Leibniz Universität Hannover

Programmieren 1

Arrays and Pointers

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations, intervals, and itemizations	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	

Review

- Formatted Output
  - printf, placeholders (%d, %f) must match variable types
- Assertions, Preconditions, Postconditions
  - Assertions check for conditions that must be true at that point
  - Preconditions constrain function arguments, protect function against bugs in caller
  - Postconditions protect function against bugs in function implementation
- Structures and Unions
  - Compound types (structures)
  - Variant types (tagged unions)
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)
- Makefiles (optional topic)
  - Manage the build process

Review: Signed and Unsigned Data Types

```
char c = 0xff;
int i = c;
printiln(i);              Ausgabe?     -1
printbln(i == 0xff);      Ausgabe?     false

unsigned char u = 0xff;
int i = u;
printiln(i);              Ausgabe?     255
printbln(i == 0xff);      Ausgabe?     true
```

- C standard allows char to be signed or unsigned
- Platform- and compiler-specific
- x86 GNU/Linux and Microsoft Windows: signed char
- PowerPC and ARM processors: unsigned char

Think, Compile, Inspect, Locate Errors

- Think carefully
- Compile often
- Check for syntax errors

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."
Brian Kernighan, "Unix for Beginners"

- Use printf to see whether variables have the values that you expect
- Use printf to locate errors, locating errors is an important skill
- Write examples to test your functions

- IDEs and debuggers offer easier ways of
  - inspecting/modifying variables
  - looking at the call stack

Style

- Do not write:
  `if (sorted(a, length) == true) ...`
- Rather write:
  `if (sorted(a, length)) ...`

- Do not write:
  `if (sorted(a, length) == false) ...`
- Rather write:
  `if (!sorted(a, length)) ...`

Preview

- Standard input and standard output
- Arrays
- Linear search and binary search
- Pointers

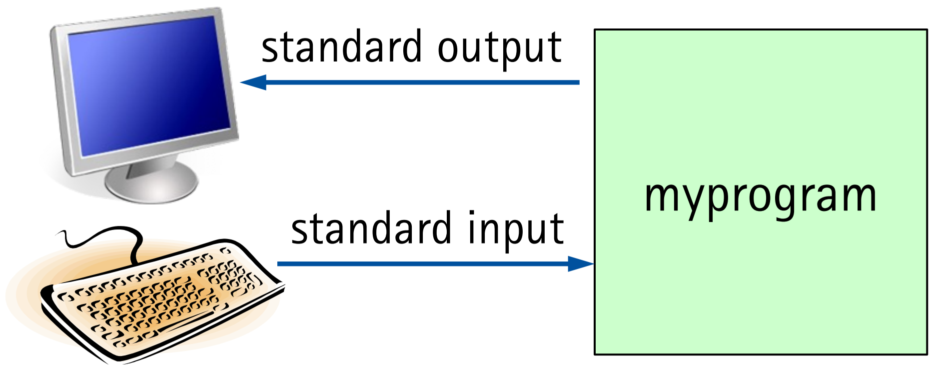
STANDARD INPUT AND OUTPUT, FILE I/O

Standard Input and Output

- Connect a program to its environment
- Input and output as streams of characters

- Standard input
  - From keyboard

- Standard output
  - To screen



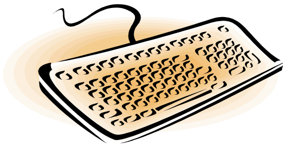
Input and Output of Single Characters

- Predefined character streams
  - stdin: Standard input stream
  - stdout: Standard output stream
  - stderr: Standard error stream

- Character input

```
char c;
c = getchar();
```

read a single character from **standard input** stream, e.g. from the keyboard



- Character output

```
putchar(c);
putchar('\n');
```

write a single character to **standard output** stream, e.g. to the display

Standard Input and Output Redirection

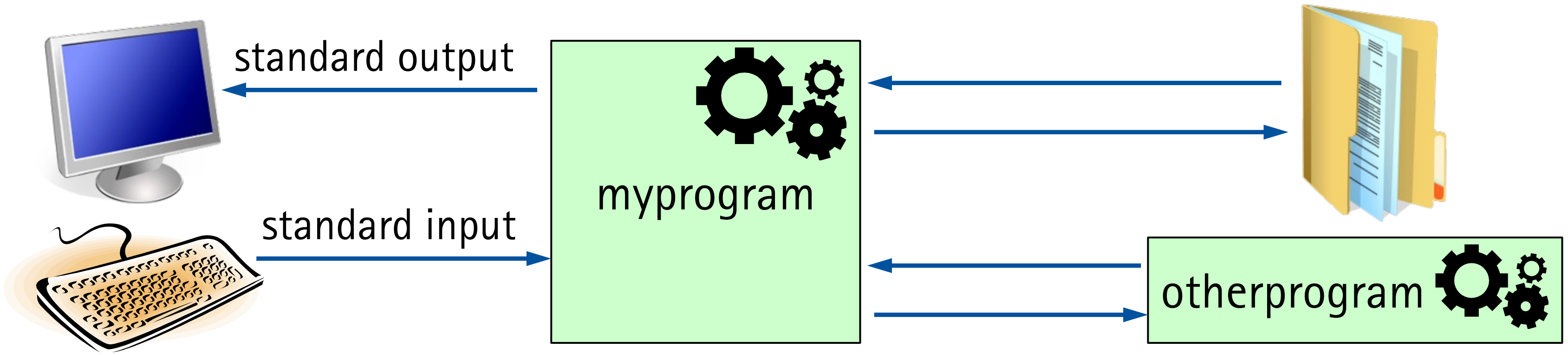
- Standard input
  - from keyboard
  - from file: **myprog.exe < file.txt**
  - from standard output of another program: **otherprog.exe | myprog.exe**
- Standard output
  - to screen
  - to file: **myprog.exe > file.txt**
  - to standard input of another program: **myprog.exe | otherprog.exe**

