Chapter 7

Basic Types

Last item in Page 89: check stdin.h

Basic Types

- C's ***basic*** (built-in) ***types:***
  - Integer types, including long integers, short integers, and unsigned integers
  - Floating types (`float`, `double`, and `long double`)
  - `char`
  - `_Bool` (C99)

Integer Types

- C supports two fundamentally different kinds of numeric types: **integer** types and **floating** types.
- Values of an ***integer type*** are whole numbers(**整數**)
- Values of a floating type can have a fractional part as well.
- The integer types, in turn, are divided into two categories: signed and unsigned.

Signed and Unsigned Integers

- The leftmost bit of a ***signed*** integer (known as the ***sign bit***) is 0 if the number is positive or zero, 1 if it's negative.
- The largest 16-bit integer has the binary representation 0111111111111111, which has the value 32,767 ($2^{15} – 1$).
- The largest 32-bit integer is

  01111111111111111111111111111111

  which has the value 2,147,483,647 ($2^{31} – 1$).
- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be ***unsigned.***
- The largest 16-bit unsigned integer is 65,535 ($2^{16} – 1$).
- The largest 32-bit unsigned integer is 4,294,967,295 ($2^{32} – 1$).

Signed Integers: ***2's complement***

- 0111-1111-1111-1111 which has the value 32,767 (2^{15} – 1).
- 1000-0000-0000-0000

  0111-1111-1111-1111, after ***1's complement***

  1000-0000-0000-0000, (1's complement) +1 => **–2^{15}**
- 1111-1111-1111-1111

  0000-0000-0000-0000, after ***1's complement***

  0000-0000-0000-0001, (1's complement) +1 => **–1**

Signed and Unsigned Integers

- By default, integer variables are signed in C—the leftmost bit is reserved for the sign.
- To tell the compiler that a variable has no sign bit, declare it to be `unsigned`.
- Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications.

Integer Types

- The `int` type is usually 32 bits, but may be 16 bits on older CPUs.
- ***Long*** integers may have more bits than ordinary integers; ***short*** integers may have fewer bits.
- The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form integer types.
- Only six combinations produce different types:

```
short int     unsigned short int
int           unsigned int
long int      unsigned long int
```

- **The order of the specifiers doesn't matter. Also, the word `int` can be dropped** (`long int` can be abbreviated to just `long`).

Integer Types

- The range of values represented by each of the six integer types varies from one machine to another.
- However, the C standard requires that `short int`, `int`, and `long int` must each cover a certain minimum range of values.
- Also, `int` must not be shorter than `short int`, and `long int` must not be shorter than `int`.

Integer Types

- Typical ranges of values for the integer types on a 16-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–32,768	32,767
`unsigned int`	0	65,535
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 32-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
`unsigned int`	0	4,294,967,295 ($2^{32}-1$)
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	-2^{63}	$2^{63}-1$

LONG_MAX=9,223,372,036,854,775,807L (exa bytes)

LONG_MIN=(-LONG_MAX - 1L) =9,223,372,036,854,775,808L

`unsigned long int`	0	$2^{64}-1$

ULONG_MAX=18,446,744,073,709,551,615UL

- The `<limits.h>` header defines macros that represent the smallest and largest values of each integer type.

Integer Types in C99

- C99 provides two additional standard integer types, ***`long long int`*** and ***`unsigned long long int`***.
- Both `long long` types are required to be at least **64 bits** wide.
- The range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807).
- The range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

Integer Types in C99

- The `short int`, `int`, `long int`, and `long long int` types (along with the `signed char` type) are called ***standard <u>signed</u> integer types*** in C99.
- The `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` types (along with the `unsigned char` type and the `_Bool` type) are called ***standard <u>unsigned</u> integer types.***
- In addition to the **standard integer types**, the C99 standard allows implementation-defined ***extended integer types*****,** both signed and unsigned.

