1 Basic Data Types

type	bytes	example	notes
`char`	1	`'a'`	`signed` or `unsigned`
`short`	2	`32000`	`signed` or `unsigned`
`int`	4	`2000000000`	`signed` or `unsigned`
`long`	8	19-digit number	`signed` or `unsigned`
`float`	4	6-7 significant digits	
`double`	8	15 significant digits	
`pointer`	8 (on 64-bit machine)	`int *ip`	use `void *p` for "wildcard" pointer
`array`		`double arr[10]`	*cannot* be used as a variable

Note. Ranges of `signed` integer ranges are slightly assymmetric (ex, `-128` to `127`) because of two's complement representation. If the qualifier `unsigned` is used, values are always nonnegative (for example, `char` ranges from `0` to `255`).

Booleans: C has no keywords for boolean values; instead, all nonnegative values are taken to be "true" and zero is taken to be "false"

Strings: C has no string type; instead, strings are stored as *character arrays* → note that strings are stored internally with a `'\0'` at the end (to delimit the end of the string)

Note. Variables which are only *declared* and not *initialized* will contain some "garbage" value.

Casting: syntax is `(type) variable`

2 Control Structures

C has many of the same control structures as Java. In particular, the following are valid in C:

1. the `if-else` statement block:

```
if ((something)&&(another)) {
    statement1;
    statement2; }
else if ((one)||(two)) {
    statement 3; }
else {
    statement 4; }
```

2. the `while` loop:

```
while (something) {
    statements;
}
```

3. the `for` loop:

```
for (i = 0; i < 100; i++) {
    statements;
}
```

4. keywords: `break`, `continue`

One piece of control that is unique to C is the `goto` and `label` syntax. These two keywords can be used to break out of multiple nested loops. For example:

```
while (condition) {
    while (condition) {
        if (disaster) {
            goto error;
        }
    }
}
error:
    do something;
```

Note that the scope of the `label` is the entire function and labels are used just like variable names.

3 Basic I/O

The `printf` function is used to output information to the terminal. It takes 2+ arguments: (1) a string indicating how the output should be formatted, and (2) a list of the data to be outputted. For example,

```
printf("%d %d", 7, 10) // prints 7 10
```

specifier	meaning
"%d"	print as decimal integer
"%6d"	decimal int, 6 characters wide
"%f"	floating point value
"%.2f"	floating point, to 2 characters after decimal point
"%c"	character
"%s"	character string

The `scanf` function is used to accept input from the command line.

Standard functions:

```
FILE *fopen(char *fname, char *mode);
int fclose(FILE *fh);
int fscanf(FILE *fh, char *format, addr1, ...);
int fprintf(FILE *fh, char *format, arg1, ...);
void rewind(FILE *fh);
```

Binary data I/O:

```
size_t fread(void *dest, size_t byte_size, size_t quantity, FILE *fh);
size_t fwrite(void *src, size_t byte_size, size_t quantity, FILE *fh);
```

4 Functions in C

C uses **header** files to declare functions that exist. If you don't want to use a header file, you need to declare functions one by one within the file with the `main` function.

5 Internal Representation

Computers have three possible internal representations of numbers:

1. **two's complement**
2. unsigned - can lead to overflow, obeys arithmetic rules (associativity and commutativity)
3. floating-point - uses $+\infty$ symbol, *not* associative

5.1 Binary and Hexadecimal

A **byte** of memory (8 bits) is the smallest addressable form of memory. Every byte of memory is labeled with a unique address. There are three different ways to represent a single byte of information:

1. decimal notation - ranges from 0 to 255
2. binary notation - ranges from 00000000 to 11111111
3. hexadecimal notation - ranges from 00 to FF

