

Types, Operators and Expressions

CSE 2031
Fall 2012

September 17, 2012

1

Variable Names (2.1)

- Combinations of letters, numbers, and underscore character (`_`) that
 - do not start with a number;
 - are not a keyword.
- Upper and lower case letters are distinct (`x` \neq `X`).
- Examples: Identify valid and invalid variable names
`abc`, `aBc`, `abc5`, `aA3_`, `char`, `_360degrees`,
`5sda`, `my_index`, `_temp`, `string`, `struct`,
`pointer`

2

Variable Names: Recommendations

- Don't begin variable names with underscore `_`
- Limit the length of a variable name to 31 characters or less.
- Function names, external variables: may be less than 31 characters allowed, depending on systems.
- Lower case for variable names.
- Upper case for symbolic constants
 - `#define MAX_SIZE 100`
- Use short names for local variables and long names for external variables.

3

Data Types and Sizes (2.2)

4 basic types in C

- **char** – characters (8 bits)
- **int** – integers (either 16 or 32 bits)
- **float** – single precision floating point numbers (4 bytes)
- **double** – double precision floating point numbers (8 bytes)

4

Qualifiers

- **signed char** `sc;` /* -127 - +128 */
- **unsigned char** `uc;` /* 0 - +255 */
- **short** `s;` /* 16 bits, -32,768 - +32,767 */
 - `short int s;`
- **long counter;** /* 32 bits */
 - `long int counter;`
 - `int` is either 16 or 32 bits, depending on systems.
- **signed int** `sint;` /* same as `int sint;` */
- **unsigned int** `uint;`
 - 0 - +4,294,967,295, assuming 4-byte int
- **long double** `ld;` /* 12 or 16 bytes */

5

Qualifiers (cont.)

- `<limits.h>` and `<float.h>` contain
 - Symbolic constants for all of the above sizes,
 - Other properties of the machine and compiler.
- To get the size of a type, use `sizeof ()`

```
int_size = sizeof( int );
```

6

Characters

- 8 bits
- Included between 2 single quotes


```
char x = 'A'
```
- Character string: enclosed between 2 double quotes


```
"This is a string"
```
- Note: `'A' ≠ "A"`

A	A	\0
---	---	----

- `c = '\012'` /* 10 decimal; r/n new line character */

7

Characters

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

Source: www.LookUpTables.com

Constants (2.3)

- Numeric constants
- Character constants
- String constants
- Constant expressions
- Enumeration constants

9

Integer Constants

- Decimal numbers
 - 123487
- Octal: starts with 0 (zero)
 - 0654
- Hexadecimal: starts with 0x or 0X
 - 0x4Ab2, 0X1234
- long int: suffixed by L or l
 - 7L, 106l
- unsigned int: suffixed by U or u
 - 8U, 127u

10

Floating-point Constants

15.75		100.0L /* long double */
1.575E1	/* = 15.75 */	100.0F /* float */
1575e-2	/* = 15.75 */	
-2.5e-3	/* = -0.0025 */	● You can omit the integer portion of the floating-point constant.
25E-4	/* = 0.0025 */	

- If there is no suffix, the type is considered **double** (8 bytes).
- To specify **float** (4 bytes), use suffix F or f.
- To specify **long double** (12 or 16 bytes), use suffix L or l.

.0075e2
0.075e1
.075e1
75e-2

11

Numeric Constants

- 2010
- 100000
- 729L or 729l
- 2010U or 2010u
- 20628UL or 20628uL
- 24.7 or 1e-2
- 24.7F or 24.7f
- 24.7L or 24.7l
- 037
- 0x1f, 0X1f, 0x1F
- 0XFUL
- int
- taken as long if 16-bit int
- long (int)
- unsigned
- unsigned long
- double
- float
- long double
- octal (= 31 decimal)
- hexadecimal (= 31)
- What is this?