Integer Constants

- ***Constants*** are numbers that appear in the text of a program.
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

- Octal numbers use only the digits 0 through 7.
- Each position in an octal number represents a power of 8.
  - The octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.
- A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  - The hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

Integer Constants

- ***Decimal*** constants contain digits between 0 and 9, but must not begin with a zero:

  ```
  15   255   32767
  ```

- ***Octal*** constants contain only digits between 0 and 7, and must begin with a **zero**:

  ```
  017   0377   077777
  ```

- ***Hexadecimal*** constants contain digits between 0 and 9 and letters between `a` and `f`, and always begin with **`0x`**:

  ```
  0xf   0xff   0x7fff
  ```

- The letters in a hexadecimal constant may be either upper or lower case:

  ```
  0xff   0xfF   0xFf   0xFF   0Xff   0XfF   0XFf   0XFF
  ```

Integer Constants

- The type of a *decimal* integer constant is normally **`int`**.(1)
- If the value of the constant is too large to store as an `int`, the constant has type **`long`**(2)**`int`** instead.
- If the constant is too large to store as a `long int`, the compiler will try (3)**`unsigned long int`** as a last resort.
- For an *octal* or *hexadecimal* constant, the rules are slightly different: the compiler will go through the types **`int`**, **`unsigned int`**, **`long int`**, and **`unsigned long`**(1)**`int`**(2)until it finds one ca(3)pable of representing the (4)constant.

Integer Constants

- To force the compiler to treat a constant as a long integer, just follow it with the letter `L` (or `l`):

```
15L   0377L   0x7fffL
```

- To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

```
15U   0377U   0x7fffU
```

- `L` and `U` may be used **in combination**:

```
0xffffffffUL
```

  The order of the `L` and `U` doesn't matter, nor does their case.

Integer Constants in C99

- In C99, integer constants that end with either `LL` or `ll` (the case of the two letters must match) have type `long long int`.
- Adding the letter `U` (or `u`) before or after the `LL` or `ll` denotes a constant of type `unsigned long long int`.
- C99's general rules for determining the type of an integer constant are a bit different from those in C89.
- The type of a decimal constant with no suffix (`U`, `u`, `L`, `l`, `LL`, or `ll`) is the "smallest" of the types `int`, `long int`, or `long long int` that can represent the value of that constant.

Integer Constants in C99

- For an octal or hexadecimal constant, the list of possible types is `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, and `unsigned long long int`, in that order.
- Any suffix at the end of a constant changes the list of possible types.
  - A constant that ends with `U` (or `u`) must have one of the types `unsigned int`, `unsigned long int`, or `unsigned long long int`.
  - A decimal constant that ends with `L` (or `l`) must have one of the types `long int` or `long long int`.

Integer Overflow

- When arithmetic operations are performed on integers, it's possible that the result will be too large to represent.
- For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`.
- If the result can't be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred.

Integer Overflow

- The behavior when integer overflow occurs depends on whether the operands were signed or unsigned.
  - When **overflow occurs** during an operation on ***signed*** integers, the program's behavior is ***undefined***.
  - When **overflow occurs** during an operation on ***unsigned*** integers, the result *is* ***defined***: we get the correct answer **modulo 2^n**, where *n* is the number of bits used to store the result.

Reading and Writing Integers

- Reading and writing unsigned, short, and long integers requires new conversion specifiers.
- When **reading or writing an *unsigned* integer**, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification.

```
unsigned int u;

scanf("%u", &u);  /* reads  u in base 10 */
printf("%u", u);  /* writes u in base 10 */
scanf("%o", &u);  /* reads  u in base  8 */
printf("%o", u);  /* writes u in base  8 */
scanf("%x", &u);  /* reads  u in base 16 */
printf("%x", u);  /* writes u in base 16 */
```

Reading and Writing Integers

- When reading or writing a *short* integer, put the letter **h** in front of d, o, u, or x:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter **l** ("**ell**," not "one") in front of d, o, u, or x.
- When reading or writing a *long long* integer (C99 only), put the letters **ll** in front of d, o, u, or x.

