C revision (contd.)

Computing Lab

Indian Statistical Institute

Outline

Basic output to the terminal

- `putchar(int c)` : print `c`, cast to an unsigned char, to stdout (terminal)
- `puts(const char *s)` : print string `s` + newline to stdout
- `printf(const char *format, ...)` : contains zero or more *conversion specifiers*

optional

`printf`: conversion specifiers

flags	width	.precision	length modifier	type

- flags
  - `0` : right align, with zero padding on the left
  - `-` : left align
- width: *minimum* field width
- precision: usually, number of significant digits (for floating point numbers) /
  maximum number of characters to be printed (for string)
- length modifier: `l`, `ll`, `L`, `z`
- type: `d`, `i`, `c`, `f`, `s`

section no. in manual

See man pages for detailed documentation, e.g., `$ man 3 printf`

Reading from the “terminal”

Character at a time:

- Functions: `getchar()` OR `fgetc(stdin)` (equivalent)
- Return value: reads the next character and returns it as an `unsigned char` cast to an `int`, or `EOF` on end of file or error.
- Typical usage: `while (EOF != (c = fgetc(fp))) ...`
- Caution: Do not forget to declare `c` as `int` type.

  If `c` is of type `char`:
  - reading from the terminal: will probably work without any problem
  - reading from a file: in some (rare) cases, problems could occur

Reading from the “terminal”

Line at a time:

- Function: `fgets(s, n, stdin)`
- Return value
  - reads at most `n-1` characters or one line (whichever is shorter), stores input in character array `s` and terminates `s` using `'\0'`
  - if a newline is read, it is stored in `s`
  - returns `s` or `NULL` on end of file (i.e., there is nothing to be read) or error
- Typical usage: `while (NULL != fgets(s, n, stdin)) ...`
- Caution: **Without exception, do not use `gets()`!**

Reading from the terminal: `scanf`

Format string:

`" "`, `"\t"`, `"\n"` or any sequence of these characters	matches any amount of white space, including none
`"%d"`, `"%ld"`, `"%lld"`	read an `int`, `long int` or `long long int`, possibly with leading `+` or `-` sign
`"%u"`, `"%lu"`, `"%llu"`	read an `unsigned int`, `unsigned long int` or `unsigned long long int`
`"%f"`, `"%lf"`, `"%llf"`	read a `float`, `double` or `long double`, possibly with leading `+` or `-` sign
`"%c"`	read a single character (*including white-space*)

Reading from the terminal: `scanf`

`"%s"`: read a sequence of *non-white-space* characters

- Examples:
  - SAFE: `char a[10]; ... scanf("%9s", a);`

    field width: read at most so many characters
  - UNSAFE:
    `char *a; a = (char *) malloc(...); ... scanf("%s", a);`
  - SAFE: `char *a; scanf("%ms", &a); ... free(a);`

    NOTE: need to pass `&a` instead of `a`
- Return value
  - on success: number of input items successfully matched and assigned (*may be less than requested*)
  - on end of input or error: `EOF`

Reading from the terminal: `ungetc`

- Function: `ungetc(c, stdin);`
- Return value
  - "unreads" `c`, i.e., pushes it back to the input so that it can be read later
  - only one pushback is guaranteed
  - returns `c` or `EOF`

Calls to above functions can safely be mixed with each other: each function moves position for next read depending on what it has read.

File handling

Opening a file:

```
fopen(filename, mode)
```

`filename`: string (`char *`) containing name of file

`mode`: string specifying whether file is to be opened in read/write mode

- `"r"`, `"w"`, `"a"`: read mode, write mode, append mode
- `"r+"`, `"w+"`, `"a+"`: read/write mode, write/read mode, read/append mode (see man page for more details)

Example:

```
FILE *fp;
if (NULL == (fp = fopen("a.txt", "r")))
   ERR_MESG("Error opening file");
```

Closing a file:

```
fclose(fp);
```

File handling

Reading / writing text:

`fgetc(fp)`: similar to `getchar()`

Typical usage: `while (EOF != (c = fgetc(fp))) ...`

`fgets(s, n, fp)`: see discussion for terminal input

Typical usage: `while (NULL != fgets(s, n, fp)) ...`

`fscanf(fp, "...", ...)`: as for `scanf`

`fputc(c, fp)`: writes `c` to `fp`

`fputs(s, fp)`: writes string `s` to `fp`

`fprintf(fp, "...", ...)`: as for `printf`

File handling

Reading / writing data:

`fread((void *) buffer, sz, n, fp)`: reads `n` elements of data, each of size `sz` bytes from `fp`, stores them in `buffer`; returns number of elements read.

`fwrite((void *) buffer, sz, n, fp)`: writes `n` elements of data from `buffer`, each of size `sz` bytes to `fp`; returns number of elements written.

Review questions I

1. Write a program that reads the given file (`getc-input.txt`) one character at a time using `fgetc`. After each character is read, print it along with its ASCII value to the screen.

   Try storing the return value of `fgetc` in an `int` type variable and a `char` type variable in turn, and report any observed difference in the behaviour of your program.

   NOTE: Do NOT open `getc-input.txt` in the browser, and save / copy-paste the content into a local file. Right-click on the link, save the file locally, and run your program on the downloaded file using input redirection (see below).

   ```
   $ ./a.out <  getc-input.txt
   ```

2. Read and understand `robust-scanf.c`. Test your understanding by compiling and running the program on `getc-input.txt`. Also try creating your own test file by mixing various kinds of input including characters, white-space, digit sequences, punctuation marks, etc.