data comes from file

data is produced by otherprog.exe

data will be stored in file

data will be sent to otherprog.exe



Standard Output and Standard Error

```c
#include "base.h"

int main(void) { // my_stdout_stderr.c
    printf("hello (implicitly on stdout)\n");
    fprintf(stdout, "hello (on stdout)\n");
    fprintf(stderr, "hello (on stderr)\n");
}
```

fprintf: formatted output to a specific stream

```
./my_stdout_stderr 1> out.txt 2> err.txt
```

redirect output stream to out.txt and error stream to err.txt

err.txt:

```
hello (on stderr)
```

out.txt:

```
hello (implicitly on stdout)
hello (on stdout)
```

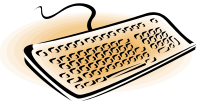
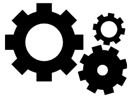
File Copying in C

putchar and getchar headers are in defined in stdio.h:
int getchar(void);
int putchar(int);

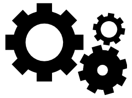
```
#include <stdio.h>
int main(void) {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

int!

end-of-file (EOF) == -1

read a character  or  or 

while character is not end-of-file indicator

output the character  or  or 

read next character

```
./my_copy.exe < file1.txt > copy_of_file_1.txt
```

Line Counting

```
#include <stdio.h>
int main(void) {
    int c;
    int lines = 0;
    while ((c = getchar()) != EOF) {
        if (c == '\n') {
            lines++;
        }
    }
    printf("%d lines\n", lines);
    return 0;
}
```

if condition is true, then increment lines

counting the last line?

foo.txt:

```
hello
world
in C
```

```
$ linecount < foo.txt
3 lines
```

Was ist der Unterschied zwischen:

- `myprogram.exe < file.txt`
- `otherprog.exe | myprogram.exe`

Was ist der Unterschied zwischen:

- `myprogram.exe > a.txt`
- `myprogram.exe 2> a.txt`

Read from and Write to a File (prog1lib)

- Read a file as a string
- Write a string to a file
  - Caution: File will be overwritten!
- Example:

```
#include "base.h"
int main(void) {
    String s = s_read_file("read_write_file.c");
    printf("This file has %d characters.\n", s_length(s)); // length of string
    String t = s_upper_case(s); // convert each character to upper case
    s_write_file("MY_UPPER.TXT", t);
    return 0;
}
```

https://postfix.hci.uni-hannover.de/files/prog1lib/base_8h.html

String	**s_read_file** (String name)
	Read the contents of a file into a String. More...
void	**s_write_file** (String name, String data)
	Write a String to a file. More...

ARRAYS

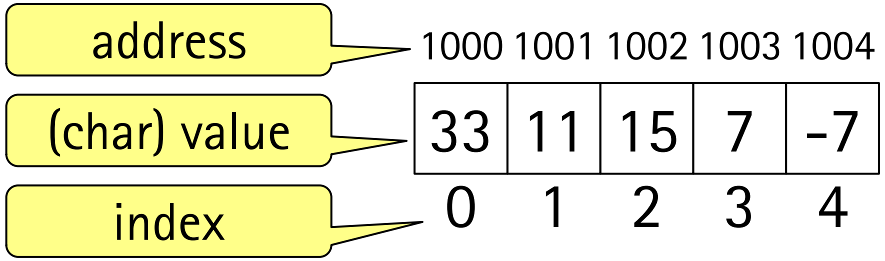
Arrays in C

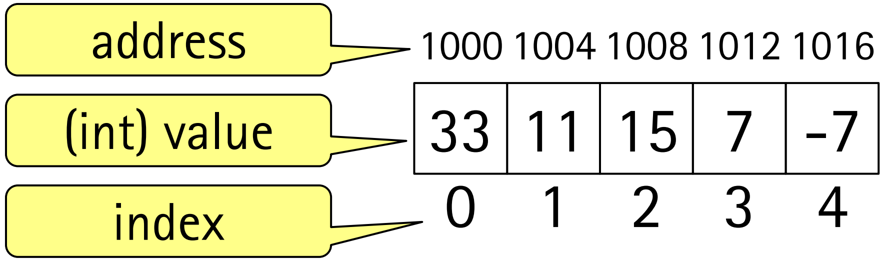
- Arrays: Sequences of elements of the same type
- Declaring an array of five integers
  - int numbers[5];
- Elements accessed by index
  - First element has index 0, second element has index 1, etc.
  - C does not check, whether index is in range!
- Accessing the array element at index i
  - int x = numbers[i];
  - first element at numbers[0]
- Storing a value at index i
  - numbers[i] = 42;

33	11	15	7	-7
0	1	2	3	4

Arrays

- Array elements are stored in subsequent memory cells

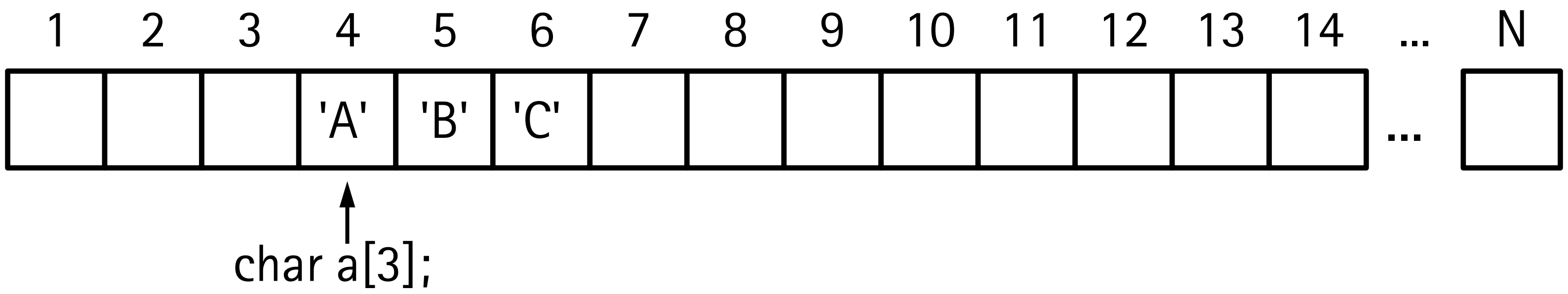




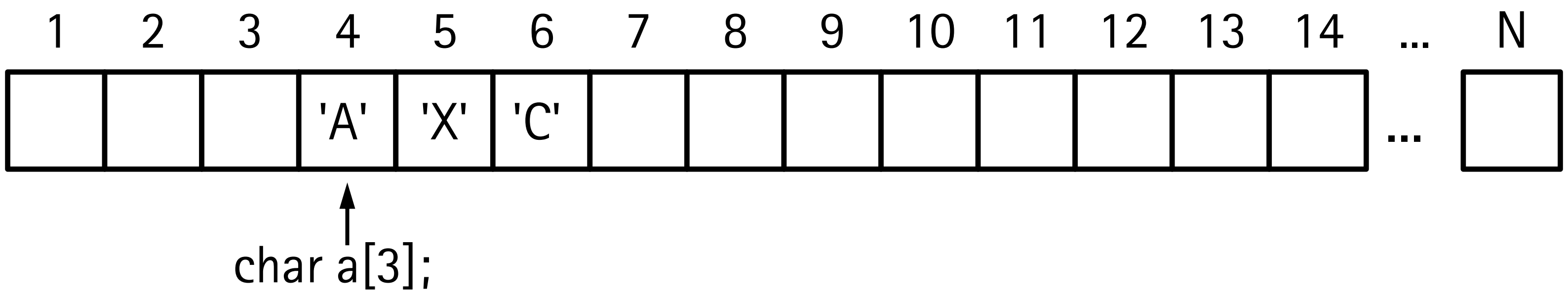
- The array length is fixed (does change)
- C arrays do not store their length

Arrays in C

- Memory is a linear sequence of bytes, numbered from 1 to N

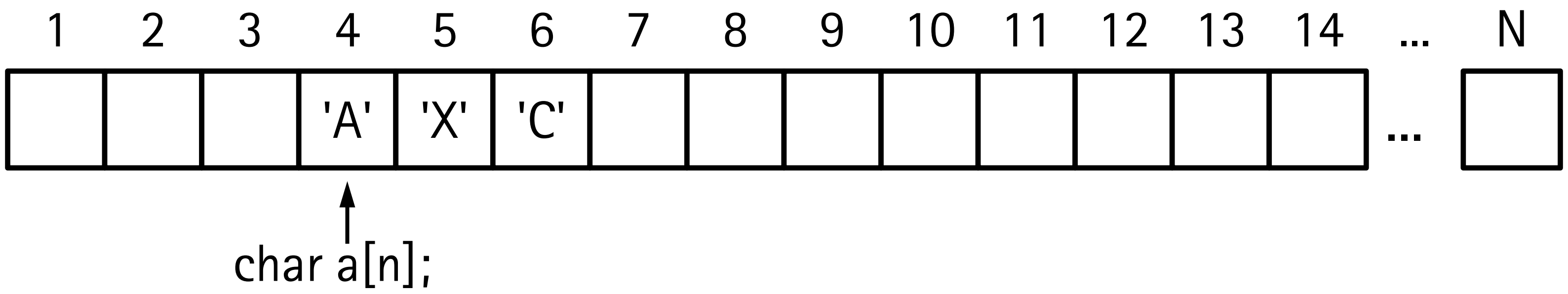


- An array points to some memory location (the start of the array)
- a[1] = 'X';



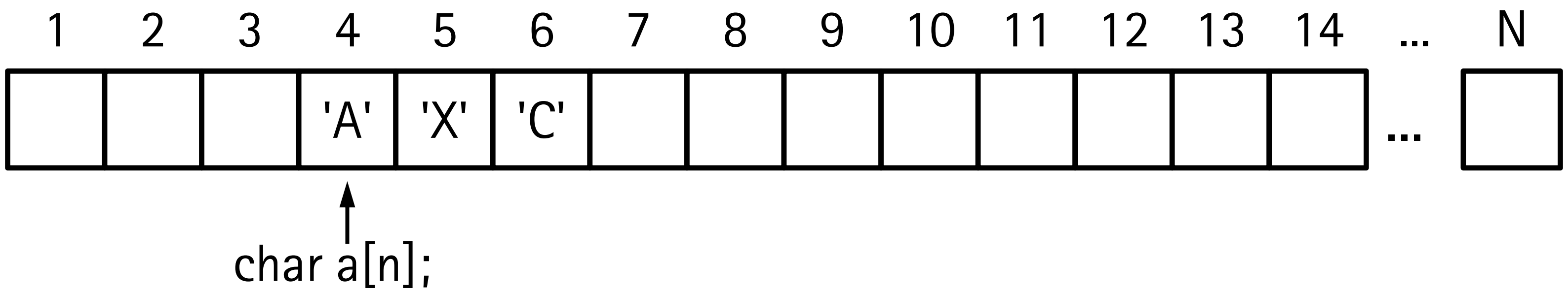
Arrays in C

- How to delete 'X' at index 1 in array a of length n?

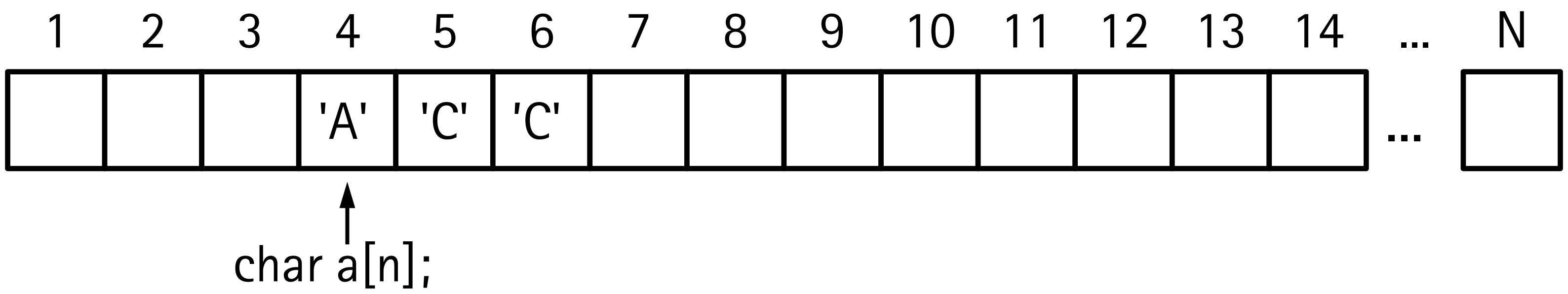


Arrays in C

- How to delete 'X' at index 1 in array a of length n?



- Need to copy elements down to overwrite 'X'
- a[1] = a[2]; ...; a[n-2] = a[n-1];



No Bounds Checking with Arrays

