Computer Organization and Architecture

C types and expressions

Dr Russ Ross
Utah Tech University—Department of Computing

Spring 2026

An example

```c
#include <stdio.h>

double circularArea(double r);

int main(void) {
    double radius = 1.0, area = 0.0;
    printf("    Areas of Circles\n\n");
    printf("     Radius          Area\n"
           "------------------------\n");
    area = circularArea(radius);
    printf("%10.1f    %10.2f\n", radius, area);
    radius = 5.0;
    area = circularArea(radius);
    printf("%10.1f    %10.2f\n", radius, area);
    return 0;
}

double circularArea(double r) {
    const double pi = 3.1415926536;
    return pi * r * r;
}
```

Notes

- Header files
- Preprocessor
- Prototypes and forward/external references
- `main`
- `printf`
- Variable declarations

The most common `printf` verbs:

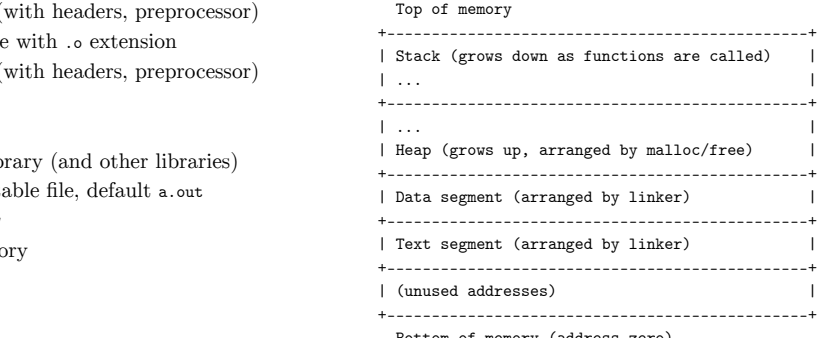
`%d, %u`	decimal integer
`%x, %X, %o`	hexadecimal, octal
`%f, %e, %g`	floats: 392.65, 3.9265e+2, shortest of %f/%e
`%c`	single characacter
`%s`	string
`%p`	pointer address
`%%`	literal percent sign (no operand)

Compiling

The compilation process:

1. Compile source file 1 (with headers, preprocessor)
   - Creates an *object* file with `.o` extension
2. Compile source file 2 (with headers, preprocessor)
   - Another object file
3. ...
4. Link with standard library (and other libraries)
   - Output is an executable file, default `a.out`
   - Also called a *binary*
   - See also: `/bin` directory

Memory map



Expression statements

Any expression followed by a semicolon is a statement. This is normally only used when the expression has a side effect.

```
y = x;                          // An assignment
sum = a + b;                    // Calculation and assignment
++x;
printf("Hello, world\n");       // A function call

100;                            // Correct, but not very useful
y < x;

(void) unused_variable;         // suppress warnings about an unused variable
```

A statement can also consist of a semicolon by itself: this is called a *null statement*:

```
for (i = 0; s[i] != '\0'; i++)
    ;
```

Block statements

A *compound statement*, call a *block* for short, groups statements and declarations between braces to form a single statement. Note that it is **not** terminated by a semicolon:

```
{
    double result = 0.0, x = 0.0;           // declarations
    static long status = 0;                 // these variables are scoped
    extern int limit;                       // to the current block

    x++;
    if (status == 0) {                      // new block
        int i = 0;                          // i is scoped to the if block
        while (status == 0 && i < limit) {  // another new block
            // ...
        }
    } else {                                // and another
        // ...
    }
}
```

while loops

A `while` loop executes a statement repeatedly as long as the controlling expression is true:

- `while (`*expression*`)` *statement*

```c
#include <stdio.h>

int main(void) {
    double x = 0.0, sum = 0.0;
    int count = 0;
    printf("\t--- Calculate Averages ---\n");
    printf("\nEnter some numbers:\n"
           "(Type a letter to end your input)\n");
    while (scanf("%lf", &x) == 1) {
        sum += x;
        count++;
    }
    if (count == 0)
        printf("No input data!\n");
    else
        printf("The average of your numbers is %.2f\n", sum/count);
    return 0;
}
```

for loops

A `for` loop has more loop logic contained in the statement itself:

- `for` (*expression1*; *expression2*; *expression3*) *statement*
  - *expression1*: **Initialization**. Evaluated only once, before the first evaluation of the controlling expression, to perform any necessary initialization
  - *expression2*: **Controlling expression**. Tested before each iteration. Loop execution ends when this expression evaluates to false.
  - *expression3*: **Adjustment**. An adjustment, such as incrementation of a counter, performed *after* each loop iteration and before *expression2* is tested again.

```
for (i = 0; i < LENGTH; i++)
    arr[i] = 2*i;

for (;;) { ... }

for (; more_to_do(x); ) { }

i = 0;
while (i < LENGTH) {
    arr[i] = 2*i;
    i++;
}
```

```
// starting with C99 you can declare a variable
// in place of expression1
for (int i = 0; i < LENGTH; i++)
    arr[i] = 2*i;

for (int i = 0, j = strlen(str)-1; i < j; i++, j--)
    ch = str[i], str[i] = str[j], str[j] = ch;
```

do while loops

The `do` ... `while` loop is *bottom driven*

- `do` *statement* `while (`*expression*`);`

The body statement is executed once before the controlling expression is evaluated for the first time, so at least one iteration of the loop is always performed.

```
do {
    int command = getCommand();
    performCommand(command);
} while (command != END);

char *strcpy(char *s1, const char *s2) {
    int i = 0;
    do
        s1[i] = s2[i];
    while (s2[i++] != '\0');
    return s1;
}
```