Conversion from one type to another:

type	how/example
decimal to binary	divide repeatedly by 2, take the remainders and arrange from right to left ex. $38 = 32 + 4 + 2 = 00100110$
binary to decimal	starting from the right, multiply each digit by 2^i where i is the place of the digit ex. $01001111 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^6 = 79$
decimal to hex	divide repeatedly by 16, take the remainders and arrange from right to left ex. $50 = 3 \cdot 16 + 2 = 32$
hex to decimal	starting from right, multiply each digit by 16^i where i is place of digit ex. $\text{FF} = 15 \cdot 16^1 + 15 = 255$
binary to hex	break into "blocks" of four from the right and convert each to decimal ex. $1011000111 = 0010\ 1100\ 0111 = 2\text{C}7$
hex to binary	convert each digit to a four-block of binary and concatenate ex. $39\text{A}7\text{F}8 = 0011\ 1001\ 1010\ 0111\ 1111\ 1000 = 001110011010011111111000$

Note. In C (and other languages), hexadecimal values are denoted as beginning with the string `"0x"` and letters can either be capitalized, lowercase, or a mixture.

Note. **Octal** typically arises in association with Unix file permissions. Every file has 3 permissions (read, write, execute) for 3 entities (user, group, and other). The readable version is to use `chmod u=rw g=rw o=rx`.

Word size: refers to the size of integer and pointer data; for example, most machines today are 64-bit which means that virtual addresses can range from 0 to $2^{64} - 1$.

Byte storage: A multi-byte object is stored as a contiguous sequence of bytes, with the address of the object pointing to the smallest address of bytes used. The two common conventions for ordering bytes are: (1) *little endian* where the least significant byte comes first, and (2) *big endian* where the most significant byte comes first.

5.2 Bitwise Operations

symbol	name	example
`&`	AND	`00111100 & 00001101 = 00001100`
`|`	OR	`00111100 | 00001101 = 00111101`
`^`	EXOR	`00111100 ^ 00001101 = 00110001`
`<<`	left shift	`00111100 << 2 = 11110000`
`>>`	right shift	`00111100 >> 2 = 00001111`
`~`	one's complement	`~00111100 = 11000011`

Note. There are technically two different variations of right shift. The first, **logical** right shift, fills the shifted places with zeroes while the second, **arithmetic** right shift, fills the shifted places with copies of the most significant bit. Note also that shifts only work on *integral* types (*not* floating point values).

To pull it all together, consider the following example:

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

How does this function work? The first portion (before the `&`) shifts the bits that we want to the rightmost location. The next portion creates a bit pattern with `0`s in all of the locations that we *don't* want and `1`s in all remaining locations. Finally, the `&` operator removes the unwanted bits.

Bit vectors: Let the set $A = \{0, 1, \ldots, w-1\}$ correspond to the bit vector $[a_{w-1}, \ldots, a_1, a_0]$ where $a_i = 1$ if and only if $i \in A$. For example, $A = \{0, 3, 5, 6\}$ corresponds to the bit vector [01101001].

Note. There is a nice correspondence between bitwise operations and set operations.

bitwise operation	set operation
`&` (AND)	$\bigcap$ (intersection)
`|` (OR)	$\bigcup$ (union)
`~` (NOT)	A^c (complement)

```
void swap_inplace(int *x, int *y) {
    *y = *x ^ *y; // y = x ^ y
    *x = *x ^ *y; // x = x ^ (x ^ y) = y
    *y = *x ^ *y; // y = y ^ (x ^ y) = x
}
```

5.3 Signed and Unsigned Integers

Two's complement encoding: the most common way to represent signed integers

- if $\mathbf{x} = [x_{w-1}x_{w-2}\cdots x_1x_0]$, then the encoding is $-x_{w-1}2^{w-1} + \cdots + x_1 2^1 + x_0$
- ex. $1011 = -1 \cdot 2^3 + 1 \cdot 2^1 + 1 = -5$
- maximum value: $[011\cdots1] = 2^{w-1} - 1$

- minimum value: $[100\ldots0] = -2^{w-1}$

Conversion between signed and unsigned: casting maintains the bit-level representations and simply changes the decimal value based on binary or two's complement encoding

- on the surface, C uses `(unsigned) x` and `(int) x` syntax
- signed → unsigned: if we have $\mathbf{x} = [x_{w-1}x_{w-2}\cdots x_1x_0]$ with signed encoding $\mathbf{x}'$, then the unsigned encoding is $x_{w-1}2^w + \mathbf{x}'$ or

$$x_{\text{unsigned}} = \begin{cases} x_{\text{signed}} + 2^w & \text{if } x < 0 \\ x_{\text{signed}} & \text{if } x \geq 0. \end{cases}$$