Program: Summing a Series of Numbers (Revisited)

- The `sum.c` program (Chapter 6) sums a series of integers.
- One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable.
- Here's what might happen if the program is **run on a machine whose integers are 16 bits long**:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

- When overflow occurs with signed numbers, the outcome is undefined.
- The program can be improved by using `long` variables.

sum2.c

```
/* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
  long n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%ld", &n);
  while (n != 0) {
    sum += n;
    scanf("%ld", &n);
  }
  printf("The sum is: %ld\n", sum);

  return 0;
}
```

Floating Types

- C provides three ***floating types,*** corresponding to different floating-point formats:
  - `float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in ***IEEE Standard 754*** (also known as IEC 60559).

The IEEE Floating-Point Standard

- IEEE Standard 754 was developed by the Institute of Electrical and Electronics Engineers (IEEE).
- It has two primary formats for floating-point numbers: **single precision** (32 bits) / ***double precision*** (64 bits).
- Numbers are stored in a form of scientific notation, with each number having a ***sign,*** an ***exponent (指數),*** and a ***fraction (分數) or mantissa (尾數， 浮點數).***
- In single-precision format, the **exponent is 8 bits** long, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.

The IEEE Floating-Point Standard

- **Precision** ***Sign, Exponent, Fraction, Bias***
- **Single** 1 [31], 8 [30-23], 23 [22-00], 127
- **Double** 1 [63], 11 [62-52], 52 [51-00], 1023
- Stored as a ***normalized base-2*** number
- Stored exponent = real exponent + bias
  - leading digit of 1 of the fraction is not stored
- 0.5 (1.0×$\mathbf{2^{-1}}$): 00111111000000000000000000000000
- 0.25 (1.0×$\mathbf{2^{-2}}$): 00111110100000000000000000000000

The IEEE Floating-Point Standard

- 0.1 ($\underline{1.1001}\ldots\times 2^{-4}$):

  $0\underbrace{\mathbf{01111011}}_{124}\underline{10011001100110011001100110 1}$

- 100.0 ($\underline{1.1001}\times 2^{6}$):

  $\underline{0}\,\mathbf{10000101}\,\underline{10010000000000000000000}$

  $100 = 64 + 32 + 4 = \mathbf{2^6} + \mathbf{2^5} + \mathbf{2^2}$

  $= 1100100_2 = 1.1001\times\mathbf{2^6}$

  $6 + 127 = \mathbf{10000101}$

- Integer

  100 = $\underline{00000000\text{-}00000000\text{-}00000000\text{-}01100100}$

0.5	0
0.25	0
0.125	0
0.0625	1
0.03125	1
0.015625	0
0.0078125	0
0.00390625	1
0.001953125	1
0.0009765625	0
0.00048828125	0
0.000244140625	1
0.0001220703125	1
…	

Floating Types

- Characteristics of `float` and `double` when implemented according to the IEEE standard:

Type	***Smallest Positive Value***	***Largest Value***	***Precision***
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	2.22507×10^{-308}	1.79769×10^{308}	15 digits

- On computers that don't follow the IEEE standard, this table won't be valid.
- In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`.

Floating Types

- Macros that define the characteristics of the floating types can be found in the `<float.h>` header.
- In C99, the floating types are divided into two categories.
  - ***Real floating types*** (`float`, `double`, `long double`)
  - ***Complex types*** (`float _Complex`, `double _Complex`, `long double _Complex`)

Floating Constants

- Floating constants can be written in a variety of ways.
- Valid ways of writing the number 57.0:

```
57.0   57.   57.0e0   57E0   5.7e1   5.7e+1
.57e2  570.e-1
```

- A floating constant must contain a decimal point and/or an exponent; the exponent indicates the **power of 10** by which the number is to be scaled.
- If an exponent is present, it must be preceded by the letter `E` (or `e`). An optional `+` or `-` sign may appear after the `E` (or `e`).

