Programming basics
(GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

https://github.com/sze-info/ProgrammingBasics

November 6, 2020

Function to swap the values of variables in the caller

Goal:

> Write a function to swap the values of two variables! The effect must be visible in the caller!

Problem:

- Actual parameters are passed by value (the formal parameters are copies of the actual parameters and our goal is to swap the values of the *original* variables and not their *copies*).
- A function may have at most one return value.

swap1.c

```
1 #include <stdio.h>
2
3 void display(int a, int b) {
4   printf("a = %d, b = %d\n", a, b);
5 }
```

Function to swap the values of variables in the caller

swap1.c – First attempt, swap1

```
7  void swap1(int a, int b) {
8    int temp = a;
9    a = b;
10   b = temp;
11 }
```

swap1.c – First lines of main

```
26 int main(void) {
27   int a = 1, b = 2;
28   printf("Original values:\t"); display(a, b);
29   swap1(a, b); printf("After swap1:\t\t"); display(a, b);
```

First lines of the output

```
Original values:  a = 1, b = 2
After swap1:      a = 1, b = 2
```

Function to swap the values of variables in the caller

swap1.c – Second attempt, swap2

```
13 struct twoNumbers { int a, b; };
14
15 struct twoNumbers swap2(int a, int b) {
16   struct twoNumbers temp = {b, a};
17   return temp;
18 }
```

swap1.c – Some lines of main

```
30   struct twoNumbers tn = swap2(a, b); a = tn.a; b = tn.b;
31   printf("After swap2:\t\t"); display(a, b);
```

Corresponding line of output

```
After swap2:        a = 2, b = 1
```

Function to swap the values of variables in the caller

swap1.c – Third attempt, swap3

```
20 void swap3(int* a, int* b) {
21   int temp = *a;
22   *a = *b;
23   *b = temp;
24 }
```

swap1.c – Some lines of main

```
32   swap3(&a, &b); printf("After swap3:\t\t"); display(a, b);
33   return 0;
34 }
```

A snippet of output

```
After swap2:    a = 2, b = 1
After swap3:    a = 1, b = 2
```

Function to swap the values of variables in the caller

swap2.c – Swapping functions

```
3  void swap1(int a, int b) {
4    printf("swap1: address of 'a': %p, address of 'b': %p\n", &a, &b);
5    printf("swap1: value of 'a': %d, value of 'b': %d\n", a, b);
6    int temp = a;
7    a = b;
8    b = temp;
9  }
10
11 void swap3(int* a, int* b) {
12   printf("swap3: address of 'a': %p, address of 'b': %p\n", &a, &b);
13   printf("swap3: value of 'a': %p, value of 'b': %p\n", a, b);
14   printf("swap3: value@address 'a': %d, "
15          "value@address 'b': %d\n", *a, *b);
16   int temp = *a;
17   *a = *b;
18   *b = temp;
19 }
```

Function to swap the values of variables in the caller

swap2.c – The main function

```
21 int main(void) {
22   int a = 1, b = 2;
23   printf("main: address of 'a': %p, address of 'b': %p\n", &a, &b);
24   printf("main: value of 'a': %d, value of 'b': %d\n", a, b);
25   swap1(a, b);
26   printf("main, after calling swap1: "
27          "value of 'a': %d, value of 'b': %d\n", a, b);
28   swap3(&a, &b);
29   printf("main, after calling swap3: "
30          "value of 'a': %d, value of 'b': %d\n", a, b);
31   return 0;
32 }
```

Function to swap the values of variables in the caller

Output

```
main: address of 'a': 0x7ffd85320ef0, address of 'b': 0x7ffd85320ef4
main: value of 'a': 1, value of 'b': 2
swap1: address of 'a': 0x7ffd85320ecc, address of 'b': 0x7ffd85320ec8
swap1: value of 'a': 1, value of 'b': 2
main, after calling swap1: value of 'a': 1, value of 'b': 2
swap3: address of 'a': 0x7ffd85320ec8, address of 'b': 0x7ffd85320ec0
swap3: value of 'a': 0x7ffd85320ef0, value of 'b': 0x7ffd85320ef4
swap3: value@address 'a': 1, value@address 'b': 2
main, after calling swap3: value of 'a': 2, value of 'b': 1
```

Drawing rectangles

rectangle2.c `readTLX`: Do you want to enter further rectangles? If yes, what is the X coord. of the TL corner?

```
39 bool readTLX(int count, int min, int max, int* k) {
40   bool goon;
41   do {
42     printf("X coordinate of the top left corner of rectangle #%d [%d, %d] "
43       "(exits to a negative value) ", count, min, max);
44     scanf("%d", k);
45     goon = *k>=0;
46   } while(goon && (*k<min or *k>max));
47   return goon;
48 }
49
50 int read(int count, char s[], int min, int max) {
51   int k;
52   do {
53     printf("%s rectangle #%d [%d, %d] ",
54       s, count, min, max);
55     scanf("%d", &k);
56   } while(k<min or k>max);
57   return k;
58 }
```

Drawing rectangles

rectangle2.c

```
60 int main(void) {
61   struct rectangle ar[MAXSHAPE]; int count; bool goon = true;
62   printf("Please enter the data of rectangles!\n");
63   for(count=0; count<MAXSHAPE and goon; count++) {
64     goon = readTLX(count+1, MINX, MAXX-1, &ar[count].tl.x);
65     if(goon) {
66       ar[count].tl.y = read(count+1, "Y coordinate of the top left corner", MINY, MAXY-1);
67       ar[count].br.x = read(count+1, "X coordinate of the bottom right corner",
68         ar[count].tl.x+1, MAXX);
69       ar[count].br.y = read(count+1, "Y coordinate of the bottom right corner",
70         ar[count].tl.y+1, MAXY);
71       printf("Drawing character of rectangle #%d: ", count+1);
72       scanf(" %c", &ar[count].c);
73       count++;
74     }
75   }
76   draw(ar, count-1);
77   return 0;
78 }
```