- unsigned → signed: if we have $\mathbf{x} = [x_{w-1}x_{w-2}\cdots x_1x_0]$ with unsigned encoding $\mathbf{x}'$, then the signed encoding is $-x_{w-1}2^w + \mathbf{x}'$ or

$$x_{\text{signed}} = \begin{cases} x_{\text{unsigned}} - 2^w & \text{if } x \geq 2^{w-1} \\ x_{\text{unsigned}} & \text{if } x < 2^{w-1}. \end{cases}$$

Quirks that are specific to C:

- implicit casting: if you assign a signed integer to an unsigned variable or vice versa, C will cast implicitly; this also occurs if you print a signed integer with `"%u"` or vice versa
- arithmetic operations: using `+, -, *`, etc on a combination of an unsigned and signed will cause the signed integer to be converted to unsigned

Expanding and compressing bits:

- expanding an unsigned integer: **zero extension** (add zeroes in front)
- expanding a signed integer: **sign extension** (add copies of the most significant bit)
  - Why does this work? Because if the most significant bit is `0`, we have the same situation as the unsigned case. If the MSB is `1`, then we see that

    $$-1 \cdot 2^w + 2^{w-1} = -2^{w-1}$$

    and the rest of the proof proceeds by induction.
- compressing an unsigned integer: compute $x \bmod 2^k$ if compressing to k bits
  - Why does this work? Because the the first $w - k$ bits are dropped.
- compressing a signed integer: convert to unsigned → compute $x \bmod 2^k$ → convert back to signed

5.4 Integer Arithmetic

Unsigned addition: if we have two w-bit integers x and y, their sum could potentially "overflow" (giving $w + 1$ bits)

- Solution? Compute **modulo** 2^w (can also think of this as discarding the most significant bit if overflow occurs).

Signed addition: if x and y are w-bit signed integers, then

$$x + y = \begin{cases} x + y - 2^w & \text{if } 2^{w-1} \leq x + y \\ x + y & \text{if } -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w & \text{if } x + y < -2^{w-1}. \end{cases}$$

This is essentially a variation of computing modulo 2^w where negative results are also reduced.

Signed negation: if x and y are w-bit signed integers, then

$$-x = \begin{cases} -2^{w-1} & \text{if } x = -2^{w-1} \\ -x & \text{if } -2^{w-1} < x < 2^{w-1}. \end{cases}$$

Unsigned multiplication: if x and y are w-bit unsigned integers, then

$$x \cdot y = (x \cdot y) \mod 2^w.$$

This means that any "overflow" bits are simply truncated off (from the most significant bit).

Signed multiplication: convert to unsigned integers → multiply as unsigned integers (including truncation) → convert back to signed integers

Why does this work? The low-order bits (post-truncation) of the result are the same, whether we multiply as unsigned integers or signed integers.

Multiplying by constants: for both unsigned and signed integers, multiplying by the kth power of 2 is equivalent to a left shift by k bits, i.e.,

$$x \cdot 2^k \quad \cong \quad \texttt{x << k}.$$

Dividing by constants:

- **logical** right shift by k bits is equivalent to division by 2^k for unsigned integers
- **arithmetic** right shift by k bits is equivalent to division by 2^k for signed integers

5.5 Floating Point Representation

IEEE floating point is the standardized way to represent floating point numbers. In 64-bit machines, the allocations are as follows:

name	sign	exponent	significand
# of bits	1	11	52
representation	s	$e_{k-1} \cdots e_1 e_0$	$f_{n-1} \cdots f_1 f_0$

The formulas for conversion are given below:

name	use	significand	exponent
normalized	exponent bits have 0s and 1s	$1 + \sum_{i=0}^{n-1} f_i \cdot 2^{-(n-i)}$	$e_{k-1} \cdots e_0$ (unsigned) $-2^{k-1} - 1$
denormalized	exponent bits all 0	$\sum_{i=0}^{n-1} f_i \cdot 2^{-(n-i)}$	$1 - (2^{k-1} - 1)$
special	exponent bits all 1	$+\infty, -\infty$, `NaN`	

5.6 Character Representation

The **ASCII standard** is most commonly used: uses 7 bits to encode a character. The more general form is **UTF-8** which uses the 8th bit to indicate an extension to more than a single byte.