Floating Constants

- By default, floating constants are stored as double-precision numbers.
- To indicate that only single precision is desired, put the letter `F` (or `f`) at the end of the constant (for example, `57.0F`).
- To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

Reading and Writing Floating-Point Numbers

- The conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers.
- When reading a value of type **`double`**, put the letter **`l`** in front of `e`, `f`, or `g`:

```
double d;

scanf("%lf", &d);
```

- *Note:* Use `l` only in a `scanf` format string, not a `printf` string.
- In a `printf` format string, the `e`, `f`, and `g` conversions can be used to write either `float` or `double` values.
- When reading or writing a value of type **`long double`**, put the letter **`L`** in front of `e`, `f`, or `g`.

Character Types

- The only remaining basic type is `char`, the character type.
- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character Sets

- Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.
- ASCII is often extended to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

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

Source: www.LookupTables.com

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Character Sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero         */
ch = ' ';    /* space        */
```

- Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers*.
- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.
- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.
- Character constants actually have `int` type rather than `char` type.

Operations on Characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';           /* i is now 97   */
ch = 65;           /* ch is now 'A' */
ch = ch + 1;       /* ch is now 'B' */
ch++;              /* ch is now 'C' */
```

Operations on Characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.
- These values depend on the character set in use, so programs that use <, <=, >, and >= to compare characters may not be portable.

Operations on Characters

- The fact that characters have the same properties as numbers has advantages.
- For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:

  ```
  for (ch = 'A'; ch <= 'Z'; ch++) …
  ```
- Disadvantages of treating characters as numbers:
  - Can lead to errors that won't be caught by the compiler.
  - Allows meaningless expressions such as `'a' * 'b' / 'c'`.
  - Can hamper portability, since programs may rely on assumptions about the underlying character set.

Signed and Unsigned Characters

- The `char` type—like the integer types—exists in both signed and unsigned versions.
- Signed characters normally have values between –128 and 127. Unsigned characters have values between 0 and 255.
- Some compilers treat `char` as a signed type, while others treat it as an unsigned type. Most of the time, it doesn't matter.
- C allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;
```

Signed and Unsigned Characters

- C89 uses the term ***integral types*** to refer to both the integer types and the character types.
- **Enumerated types** are also integral types.
- C99 doesn't use the term "integral types."
- Instead, it expands the meaning of "integer types" to include the character types and the enumerated types.
- C99's `_Bool` type is considered to be an unsigned integer type.

Arithmetic Types

- The integer types and floating types are collectively known as ***arithmetic types*.**
- A summary of the arithmetic types in C89, divided into categories and subcategories:
  - Integral types
    - `char`
    - Signed integer types (`signed char`, `short int`, `int`, `long int`)
    - Unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`)
    - Enumerated types
  - Floating types (`float`, `double`, `long double`)

Arithmetic Types

- C99 has a more complicated hierarchy:
  - Integer types
    - `char`
    - Signed integer types, both standard (`signed char`, `short int`, `int`, `long int`, `long long int`) and extended
    - Unsigned integer types, both standard (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `_Bool`) and extended
    - Enumerated types
  - Floating types
    - Real floating types (`float`, `double`, `long double`)
    - Complex types (`float _Complex`, `double _Complex`, `long double _Complex`)

Escape Sequences

- A character constant is usually one character enclosed in single quotes.
- However, certain special characters—including the new-line character—can't be written in this way, because they're invisible (nonprinting) or because they can't be entered from the keyboard.
- ***Escape sequences*** provide a way to represent these characters.
- There are two kinds of escape sequences: ***character escapes*** and ***numeric escapes***.

Escape Sequences

- A complete list of character escapes:

Name	***Escape Sequence***
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Question mark	`\?`
Single quote	`\'`
Double quote	`\"`

Escape Sequences

- Character escapes are handy, but they don’t exist for all nonprinting ASCII characters.
- Character escapes are also useless for representing characters beyond the basic 128 ASCII characters.
- Numeric escapes, which can represent any character, are the solution to this problem.
- A numeric escape for a particular character uses the character’s octal or hexadecimal value.
- For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex.

Escape Sequences

- An ***octal escape sequence*** consists of the `\` character followed by an octal number with at most three digits, such as `\33` or `\033`.
- A ***hexadecimal escape sequence*** consists of `\x` followed by a hexadecimal number, such as `\x1b` or `\x1B`.
- The **x must be in lower case**, but the hex digits can be upper or lower case.

Escape Sequences

- When used as a character constant, an escape sequence must be enclosed in **single quotes**.
- For example, a constant representing the escape character would be written `'\33'` (or `'\x1b'`).
- Escape sequences tend to get a bit cryptic, so it's often a good idea to use `#define` to give them names:

  ```
  #define ESC '\33'
  ```
- Escape sequences can be embedded in strings as well.

Character-Handling Functions

- Calling C's `toupper` library function is a fast and portable way to convert case:

```
ch = toupper(ch);
```

- `toupper` returns the upper-case version of its argument.
- Programs that call `toupper` need to have the following `#include` directive at the top:

```
#include <ctype.h>
```

- The C library provides many other useful character-handling functions.

Reading and Writing Characters Using `scanf` and `printf`

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);   /* reads one character */
printf("%c", ch);   /* writes one character */
```

- ***`scanf` doesn't skip white-space characters.***
- To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:

```
scanf(" %c", &ch);
```

Reading and Writing Characters Using **`scanf`** and **`printf`**

- Since `scanf` doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character.
- A loop that reads and ignores all remaining characters in the current input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters Using **`getchar`** and **`putchar`**

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- **`getchar` returns an `int` value** rather than a `char` value, so `ch` will often have type `int`.
- Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Reading and Writing Characters Using **`getchar`** and **`putchar`**

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves execution time.
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats.
  - They are usually implemented as macros for additional speed.
- `getchar` has another advantage. Because it returns the character that it reads, `getchar` lends itself to various C idioms.

Reading and Writing Characters Using **`getchar`** and **`putchar`**

- Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\n');
```

Reading and Writing Characters Using **`getchar`** and **`putchar`**

- Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
  ;
```

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character:

```
while (getchar() != '\n')
  ;
```

Reading and Writing Characters Using `getchar` and `putchar`

- `getchar` is useful in loops that skip characters as well as loops that search for characters.
- A statement that uses `getchar` to skip an indefinite number of **space** characters:

```
while ((ch = getchar()) == ' ')
  ;
```

- When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Reading and Writing Characters Using **getchar** and **putchar**

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

  `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character.
- `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`.

length.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

length2.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

Type Conversion

- For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way.
- When operands of **different types are mixed in** expressions, the C compiler may have to generate instructions that change the types of some operands so that hardware will be able to evaluate the expression.
  - If we add a 16-bit `short` and a 32-bit `int`, the compiler will arrange for the `short` value to be converted to 32 bits.
  - If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format.

Type Conversion

- Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions*.**
- C also allows the programmer to perform ***explicit conversions*,** using the **cast** operator.
- The rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Type Conversion

- Implicit conversions are performed:
  1. When the operands in an arithmetic or logical expression don’t have the same type. (C performs what are known as the ***usual arithmetic conversions*.**)
  2. When the type of the **expression** on the **right side** of an **assignment** doesn’t match the type of the variable on the ***left side***.
  3. When the type of **an argument in a function call** doesn’t match the type of the corresponding parameter.
  4. When the type of the **expression in a `return` statement** doesn’t match the function’s return type.
- Chapter 9 discusses the last two cases.

