System-Level Programming

21 Supplements: Pointers

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24
Pointers, Arrays, and Strings

Strings are arrays of single characters (char) that are internally terminated by the ‘\0’-character

Example: Determining the length of a string – call strlen(x);

/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n = 0; *s != '\0'; n++) {
        s++;
    }
    return n;
}

/* 2. Version */
int strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return p - s;
}
If a string is used for the initialization of a `char`-array, the identifier of the array is a constant pointer to the start of the string.

```c
char amessage[] = "now is the time";
```

- a memory space of size 16 bytes is allocated and the characters are copied to this area
- `amessage` is a *constant pointer* to the start of the memory area, this pointer cannot be modified
- however, the *contents* of the memory area can be modified

```c
amessage[0] = 'h';
```
If a string is used for the initialization of a `char` pointer, the pointer is a variable that is initialized with the starting address of the string.

```c
const char *pmessage = "hello world"; /*(1)*/
```

The string itself is placed in memory as a constant value (string literal) by the compiler.

```c
pmessage++; /*(2)*/
printf("%s\n", pmessage); /* prints "ello world" */
```

- the string itself is placed in memory as a constant value (string literal) by the compiler
- the memory space for a pointer is reserved (e.g., 4 byte) and then initialized with the address of the string.
Pointer, Arrays and Strings (4)

```c
const char *pmessage = "hello world"; /*(1)*/

pmessage++; /*(2)*/
printf("%s\n", pmessage); /* prints "ello world" */
```

- `pmessage` is a variable pointer that is initialized with a certain address, but can be modified (`pmessage++;`)
- It is not allowed to modify the memory area of "hello world"
  - The compiler detects this use of the keyword `const` and prevents write access via the pointer
  - Some compilers place such strings in the write-protected area of the memory (⇒ memory-protection violation when the content is accessed and the pointer has not been declared as a `const` pointer)
Assigning a `char` pointer or string to another `char` pointer does not copy the string!

```c
pmessage = amessage;
```

The pointer `pmessage` only gets assigned the address of the string "now is the time".

When passing a string as an actual parameter to a function, the function only receives a copy of the pointer to the string.
To assign a whole string to another `char` array, the string has to be copied: Function `strcpy` from the standard C library

Examples for implementation:

```c
/* 1. Version */
void strcpy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') {
        i++;
    }
}

/* 2. Version */
void strcpy(char *s, char *t) {
    while ((s = *t) != '\0') {
        s++, t++;
    }
}

/* 3. Version */
void strcpy(char *s, char *t) {
    while (*s++ = *t++) {
    }
}
```
Arrays of pointers can also be created.

Declaration

```c
int *pfield[5];
int i = 1;
int j;
```
Arrays of pointers can be created also

**Declaration**

```c
int *pfield[5];
int i = 1;
int j;
```

**Access to a pointer of the array**

```c
pfield[3] = &i;
```
Arrays of pointers can be created also

**Declaration**
```c
int *pfield[5];
int i = 1;
int j;
```

**Access to a pointer of the array**
```c
pfield[3] = &i;
```

**Access to the object that the pointer of the array points to**
```c
j = *pfield[3];
```
Example: Definition and initialization of a pointer array:

```c
const char *
month_name(int n)
{
    static const char *name_of_month[] = {
        "invalid month",
        "January",
        ...
        "December"
    };

    return (n < 1 || 12 < n) ? name_of_month[0] : name_of_month[n];
}
```
Arguments from the Command Line

- Usually, when a program is called, arguments are passed to the program.

- The access to these arguments is provided in the function `main()` by two parameters (both variants are equivalent):

  ```
  int main(int argc, char *argv[]) {
  ...
  }

  int main(int argc, char **argv) {
  ...
  }
  ```

- The parameter `argc` contains the number of arguments that were given when calling the program.

- The parameter `argv` is a field of pointers to the respective arguments (strings).

- The name of the program is always passed as the first argument (`argv[0]`)
Arguments from the Command Line

Command:
```sh
gcc -Wall -o test test.c
```

C-file:
```c
... int main(int argc, char *argv[]) ...
```
```c
... int main(int argc, char **argv) ...
```
Arguments – Example

Example: echo program

```
~> echo Hello SLP!
Hello SLP!
~>
```

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}
```
Composite Data Types / Structures

- Combination of multiple values to one unit
- Declaration of structures
  ```
  struct person {
    char name[20];
    int age;
  };
  ```
- Definition of a variable of type `struct`
  ```
  struct person p1;
  ```
- Access to an element of the structure
  ```
  strcpy(p1.name, "Peter Pan");
  p1.age = 12;
  ```
Pointers to Structures

- Concept analogous to “pointer to variable”
  - Address of a structure can be determined with the & operator

Example

```c
struct person stud1;
struct person *pstud;
pstud = &stud1;
```

- Especially useful when building linked structures (lists, trees, ...)
  - a structure can contain addresses to further structures of the same (and other) types
Access to components of the structure via the pointer

Known approach

- "*"-operator yields structure itself
- "."-operator yields an element of the structure
- **However:** Keep in mind the order of the operators!

```
(*pstud).age = 21;
```

Syntactically nicer:

- "->"-operator

```
pstud->age = 21;
```
Nested/Linked Structures

Structures inside of structures are allowed – however:

- the size of the structure has to be determinable by the compiler
  ⇒ structure cannot contain itself
- the size of a pointer is always known
  ⇒ structure can contain a pointer to the same structure

Examples:

Linked list:
```c
struct list {
    struct list *next;
    struct person stud;
};
struct list *head;
```

Tree:
```c
struct tree {
    struct tree *left;
    struct tree *right;
    struct person stud;
};
struct tree *root;
```
Multiple structures of the same type can be linked via pointers

```c
struct list { struct list *next; int val; }

struct list el1, el2, el3;
struct list *head;

head = &el1;
el1.next = &el2; el2.next = &el3; el3.next = NULL;
el1.val = 10; el2.val = 20; el3.val = 30;
```

Iterating over a linked list

```c
int sum = 0;
for (struct list *curr = head; curr != NULL; curr = curr->next) {
    sum += curr->val;
}
```