6 Pointers and Arrays

Use the syntax `&c` to indicate that you want the **address** of `c`. This creates a **pointer** to the memory location at which `c` is stored. Note that this can only be applied to variables, objects, and arrays. Use the syntax `*c` to **dereference** a pointer and access the value to which it is pointing. For example:

```
int x = 1, y = 2;
int *ip; // indicates that ip is a pointer to an integer
ip = &x; // ip points to the location where x is stored
y = *ip; // set y to the same object as x
*ip = 0; // set whatever is at the location pointed to by ip to 0
```

Using pointers and functions:

```
double myfunc(char *); // myfunc takes in a pointer to a char and returns a double
```

Note. If `ip` points to the integer `x`, then `*ip` can be used in any context where `x` can. For example, `*ip = *ip + 10` is equivalent to `x = x + 10`.

Arrays: The syntax `int a[SIZE]` defines a block of `SIZE` consecutive objects `a[0], a[1],..., a[SIZE-1]`. By definition, the value of a variable of type array is the address of the 1st (0-index) element of the array. Because of this, the following pairs of statements are identical:

```
pa = &a[0];     pa = a;
a[i];           *(a+i);
&a[i];          a+i;
pa[i];          *(pa+i);
```

Essentially, an array-and-index expression is equivalent to one written as a pointer-and-offset.

However, arrays are *not* variables. The variable pointing to an array cannot be set equivalent to any other expression and cannot be incremented. Passing an array into a function is equivalent to passing a pointer to the 1st (0-index) element. For example:

```
int strlen(char *s) {
    int n;
    for (n = 0; *s != '\0'; s++) {
        n++;
    }
    return n;
}
/* this function "works" because passing in a string is equivalent to passing in a
character array which is equivalent to passing in a pointer to the 1st character */
```

Comparison of pointers and arrays:

property	pointer	array
location is:	changeable	fixed
location is:	assigned by user	determined by compiler
brace indexing?	`int z = p[0];`	`int z = a[0];`
dereferencing?	yes	no
assign to array?	yes	no
tracks length?	no	no

Note. Often, the difference between a pointer and an array won't be obvious and they can often be used interchangeably. For example, a function which takes an array as a formal parameter can also take a pointer and vice versa.

Initializing an array of characters: (i.e., a string)

```
char str[8] = "hello";
// stored implicitly as 'h' 'e' 'l' 'l' 'o' '\0'
printf("%s\n", str); // prints "hello"

// the following also works the same way
char *str = "hello";
```

6.1 Dynamic Memory Allocation

C uses `malloc()` and `free()` for the dynamic allocation and de-allocation of memory. In general, use these functions when:

- You have a data structure which needs to store an arbitrary set of data values of indeterminate length (linked list versus fixed array list).
- You need to pass a pointer to an object (an array, a string, etc) outside of a function.

```
int *ptr = malloc(4 * sizeof(int)); // mallocs 16 bytes
```

Note that memory that is `malloc()`'d is located in the heap, *not* the stack.

7 Data Structures

The `struct` is C's way of defining a collection of heterogeneous data. Each **field** in a struct has its own type. You can access elements with the dot notation. Access elements with the `->` notation if you want a pointer to the element.

```
typedef struct {
    int field1;
    double field2;
    int field3[6];
} thing_t

thing_t thing_a = {
    .field1 = 15,
    .field2 = 10.5,
```

```
    .field3 = {17, 27, 13},
}
```

Structs are stored in a sequential, contiguous block. However, there may be some "padding" so that the ends of each field align with word boundaries or even addresses.

Structs use both dot `.` and arrow `->` notation for access to elements. Some examples:

```
mystruct_t *ptr = malloc(sizeof(mystruct_t));
ptr->length = 0; // ptr->x is equal to *(ptr.x)
&ptr->length; // gives the actual memory address
mystruct_t *fiveptr = malloc(5 * sizeof(mystruct_t));
fiveptr[1].length = 0; // does same thing as above
// because brackets are implicit dereference
```