Computer Organization and Architecture

C arrays and pointers

Utah Tech University

Spring 2025

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

```
#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

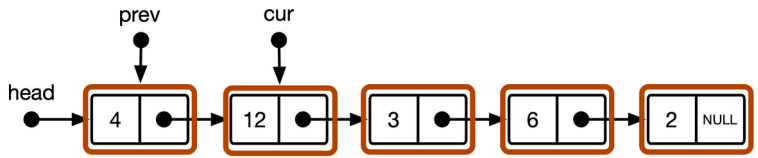

Figure 1: Conventional linked list

```
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

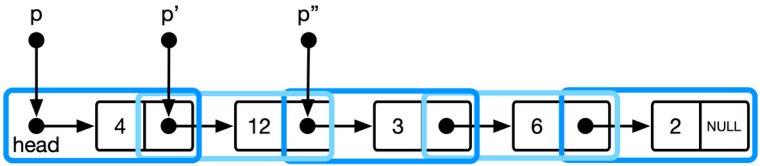

Figure 2: Link list with a single indirect pointer

```
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.