Nested loops

A loop body can be any simple or block statement and may include other loop statements.

```
void bubbleSort(float arr[], int len) {
    int isSorted;
    do {
        isSorted = 1;
        len--;
        for (int i = 0; i < len; i++) {
            if (arr[i] > arr[i+1]) {
                isSorted = 0;
                float temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    } while (!isSorted);
}
```

if statements

An `if` statement has the following form:

- `if (`*expression*`)` *statement1* with optional `else` *statement2*

```
double power(double base, unsigned int exp) {
    if (exp == 0) return 1.0;
    else return base * power(base, exp-1);
}
```

If several `if` statements are nested, then an `else` clause always belongs to the last `if` (on the same block nesting level that does not yet have an `else` clause. If there is any doubt or the nesting is complex, use a block and make sure your indentation matches the code.

```
if (n > 0)
    if (n%2 == 0)
        puts("n is positive and even");
    else
        puts("n is positive and odd");
```

```
if (n > 0) {
    if (n%2 == 0)
        puts("n is positive and even");
} else
    puts("n is negative or zero");
```

if statements

To select one of more than two alternative statements, `if` statements can be cascaded in an `else if` chain. Each new `if` statement is simply nested in the `else` clause of the preceding `if` statement:

```
double spec = 10.0, measured = 10.3, diff;
/* ... */
diff = measured - spec;

if (diff >= 0.0 && diff < 0.5)
    printf("Upward deviation: %.2f\n", diff);
else if (diff < 0.0 && diff > -0.5)
    printf("Downward deviation: %.2f\n", diff);
else
    printf("Deviation out of tolerance!\n");
```

switch statements

A `switch` statement causes a jump to one of several statements according to the value of an integer expression. The cases must all be constants:

- `switch (`*expression*`)` *statement*

```
switch (menu()) {   // menu() returns an int
case 'a':
case 'A':
    action1();
    break;
case 'b':
case 'B':
    action2();
    break;
default:
    putchar('\a');
}
```

- Cases *fall through* so you must add a `break` statement to exit the `switch`
- To declare variables in a case you must introduce a nested block

break

The `break` statement can occur only in the body of a loop or a `switch` statement and jumps to the first statement after the loop or `switch` in which it is immediately contained:

- `break;`

```
int getScores(short scores[], int len) {
    puts("Please enter scores between 0 and 100.\n"
         "Press <Q> and <Enter> to quit.\n");
    int i;
    for (i = 0; i < len; i++) {
        printf("Score No. %2d: ", i+1);
        if (scanf("%hd", &scores[i]) != 1)
            break;
        if (scores[i] < 0 || scores[i] > 100) {
            printf("%d: Value out of range.\n", scores[i]);
            break;
        }
    }
    return i;
}
```

continue

The `continue` statement can be used only within the body of a loop, and causes the program to skip over the rest of the current iteration of the loop:

```
int getScores(short scores[], int len) {
    puts("Please enter scores between 0 and 100.\n"
         "Press <Q> and <Enter> to quit.\n");
    int i;
    while (i < len) {
        printf("Score No. %2d: ", i+1);
        if (scanf("%hd", &scores[i]) != 1)
            break;
        if (scores[i] < 0 || scores[i] > 100) {
            printf("%d: Value out of range.\n", scores[i]);
            continue;   // discard this value and read in another
        }
        i++;
    }
    return i;
}
```

- In a `while` or `do ... while` loop, the program jumps to the next evaluation of the controlling expression
- In a `for` loop, the program jumps to the evaluation of the third expression (the increment operation, typically)

goto

The `goto` statement causes an unconditional jump to another statement in the same function. The destination is specified by the name of a label.

```
bool calculate(double arr[], int len, double *res) {
    bool error = false;
    if (len < 1 || len > MAX_ARR_LENGTH)
        goto error_exit;
    for (int i = 0; i < len; i++) {
        // do stuff that might set error
        if (error)
            goto error_exit;
        // continue calculation and set *res
    }
    return true;

error_exit:
    *res = 0.0;
    return false;
}
```

- You should never use `goto` to jump into a block from outside if the jump skips over declarations or statements that initialize variables
- Code that makes heavy use of `goto` statements is hard to read and should be avoided. The most common uses for `goto` are:
  - Consolidating exits from a function (see example)
  - Simulating a `break` or `continue` from an inner loop directly to an outer loop

return

The `return` statement ends execution of the current function and jumps back to where the function was called:

- `return` [*expression*];

The *return value* is converted to the function's return type if necessary.

```
int min(int a, int b) {
    if (a < b) return a;
    else       return b;
}
```

See also

```
int min(int a, int b) {
    return a < b ? a : b;
}
```

For functions with `void` return type the expression is omitted.

