Types, Operators and Expressions

Ritankar Mandal

Data Types

Type	Description	Size (bits)	Format specifier
`char`	holds one character	8	%c
`int`	holds one integer	16	%d
`float`	single-precision floating point	32	%f
`double`	double-precision floating point	64	%lf

Qualifiers: short & long

Qualifiers `short` and `long` can be applied to integers.

```
1 short int sh;
2 long int counter;
```

The sizes depend on the compiler for its own hardware, subject to the restriction that

1. `short` and `int` are at least 16 bits,
2. `long` is at least 32 bits,
3. `short` is no longer than `int`, which is no longer than `long`.

Qualifiers: signed & unsigned

1. signed variables can be positive, zero or negative. So they have the range between (-2^{n-1}) and $2^{n-1} - 1$ (in a two's complement machine), where n is the number of bits in the type. E.g., signed chars have values between -128 and 127

2. unsigned numbers are always positive or zero. So they obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255.

Declaration

All variables must be declared before use.

```
1 int lower;
2 int upper;
3 char c;
4 char line[1000];
```

You can combine all the variables of the same type to a single line.

```
1 int lower, upper;
2 char c, line[1000];
```

Declaration & Initialization

```
1 /*Declaration*/
2 char c;
3 int i;
4 /*Initialization*/
5 c = 'a';
6 i = 0;
```

A variable may also be initialized in its declaration.

```
1 char c = 'a';
2 int i = 0;
```

Qualifier: const

The qualifier `const` can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the `const` qualifier says that the elements will not be altered.

The following code will generate an error (implementation-defined).

```
1 const msg[] = "Hello";
2 const int i = 10;
3 printf("i = %d\n", i);
4 i = 20; // Trying to change the value of a const variable
5 printf("i = %d\n", i);
```

Operators: Arithmetic, Relational and Logical

Binary arithmetic operators are $+, -, *, /, \%$ (the modulus operator). The % operator cannot be applied to a `float` or `double`.

Example: If a year is divisible by 4 but not by 100, or by both 100 and 400 is a leap year.

```
1 if(((year%4 == 0) && (year%100 != 0)) || (year%400 == 0))
2     printf("%d is a leap year. \n", year);
3 else
4     printf("%d is not a leap year. \n", year);
```

Relational operators are $>, >=, <, <=, ==, !=$. Logical operators are && and ||.

Relational operators have lower precedence than arithmetic operators, so $i < lim - 1$ is taken as $i < (lim - 1)$.

Integer division

Integer division truncates any fractional part.

Check the following example to find out what erroneous results might be generated by using it.

```
1  int a = 1, b = 2;
2  float c = 1.0, d = 2.0;
3  float e, f, g, h;
4
5  e = a/b; /* Integer Division */
6  printf("a/b = %f\n", e);
7  f = c/b;
8  printf("c/b = %f\n", f);
9  g = a/d;
10 printf("a/d = %f\n", g);
11 h = (float)a/b; /* Type casting Integer to Float */
12 printf("(float)a/b = %f\n", h);
```

A common mistake

```
1  int a = 9, b = 0;
2  if(b = 1)
3      printf("b = %d.\n", b);
4  if((a == 10) || (b = 2))
5      printf("a = %d. b = %d.\n", a, b);
```

A common mistake

```
1  int a = 9, b = 0;
2  if(b = 1)
3      printf("b = %d.\n", b);
4  if((a == 10) || (b = 2))
5      printf("a = %d. b = %d.\n", a, b);
```

Expressions connected by && or || are evaluated from left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules.

A "narrower" operand is converted into a "wider" one without losing information automatically. Assigning a longer integer type to a shorter, or a `float` to an `int`, may draw a warning, but they are not illegal.
`float` to `int` causes truncation of any fractional part. In `double` to `float` conversion, the value is rounded or truncated depending on the implementation.

`char` is a small integer and can be used in arithmetic expressions.

```
1 char s[] = "329";
2 int i, n = 0;
3 for(i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
4     n = 10 * n + (s[i] - '0');
5 printf("n = %d \n", n);
```

Increment and Decrement Operators

$++n$ **(prefix):** Increments n before its value is used.
$n++$ **(postfix):** Increments n after its value is used.

```
1 int n, x;
2 n = 5;
3 x = n++;
4 printf("n = %d, x = %d\n", n, x);
5 n = 5;
6 x = ++n;
7 printf("n = %d, x = %d\n", n, x);
```

Bitwise Operators

Operator	Description
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

These can only be applied to integral operands, i.e., `char`, `short`, `int`, `long`, whether `signed` or `unsigned`.

Assignment Operators

`i += 2` is the same as `i = i + 2`.

`x *= y + 1` is the same as `x = x * (y + 1)`, but not `x = x * y + 1`.

Conditional Expressions

In the expression

$$expr1 \ ? \ expr2 \ : \ expr3$$

expr1 is evaluated first. If it is non-zero (true), then *expr2* is evaluated, else *expr3* is evaluated.

```
1 /* z = max(a, b) */
2 if (a > b)
3     z = a;
4 else
5     z = b;
```

is the same as

```
1 z = (a > b) ? a : b; /* z = max(a, b) */
```

Precedence and Associativity of Operators

Operators on the same line have the same precedence; rows are in order of decreasing precedence. For example, $*$, $/$, % all have the same precedence, which is higher than that of binary $+$ and $-$.

The "operator" () refers to function call. The operators $->$ and . are used to access members of structures, along with sizeof (size of an object), $*$ (indirection through a pointer) and & (address of an object).

Unary operators have higher precedence than the binary forms.

Precedence and Associativity of Operators

Operators	Associativity
(), [], $->$, .	left to right
!, $\sim$, ++, $--$, +, $-$, $*$, (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
||	left to right
?:	right to left
$=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $\sim=$, $\|=$, $<<=$, $>>=$	right to left
,	left to right

Associativity of Operators

Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

```
1 float a = 100.0 / 10.0 * 10.0;
2 float b = 10.0 * 10.0 / 100.0;
3 printf("a = %f, b = %f\n", a, b);
```

Associativity of Operators

```
1  int a = 10, b = 20, c = 30;
2  if (c > b > a)
3      printf("TRUE");
4  else
5      printf("FALSE");
```

Associativity of Operators

```
1  int a = 10, b = 20, c = 30;
2  if (c > b > a)
3      printf("TRUE");
4  else
5      printf("FALSE");
```

Associativity of '$>$' is left to right, so $(c > b > a)$ is treated as $((c > b) > a)$. Therefore the expression becomes $((30 > 20) > 10)$ which becomes $(1 > 10)$.

Precedence and Associativity of Operators

$x = f() + g();$. $f()$ may be evaluated before $g()$ or vice versa.

```
 1 #include <stdio.h>
 2 int i = 0;
 3 int f(){
 4     i = 1;
 5     return i;
 6 }
 7 int g(){
 8     i = 2;
 9     return i;
10 }
11 int main(){
12     int a = f() + g();
13     printf("%d ", i);
14     return 0;
15 }
```

Conclusion

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don't know how they are done on various machines, you won't be tempted to take advantage of a particular implementation.