12

Character Constants

`'x'` ● letter **x**
`'2'` ● numeric value 50
`'\0'` ● NULL char, value 0

`#define NEW_LINE '\012'` ● octal, 10 in decimal
`#define NEW_LINE '\12'` ● `'\ooo'` 1 to 3 octal digits

`#define SPACE '\x20'` ● hex, 32 in decimal

13

Escape Sequences

<code>\a</code>	alert (bell) character	<code>\\</code>	backslash
<code>\b</code>	backspace	<code>\?</code>	question mark
<code>\f</code>	formfeed	<code>\'</code>	single quote
<code>\n</code>	newline	<code>\"</code>	double quote
<code>\r</code>	carriage return	<code>\ooo</code>	octal number
<code>\t</code>	horizontal tab	<code>\xhh</code>	hexadecimal number
<code>\v</code>	vertical tab		

14

String Constants

`"hello, world\n"`

`"" /* empty string */`

`"\" /* double quote character */`

`"hello," " world"` same as `"hello, world"`

- concatenated at compile time
- useful for splitting up long strings across several source lines.

15

Constant Expressions

- Expressions that involve only constants.
- Evaluated during compilation.

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

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

16

Enumeration Constants

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

- The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified.

```
enum colours { black, white, red, blue, green };
```

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

- If not all values are specified, unspecified values continue the progression from the last specified value.

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

17

Limits

- File `limits.h` provides several constants

- `char CHAR_BIT, CHAR_MIN, CHAR_MAX, SCHAR_MIN, ...`

- `int INT_MIN, INT_MAX, UINT_MAX`

- `long LONG_MIN, ...`

- You can find `FLOAT_MIN, DOUBLE_MIN, ...` in `<float.h>`

18

Declarations (2.4)

- All variables must be declared before use
- A variable may also be initialized in its declaration.

```
char esc = '\\';  
int i = 0;  
int limit = MAXLINE+1;  
float eps = 1.0e-5;
```

19

Qualifier `const`

- Indicates that the value of a variable will not be changed.
- For an array: the elements will not be altered.

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

- Used with array arguments, to indicate that the function does not change that array.

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

20

Arithmetic Operators (2.5)

+ - * / %

Examples:

```
abc = x + y * z;
```

```
j = a % i;
```

```
++x;
```

```
x++;
```

```
x += 5; /* x = x + 5; */
```

```
y /= z; /* y = y / z */
```

What is `x *= y + 1` ?

21

Precedence and Associativity (Pg 53)

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

22

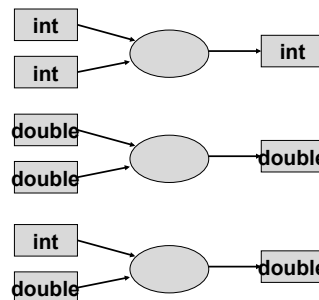
Type Conversion (2.7)

- `float f; int i;` What is the type of `f+i` ?
- General rule: convert a "narrower" operand into a "wider" one without losing information.
- So `i` is converted to float before the addition.
- `char` may be freely used in arithmetic expressions.

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

23

Arithmetic Conversion: Examples



```
int a=5, b=2, c;
double x, y = 2;

x = a/b;
// x = 2.0
c = a/b;
// c = 2
x = a/y;
// x = 2.5
c = a/y;
// c = 2
```

24

More Examples

- $17 / 5$
 - 3
- $17.0 / 5$
 - 3.4
- $9 / 2 / 3.0 / 4$
 - $9 / 2 = 4$
 - $4 / 3.0 = 1.333$
 - $1.333 / 4 = 0.333$

25

Type Conversion: More Rules

- 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.
- `float` to `int` causes truncation of any fractional part.
- Example:

```
float x, y = 2.7;
int i = 5;
x = i; /* x = 5.0 */
i = y; /* i = 2 */
```
- Example:

```
int a;
float x = 7, y = 2;
a = x / y;
```

26

Type Conversion: Even More Rules

- Longer integers are converted to shorter ones or to chars by dropping the excess high-order bits.

```
int i;          int i;
char c;        char c;
i = c;         c = i;
c = i;         i = c;
/* c unchanged */ /* i may be changed */
```

27

Casting

```
int A = 9, B = 2;
double x;
x = A / B; /* x is 4.0 */
x = A / (double)B; /* C is 4.5 */
```

```
int n;
sqrt(double(n))
```

Doesn't change the value of B,
just changes the type to double

- The cast operator has the same high precedence as other unary operators.

