# Engineering Computing I

The C programming Language

Chapter 2

Types, Operators, and Expressions

## Chapter 2

- **Types**
- **Operators**
- **Expressions**

# 2.1 Variable Names

- Names are made up of letters and digits
- The first character must be a letter
- The underscore ‘‘_’’ counts as a letter
- Don’t begin variable names with ‘‘_’’, reserved library routines
- Upper and lower case letters are distinct
- Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants



# 2.1 Variable Names *continued*

- At least the first 31 characters of an internal name are significant
- Keywords like *if, else, int, float, etc.*, are reserved
- It’s wise to choose variable names that are related to the purpose of the variable and that are unlikely to get mixed up typographically

# Different Types

| Type | Description |
|---|---|
| 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 |
| float | single-precision floating point |
| double | double-precision floating point |



# Other Qualifiers

- short
- long

- short int
- long int

# Other Qualifiers

- signed
- unsigned

- signed char
- unsigned int



# Floating Point

In binary
$\pm 1.xxxxxxx_2 \times 2^{yyyy}$
Types `float` and `double` in C

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 \times 10^9$
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: sign bit | 8 bit exponent | 23 bit significand
  - double precision: sign bit | 11 bit exponent | 52 bit significand

# Float, double

| Name | Common name | Base | Digits | E min | E max |
|---|---|---|---|---|---|
| binary32 | Single precision | 2 | 23+1 | -126 | +127 |
| binary64 | Double precision | 2 | 52+1 | -1022 | +1023 |



# Testing Different Sizes

Using ‘sizeof()’ function, write a short program to determine the size of different types of variables on your machine.

# Constants

| Suffix | Prefix | Example | Description |
|---|---|---|---|
| - | - | 1234 | integer |
| - | - | 1.234 | double |
| - | - | 1.23e-3 | double |
| l, L | - | 12345678L | long integer |
| u, U | - | 1234U | unsigned int |
| ul, UL | - | 12345678UL | unsigned long |
| - | 0 | 037 | octal |
| - | 0x, 0X | 0x10 | hexadecimal |
| l, L | 0, 0x, 0X | | long octal/hexadecimal |
| u, U | 0, 0x, 0X | | unsigned octal/hexadecimal |
| ul, UL | 0, 0x, 0X | | unsigned long octal/hexadecimal |
| ‘ | ‘ | ‘x’ | character |



# 2.3 Constants continued

| Suffix | Prefix | Example | Description |
|---|---|---|---|
| \ | - | \n | Special characters |
| \0 | - | \7 | Special characters - octal |
| \00 | - | \17 | Special characters - octal |
| \000 | - | \117 | Special characters - octal |
| \xh | - | \xd | Special characters - hexadecimal |
| \xhh | - | \x10 | Special characters - hexadecimal |
| - | 0 | 037 | octal |
| - | 0x, 0X | 0x10 | hexadecimal |
| l, L | 0, 0x, 0X | | long octal/hexadecimal |
| u, U | 0, 0x, 0X | | unsigned octal/hexadecimal |
| ul, UL | 0, 0x, 0X | | unsigned long octal/hexadecimal |
| ‘ | ‘ | ‘x’ | character |

# Escape Characters

| | | | |
|---|---|---|---|
| \a | alert (bell) character | \\ | backslash |
| \b | backspace | \? | question mark |
| \f | formfeed | \' | single quote |
| \n | newline | \" | double quote |
| \r | carriage return | *ooo* | octal number |
| \t | horizontal tab | \x*hh* | hexadecimal number |
| \v | vertical tab | | |



# Constant Expressions

A *constant expression* is an expression that involves only constants

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

A *string constant*, or *string literal*

```
"I am a string"        ""        "hello, " "world"
```

# String Constants

Technically, a string constant is an array of characters

The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal '\0'. Here is our version:

```
/* strlen: return length of s */
int strlen(char s[])
{
        int i;
        while (s[i] != '\0')
        ++i;
        return i;
}
```



# enumeration

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                    /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

# Exercise

Enumerate Days Of the Week, Sunday being 7



# Arithmetic Operators

unary operations:
**+**, **-**

Binary operators:
**+**, **-**, ***, /**
**%** (modulus)

# Arithmetic Operators continued

| Precedence |
|---|
| Unary Operation +, - |
| Binary Operation +, -, *, /, % |
| Left → Right |



# Relational and Logical Operators

| Category | Operation |
|---|---|
| Relational Operation | > >= < <= |
| Relational Operation | == != |
| Logical Operators | && \|\| |

| Precedence |
|---|
| > >= < <= |
| == != |
| &&[1] |
| \|\|[1] |
| Left → Right |

1- evaluation stops as soon as the truth or falsehood of the result is known

# Relational and Logical Operators

| Precedence |
| --- |
| Unary Operation +, - |
| Binary Operation +, -, *, /, % |
| Left → Right |

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

Parentheses are NOT Needed!

```
(c=getchar()) != '\n'
```

Parentheses ARE Needed!