Operators

Precedence	Operators	Associativity
1.	Postfix operators: `[] () . -> ++ -- (type-name){list}`	Left to right
2.	Unary operators: `++ -- ! ~ + - * & sizeof`	Right to left
3.	The cast operator: `(type-name)`	Right to left
4.	Multiplicative operators: `* / %`	Left to right
5.	Additive operators: `+ -`	Left to right
6.	Shift operators: `<< >>`	Left to right
7.	Relational operators: `< <= > >=`	Left to right
8.	Equality operators: `== !=`	Left to right
9.	Bitwise AND: `&`	Left to right
10.	Bitwise exclusive OR: `^`	Left to right
11.	Bitwise OR: `|`	Left to right
12.	Logical AND: `&&`	Left to right
13.	Logical OR: `||`	Left to right
14.	The conditional operator: `? :`	Right to left
15.	Assignment operators: `= += -= *= /= %= &= ^= |= <<= >>=`	Right to left
16.	The comma operator: `,`	Left to right

Memory addressing operators

Operator	Meaning	Example	Result
`&`	Address of	`&x`	Pointer to `x`
`*`	Indirection operator	`*p`	The object or function that `p` points to
`[]`	Subscripting	`x[y]`	The element with index `y` in the array `x`
`.`	Structure or union member designator	`x.y`	The member named `y` in the structure or union `x`
`->`	Structure or union member designator by reference	`p->y`	The member named `y` in the structure or union that `p` points to

```
float x, *ptr;
ptr = &x;           // OK: Make ptr point to x.
ptr = &(x+1);       // Error: (x+1) is not an lvalue

*ptr = 1.7;         // Assign the value 1.7 to the variable x
++(*ptr);           // and add 1 to it. (note: parentheses are superfluous)
```

Element of arrays

The *subscript operator* `[]` allows you to access individual elements of an array. In its simplest form:

```
myarray[i]  // note: arrays always start with element 0
```

An expression of the form `x[y]` is equivalent to

```
(*((x)+(y)))
```

Either `x` or `y` must have a type that is a pointer to an object type, and the other must have an integer type. This follows the rules of *pointer arithmetic* and means that `x[y]` and `y[x]` are equivalent.

```
#include <stdlib.h>
#define ARRAY_SIZE 100
// ...
    double *pArray = NULL; int i = 0;
    pArray = malloc(ARRAY_SIZE * sizeof(double));
    if (pArray != NULL) {
        for (i = 0; i < ARRAY_SIZE; i++)
            pArray[i] = (double) rand() / RAND_MAX;
    }
// note: pArray[i] or i[pArray] or *(pArray + i)
```

Compound literals

A *compound literal* lets you define literals with any object type and consiste of an object type in parentheses followed by an initialization list in braces:

```
float *fPtr = (float []){ -0.5, 0.0, +0.5 };
```

```
#include "database.h"
// includes: struct Pair { long key; char value[32]; };
insertPair(&db, &(struct Pair){ 1000L, "New York JFK Airport" });
```

Integer types

The basic signed integer types

Type	Synonyms
signed char	
int	signed, signed int
short	short int, signed short, signed short int
long	long int, signed long, signed long int
long long	long long int, signed long long, signed long long int

For each signed type, there is a corresponding unsigned type of the same memory size and alignment

Type	Synonyms
_Bool	bool (defined in stdbool.h)
unsigned char	
unsigned int	unsigned, unsigned int
unsigned short	unsigned short int
unsigned long	unsigned long int
unsigned long long	unsigned long long int

Note: `char` can be signed or unsigned. Be explicit if it matters.

Integers with exact width

The header `stdint.h` defines some aliases for when *width* matters

Type	Meaning	Implementation
`intN_t`, `uintN_t`	An integer type whose width is exactly N bits	Optional
`int_leastN_t`, `uint_leastN_t`	An integer type whose width is a least N bits	Required for `N = 8, 16, 32, 64`
`int_fastN_t`, `uint_fastN_t`	The fastest process with width at least N bits	Required for `N = 8, 16, 32, 64`
`intmax_t`, `uintmax_t`	The widest integer type implemented	Required
`intptr_t`, `uintptr_t`	An integer type wide enough to store a pointer	Optional

Floating-point types

Floating-point types:

- `float`: 32 bits, $\pm 3.4 \times 10\hat{}38$, 6–7 digits of precision
- `double`: 64 bits, $\pm 1.7 \times 10\hat{}308$, 15–16 digits of precision
- `long double`: 80 bits (we will mostly ignore these)

Enumerated types

A special type of integer where you name and list all of the defined values:

```
enum color { black, red, green, yellow, blue, white=7, gray };
```

Here `color` is the *tag* and the color names are the *enumeration constants*.

To use it:

```
enum color bgColor = blue, fgColor = yellow;
void setFgColor(enum color fgc);
```

Different constants may have the same value:

```
enum { OFF, ON, STOP=0, GO=1, CLOSED=0, OPEN=1 };
```

Note there is no tag. This is useful when you just want to define some constants, but not necessarily a new type to go with them.