The Usual Arithmetic Conversions

- The usual arithmetic conversions are applied to the operands of most binary operators.
- If `f` has type `float` and `i` has type `int`, the usual arithmetic conversions will be applied to the operands in the expression `f + i`.
- Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type).
  - When an integer is converted to `float`, the worst that can happen is a minor loss of precision.
  - Converting a floating-point number to `int`, on the other hand, causes the fractional part of the number to be lost. Worse still, the result will be meaningless if the original number is larger than the largest possible integer or smaller than the smallest integer.

The Usual Arithmetic Conversions

- Strategy behind the usual arithmetic conversions: convert operands to the "narrowest" type that will safely accommodate both values.
- Operand types can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***).
- Common promotions include the ***integral promotions,*** which convert a character or short integer to type `int` (or to `unsigned int` in some cases).
- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.

The Usual Arithmetic Conversions

- ***The type of either operand is a floating type.***
  - If one operand has type `long double`, then convert the other operand to type `long double`.
  - Otherwise, if one operand has type `double`, convert the other operand to type `double`.
  - Otherwise, if one operand has type `float`, convert the other operand to type `float`.
- Example: If one operand has type `long int` and the other has type `double`, the `long int` operand is converted to `double`.

The Usual Arithmetic Conversions

- ***Neither operand type is a floating type.*** First perform integral promotion on both operands.
- Then use the following diagram to **promote the operand whose type is narrower**:

$$\begin{tikzcd}
\texttt{unsigned long int} \\
\texttt{long int} \arrow[u] \\
\texttt{unsigned int} \arrow[u] \\
\texttt{int} \arrow[u]
\end{tikzcd}$$

The Usual Arithmetic Conversions

- When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value.
- This rule can cause obscure programming errors.
- It's best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

The Usual Arithmetic Conversions

- Example of the usual arithmetic conversions:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;     /* c is converted to int               */
i = i + s;     /* s is converted to int               */
u = u + i;     /* i is converted to unsigned int      */
l = l + u;     /* u is converted to long int          */
ul = ul + l;   /* l is converted to unsigned long int */
f = f + ul;    /* ul is converted to float            */
d = d + f;     /* f is converted to double            */
ld = ld + d;   /* d is converted to long double       */
```

Conversion During Assignment

- The usual arithmetic conversions don't apply to assignment which is also an **operator**.
- Instead, the expression on the right side of the assignment is converted to the type of the variable on the left side:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int    */
f = i;   /* i is converted to float  */
d = f;   /* f is converted to double */
```

Conversion During Assignment

- Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;     /* i is now 842 */
i = -842.97;    /* i is now -842 */
```

- Assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;     /*** WRONG ***/
i = 1.0e20;    /*** WRONG ***/
f = 1.0e100;   /*** WRONG ***/
```

Conversion During Assignment

- It's a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable:

  ```
  f = 3.14159f;
  ```

- Without the suffix, the constant `3.14159` would have type `double`, possibly causing a warning message.

Implicit Conversions in C99

- C99's rules for implicit conversions are somewhat different.
- Each integer type has an "integer conversion rank."
- Ranks from highest to lowest:
  1. `long long int`, `unsigned long long int`
  2. `long int`, `unsigned long int`
  3. `int`, `unsigned int`
  4. `short int`, `unsigned short int`
  5. `char`, `signed char`, `unsigned char`
  6. `_Bool`
- C99's "integer promotions" involve converting any type whose rank is less than `int` and `unsigned int` to `int` (provided that all values of the type can be represented using `int`) or else to `unsigned int`.

Implicit Conversions in C99

- C99's rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.
- ***The type of either operand is a floating type.*** As long as neither operand has a complex type, the rules are the same as before. (The conversion rules for complex types are discussed in Chapter 27.)