Review questions II

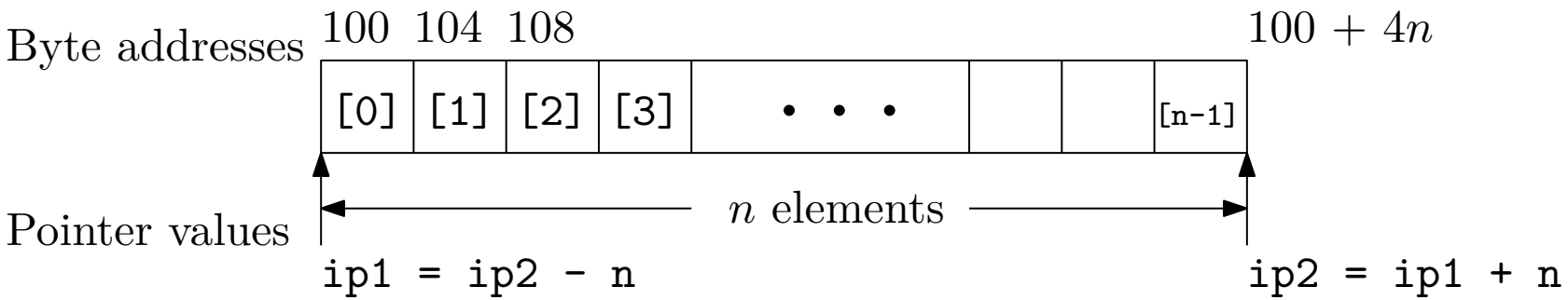
3. Read and understand the various parts of `basic-io.c`. Compile the program using the following commands in turn.

```
$ gcc -g -Wall basic-io.c
$ gcc -g -Wall -DOPTION=1 basic-io.c
$ gcc -g -Wall -DOPTION=2 basic-io.c
```

In each case, run the resulting executable, and explain its behaviour. For the first part of `basic-io.c`, you may use `test-input.txt` as a test input file.

Outline

Pointer arithmetic



`ip1 + n` points to n-th *element* (of the proper type) after what `ip` is pointing to

`ip2 - n` points to n-th *element* (of the proper type) before what `ip` is pointing to

`ip2 - ip1` number of elements between `ip1` and `ip2`

Pointers and arrays

An array name is synonymous with the address of its first element.

Conversely, a pointer can be regarded as an array of elements starting from wherever it is pointing.

```
int a[10] = {...}, *p;

p = a;       /* same as p = &(a[0]); */
*p = 5;      /* same as a[0] = 5; */
p[2] = 6;    /* same as a[2] = 6; */
*(a+3) = 7; /* same as a[3] = 7; */
```

But:

CORRECT	INCORRECT
`&p`	`&a`
`p = a;`	`a = p;`
`p++;`	`a++;`

Pointer-array equivalence (contd.)

Using pointer arithmetic	Using array elements
`p = a + i`	`p = &(a[i])`
`*p = x`	`a[i] = x`
`*(p+j) = x`	`p[j] = x` or `a[i+j] = x`

Review questions

1. What does the following code do and why? (see `strcpy.c`)

```
1  char a[32] = "Introduction", b[32] = "Programming", *s, *t;
2  s = a; t = b;
3  while (*s++ = *t++);
```

2. What output is generated by the following code and why?

```
for (i=0; i < 10; i++)
    printf("abcdefghijklmnop\n" + i);
```

Another review question – slide I

Which of `read_data1`, `read_data2`, `read_data3` is best?

```
typedef struct {
    char name[64];
    int roll, rank;
    float percent;
} STUDENT;

STUDENT *read_data1(void)
{ STUDENT s;
  scanf("%s %d %d %f",
    &(s.name[0]), &(s.roll),
    &(s.rank), &(s.percent));
  return &s;
}
```

Another review question – slide II

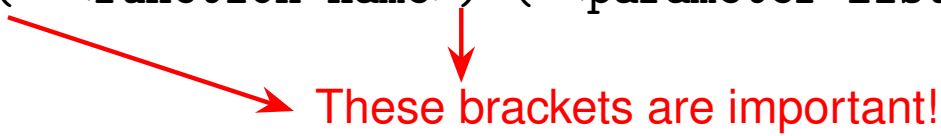
```
STUDENT read_data2(void)
{ STUDENT s;
  scanf("%s %d %d %f",
    &(s.name[0]), &(s.roll),
    &(s.rank), &(s.percent));
  return s;
}

STUDENT *read_data3(STUDENT *s)
{ scanf("%s %d %d %f",
    &(s->name[0]), &(s->roll),
    &(s->rank), &(s->percent));
  return s;
}
```

Outline

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);
}
```

Using qsort and bsearch

```
char **strings;
int *a;
int num_strings, N;

qsort(a, N, sizeof(int), compare_int);
qsort(strings, num_strings, sizeof(char *), compare_strings);
```

Outline

Command-line arguments

Code:

```
for (i = 0; i < ac; i++)
    printf("Argument no. %2d: %s\n", i, av[i]);
```

Running the program:

```
$ ./a.out 10 inputfile-name 100 -k "arg with space" \
    'another arg with space' yet\ another\ arg\ with\ space
```

Output:

```
Argument no.  0: ./a.out
Argument no.  1: 10
Argument no.  2: inputfile-name
Argument no.  3: 100
Argument no.  4: -k
Argument no.  5: arg with space
Argument no.  6: another arg with space
Argument no.  7: yet another arg with space
```

Each `av[i]` is a *string*.
Use `atoi()`, `atol()`, `atoll()`, `atof()` (or `strtol()`, `strtoul()`, `strtod()`) as needed (see the man pages).

Outline

Exercises

1. Experimentally determine / demonstrate the *endianness* of your processor.
2.