The void type

The `void` type specifier means no type is available. This is useful:

```
// in function declarations
void perror(const char *);

// to explicitly discard a value
(void) printf("I don't need the return value\n");

// pointers to void
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Literals

A *literal* value is a value that is written directly, as opposed to one that is computed by an expression.

Integer literals

Integers can be written in:

- decimal (starts with a non-zero digit)
- hexadecimal (starts with `0x` or `0X`
- octal (starts with a `0`)

Constants usually default to `int`, but if the value is too big the compiler will choose a bigger size. You can also be explicit with suffixes:

Integer constant	Type
`0x200`	int
`512U`	unsigned int
`0L`	long
`0XfOfUL`	unsigned long
`0777LL`	long long
`0xAAAllu`	unsigned long long

Float literals

A float literal consists of a sequence of decimal digits with a decimal point, optionally with an exponent:

Floating-point constant	Value
`10.0`	10
`2.32E5`	2.34×10^5
`67e-12`	67.0×10^{-12}

The default size of a constant is a `double`. You can append `f` or `F` to get a single-precision float instead.

Character literals

A character constant is written inside single-quote marks:

```
'a'   '0'   '*'
```

A few characters are written using a backslash escape sequence:

```
'\''  '\\'  '\n'  '\t'
```

A character constant has type `int`, and the value is the ASCII code of the character (there are some other cases that we will ignore here).

```
int c = getchar();
if (c != EOF && c >= '0' && c <= '9') {
    // the user entered a digit
}
```

You can also enter an explicit numeric code:

```
'\xA3'   // the character with value 163
```

Escape sequences

Escape sequence	Value	Action when printed
`\'`	single quotation mark (')	
`\"`	double quotation mark (")	
`\?`	question mark (?)	
`\\`	backslash character (\)	
`\a`	alert	beep or other signal
`\b`	backspace	move left one character
`\f`	form feed	move to next page/clear screen
`\n`	line feed (newline)	move to beginning of next line
`\r`	carriage return	move to beginning of current line
`\t`	horizontal tab	move to next horizontal tab stop
`\v`	vertical tab	move to next vertical tab stop
`\o`, `\oo`, or `\ooo` (octal digits)	character with given octal value	
`\xh[h...]` (hex digits)	character with given hex value	
`\uhhhh`, `\Uhhhhhhhh`	character with given universal char name	

String literals

A *string literal* consists of a sequence of characters (and/or escape sequences) enclosed in double quotation marks:

```
"Hello, world!\n"
```

A string literal can be used to initialize a character array:

```
char msg1[100] = "the array will have space for 100 characters";
char msg2[] = "this array will be just the right size";
```

It can also be used to initialize a character pointer:

```
char *msg3 = "this is a string constant and msg3 points to it";
```

Multiple string literals in the source will be contatenated at compile time into a single string:

```
printf("ID   | Name\n"
       "-----|-------\n");

#define MY_EMAIL_ADDRESS "jdoe@example.com"
printf("Email me at " MY_EMAIL_ADDRESS " with suggestions\n");
```

Type conversions

An expression may involve values of different types:

```
double dVar = 2.5;
dVar *= 3;
if (dVar < 10L) { ... }
```

At each step, the compiler converts the two values into one compatible type before performing the operation. This only works for scalar types (integers, floats, booleans, pointers) not structs.

You can explicitly *type cast* a value:

```
int sum = 22, count = 5;
double mean = (double) sum / count;
```

Type casting has the same precedence level as most unary operators (which beats most binary operators) so `sum` is first converted to a `double` and then the division is performed. A few basic rules help understand what happens next:

- Operations between two integers will always yield an integer
- Operations between two floats will always yield a float
- Operations that mix an integer and a float will yield a float

Implicit type conversion

The hierarchy of types:

- Any two unsigned integer types have different conversion ranks. If one is wider than the other, then it has a higher rank.
- Each signed integer type has the same rank as the corresponding unsigned type. The type `char` has the same rank as `signed char` and `unsigned char`.
- The standard integer types are ranked in the order:

  ```
  _Bool < char < short < int < long < long long
  ```

- Every enumerated type has the same rank as its corresponding integer type
- The floating-point types are ranked in the following order:

  ```
  float < double < long double
  ```

- The lowest-ranked floating-point type, `float`, has a higher rank than any integer type.

Implicit type conversion

There are many details, but the general goal is to pick the type that can represent a wider range of values. A couple exceptions:

- Converting to a float can lose precision for large numbers
- Converting to an unsigned can lose negative numbers

A few other special cases. The common theme is that data must fit in its container:

- Assignments and initializations: the value of the right operand is always converted to the type of the left operand
- Function calls: arguments are converted to the types of the formal parameters
- Return statements: value is converted to the function's return type

Arrays

The definition of an array determines its name, the type of its elements, and the number of elements in the array (the *capacity*):

```
char buffer[4*512];
```

The size of an array is always the size of a single element times the number of elements in the array. There is no hidden metadata:

```
sizeof(buffer) == sizeof(char) * 2048
```

When defining an array, the number of elements must be a constant expression except under certain limited circumstances. The result is a *fixed-length* or a *variable-length* array.

Fixed-length arrays

Most arrays specify the number of elements as a constant expression. Such arrays can have any storage class. You can define them:

- Outside all functions
- Within a block
- With or without `static` storage class specifier
- Inside a `struct`

An important restriction: arrays cannot be used as function parameters or return values. An array argument passed to a function is always **converted to a pointer** to the first array element.

Variable-length arrays

An array with a nonconstant expression for the number of elements:

- Has *automatic* storage duration, i.e., it must be defined inside a block
- Cannot have a `static` storage class
- Is allocated on the stack
- Name must be an *ordinary identifier*, so it cannot be a member of a `struct` or `union`