Implicit Conversions in C99

- ***Neither operand type is a floating type.*** Perform integer promotion on both operands. Stop if the types of the operands are now the same. Otherwise, use the following rules:
  - If both operands have signed types or both have unsigned types, convert the operand whose type has lesser integer conversion rank to the type of the operand with greater rank.
  - If the unsigned operand has rank greater or equal to the rank of the type of the signed operand, convert the signed operand to the type of the unsigned operand.
  - If the type of the signed operand can represent all of the values of the type of the unsigned operand, convert the unsigned operand to the type of the signed operand.
  - Otherwise, convert both operands to the unsigned type corresponding to the type of the signed operand.

Implicit Conversions in C99

- All arithmetic types can be converted to `_Bool` type. The result of the conversion is 0 if the original value is 0; otherwise, the result is 1.

Casting

- Although C's implicit conversions are convenient, we sometimes **need a greater degree of control** over type conversion.
- For this reason, C provides ***casts.***
- A cast expression has the form

  (*type-name*) *expression*

  type-name specifies the type to which the expression should be converted.

Casting

- Using a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;

frac_part = f - (int) f;
```

- The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.
- Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f;  /* f is converted to int */
```

Casting

- Cast expressions also let us force the compiler to perform conversions.
- Example:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

- To avoid truncation during division, we need to cast one of the operands:

```
quotient = (float) dividend / divisor;
```

- Casting `dividend` to `float` causes the compiler to convert `divisor` to `float` also.

Casting

- C regards (*type-name*) as a unary operator.
- Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

- Other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
quotient = (float) dividend / (float) divisor;
```

Casting

- Casts are sometimes necessary to avoid overflow:

```
long i;
int j = 1000; //assume 16 bits wide
i = j * j;/* LHS overflow may occur */
```

- Using a cast avoids the problem:

```
i = (long) j * j;
```

- The statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.

Example

- int i=0;
- short j = 10000, k=0;

- sizeof(i)=4, sizeof(j)=2, sizeof(k)=2

- i=0, j=10000, k=0
- i = j * j;--> j=646045128, i=1000
- k = j * j;--> i=100000000, j=10000, k=16960

Type Definitions

- The `#define` directive can be used to create a "Boolean type" **macro**:

```
#define BOOL int
```

- There's a better way using a feature known as a ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names.
- Example:

```
Bool flag;   /* same as int flag; */
```

Advantages of Type Definitions

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

  ```
  typedef float Dollars;
  ```

  and then writing

  ```
  Dollars cash_in, cash_out;
  ```

  is more informative than just writing

  ```
  float cash_in, cash_out;
  ```

Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine `Dollars` as `double`, only the type definition need be changed:

  ```
  typedef double Dollars;
  ```
- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

  ```
  i = 100000;
  ```

  is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

Type Definitions and Portability

- For greater portability, consider using **`typedef`** to define new names for integer types.
- Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000.
- We could use `long` variables for this purpose, but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values. Also, `int` variables may take up less space.

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

  ```
  typedef int Quantity;
  ```

  and use this type to declare variables:

  ```
  Quantity q;
  ```

- When we transport the program to a machine with shorter integers, we'll change the type definition:

  ```
  typedef long Quantity;
  ```

- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with suffix `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

- In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits.

The **`sizeof`** Operator

- The value of the expression

  `sizeof (` *type-name* `)`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*.
- `sizeof(char)` is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, `sizeof(int)` is normally 4.

The **`sizeof`** Operator

- The `sizeof` operator can also be applied to constants, variables, and expressions in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.
- When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses.
  - We could write `sizeof i` instead of `sizeof(i)`.
- Parentheses may be needed anyway because of operator precedence.
  - The compiler interprets `sizeof i + j` as `(sizeof i) + j`, because `sizeof` takes precedence over binary `+`.

The **sizeof** Operator

- Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named **`size_t`**.
- In C89, it's best to convert the value of the expression to a known type before printing it:

```
printf("Size of int: %lu\n",
       (unsigned long) sizeof(int));
```

- The `printf` function in C99 can display a `size_t` value directly if the letter **`z`** is included in the conversion specification:

```
printf("Size of int: %zu\n", sizeof(int));
```