```
int main(void) {
    int a[ ] = { 33, 11, 15, 7, -7 };
    printiln(a[0]);  // output: 33
    printiln(a[4]);  // output: -7
    int i = -1;      // out of bounds!
    printiln(a[i]);  // output: 0 (one left of array)
    i = 5;           // out of bounds!
    printiln(a[i]);  // output: 32767 (one right of array)
    return 0;
}
```

whatever happens to be at that point in memory

Character Arrays, Strings

- String literals are arrays of characters, terminated by '\0' = 0
- String literal “hello!” is stored as a character array containing
  - The characters of the string
  - Terminating '\0'-character

h	e	l	l	o	!	\0
0	1	2	3	4	5	6

- Why terminating '\0' character?
  - Arrays don’t store their length
  - '\0' character needed to mark end of string
  - Thus '\0’ is not part of a string

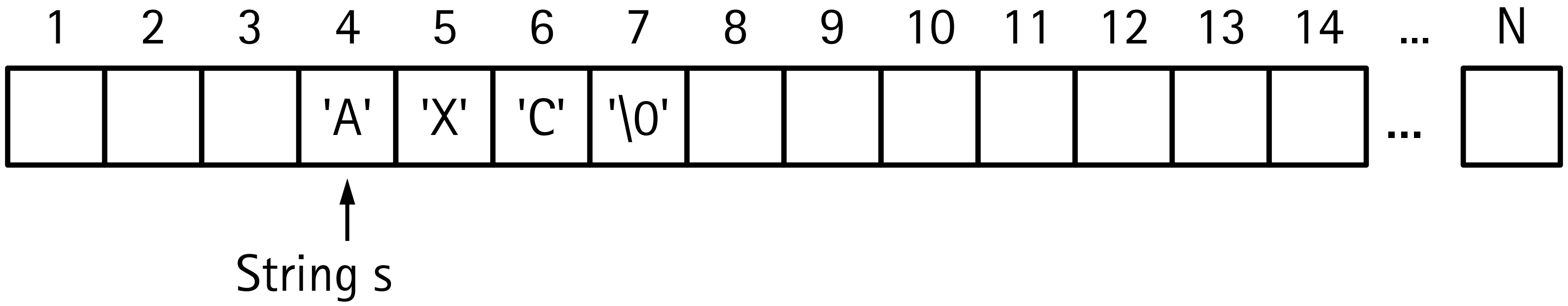
length of "hello!" is 6, not 7

even though 7 bytes are needed to represent it in memory

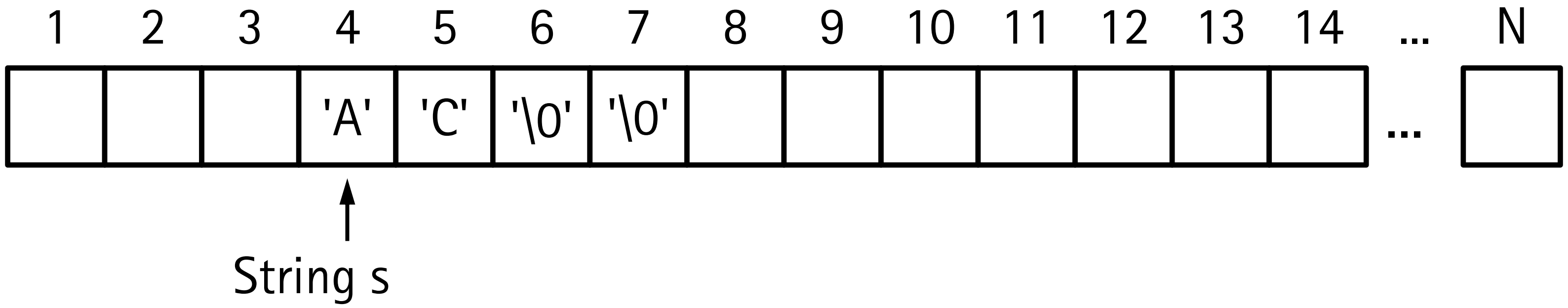
- Example: `char line[101];`
  - Can store a string of 100 characters followed by terminating '\0' character

Arrays in C

- How to delete 'X' in string s?



- Need to copy elements down to overwrite 'X'
- a[1] = a[2]; …; a[n-1] = a[n];



Arrays Example: Counting Digits ‘0’..’9’