```
void func(int n) {
    int vla[2*n];           // OK: storage duration is automatic
    static int e[n];        // Illegal: a variable-length array
                            //   cannot have static storage duration
    struct S { int f[n]; }; // Illegal: f is not an ordinary identifier
    /* ... */
}
```

Accessing array elements

The *subscript operator* `[]` provides an easy way to address the individual elements of an array by index:

```
#define A_SIZE 4
long myArray[A_SIZE];
for (int i = 0; i < A_SIZE; i++)
    myArray[i] = 2 * i;
```

An array index can be any integer expression. The subscript operator `[]` does **not** bring any range checking with it.

```
long myArray[4];
myArray[4] = 8;             // Error: subscript must not exceed 3
```

Pointer arithmetic

Another way to access array elements is to use *pointer arithmetic*.

> The name of an array is implicitly converted into a pointer to the first element of the array in all expressions except `sizeof` operations.

The following uses a pointer instead of an index to step through an array and double the value of each element:

```
for (long *p = myArray; p < myArray + A_SIZE; p++)
    *p *= 2;
```

When you add an integer to a pointer, the integer is automatically scaled to the size of the type of element the pointer refers to, so:

```
p++                         // adds sizeof(*p) == sizeof(long) to p
myArray + A_SIZE            // adds sizeof(long) * A_SIZE to myArray
```

Memory addressing operators

Operator	Meaning	Example	Result
`&`	Address of	`&x`	Pointer to `x`
`*`	Indirection operator	`*p`	The object or function that `p` points to
`[]`	Subscripting	`x[y]`	The element with index `y` in the array `x`
`.`	Structure or union member designator	`x.y`	The member named `y` in the structure or union `x`
`->`	Structure or union member designator by reference	`p->y`	The member named `y` in the structure or union that `p` points to

```
float x, *ptr;
ptr = &x;           // OK: Make ptr point to x.
ptr = &(x+1);       // Error: (x+1) is not an lvalue

*ptr = 1.7;         // Assign the value 1.7 to the variable x
++(*ptr);           // and add 1 to it. (note: parentheses are superfluous)
```

Initializing arrays

If you do not explicitly initialize an array:

- If the array has automatic storage duration then its elements have undefined values
- Otherwise (global and `static`), all elements are initialized by default to 0 (pointers are initialized to `NULL`)

You can initialize an array when you define it:

```
int a[4] = { 1, 2, 4, 8 };
```

Initialization lists

The following rules apply:

- You cannot include an initialization in the definition of a variable-length array.
- If the array has `static` storage duration, then the array initializers must be constant expressions. If the array has automatic storage duration, then you can use variables in its initializers.
- You may omit the length of the array in the definition if you supply an initialization list. The length is then determined by the index of the last array element for which the list contains an initializer.
- If the definition contains both a length specification and an initialization list, then the length is that specified by the expression between the brackets. Any elements for which there is no initializer in the list are initialized to zero (or NULL). If the list contains extra initializers they are ignore.
- A superflous comma after the last initializer is ignored.

As a result, these are all equivalent:

```
int a[4] = { 1, 2 };
int a[]  = { 1, 2, 0, 0 };
int a[]  = { 1, 2, 0, 0, };
int a[4] = { 1, 2, 0, 0, 5 };
```

Initialization lists

Array initializers must have the same type as the array elements. If the elements' type is a union, structure, or array type, then each initializer is generally another initialization list:

```
typedef struct {
    unsigned long pin;
    char name[64];
    /* ... */
} Person;
Person team[6] = { {1000, "Mary"}, {2000, "Harry"} };
```

The other four elements of the array `team` are initialized to 0, or in this case to `{0, ""}`.

Strings

A *string* is a continuous sequence of characters terminated by `\0`, the null character. The *length* of the string is considered to be the number of characters **excluding** the terminating null character.

There is no string type in C. Consequently there are no operators that accept strings as operands. Instead strings are stored in arrays of type `char` and the standard library provides numerous functions to perform basic operations on strings, such as copying, comparing, and concatenating them.

You can initialize arrays of `char` using string literals. The following are equivalent:

```
char str1[30] = "Let's go";     // string length: 8, array length: 30
char str1[30] = { 'L', 'e', 't', '\'', 's', ' ', 'g', 'o', '\0' };
```

An array holding a string must always be at least one element longer than the string length to accomodate the terminating null character. If you define a character array without an explicit length and initialize it with a string literal, the array created is one element longer than the string length:

```
char str2[] = " to London!";    // string length 11, array length 12
```

String concatenation

The following uses the standard library function `strcat()` to append the string in `str2` to the string in `str1`. The array `str1` must be large enough to hold all the characters in the concatenated string:

```
#include <string.h>

char str1[30] = "Let's go";
char str2[] = " to London!";

/* ... */

strcat(str1, str2);
puts(str1);
```

This prints:

```
Let's go to London!
```

The names `str1` and `str2` are pointers to the first character of the string stored in each array. This is called a *pointer to a string* or a *string pointer* for short. String manipulation functions receive the beginning addresses of strings as their arguments, and they generally process a string character by character until they reach the terminating null.

String concatenation

Here is a possible implementation of `strcat()`:

```
// The function strcat() appends a copy of the second string
// to the end of the first string.
// Arguments: Pointers to the two strings.
// Return value: A pointer to the first string, now concatenated with the second.
char *strcat(char *s1, const char *s2) {
    char *rtnPtr = s1;
    while (*s1 != '\0')             // Find the end of string s1.
        s1++;
    while ((*s1++ = *s2++) != '\0') // The first character from s2 replaces
        ;                           // the terminator of s1.
    return rtnPtr;
}
```

To test if this is safe, you must make sure there is enough space in `s1` for both strings plus the terminating null:

```
if (sizeof(str1) >= (strlen(str1) + strlen(str2) + 1))
    strcat(str1, str2);
```

Multidimensional arrays

A *multidimensional array* in C is array whose elements are themselves arrays. The elements of an n-dimensional array are $(n-1)$-dimensional arrays.

A multidimensional array declaration has a pair of brackets for each dimension:

```
char screen[10][40][80];            // a three-dimensional array
```

`screen` has 10 elements: `screen[0]` to `screen[9]`. Each is a two-dimensional array, consisting in turn of 40 one-dimensional arrays of 80 characters each. The array `screen` contains a total of 32,000 elements with type `char`.

To access a `char` element you must specify three indices:

```
screen[9][39][79] = 'Z';            // set the very last element to 'Z'
```

Matrices

Two-dimensional arrays are also called *matrices*. They are used frequently and merit some extra attention. It is helpful to think of the elements of a matrix as being arranged in rows and columns:

```
float mat[3][5];                    // three rows and 5 columns
```

The three elements `mat[0]`, `mat[1]`, and `mat[2]` are the rows of `mat`. Each row is an array of five `float` elements:

mat	0	1	2	3	4
mat[0]	0.0	0.1	0.2	0.3	0.4
mat[1]	1.0	1.1	1.2	1.3	1.4
mat[2]	2.0	2.1	2.2	2.3	2.4

We can initialize this using nested loops. The first index specifies the row and the second addresses a column in the row:

```
for (int row = 0; row < 3; row++)
    for (int col = 0; col < 5; col++)
        mat[row][col] = row + (float)col / 10;
```

In memory the three rows are stored consecutively since they are the elements of the array `mat`. This is called *row-major order*.

Declaring multidimensional arrays

In an array declaration that is not a definition, the array type can be incomplete. Such a declaration is a reference to an array that you must define with a specified length somewhere else in the program.

- You must always declare the complete type of an array's elements
- For a multidimensional array declaration, only the first dimension can have an unspecified length
- For example, in a two-dimensional matrix you must always specify the number of columns

If `mat` is a global array defined in one file, you can use it from a different file by declaring its type:

```
extern float mat[][5];              // external declaration
```

The external object so declared as an incomplete type-dimensional array type.

Arrays as arguments of functions

When the name of an array appears as a function argument, the compiler implicitly converts it into a pointer to the array's first element.

- The parameter of the function is always a pointer to the same object type as the type of the array elements
- You can declare the parameters either in array form or in pointer form:
  - `type name[]`
  - `type *name`

When you pass a multidimensional array as a function argument, the function receives a pointer to an array type. Because this array type is the type of the elements of the outermost array dimension, it must be a complete type. For this reason you must specify all dimensions of the array elements in the corresponding function parameter declaration.

For example, the type of a matrix parameter is a pointer to a "row" array and the length of the rows (the number of "columns") must be included in the declaration:

```
#define NCOLS 10
/* ... */
void somefunction(float (*pMat)[NCOLS]);    // pointer to a row array
void somefunction(float pMat[][NCOLS]);     // equivalent
```

The parentheses in `(*pMat)` are necessary here. Without them, `float *pMat[NCOLS]` would declare the identifier `pMat` as an array whose elements have the type `float *`, or pointers to `float`.

Working with multidimensional arrays

It is a good idea to give a name to the $(n-1)$-dimensional elements of an n-dimensional array. Such `typedef` names can make your program more readable and your arrays easier to handle:

```
typedef float ROW_t[NCOLS];                 // a type for the row arrays

void printRow(ROW_t pRow) {
    for (int c = 0; c < NCOLS; c++)
        printf("%6.2f", pRow[c]);
    putchar('\n');
}

void printMatrix(ROW_t *pMat, int nRows) {
    for (int r = 0; r < nRows; r++)
        printRow(pMat[r]);
}
```

Working with multidimensional arrays

The following defines and initializes an array of rows with type `ROW_t` and then calls `printMatrix()`:

```
ROW_t mat[] = {
    { 0.0F, 0.1F },
    { 1.0F, 1.1F, 1.2F },
    { 2.0F, 2.1F, 2.2F, 2.3F },
};
int nRows = sizeof(mat) / sizeof(ROW_t);
printMatrix(mat, nRows);
```

Declaring pointers

A *pointer* is a reference to a data object or a function. They have many uses:

- Sharing data between two functions (caller and callee)
- Implementing trees, graphs, linked lists, and other dynamic data structures (including recursive data structures)
- Reducing the amount of data copied when sharing objects

A pointer represents both the **address** and the **type** of an object or function.

