Types, Operators and Expressions

Variable Names

some restriction on the names of variables and symbolic constants:

- made up of **letters**, **digits** and **underscore**
- the first character must be a **letter** or **underscore**
- upper case and lower case letters are distinct
- cannot use **keywords** as variables names

Data Types and Sizes

`char`	**a single byte, capable of holding one character in the local character set.**
`int`	**an integer, typically reflecting the natural size of integers on the host machine.**
`double`	**double-precision floating point.**
`float`	**single-precision floating point.**

Qualifiers

`short` and `long`

- **`short int`**
- **`long int`**
- **`long double`**

`short int`(16 bits) ≤ **`int`**(16 or 32 bits) ≤ **`long int`**(32 bits)

`signed` and `unsigned`

- **`signed char`**
- **`unsigned char`**
- **`signed (long/short) int`**

- **`unsigned (long/short) int`**

Whether plain chars are signed are signed or unsigned is machine-dependent, but printable characters are always positive.

> The standard headers **`<limits.h>`** and **`<float.h>`** contain symbolic constants for all of these sizes, along with other properties of the machine and compilers.

Constants

- integer constants

```
int int_constant = 12345;
```

- long constants

```
long long_constant = 123456789L;
long long_constant_2 = 1234567881;
```

- unsigned constants

```
unsigned unsigned_constant = 201802152330uL;
```

- float-point constants

```
double d1 = 3.14;
double d2 = 3e8;
float f1 = 3.14f;
float f2 = 3.14F;
double d3 = 3.141;
double d4 = 3e8L;
```

- octal constants

```
char char_A = 0101u;
int octal = 02333;
```

- hexadecimal constants

```
int hex1 = 0x1f;
unsigned hex2 = 0x1fu;
long hex3 = 0x1fl;
unsigned long hex4 = 0xful;
```

- character constants

```c
char ch1 = 65;
char ch2 = 'A';
char ch3 = '\101';
char ch4 = '\x41';

printf("%c %c %c %c\n", ch1, ch2, ch3, ch4);
```

- set of escape sequence:
  - **\a** alert (bell) character
  - **\b** backspace
  - **\f** formfeed
  - **\n** newline
  - **\r** carriage return
  - **\t** horizontal tab
  - **\v** vertical tab
  - **** backslash
  - **\?** question mark
  - **\'** single quote
  - **\"** double quote
  - **\ooo** octal number
  - **\xhh** hexadecimal number
- constant expression

```c
#define MAX_CAPACITY 10000;
'x';
"x";
enum escapes {
    BELL = '\a', BACKSPACE = '\b', TAB = '\t',
    NEWLINE = '\n', VTAB = '\v', RETURN = '\r'
};
```

Declarations

- all variables must be declared before use.
- external and static variables are initialized to zero by default
- automatic variables for which there is no explicit initializer have undefined values
- **const** specifies that variable will not be changed

Arithmetic Operators

+, -, *, /, %

- **%** cannot be applied to **float** or **double**
- The direction of truncation for **/** and the sign of the result for **%** are machine-dependent for negative operands, as is the action taken on overflow or underflow
- Arithmetic operators associate left to right

- `%` ≤ `+, -` ≤ `*, /`

Relational and Logical Operators

relational operators: `>, >=, <, <=`

equality operators: `==, !=`

- relational operators have lower precedence than arithmetic operators

logical operators: `&&, ||, !`

- The precedence of `&&` is higher than `||`
- logical operators < equality operators < relation operators
- `!` converts a non-zero operand into 0 and a zero operand into 1.

Type conversions

Automatic Conversions

- `char` is a small integer, so it may be freely used in arithmetic
- Whether a conversion from a `char` to a `int` produces a signed or unsigned quantities is machine-dependent.
- Any characters in the machine's standard printing character set will never be negative.
- Relational expressions and logical expressions connected by `&&` or `||` are defined to have value `1` if true, and `0` if false.
- Functions may return non-zero value for true.
- implicit arithmetic conversions without unsigned operands
  - if either operand is `long double`, convert the other to `long double`.
  - Otherwise, if either operand is `double`, convert the other to `double`.
  - Otherwise, if either operand is `float`, convert the other to `float`.
  - Otherwise, convert `char` and `short` to `int`.
  - Then, if either operand is `long`, convert the other to `long`
- conversions with `unsigned` operands is complicated
- conversions take place across assignments.
  - The value of right side is converted the type of the left, which is the type of the result.
  - Longer integers are converted to shorted ones or to `chars` by dropping the excess high-order bits.
  - `float` to `int` causes truncation of any fractional part.
  - When `double` is converted to `float`, whether the value is rounded or truncated is implementation-dependent.
- conversions in function
  - Since an argument of a function call is an expression, type conversions also take place when arguments are passed to functions
  - In the absence of a function prototype, `char` and `short` become `int`, and `float` becomes `double`.

Forced Conversions

`(type name) expression`

Increment and Decrement Operators

`++, --`

- prefix operators increment variables before their value is used.
- postfix operators decrement variables after their value is used.
- The increment and decrement operators can only be applied to variables.

```
1  int i = 1, j = 2;
2  int k = (i + j)++; // error: lvalue required as increment operand.
```

Bitwise Operators

operators	state
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exlusive OR
<<	left shift
>>	right shift
~	one's compliment (unary)

- right shift
  - right shift a signed quantity will fill with sign bits ("arithmetic shift") on some machines
  - and with 0-bits ("logical shift") on others.

```
1  /* getbits: get n bits from position p */
2  unsigned getbits(unsigned x, int p, int n) {
3      return (x >> p + 1 - n) & ~(~0 << n);
4  }
```

Assignment Operators and Expressions

If $expr_1$ and $exper_2$ are expressions, then

$$expr_1\ op = expr_2$$

is equivalent to

$$expr_1 = (expr_1)\ op\ (expr_2)$$

where op is one of

`+ - * / % << >> & ^ |.`

Conditional Expressions

```
if (a > b)
    z = a;
else
    z = b;

z = (a > b)? a : b;

float f;
int n;
(n > 0)? f : n; // is of type float regradless of whether n is positive.
```

Precedence and Order of Evaluation

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

OPERATORS	ASSOCIATIVITY
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

- Unary `+`, `-`, and `*` have higher precedence than the binary forms.