Generic functions, generic data structures

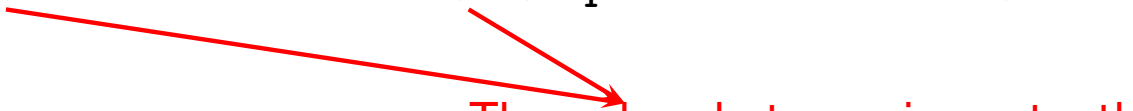
Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2019/index.html

Function pointers

- Declaring function pointers

```
<return type> (* <function name>) ( <parameter list> )
```



Example:

```
int *aFunction(int), *(*aFunctionPointer)(int);
```

- Using function pointers

```
(*f)(...)
```

- Setting function pointer variables / passing function pointers as arguments: simply use the name of the function

Example:

```
aFunctionPointer = aFunction;
```

Generic sort/search routines

```
#include <stdlib.h>
```

Sorting

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Searching

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Comparator routine: examples

```
int compare_int (void *elem1, void *elem2)
{
    int *ip1 = elem1;
    int *ip2 = elem2;
    return *ip1 - *ip2;
    /* Or more explicitly:
       int i1 = *((int *) elem1);
       int i2 = *((int *) elem2);
       return i1 - i2;
     */
}

int compare_strings (void *elem1, void *elem2)
{
    char **s1 = elem1; // Alt.: char *s1 = *((char **) elem1);
    char **s2 = elem2; // Alt.: char *s2 = *((char **) elem2);
    return strcmp (*s1, *s2);   // Alt.: return strcmp(s1, s2);
}
```

Generics: useful functions

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

- `memcmp()`: compares the first `n` bytes (each interpreted as unsigned char) of the memory areas `s1` and `s2`
- `memcpy()`: copies `n` bytes from `src` to `dest` (memory areas must not overlap)
- `memmove()`: copies `n` bytes from `src` to `dest` (memory areas may overlap)

Generic stacks

```
1  #ifndef _GSTACK_
2  #define _GSTACK_
3
4  typedef struct {
5      void *elements;
6      size_t element_size, num_elements, max_elements;
7  } STACK;
8
9  STACK newStack(int element_size);
10 // OR
11 void initStack (STACK *s, int element_size);
12 void freeStack(STACK *s);
13 bool isEmpty(const STACK *s);
14 void push(STACK *s, const void *eptr);
15 void pop(STACK *s, void *eptr);
16
17 #endif // _GSTACK_
```

Implementation notes

- Choose a default stack size initially (`max_elements`); `realloc()` to double the current size as needed
- Use `memcpy()` for `push()` and `pop()`

  Example:

```
stackElementAddress = (char *) s->elements + s->num_elements *
    s->element_size;
memcpy(stackElementAddress, argument, s->element_size); // for
    push()
memcpy(argument, stackElementAddress, s->element_size); // for
    pop()
```

Review question

Compile and run `function-pointers.c`. Explain the output.

Answer to review question

Sub-question I: Are the following assignments permissible?

```
array1[0] = "Hello";
array1[0][4] = '!';
```

Answer to review question

Sub-question I: Are the following assignments permissible?

```
array1[0] = "Hello";
array1[0][4] = '!';
```

Sub-question II:

Where is `array1` stored?

Where are the strings `"Hello"`, etc. stored?

Answer to review question

Sub-question I: Are the following assignments permissible?

```
array1[0] = "Hello";
array1[0][4] = '!';
```

Sub-question II:
Where is `array1` stored?
Where are the strings `"Hello"`, etc. stored?

Answer to sub-question II:
`array1` $\longrightarrow$ stack
`"Hello"` $\longrightarrow$ ***read-only data***

If you are adventurous, run
`gcc -g -O0 -c -fverbose-asm -Wa,-adhln function-pointers.c`
and study the output carefully (also see what the flags mean).

Answer to review question

```
array1: 0x7fff11769a20 (140733486373408)
key1: 0x7fff11769a18 (140733486373400)
status: 0x7fff11769a10 (140733486373392)

0x4008db
0x4008de
0x4008e1
0x4008e3
0x4008e6

0x4008de
```

Problems - I

1. Complete the implementation of the functions in the header file. Test it using different types (e.g., `int`, `float` and strings).
2. Implement a generic data structure SEQUENCE to hold a list of elements of any one of the following types: `int`, `float`, or null-terminated strings (`char *`) (all elements in a particular list will be of the same type). Your data structure should support the operations given below.
   - `void get_element(SEQUENCE s, size_t i)`: prints the numerical value of the i-th element of the sequence `s`, if it exists. The function should print an error message if the i-th element does not exist. The numerical value of an integer n is n itself; the numerical value of a floating point number f is the integer nearest to f; the numerical value of a string s of alphanumeric characters is the sum of the ASCII values of all the characters contained in s. Note that the numerical value of an empty string is 0.

Problems - II

- `size_t length(SEQUENCE s)`: returns the number of elements in the sequence `s`.
- `void summation(SEQUENCE s)`: prints the sum of the numerical values of the elements in the sequence `s`. Please see above for the definition of numerical value for sequences of various types. For a sequence containing no elements, `summation(s)` should print 0.

3. Write a program that takes N sequences as input (from stdin), and prints the sequence `s` for which `summation(s)` is the maximum.

   Input format: A positive integer N, followed by N sequences, one per line. Each of these lines will begin with `i`, `f` or `s` to specify whether the sequence on that particular line consists of `int`, `float` or string elements. This single letter will be followed by a non-negative integer that specifies the number of elements in the sequence, which in turn will be followed by the elements of the sequence.

Problems - III

4. Write a program that can accept a generic but homogeneous list of elements from the user and report the next greater element for every element in the list in linear time. Consider a lexicographic comparison of the elements (i.e., use `memcmp()`).