) */
#include <stdio.h>

int main() {
   int mark = 76;           // declare & assign
   printf("%d\n", mark); // 76

   mark++;                  // increase by 1 (post-increment)
   printf("%d\n", mark); // 77

   ++mark;                  // increase by 1 (pre-increment)
   printf("%d\n", mark); // 78
```

```

   mark = mark + 1;       // also increase by 1 (or mark += 1)
   printf("%d\n", mark); // 79

   mark--;                // decrease by 1 (post-decrement)
   printf("%d\n", mark); // 78

   --mark;                // decrease by 1 (pre-decrement)
   printf("%d\n", mark); // 77

   mark = mark - 1;       // also decrease by 1 (or mark -= 1)
   printf("%d\n", mark); // 76
   return 0;
}
```

The increment/decrement unary operator can be placed before the operand (prefix operator), or after the operands (postfix operator). They takes on different meaning in operations.

Operator	Description	Example	Result
++var	Pre-Increment Increment *var*, then use the new value of *var*	y = ++x;	same as x=x+1; y=x;
var++	Post-Increment Use the old value of *var*, then increment *var*	y = x++;	same as oldX=x; x=x+1; y=oldX;
--var	Pre-Decrement	y = --x;	same as x=x-1; y=x;
var--	Post-Decrement	y = x--;	same as oldX=x; x=x-1; y=oldX;

If '++' or '--' involves another operation, then pre- or post-order is important to specify the order of the two operations. For examples,

```
x = 5;
printf("%d\n", x++);  // Save x (5); Increment x (=6); Print old x (5).
x = 5;
printf("%d\n", ++x);  // Increment x (=6); Print x (6).
// This is confusing! Try to avoid! What is i=++i? What is i=i++?
```

Prefix operator (e.g, `++i`) could be more efficient than postfix operator (e.g., `i++`) in some situations.

02.7 Implicit Type-Conversion vs. Explicit Type-Casting

Converting a value from one type to another type is called *type casting* (or *type conversion*). There are two kinds of type casting:

1. Implicit type-conversion performed by the compiler automatically, and
2. Explicit type-casting via an unary *type-casting operator* in the form of (*new-type*)*operand*.

Implicit (Automatic) Type Conversion

When you assign a value of a fundamental (built-in) type to a variable of another fundamental type, C automatically converts the value to the receiving type, if the two types are compatible. For examples,

- If you assign an `int` value to a `double` variable, the compiler automatically casts the `int` value to a `double` double (e.g., from 1 to 1.0) and assigns it to the `double` variable.
- if you assign a `double` value of to an `int` variable, the compiler automatically casts the `double` value to an `int` value (e.g., from 1.2 to 1) and assigns it to the `int` variable. The fractional part would be truncated and lost. Some compilers issue a warning/error "possible loss in precision"; others do not.

```
/*
 * Test implicit type casting (TestImplicitTypeCast.c)
 */
#include <stdio.h>

int main() {
   int i;
   double d;

   i = 3;
   d = i;    // Assign an int value to double
   printf("d = %lf\n", d); // d = 3.0

   d = 5.5;
   i = d;    // Assign a double value to int
   printf("i = %d\n", i); // i = 5 (truncated, no warning!)

   i = 6.6; // Assign a double literal to int
   printf("i = %d\n", i); // i = 6 (truncated, no warning!)
}
```

C will not perform automatic type conversion, if the two types are not compatible.

Explicit Type-Casting

You can explicitly perform type-casting via the so-called unary *type-casting operator* in the form of (*new-type*)*operand*. The type-casting operator takes one operand in the particular type, and returns an equivalent value in the new type. Take note that it is an operation that yields a resultant value, similar to an addition operation although addition involves two operands. For example,

```
printf("%lf\n", (double)5); // int 5 -> double 5.0
printf("%d\n", (int)5.5);   // double 5.5 -> int 5

double aDouble = 5.6;
int anInt = (int)aDouble; // return 5 and assign to anInt. aDouble does not change!
```

Example: Suppose that you want to find the average (in `double`) of the integers between `1` and `100`. Study the following codes:

```
/*
 * Testing Explicit Type Cast (Average1to100.c).
 */
#include <stdio.h>

int main() {
   int sum = 0;
   double average;
   int number = 1;
   while (number <= 100) {
      sum += number;      // Final sum is int 5050
      ++number;
   }
   average = sum / 100;   // Won't work (average = 50.0 instead of 50.5)
   printf("Average is %lf\n", average); // Average is 50.0
   return 0;
}
```

You don't get the fractional part although the `average` is a `double`. This is because both the `sum` and `100` are `int`. The result of division is an `int`, which is then implicitly casted to `double` and assign to the `double` variable `average`. To get the correct answer, you can do either:

```
average = (double)sum / 100;     // Cast sum from int to double before division
average = sum / (double)100;     // Cast 100 from int to double before division
average = sum / 100.0;
average = (double)(sum / 100);   // Won't work. why?
```

Example:

```
/*
 * Converting between Celsius and Fahrenheit (ConvertTemperature.c)
 *   Celsius = (5/9)(FahrenheitɃ32)
 *   Fahrenheit = (9/5)Celsius+32
 */
#include <stdio.h>

int main() {
   double celsius, fahrenheit;

```

```
  printf("Enter the temperature in celsius: ");
  scanf("%lf", &celsius);
  fahrenheit = celsius * 9 / 5 + 32;
 // 9/5*celsius + 32 gives wrong answer! Why?
  printf("%.2lf degree C is %.2lf degree F\n", celsius, fahrenheit);

  printf("Enter the temperature in fahrenheit: ");
  scanf("%lf", &fahrenheit);
  celsius =  (fahrenheit - 32) * 5 / 9;
 // 5/9*(fahrenheit - 32) gives wrong answer! Why?
  printf("%.2lf degree F is %.2lf degree C\n", fahrenheit, celsius);
  return 0;
}
```

02.8 Relational and Logical Operators

Very often, you need to compare two values before deciding on the action to be taken, e.g., if mark is more than or equal to 50, print "PASS".

C provides six *comparison operators* (or *relational operators*):

Operator	Description	Usage	Example (x=5, y=8)
==	Equal to	*expr1 == expr2*	(x == y) → false
!=	Not Equal to	*expr1 != expr2*	(x != y) → true
>	Greater than	*expr1 > expr2*	(x > y) → false
>=	Greater than or equal to	*expr1 >= expr2*	(x >= 5) → true
<	Less than	*expr1 < expr2*	(y < 8) → false
<=	Less than or equal to	*expr1 >= expr2*	(y <= 8) → true

Each comparison operation involves two operands, e.g., `x <= 100`. It is invalid to write `1 < x < 100` in programming. Instead, you need to break out the two comparison operations `x > 1`, `x < 100`, and join with with a logical AND operator, i.e., `(x > 1) && (x < 100)`, where && denotes AND operator.

C provides four logical operators:

Operator	Description	Usage
&&	Logical AND	*expr1 && expr2*
||	Logical OR	*expr1 || expr2*
!	Logical NOT	*!expr*
^	Logical XOR	*expr1 ^ expr2*

The truth tables are as follows:

AND (&&)	true	false
true	true	false
false	false	false

OR (||)	true	false
true	true	true
false	true	false

NOT (!)	true	false
	false	true

XOR (^)	true	false
true	false	true
false	true	false

Example:

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)
// wrong to use 0 <= x <= 100

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)   //or
!((x >= 0) && (x <= 100))

// Return true if year is a leap year
// A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Exercise: Given the year, month (1-12), and day (1-31), write a boolean expression which returns true for dates before October 15, 1582 (Gregorian calendar cut over date).

Ans: `(year < 1582) || (year == 1582 && month < 10) || (year == 1582 && month == 10 && day < 15)`