```
#include <stdio.h>

int main(void) {
    int ndigit[10];
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
        .
        .
        .
        .
        .
        .
    return 0;
}
```

array of 10 ints: a counter for each digit ‘0’..’9’

reset each counter to 0

read data and count digits

output digit counts (number of times each digit occurred)

Arrays Example: Counting Digits '0'..'9'

```
#include <stdio.h>

int main(void) {
    int ndigit[10];
    int i = 0;
    while (i < 10) {
        ndigit[i] = 0;
        i++;
    }
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    .
    return 0;
}
```

array of 10 ints, not initialized!

set each counter to 0

read data and count digits

output digit counts (number of times each digit occurred)

Arrays Example: Counting Digits '0'..'9'

```
#include <stdio.h>

int main(void) {
    int ndigit[10];
    int i = 0;
    while (i < 10) {
        ndigit[i] = 0;
        i++;
    }
    while ((i = getchar()) != EOF) {
        if (i >= '0' && i <= '9') {
            ndigit[i - '0']++;
        }
    }
    .
    .
    .
    .
    .
    .
    return 0;
}
```

array of 10 ints, not initialized!

set each counter to 0

int getchar(void);

end-of-file (EOF) == -1

character '0' is represented by its ASCII value (48), thus we can compare it to an int...

... and we can compute with characters

output digit counts (number of times each digit occurred)

Arrays Example: Counting Digits '0'..'9'

```
#include <stdio.h>

int main(void) {
    int ndigit[10];
    int i = 0;
    while (i < 10) {
        ndigit[i] = 0;
        i++;
    }
    while ((i = getchar()) != EOF) {
        if (i >= '0' && i <= '9') {
            ndigit[i – '0']++;
        }
    }
    i = 0;
    while (i < 10) {
        printf("%d: %4d times\n", i, ndigit[i]);
        i++;
    }
    return 0;
}
```

array of 10 ints, not initialized!

set each counter to 0

int getchar(void);

end-of-file (EOF) == -1

character '0' is represented by its ASCII value (48), thus we can compare it to an int...

... and we can compute with characters

output digit frequencies

Arrays Example: Counting Digits from File