```
int *ptr;                   // ptr is a pointer to an int
ptr = &var;                 // let ptr point to the variable var

// * is part of an individual declarator
int var = 77, *ptr = &var;
int* a, b;                  // a is a pointer, b is not

// printing pointers
printf("Value of ptr (the address of var): %p\n"
       "Address of ptr:                    %p\n", ptr, &ptr);
```

Every pointer is the same size in memory.

Null pointers

A *null pointer* is a pointer with the value 0. The macro `NULL` is defined in `stdlib.h`, `stdio.h` and other header files as a null pointer constant.

- A null pointer is always unequal to any valid pointer to an object or function.
- Functions that return a pointer type usually use a null pointer to indicate a failure condition

```c
#include <stdio.h>
/* ... */
FILE *fp = fopen("demo.txt", "r");
if (fp == NULL) {
    // Error: unable to open the file for reading
}
```

The null pointer constant is implicitly converted to other pointer types as necessary for assignments and comparisons, so no cast operator is necessary in the example.

void pointers

A pointer to `void` or *void pointer*, is a pointer with the type `void *`. There are no objects with type `void` so a `void *` is a generic pointer type that can represent any object address (but not its type).

To declare a function that can be called with different types of pointer arguments, you can declare the parameters as pointers to `void`:

```
void *memset(void *s, int c, size_t n);

struct Data { /* ... */ } record;
memset(&record, 0, sizeof(record));
```

The compiler implicitly converts any pointer type to `void *` as needed. The implicit conversion works the other way, too:

```
int *ptr = malloc(1000 * sizeof(int));
```

Using pointers to read and modify objects

The indirection operator * yields the location in memory whose address is stored in a pointer. If `ptr` is a pointer then `*ptr` is the object (or function) that `ptr` points to. This is called *dereferencing* a pointer.

```
double x, y, *ptr;          // two double variables and a pointer to double
ptr = &x;                   // let ptr point to x
*ptr = 7.8;                 // assign the value 7.8 to the variable x
*ptr *= 2.5;                // multiply x by 2.5
y = *ptr + 0.5;             // assign y the result of adding 0.5 to x
```

Do not confuse the * is a pointer declaration with the indirection operator. Think of the syntax of the declaration as an illustration of how to use the value:

```
double *ptr;                // how to declare a pointer to double
ptr = &x;                   // how to assign the value of the pointer
*ptr = 5;                   // how to assign the value of the double
```

Pointers to pointers

A pointer variable is itself an object in memory:

```
char c = 'A';
char *ptr = &c;
char **ptrptr = &ptr;
```

`ptrptr` points to where `ptr` is located in memory

```
**ptrptr = 'B';             // change the value of c
char c2;
*ptrptr = &c2;              // change ptr to point to c2
*ptr = 'C';                 // change the value of c2
**ptrptr = 'D';             // also change the value of c2
```

Linked list example

Consider a basic linked list:

```
struct IntListItem {
    int value;
    struct IntListItem *next;
};
typedef struct IntListItem IntListItem;

typedef struct IntList {
    IntListItem *head;
} IntList;
```

Linked list example

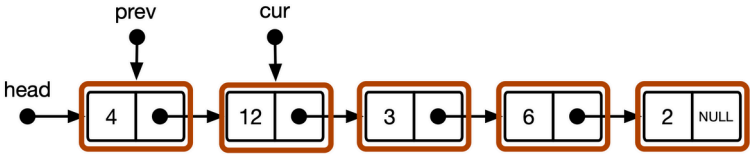


Figure 1: Conventional linked list

```
void remove(IntList *list, IntListItem *target) {
    IntListItem *cur = list->head, *prev = NULL;
    while (cur != target) {
        prev = cur;
        cur = cur->next;
    }
    if (prev) {
        prev->next = cur->next;
    } else {
        list->head = cur->next;
    }
}
```

Linked list example

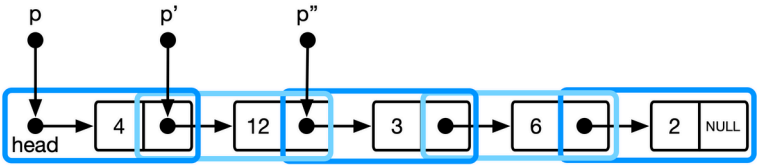


Figure 2: Link list with a single indirect pointer

```
void remove(IntList *list, IntListItem *target) {
    IntListItem **indirect = &l->head;
    while ((*indirect) != target) {
        indirect = &(*indirect)->next;
    }
    *indirect = target->next;
}
```

Modifying and comparing pointers

You can perform the following operations on pointers to objects. These operations are mostly useful when the pointers refer to elements of an array:

- Adding an integer to, or subtracting an integer from, a pointer
  - Pointer arithmetic applies: the integer is scaled by the size of the type the pointer points to
- Subtracting one pointer from another
  - Pointer arithmetic applies: the result is the number of *elements* that separate the two pointers (**not** necessarily the number of bytes)
- Comparing two pointers
  - Test if two pointers point to the same element, or determine which points to the element with the higher/lower index

So:

- Give a pointer `a`, `a + i` is a pointer to `a[i]`, and the value of `*(a + i)` is the element `a[i]`
- The expression `p1 - a` yields the index `i` of the element referenced by `p1`

Pointer arrays

A *pointer array*, an array whose elements are pointers, is an alternative to two-dimensional arrays. Often the pointers in the array point to elements of varying size. Compare:

```
char myStrings[100][256] = {
    "If anything can go wrong, it will.",
    "Nothing is foolproof, because fools are so ingenious.",
    "Every solution breeds new problems."
};

char *myStrPtr[100] = {
    "If anything can go wrong, it will.",
    "Nothing is foolproof, because fools are so ingenious.",
    "Every solution breeds new problems."
};
```

In the first case, we reserve exactly 256 bytes for each string, or exactly 100×256 bytes in total, which wastes space for most strings but may still be too short for others.

In the second case we have a fixed array of pointers, but the strings they point to can be of varying length, can be allocated dynamically, or could even be `NULL` to mark absent entries. It is useful to think of it as an array of strings, where each string is a `char *`, but it can also apply to non-string types.

Structures

Data structures are made up of

- *primitive types* like `int`, `double`, `char *`, etc., and
- *composite types* that combine them.

The primary composite types include arrays (lists of elements of the same type) and *structures*, also called *records*, which collect members of different types into a single object. Other more complex data structures are built up from these pieces.

```
struct Date {
    short month;
    short day;
    short year;
};
```

A `struct` must have at least one member. The *tag* is optional.

Defining structure types

The members of a `struct` may have any complete type, including previously-defined structure types. They must not be variable-length arrays or pointers to such arrays.

```
struct Song {
    char title[64];
    char artist[32];
    char composer[32];
    short duration;
    struct Date published;
};
```

Each member is embedded in the `struct`, and an instance of a `struct` includes its members laid out in memory in the order they are declared in the type.

A `struct` type cannot contain itself as a member (what would the `sizeof` such a type be?), however it can contain a pointer to its own type:

```
struct Cell {
    struct Song song;
    struct Cell *next;
};
```

Structures and typedefs

You can declare objects of `struct` types:

```
struct Song song1, song2, *pSong = &song1;
```

The type name is `struct Song`, not `Song`. You can use `typedef` to define a one-word name for a `struct` type:

```
typedef struct Song Song_t;
Song_t song1, song2, *pSong = &song1;
```

You can also define a `struct` without a tag. This is most commonly combined with `typedef`:

```
typedef struct {
    struct Cell *pFirst;
    struct Cell *pLast;
} SongList_t;
```

`struct` definitions are often placed in a header file so the type can be used across multiple source files. The definition is also commonly combined with the prototypes of functions that act on the `struct`.

Accessing structure members

The *dot operator* (.) lets you access members of a `struct` when the left operand is a structure object:

```
Song_t song1, song2, *pSong = &song1;

// passing an array to a function yields a pointer
strcpy(song1.title, "Flaming Wreck");
strcpy(song1.composer, "Joe Pernice");
song1.duration = 335;

// song1.published is a struct
song1.published.year = 2001;

// pSong is not a song object, but *pSong is
if ((*pSong).duration > 180)
    printf("The song %s is more than 3 minutes long\n", (*pSong).title);
```

Accessing structure members

If you have a pointer to a structure you can use the *arrow operator* (`->`) to access the structure's members instead of combining the indirection and dot operators (`*` and `.`), i.e., `p->m` is equivalent to `(*p).m`:

```
if (pSong->duration > 180)
    printf("The song %s is more than 3 minutes long\n", pSong->title);
pSong->published.year = 2001;
```

You can use assignment to copy the entire contents of a structure object to another object of the same type:

```
song2 = song1;
```

This copies the entire region of memory including

- The contents of embedded `struct` members
- The contents of arrays
- Pointer values, but **not** the values they point to

Similarly, passing a `struct` by value to a function (or returning one) copies the entire object. This is fine when objects are small and memory sharing is not desired, but it is common to share larger objects (or any objects when sharing is called for) using pointers to the objects.

Unions

A *union* is declared and defined like a structure, but instead of laying out members in memory one after another, the members of a union occupy the same storage, letting you select one of several alternative values for a single object.

```
union tree {
    struct leaf leaf_node;
    struct branch branch_node;
};

union tree t;
t.branch_node.key = 536;
```

The size of a `union` is the size of its largest member. A `union` must be initialized and used consistently through only one of its members (say, `branch_node`). It may be re-initialized as a different member (say `leaf_node`) and then used consistently through that member.

Discriminated unions

Unions are usually combined with structures to implement *discriminated unions*, a kind of *sum type*, where a single value may take on one of several shapes.

```
enum message_type { CREATE, READ, UPDATE, DELETE };
struct message {
    enum message_type type;
    union {
        struct create_message create;
        struct read_message read;
        struct update_message update;
        struct delete_message delete;
    } body;
};

struct message msg;
/* ... */
switch (msg.type) {
case CREATE:
    printf("create message with id %d\n", msg.body.create.id);
    break;
// ...
}
```