CS 240 Programming in C

Variable Names, Elementary Types, Bit Operations

September 25, 2019

Variable Names

- Made up of letters and digits
- Underscore (_) counts as a letter
- This is useful for improving readability of long names
- Note: Do not begin a variable name with underscore
  _names are reserved for library routines
- Variable names are case-sensitive
  Uppercase and lowercase letters are different
- The first character in the name must be a letter

C Keywords

- Reserved by C language, cannot be used as variable names
- External variables: function names, global variables
- Global variables are defined outside of { }
- Internal variables: variables defined within { }
- At the time of The C Book's publication, the first 31 characters of an internal variable are significant
- The first 6 monocase characters of external variables are significant

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

Integer Data Types

- Traditional C integer types: `char`, `int`, `short`, `long`
- They can be signed or unsigned
- ISO C99 uses a different format:
  `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- Example: 32-bit integers

$$0 \leq \texttt{uint32\_t} \leq 2^{32} - 1$$
$$-2^{31} \leq \texttt{int32\_t} \leq 2^{31} - 1$$

- Be careful of overflow and underflow

Integer Constants

- 12345
- Long 12345L (avoid 12345l with a lowercase l)
- Unsigned long 12345UL
- Octal numbers are preceded by a 0 (zero)
  Do not start a decimal number with a 0
- Hexadecimal numbers are preceded by 0x or 0X
  Use a, b, c, d, e, f for $10, \ldots, 15$, or capital letters
- `0xABCD` or `0Xabcd`
- '[c]'.(where c is a character).
  For example: 'c', '\n', '\012','\xab'.

Floating Point Numbers

- `float, double, long double`
- 32, 64, and 96 bits
- Floating point constants:
  12345. (note the decimal point)
  1.2345e6
- Floating point constants are stored as a double
- If float is desired, write 1234.5F
- See `float.h` for the values of minimum and maximum floats:
  `FLT_MIN`, `FLT_MAX`

Character and String Constants

- Char constants: 'a', '0', '\n'
- String constants: "hello", "x"
- Note 'x' does not equal "x"
- 'x' is a char, taking one byte
- "x" is a char array, taking two bytes
- integer between [0,128]
- '\012' and '\xaa'

Enumeration

```
enum boolean {FALSE, TRUE};

enum days {SUN, MON, TUE, WED, THU, FRI, SAT};

enum months {JAN = 1, FEB, MAR, APR, MAY,
  JUN, JUL, AUG, SEP, OCT, NOV, DEC};

enum boolean done = FALSE;

while (!done) {
  //do something
  //update the value of done
}
```

The const Declaration

- We can put const before a declaration to indicate that the value of the variable will not change

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";

int strlen(const char str[]);
```

- The result is implementation-defined
- Use const when you know for sure you are not going to change the value
- Help the compiler and OS to optimize your code and secure the system

Arithmetic Operator

```
 +
 -
 *
 /
 %    Modulus    10 % 2 = 0; 10 % 3 = 1.
++
--
```

Relational and Logic Operator

Relational	Logic
==	||
!=	&&
<	!
>	
<=	
>=	

Type Casting

There are two kinds of type casting in C.

1. Automatic conversions or implicit conversions.
2. Explicit type conversions or forced type castings.

Automatic Type Casting

- In general, if an operator like + or * that takes two operands (a binary operator) has operands of different types, the "lower" type is promoted to the "higher" type before the operation proceeds. The result is of the higher type.
- Section 6 of Appendix A states the conversion rules precisely.
- Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

Implicit Arithmetic Conversion

```
char -> short int -> int ->
unsigned int -> long -> long long
-> float -> double -> long double
```

- Generally promote narrow type to broader type to keep information.
- Conversions get off when overflow or underflow happens for signed types. For example, an signed char overflowed then gets converted into integer.
- In fact, the overflow and underflow actually is a conversion from signed to unsigned, and then unsigned to others.
- In all, conversion rules get more complicated when unsigned operands are involved and it may lose information. Another example,

Example

```
/* Here is another example form book concerned about
   signed long promoted to unsigned long.
*/

    /* Explain what happens and the result */
     printf("-1L < 1UL: %d\n", -1L < 1UL );

     /* Explain what happens and the result */
     printf("-1L < 1U : %d\n", -1LL < 1U);
```

Sign Extension

- When casting a smaller signed data type to a larger one, sign extension will duplicate the Most Significant Bit in this case.
- For positives there are all 0s.
- For negatives there are all 1s, which is consistent with negative's 2' complement.

  Example, let's set short 2bytes, int 4bytes;

```
short i = -1;          i:  FFFF
int j = i;             j:  FFFF,FFFF
```

```
(The sign of 1 get's extended
 and consistant with 2' complement for negatives
```

```
Note, it is hard to do demo, since printf has no
unsigned short or unsigned int data type to print out.
It has only unsigned decimal.
This time you have to just trust me.
```

Explicit Type Casting

- (type-name) expression, like (int) 3.14.
- This is the times when longer data type cut into narrower data type happens, besides the assign expression.
- For integers, the extra most significant bits are just cut out.
- For floating numbers to integer, the fraction part is cut out.
- Conversions between integer and floating have to use explicit casting, or will fail compiling.

Random number generator

```
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
     next = next * 1103515245 + 12345;
     return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
     next = seed;
}
```

Truth Tables

`p && q`

	0	1
0	0	0
1	0	1

`p || q`

	0	1
0	0	1
1	1	1

When p and q Are Bits

`sum of p + q`

`p XOR q`

	0	1
0	0	1
1	1	0

`carry of p + q`

	0	1
0	0	0
1	0	1

Bitwise Operations

- & bitwise AND
- | bitwise inclusive OR
- ^ bitwise exclusive OR
- << left shift
- >> right shift
- ~ one's complement
- – two's complement

One's Complement

- ~
- Takes the logical NOT of each bit in the operand
- Flips the value of each bit in the operand's value
- Zeros become ones
- Ones become zeros
- Example: `~10101010 == 01010101`

Two's Complement

- -
- The two's complement operator is the negative sign
- It is a unary operator that performs the following steps:

1. Take the one's complement
2. Add 1

- The two's complement generates the negative of the original value
- Example: `0x55`, 0, -2^7

Bitwise AND, Inclusive OR, Exclusive OR

- Take the logical AND, OR, and XOR
- Apply to each pair of bits

```
  01001000       01001000       01001000
& 10111000     | 10111000     ^ 10111000
----------     ----------     ----------
  00001000       11111000       11110000
```

Left Shift <<

- Form: operand << numOfBitPositions
- Shift the bits in operand to the left
- Bits that fall off the left side disappear
- 0's are shifted in from the right
- The operand is usually an unsigned integer
- The number of bit positions must be *positive*
- The value of `i << 0` is not defined
- Example: `0x55 << 3`
- Left-shifting is the same as multiplying by a power of 2

Right Shift >>

- Form: operand >> numOfBitPositions
- Shift the bits in operand to the right
- Bits that fall off the right side disappear
- If the operand is unsigned, 0's are shifted in from the left
- If the operand is signed, right shifting will fill with sign bits ("arithmetic shift") on some machines and with O-bits ("logical shift") on others. (Based on implementation)
- The number of bit positions must be *positive*
- The value of `i >> 0` is not defined
- Example: `0x55 >> 1`
- Right-shifting of unsigned is the same as dividing by a power of 2

Assignment Operators and Expressions

```
        expr1 op = expr2
<==>    expr1 = (expr1) op (expr2)

        x *= y+1
<==>    x = x * (y+1)
```

ternary operator ?

```
expr1 ? expr2 : expr3
```

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

<==>

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