```
$ gcc countdigits.c -o countdigits
$ ./countdigits < foo.txt
0:    0 times
1:    2 times
2:    2 times
3:    1 times
4:    0 times
5:    0 times
6:    0 times
7:    0 times
8:    0 times
9:    5 times
```

foo.txt:

```
1h1ello
w2orld
in2 C
399999
```

Limitations: Functions Cannot Return C-Arrays

- Cannot return c-arrays from functions

```
int[] f(void) { // does not work!
    int a[] = { 33, 11, 15, 7, -7 }; // array is stored on the stack
    return a; // error: array gets removed from stack on return
}
```

```
...: fatal error: function cannot return array type 'int []'
...: fatal error: address of stack memory associated
                  with local variable 'a' returned
```

Limitations: Functions Cannot Return C-Arrays

- Solution: Use arrays as input/output parameters

```
void f(/*inout*/ int a[], /*in*/ int n) {
    for (int i = 0; i < n; i++) {
        a[i] *= -1; // multiply each element by -1
    }
}

int main(void) {
    int a[] = { 33, 11, 15, 7, -7 };
    f(a, 5); // f will modify a
    printialn(a, 5); // print-int-array+linebreak: [-33 -11 -15 -7 7]
    return 0;
}
```

need to explicitly pass array length

in: an input parameter

inout: an input and output parameter, modification visible outside of function

need to explicitly pass array length

ITERATING OVER ARRAYS

Iterating Over Arrays: Valid Indices

```
// Returns true if the array contains negative values,
// otherwise returns false.
bool contains_negative(int a[], int n) {
    // Iterate over the elements of the array
    // and return true if a negative element is found.
    for (int i = 0; i < n; i++) {
        assert("less than length", i <= n - 1); // follows from run condition
        assert("not negative", i >= 0); // follows from initialization and update expression
        if (a[i] < 0) return true;
    }
    return false;
}
```

for-loop generates exactly the indices i that are valid for array a, $i \in 0..n-1$

edge/boundary cases often important for correctness checking

- Only generate valid indices i within for-loop
- Think about values of i in the first and the last iteration
- See Chapter 9 of https://postfix.hci.uni-hannover.de/files/prog1script/script.html

Computing the Average Temperature

```
#include <stdio.h>
int main(void) {
    int a[] = { 7, 6, 5, 6, 6, 5, 6, 6, 6, 7, 7, 9, 10,
                11, 11, 12, 13, 12, 10, 9, 9, 8, 7, 4 };
    int n = sizeof(a) / sizeof(a[0]);
    int sum = 0;
    for (int i = 0; i < n; i++) {
        // first iteration: i == 0,   last iteration: i == n - 1
        sum += a[i];
    }
    double mean = (double) sum / (double) n;
    printf("mean temperature: %.1f °C\n", mean);
    return 0;
}
```

number of elements of a

make sure indices are valid:
C won't warn you!

```
mean temperature: 8.0 °C
```

Average Temperature Function

```
double mean(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    double m = (double) sum / (double) n;
    return m;
}
```

enforce double division

- What if n == 0?

```
mean temperature: nan°
```

  - nan = not a number (division by zero)

Average Temperature Function (returning int)

```
int mean_i(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum / n;
}
```

integer division

- What if n == 0?

  `Division-by-zero error`

  - integer division by zero is undefined
- Solutions?
  - Preconditions to limit the allowed argument values
  - Option types (special case of variant types) to handle the case of no result

Average Temperature Function with Preconditions

```
/*
Computes the mean of the values in the
array. The array must exist and must
not be empty.
*/
int mean_i(int a[], int n) {
    require("not empty", n > 0);
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum / n;
}
```

- Functions should work correctly for each possible argument value
  - Here for each possible array
- If it cannot do its job with the given value, it must clearly say so
  - Must say so in the documentation
  - Must generate a clear error message
- C data types cannot express these constraints, so use preconditions

Option Types Example: IntOption

- Option types are a special case of variant types
- Either contains a value or contains nothing
- Implementation of int-option (see basedefs.h)

```
struct IntOption {
        bool none;
        int some;
};

typedef struct IntOption IntOption;
```

binary tag to indicate whether "some" is valid

```
IntOption make_int_none(void) {
    IntOption op = { true, 0 };
    return op;
}

IntOption make_int_some(int some) {
    IntOption op = { false, some };
    return op;
}
```

Average Temperature Function (returning IntOption)

There may or may not be a result. If there is a result the encapsulated value has type int.

