

Lexical Rules and Data Types

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

Contents

- Lexical Scanning
- Comments
- Identifiers and Keywords
- C Variables
- Data Types
- Fundamental C Types
- Constants

Lexical Scanning

Compiling Step #1: *Lexical Scanning*

- Divides the program into ***tokens***, which are the smallest meaningful units of a program
- Tokens in C are...
 - identifiers (e.g., **num_records, cust_name**)
 - keywords (e.g., **while, if, char**)
 - constants/strings (e.g., **3.1415, "Answer: "**)
 - operators (e.g., **+, ^, =**)
 - explicit separators (e.g., **(, }, ;**)

Scanning... (cont'd)

- **White space** (space, tabs/indentation, newlines, comments) are **ignored**, except as explicit separators
- The C compiler uses scanning when processing code

Scanning (cont'd)

- Requires precision: what are the tokens in `d=-c2++a;`;

```
d  =  -c2  +  ++a  ;  ?  
d  =  -c2++  +  a   ;  ?  
d  =-   c2++  +  a   ;  ?  
d  =-   c2  +  ++a;  ?
```

This is not a precedence issue

- we don't know or care what the precedence of `=`, `=-`, `++`, and `+` is at this point

Greedy or “Max Munch” Approach

- Scanning goes from left to right, always grabbing the **largest token possible**
- Example (again):

1. `d` `==c2+++a;` (“d=” not a token)

2. `d` `= - c2+++a;` (“=-” not a token)

3. `d` `= - c2+++a;` (“-c” not a token)

4. `d` `= - c2 +++a;` (“c2+” not a token)

5. `d` `= - c2 ++ +a;` (“+++” not a token)

6. `d` `= - c2 ++ + a;` (“+a” not a token)

7. `d` `= - c2 ++ + a ;` (“a;” not a token)

Greedy or “Max Munch” Approach (cont’d)

- String: `d=-c2+++a;`
- Tokens (one per line):

`d`

`=`

`-`

`c2`

`++`

`+`

`a`

`;`

(see **Lexical Scanning** in
Code samples and Demonstrations
in *Canvas*)

Scanning... (cont'd)

- How many tokens, and what are they?

```
j =+k2+3;
```

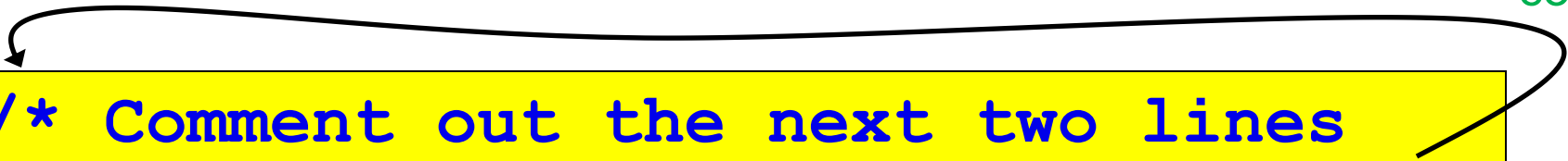
Using Code Comments

Block Style:

```
a = c - b;    /* b must be gt 0 */  
d = a * 3;
```

Great for commenting out whole sections of code, but **look out** if the code already has comments!

Terminates
comment



```
/* Comment out the next two lines  
  a = c - b;    /* b must be gt 0 */  
  d = a * 3;  
*/
```


⚠ common source of bugs (errors) ⚠
attempt to nest comments

Comments (cont'd)

To-end-of-line comments are allowed in C99:

```
r = 6 * x;    // compute radius  
d = 2 * r;    // now diameter
```

Identifiers (Names, Labels)

- Consist of letters, '_', and digits
cannot start with a digit (2_B_or_not_2_B) 
- Case sensitive!
`myVar` is not the same as `myvar`
- Unlimited length (advice: stop at 32)
`gnome_memmgt_insert_into_heap_I_modified_this_because_I_can`