# Type Conversions

## General Rules

'C' automatically converts a ''narrower'' operand into a ''wider'' one without losing information

Expressions that don't make sense, like using a float as a subscript, are disallowed

Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they Aare not illegal

# Type Conversions

A char is just a small integer!

```
/* atoi: convert s to integer */
int atoi(char s[])
{
        int i, n;
        n = 0;
        for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
        return n;
}
```

S = "235U"

Iteration 1 s[0]=50 → (0)*10 + (50-48)

Iteration 2 s[1]=51 → (2)*10 + (51-48)

Iteration 3 s[2]=53 → (2)*10*10 + (3)*10+(53-48)
→ 2*100 + 3*10 +5 = 235

Iteration 4 s[3]= 87 → Terminate the for loop



| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

# Type Conversions

Upper case to Lower Case conversion

```
/* lower: convert c to lower case; ASCII only */
int lower(int c)
{
if (c >= 'A' && c <= 'Z')
return c + 'a' - 'A';
else
return c;
}
```

c >= '0' && c <= '9'
can be replaced by
***isdigit(c)* along with <ctype.h>**



# Type Conversions

Exercise

Write a program to convert a two-digit numerical characters to integer using the ***isdigit()*** function

# Type Conversions

## Implicit arithmetic conversions

In general, if an operator like + or * that takes two operands has operands of different types, the ''lower'' type is *promoted to the ''higher'' type before the operation proceeds.*

- 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.



# Increment and Decrement Operators

The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n), or postfix operators (after the variable: n++)

**If n is 5, then**

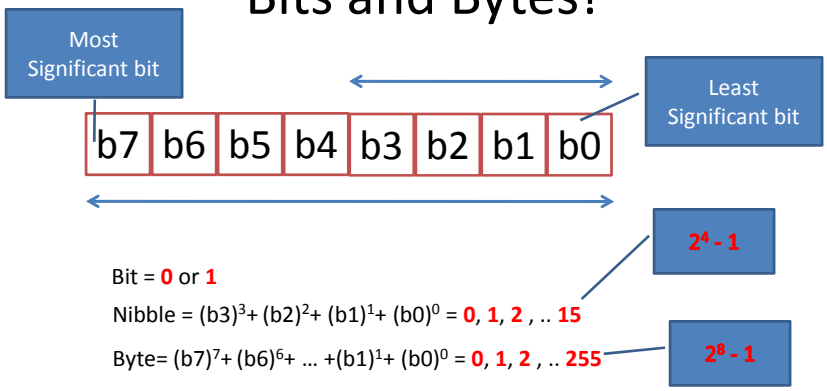
```
x = n++;
```

**sets x to 5, but**

```
x = ++n;
```

**sets x to 6**

# Bits and Bytes!




Bit = **0** or **1**

Nibble = $(b3)^3 + (b2)^2 + (b1)^1 + (b0)^0$ = **0**, **1**, **2** , .. **15**

Byte= $(b7)^7 + (b6)^6 + \ldots + (b1)^1 + (b0)^0$ = **0**, **1**, **2** , .. **255**

Spring 2012 Chapter 2 29

# Octal and Hexadecimal Arithmetic

- Octal → Groups of 3 bits, i.e. 0 through 7
- Hexadecimal→ Groups of 4 bits, i.e. 0 through 15
- Bytes can be written as 3 decimal, 3 Octal or 2 Hexadecimal digits.

Spring 2012 Chapter 2 30

# Bitwise Operators

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



# Octal and Hexadecimal Arithmetic

\034 + \567 = ?

Convert 213 (decimal) to Binary, Octal and Hexadecimal

0x1AB + 0x67 = ?

0x123 >> 5 = ?

(0xC2 | 0x 123) & 0x11 = ?

# Bitwise Operators

| | |
|---|---|
| n = n & 0177; | sets to zero all but the low-order 7 bits of n |
| x = x \| SET_ON; | sets to one in x the bits that are set to one in SET_ON |
| x = x & ~077 | sets the last six bits of x to zero |

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



# Assignment Operators and Expressions

i = i + 2 ←→ i += 2

The operator += is called an *assignment operator.*

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

***expr1 op= expr2***

is equivalent to

***expr1 = (expr1) op (expr2)***

# Conditional Expressions

***expr1 ? expr2 : expr3***

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

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



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

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