28

Increment and Decrement Operators (2.8)

- ++ or --
- Placing in front: incrementing or decrementing occurs **BEFORE** value assigned

i = 2 and k = 1

`k = ++i;`

<code>i = i + 1; 3</code>
<code>k = i; 3</code>

`k = --i;`

<code>i = i - 1; 1</code>
<code>k = i; 1</code>

- Placing after: occurs **AFTER** value assigned

i = 2 and k = 1

`k = i++;`

<code>k = i; 2</code>
<code>i = i + 1; 3</code>

`k = i--;`

<code>k = i; 2</code>
<code>i = i - 1; 1</code>

29

Examples

```
int a=2, b=3; c=5, d=7, e=11, f=3;
```

```
f += a/b/c;           3
```

```
d -= 7+c*--d/e;      -3
```

```
d = 2*a%b+c+1;       7
```

```
a += b +=c += 1+2;   13
```

Note: Do NOT write code as above. Hard to read and debug!

30

Relational and Logic Operators (2.6)

- Relational operators:
`> >= < <= == !=`
- Logical operators:
`! && ||`
- Evaluation stops as soon as the truth or falsehood of the result is known.

31

Boolean Expressions

- False is 0; any thing else is 1 (true).
- Write
`if (!valid)`
instead of
`if (valid == 0)`

32

Bitwise Operators (2.9)

- Work on individual bits
 - `&` `|` `^` `~`
- Examples:


```

a = 1;
b = 2;
c = a & b; /*c = 0*/
d = a && b; /*d = 1*/
short int i=5, j=8;
k=i&j;
k=i|j;
k=~j;
      
```

 - Application: bit masking


```

#define SET_ON 0XFFFF
int n, x;
n = n & 0177;
x = x | SET_ON;
          
```

33

Bit Shifting

- `x<<y` means shift `x` to the left `y` times.
 - equivalent to multiplication by 2^y
- `x>>y` means shift `x` to the right `y` bits.
 - equivalent to division by 2^y
- Left shifting 3 many times:

0	3
1	6
2	12
3	24
4	48
5	...
13	49512
14	32768

34

Right Shifting

- It could be logical (0) or arithmetic (signed)
- If unsigned, 0; if signed undefined in C

```

unsigned int i = 714;
357 178 89 44 22 11 5 2 1 0
      
```

- What if `i = -714` ?


```

-357 -178 -89 ... -3 -2 -1 -1 -1 -1
      
```

35

Conditional Expressions (2.11)

- ```

exp1 ? exp2 : exp3

```
- If `exp1` is true, the value of the conditional expression is `exp2`; otherwise, `exp3`.
 

```

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

```
  - If `expr2` and `expr3` are of different types, the type of the result is determined by the conversion rules discussed earlier.
 

```

int n; float f;
(n > 0) ? f : n
/* result of type float in either case */

```

36

## Conditional Expressions: Advantage

- Succinct but **hard-to-read** code

- Example 1:

```
for (i = 0; i < n; i++)
 printf("%6d%c", a[i],
 (i%10==9 || i==n-1) ? '\n' : ' ');
```

- Example 2:

```
printf("You have %d item%s.\n", n,
 n==1 ? "" : "s");
```

37

## Next time ...

- Control Flow (Chapter 3, C book)
- Functions and program structures (Chapter 4, C book)

38