Reserved Keywords

- **Do not** use as identifiers

- C89:

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

- C99 adds a few more:

`_Bool, _Complex, _Imaginary, inline, restrict`

C Variables

- A ***variable*** = a **location** in memory + its ***interpretation***
- Interpretation of a variable is based on its
 1. storage class and
 2. data type
- (*We will discuss storage classes later...*)
 - *lifetime of the variable*
 - *how variable is (or can be) initialized*
 - *scope (visibility) of the variable*

Data Types

- The **data type** of a variable defines its interpretation
- Ex: suppose a 32-bit binary value stored in memory is **01000001010000100100001101000100**
 - if type **float**, interpreted to be numerical value **781.03521728515625**
 - if type **unsigned int**, interpreted to be numerical value **1145258561**
 - if type **char**, interpreted to be the ASCII string value **ABCD**

Static or Dynamic Types

- In C variables are **statically** typed
 - A type must be specified when a variable is created, and cannot change thereafter
- Languages with **dynamic** typing (e.g., PHP, Python, Perl, Ruby, Javascript, ...) are more flexible

Fundamental C Types

- Also called built-in, primitive, basic types
- There are really **only 2!**
 - **integer** (includes **characters**)
 - **floating point**, or limited precision real number

Derived C Types

- These are composed from the fundamental types
 - arrays
 - functions
 - pointers
 - structs
 - unions
 - *these will all be discussed later...*
- **Enumerated** types: *we'll discuss later...*
- **Complex numbers** type: *we won't use this semester*

Specializations of Fundamental Types

- Integers can be...
 - **signed** or **unsigned** (**signed** by default)
 - really short (**char**), **short**, regular (**int** by default), **long**, really long (**long long**)
- Floating point (always signed) can be...
 - regular precision (**float**)
 - double precision (**double**)
 - extended precision (**long double**)

(Footnote)

- The data type of a variable defines its **usual** meaning, but the programmer may interpret it differently
- Ex.: a **char** can represent...
 - an ASCII-encoded character (most common case)
 - an 8-bit integer
 - eight 1-bit flags
 - ...

Min and Max Integer Values

The **lengths** (in bits) (and the max and min values) of these types are **platform dependent**

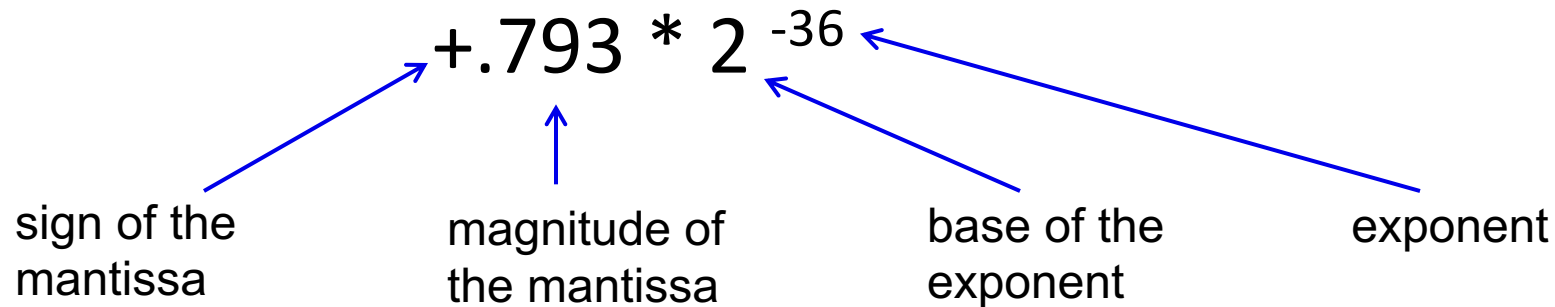
Type	# bits	Value
Min 'unsigned anything'	n.a.	0
Min 'signed char'	8	-128
Max 'signed char'	8	127
Max 'unsigned char'	8	255
Min 'signed short int'	16	-32,768
Max 'signed short int'	16	32,767
Max 'unsigned short int'	16	65,535