```
IntOption mean_io(int a[], int n) {
int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    if (n <= 0) {
        return make_int_none();
    } else {
        return make_int_some(sum / n);
    }
}
```

wrap value into option

Using Option Types as a Caller

- Checking whether the function
  - either returned nothing (none)
  - or returned some value
- Option type clearly tells a user of the function that there may not always be a result

```
IntOption io = mean_io(a, n);
if (io.none) {
    printf("no result\n");
} else {
    printf("%d\n", io.some);
}
```

access encapsulated value

```
struct IntOption {
    bool none;
    int some;
};
```

LINEAR AND BINARY SEARCH

Linear Search

```
int linear_search(int a[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            return i; // found match
        }
    }
    return -1; // no match
}
```

-1 is not a valid index. Returns -1 to indicate “no match”.

0	1	2	3	4	5	6	7	8
3	5	6	7	11	12	14	16	17

found x = 7 ⤴
at index 3

Linear search takes n/2 steps on average

Binary Search

- Search for index of value x in array a
  - Input array a (n elements)
  - Precondition: elements are in increasing order: $v_0 \leq \ldots \leq v_i \leq \ldots \leq v_{n-1}$
- Algorithm (pseudo code)

```
low = 0, high = n-1     (search range 0..n-1)
while (low ≤ high) {
        test value m of middle element (at index i_m = (low + high) / 2)
        if x == m then element found, return its index i_m
        else if x < m then search left half of range (low..i_m-1)
        else if x > m then search right half of range (i_m+1..high)
}
return "not found"
```

Binary Search

```
int binary_search(int a[], int n, int x) {
    int low = 0;
    int high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int v = a[mid];
        if (x == v) { // found match
            return mid;
        } else if (x < v) {
            high = mid - 1; // search low..mid-1
        } else if (x > v) {
            low = mid + 1; // search mid+1..high
        }
    }
    return -1; // no match
}
```

is mid a valid index?

x = 7

v =

0	1	2	3	4	5	6	7	8
3	5	6	7	11	12	14	16	17

L				M				H
L	M		H					
		LM	H					
			LMH					

found ⤴

Binary search takes considerably fewer steps than linear search: $\log_2(n)$ steps on average

Binary Search with Assertions

Leibniz Universität Hannover

```
int binary_search2(int a[], int n, int x) {
    require("array length not negative", n >= 0);
    int low = 0, high = n - 1;
    int result = -1; // no match
    while (low <= high) {
        int mid = (low + high) / 2;
        assert("valid index", 0 <= mid && mid < n);
        int v = a[mid];
        if (x == v) {
            result = mid; break; // found match
        } else if (x < v) {
            high = mid - 1; // search low..mid-1
        } else if (x > v) {
            low = mid + 1; // search mid+1..high
        }
    }
    ensure("loop termination", (low > high && result == -1) ||
                               (0 <= result && result < n && x == a[result]));
    ensure("same result", result == linear_search(a, n, x));
    return result;
}
```

alternative:

```
if (n <= 0) return -1;
```

Binary Search with Assertions

```
while (low <= high) {
    assert_code(int old_low = low);
    assert_code(int old_high = high);
    int mid = (low + high) / 2;
    assert("valid index", 0 <= mid && mid < n);
    int v = a[mid];
    if (x == v) {
        result = mid; break; // found match
    } else if (x < v) {
        high = mid - 1; // search low..mid-1
    } else if (x > v) {
        low = mid + 1; // search mid+1..high
    }
    assert("smaller range", low > old_low || high < old_high);
}
```

POINTERS

Pointers

- A pointer is a variable that contains the address of a variable
- Memory consists of consecutively numbered cells
  - Each cell has an address and contains a value

address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	1003			5				

points to

- Pointers allow accessing arbitrary memory locations
  - A lot of flexibility to access/modify data, organize memory layout
  - A lot of flexibility to shoot oneself in the foot

Pointers

- Pointer variables point to items of a specific type
  - int *p; // declares a pointer variable p that points to a variable of type int
  - char *q; // declares a pointer variable q that points to a variable of type char
  - Except:
    void *v; // declares a pointer v that points to a variable of any arbitrary type
- Pointer variables may have value NULL ("NULL pointer")
  - If a pointer variable stores value NULL (0), then this indicates that the variable does not point to any memory cell
  - Pointers are thus implicit option types:
    Either point to an item or point to nothing

Pointing to Multi-Byte Items: Endianness

- Each memory cell is a byte
- Larger objects occupy multiple cells
  - Example: an int (typically) uses 4 cells
- How is int 5 distributed across 4 memory cells?
- Depends on the processor architecture
  - "little-endian" stores least significant byte first

address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	5	0	0	0				

  - "big-endian" stores most significant byte first

address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	0	0	0	5				

Pointing to Multi-Byte Items

- For both little- and big-endian, value of pointer is lowest memory address of multi-byte object
  - "little-endian" stores least significant byte first

address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	5	0	0	0				

int 5 stored in addresses 1000-1003

⇧ pointer to int 5 contains address 1000

  - "big-endian" stores most significant byte first

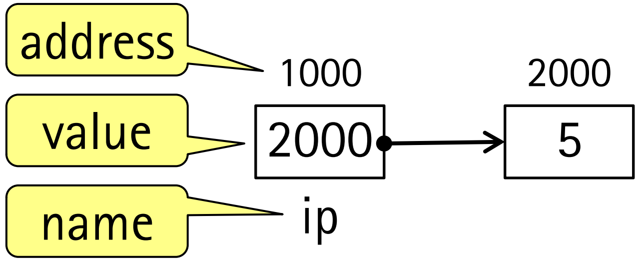
address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	0	0	0	5				

int 5 stored in addresses 1000-1003

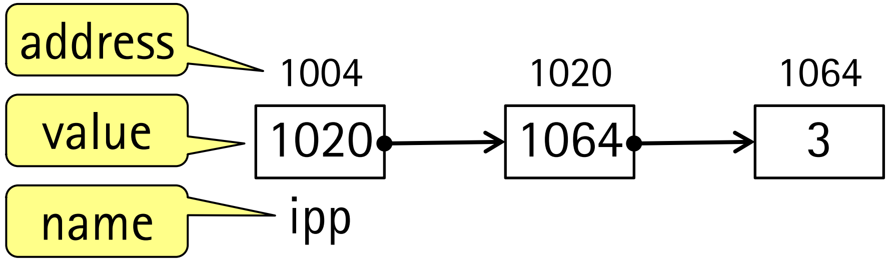
⇧ pointer to int 5 contains address 1000

Pointer Declaration, Address Operator &, Dereferencing Operator *

- Address operator **&**
  - The address of a variable (Where is the variable stored in memory?)
- Dereferencing operator *
  - The value of the variable pointed to (What value does the pointer refer to?)
- Pointer declaration
  - int * ip; says that "ip is a pointer to an int" and "*ip is an int"
  - int ** ipp; says that "ipp is a pointer to a pointer to an int"



ip == 2000. &ip == 1000. *ip == 5



ipp == 1020. &ipp == 1004.
*ipp == 1064. **ipp == 3

Pointer Declaration, Address Operator &, Dereferencing Operator *

- int x = 1;
- int y = 2;
- int *ip = NULL;

1000..1003	1004..1007	1008..1015
1	2	0
x	y	ip

- ip = &x;

1000..1003	1004..1007	1008..1015
1	2	1000
x	y	ip

- y = *ip;

1000..1003	1004..1007	1008..1015
1	1	1000
x	y	ip

- *ip = 0;

1000..1003	1004..1007	1008..1015
0	1	1000
x	y	ip

Dereferencing Operator in Assignments

- Address and dereferencing operators have high precedence
  - y = *ip +1 — takes the value of the variable ip points to, adds 1, and assigns to y
  - *ip = y + 1 — adds 1 to y and assigns the result to the variable ip points to
  - *ip = *ip + 10 — increments the variable ip points to by 10

- Pointers are variables and can be assigned to other pointers
  - int *ip, *iq; — ip an iq are pointers to int
  - ip = &x; — now ip points to x (ip contains the address of x)
  - iq = ip; — now ip and iq contain the same address

Swap

```c
#include <stdio.h>

void swap(int x, int y) {
    int h = x;
    x = y;
    y = h;
}

int main(void) {
    int x = 1, y = 2;
    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

```c
#include <stdio.h>

void swap2(int *px, int *py) {
    int h = *px;
    *px = *py;
    *py = h;
}

int main(void) {
    int x = 1, y = 2;
    swap2(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Addresses of local variables x and y in main!

Can modify variables in another scope!

char ***b;   ???

- char ***b;

  if (***b == 1) ... // what happens here?

- Evaluation diagram: (assume b is stored at address 1000)

```
  ***b == 1
*(*(*  b   )) == 1
*(*(*1003)) == 1
*(*  1001 ) == 1
*    1007    == 1
   1         == 1
        1
```

b

address:	1000	1001	1002	1003	1004	1005	1006	1007
value:	1003	1007		1001				1

Pointers as Function Parameters

- Pointers allow accessing and modifying any variable, even if defined in another scope
- Function arguments are passed by copying the value
  - Pointers: Memory address is copied, not the object pointed to
  - Arrays: Memory address of first element is copied, not the complete array
  - Structs: The struct itself is copied (sizeof struct bytes)
  - Primitive types (int, double, char, etc.):
    The bytes that make up the value are copied

Type Casts with Pointers

- Converting a pointer-to-type-A to a pointer-to-type-B
  - Allows (re-)interpreting what is pointed to in arbitrary ways
- Example

```
char a[] = { 1, 2, 3, 4, 5, 6, 7, 8 }; // memory as a sequence of bytes
char *c = a;
printf("%02x\n", *c);      // 01
short *s = (short*)a;      // type cast from Byte* to short*
printf("%04x\n", *s);      // 0201
int *i = (int*)a;          // type cast from Byte* to int*
printf("%08x\n", *i);      // 04030201
long *l = (long*)a;        // type cast from Byte* to long*
printf("%016lx\n", *l);    // 0807060504030201
```

Summary

- Standard input and standard output
  - Reading from (writing to) the console or from (to) a file
- Arrays
  - Sequences of elements of the same type, access by index, no bounds checking
- Linear search and binary search
  - Efficient lookup of ordered elements in an array
- Pointers
  - A pointer is a variable that contains a memory address (or NULL)