Integer Values... (cont'd)

Type	# bits	Value
Min 'signed int'	32	-2,147,483,648
Max 'signed int'	32	2,147,483,647
Max 'unsigned int'	32	4,294,967,295
Min/Max 'signed long int'	64	same as 'signed long long int'
Max 'unsigned long int'	64	same as 'unsigned long long int'
Min 'signed long long int'	64	-9,223,372,036,854,775,808
Max 'signed long long int'	64	9,223,372,036,854,775,807
Max 'unsigned long long int'	64	18,446,744,073,709,551,615

Floating Point (Real Numbers)

- Warning! **Platform dependent!** Lots of **gcc** options!
- Terminology



Size of the **exponent** (# bits) mainly determines the **range** of numbers that can be represented

Size of the **mantissa** (# bits) mainly determines the **precision** of numbers that can be represented

Floating Point (Real Numbers)

- **IEEE floating point standard single precision:**
 - 1-bit sign
 - 23-bit (+ 1 implied bit) mantissa
 - 8-bit biased exponent (base 2)
 - 6 digits precision
- **double precision:**
 - 1-bit sign
 - 52+1 bit mantissa
 - 11-bit biased exponent (base 2)
 - 15 digits precision

Floating Point (cont'd)

- **Min** (normalized) **positive** values (approximate)
 - single precision (**float**): 2^{-126} ($\approx 10^{-38}$)
 - double precision (**double**): 2^{-1022} ($\approx 10^{-308}$)
- **Max** (normalized) **positive** values (approximate)
 - single precision (**float**): 2^{127} ($\approx 10^{38}$)
 - double precision (**double**): 2^{1023} ($\approx 10^{308}$)

Floating Point (cont'd)

- **long double = 128 bits**
 - more bits precision than **double**, same range

Reminder: Arithmetic Problems

- Types make a difference in computer arithmetic
 - **signed vs. unsigned** max and min values (integer)
 - **overflow** (integer and floating point)
 - **underflow** and **limited precision** (floating point)

⚠ *common source of bugs* ⚠
overflow, limits of precision

Constants

- Types of constants (set once and never changed)
 - integer
 - floating point
 - character (a type of integer)
 - enumeration (*we'll talk about these later*)
- Character constants in single quotes: 'a', 'b'
 - value stored is the numeric value of the character in ASCII
- **#define <CONSTANT_NAME> <value>**

Implied Types of Constants

- Default type for **integer** constants: shortest type compatible with value, starting with
signed int -> unsigned int -> ...
- Default type for **floating point** constants: **double**

ASCII

- The ASCII code is used by computers to represent characters, such as letters, special symbols and digits
- ASCII is a specific 8-bit encoding of Western characters (punctuation, digits, upper and lower case characters)
- Only the **first 128 values** are standardized
- The interpretation of the **remaining 128 values** are **application/platform-specific**

Standardized ASCII (0-127)

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		DEL

Source: www.LookupTables.com

One Interpretation of 128-255

128	Ç	144	É	161	í	177	▤	193	⊥	209	〒	225	β	241	±
129	ü	145	æ	162	ó	178	▥	194	⌞	210	π	226	Γ	242	≥
130	é	146	Æ	163	ú	179		195	⌟	211	ℒ	227	π	243	≤
131	â	147	ô	164	ñ	180	⌞	196	—	212	⌞	228	Σ	244	∫
132	ä	148	ö	165	Ñ	181	⌞	197	+	213	ƒ	229	σ	245	∫
133	à	149	ò	166	²	182	⌞	198	⌞	214	π	230	μ	246	÷
134	â	150	û	167	°	183	π	199	⌞	215	⌞	231	τ	247	≈
135	ç	151	ù	168	¿	184	⌞	200	ℒ	216	⌞	232	Φ	248	°
136	ê	152	—	169	—	185	⌞	201	ƒ	217	∫	233	⊗	249	.
137	ë	153	Ö	170	¬	186	⌞	202	≡	218	⌞	234	Ω	250	.
138	è	154	Ü	171	½	187	⌞	203	〒	219	■	235	δ	251	√
139	ï	156	£	172	¼	188	⌞	204	⌞	220	■	236	∞	252	—
140	î	157	¥	173	¡	189	⌞	205	=	221	■	237	φ	253	²
141	ì	158	—	174	«	190	⌞	206	⌞	222	■	238	ε	254	■
142	Ä	159	ƒ	175	»	191	⌞	207	⌞	223	■	239	∧	255	
143	Å	160	á	176	▤	192	⌞	208	⌞	224	α	240	≡		

Source: www.LookupTables.com

Useful Character Constant Escape Sequences

- `\0` Null character
- `\'` Single quote
- `\"` Double quote
- `\\` Backslash
- `\n` Newline
- `\t` Horizontal tab
- `\nnn` Octal value of character
(ex: `'a' == '\141'`)
- `\xnn` Hexadecimal value of character
(ex: `'a' == '\x61'`)

(see `letter.c` in *Code samples and Demonstrations in Canvas*)

Converting ASCII digits to Integers

- You can read ASCII characters and do arithmetic on them, but results are **not** what you expect!
- Program: read a number, print it out

(see `letter.c` in Code samples and Demonstrations in Canvas)

```
int c;  
c = getchar();           // read one ascii character  
printf("%d\n", c);       // interpret c as integer  
                           // and print as ASCII  
                           // (decimal) string
```

Result

- user types: **1**
- program prints: **49** Why??

⚠ common source of bugs ⚠
**difference between
ASCII-encoded strings
and numbers**

Converting ASCII to Numbers

- Converting ASCII-encoded digit to an integer, the **right way**:

```
unsigned char c;  
c = (unsigned char) getchar();  
unsigned int n;  
n = c - '0';  
printf("%d\n", n);
```

48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9

- Converting integer to ASCII:

```
c = (char) (n + '0');
```

How would we convert an ASCII string (“12”) to an integer, and vice versa???

(see [ascii_to_number.c](#) in Code samples and Demonstrations in Canvas)

String Literals

- Strings in C are **arrays** of characters
 - Terminated (**automatically**, by the compiler) with **NULL**
- Specifying a string: **"abcdefg"**
 - Cannot contain double quote or span multiple lines (use **\"** or **\n** if quote or newline should be in the string)
- Warning: **"a"** is not the same as **'a'**
 - The first one is a string, the second one is a single character value
 - Single quotes are for one character only.
- *We will discuss strings in more detail later...*

Integer Constants

- Specifying:
<optionalsign> <stringofdecimaldigits>
 - ex: **7940**, **+7940**, **-36**
- If prefixed by **0**, interpreted as **base 8** (octal) constant
 - only 0–7 allowed as digits
- If prefixed by **0x**, interpreted as **base 16** (hexadecimal) constant
 - 0–9, **a–f** allowed as digits

(see `int_constants.c` in Code samples and Demonstrations in Canvas)

Integer Constants (cont'd)

- If suffixed by **u**, type is **unsigned int**, and value must be positive
 - ex: **123u**
- If suffixed by **L**, type is **long int**
 - ex: **456L**

Floating Point Constants

- Specifying:
<optionalsign> integerpart . fractionpart
 - either integer part or fractional part can be missing
 - all good: **22.22**, **+2.**, **-.22**
 - warning: **2** is an integer constant, **2.** is a floating point constant
- Followed (optionally) by exponent
e ***<optionalsign> <integerconstant>***
 - ex.: **23.45e-67** means $23.45 * 10^{-67}$

Floating Point... (cont'd)

- **Default type is `double`**
 - suffixed by `f`: forces type to be `float`
 - suffixed by `L`: forces type to be `long double` (extended precision)
- *More about floating point numbers, precision, and range, later...*

References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.
- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.