Further information regarding pointers

- Example: pointers.c
- Watch out! `i` is an `int`, but `pi1` and `pi2` are pointers to `int`s!
- Any number of white spaces can be placed on both sides of *
- A pointer can be initialized, too

```
4  int i=3, *pi1, *pi2;
5  pi1 = pi2 = &i; // OK
6  double d=1.5;
7  double* pd = &d; // initialization
```

Pointers

- If an object does not have a memory location/address, even operator & cannot determine it

```
8   // pd = &12.34;
9   // error: lvalue required as unary '&' operand
10  // A literal does not have a memory address, meaningless
```

- In general, assignments can be carried out with pointers of the same type

```
11  // pd = pi1; warning: assignment from
12  //                    incompatible pointer type
```

Pointers

- Exception: any pointers can be assigned to a `void*` pointer
  (≈ dropping type information)

```
13  void *pv;
14  pv = pi1; // OK
```

- It can be done in the opposite direction, but the type of data at the specified address may be incompatible

```
15  pi1 = pv; // OK, but did pv really address an int?
```

- `NULL` is a special address: no data is stored there
- It indicates an error or lack of something
- Can be assigned to any type of pointers

```
16  pv = NULL; // OK
```

Date management

Practical problem:

the structures are usually big, parameter passing needs too much time

Solution:

- pass the *address* of the structure!
- Danger! If the *called* fn. modifies the parameter, that affects the variable in the *caller*, too!
- To avoid the unintended modifications of variables in the *called* fn.: modifier `const` makes the parameter read-only (it can be used with other types as well)
- Indirection + member access: with operator `->`, eg. `(*d).day` ≡ `d->day`

Date management

calendar2.c

```
23 bool check(const struct date* d) { // content validation
24   if(d->month<1 or d->month>12) return false;
25   int days = daysOfMonth(d->year, d->month);
26   if(d->day<1 or d->day>days) return false;
27   return true;
28 }
29
30 int dayOfYear(const struct date* d) { // determining the day of the year
31   int days = d->day;                   // based on year, month and day
32   for(int month=1; month<d->month; month++) {
33     days += daysOfMonth(d->year, month);
34   }
35   return days;
36 }
```

Date management

calendar2.c

```
64 int main(void) {
65   struct date d = {23, 10, 2020};
66   printf("The given date is %s.\n"
67          "%d.%d.%d is the %dth day of the year.\n",
68          (check(&d)?"valid":"invalid"), d.day, d.month, d.year,
69          dayOfYear(&d));
70   struct date xmas = {24, 12, 2020};
71   printf("How many days are left to christmas? %d\n",
72     difference(&d, &xmas));
73   int dy = 300;
74   d = monthAndDay(d.year, dy);
75   printf("The %dth day of %d is: %d.%d\n",
76     dy, d.year, d.day, d.month);
77   return 0;
78 }
```

Bubble sort

bubble2.c

```
1  #include <stdio.h>
2
3  void bubble(int a[], int n) {
4    for(int e=n-1; e>=1; e--) {
5      for(int b=0; b<e; b++) {
6        if(a[b] > a[b+1]) {
7          int swap = a[b];
8          a[b] = a[b+1];
9          a[b+1] = swap;
10       }
11     }
12   }
13 }
```

Bubble sort

bubble2.c

```
15 int main(void) {
16   int numbers[] = {12, 3, 54, -4, 56, 4, 7, 3};
17   int n = sizeof(numbers)/sizeof(numbers[0]);
18   bubble(numbers, n);
19   printf("After sorting:\n");
20   for(int i=0; i<n; i++) {
21     printf("%d\t", numbers[i]);
22   }
23   printf("\n");
24   return 0;
25 }
```

Output

```
After sorting:
-4      3       3       4       7       12      54      56
```

Pointers and arrays

Please consider that

- The size of the array is *not always needed* to be passed, but the function needs to know the number of elements to sort
- The *called* function has modified the array!

Explanation:

- Arrays are usually big $\rightarrow$ always a pointer to the array is passed!
- The name of the array is a *constant pointer* (the pointer cannot be modified, but the data where it points to can be modified), eg.
  `int a[]` $\equiv$ `int* const a`
- The content of the array can be made read-only:
  `const int* const a` $\equiv$ `const int a[]`

Bubble sort

bubble3.c

```
15 void printArray(const int* const a, int n) {
16   for(int i=0; i<n; i++) {
17     printf("%d\t", a[i]);
18   }
19   printf("\n");
20 }
21
22 int main(void) {
23   int numbers[] = {12, 3, 54, -4, 56, 4, 7, 3};
24   int n = sizeof(numbers)/sizeof(numbers[0]);
25   bubble(numbers, n);
26   printf("After sorting:\n");
27   printArray(numbers, n);
28   return 0;
29 }
```

Pointers

Pointer arithmetics: similarly to integers, operations can be evaluated with pointers

- A pointer can be increased and decreased $\rightarrow$ the real increase/decrease of the address is the size of the data
- Pointers can be compared (relations)
- addressOfAnArrayElement = baseAddressOfTheArray + index*sizeof(typeOfArray)
- `array[index]` $\equiv$ `*(array + index)`
- The `void*` pointer is a special one: the size of the object it points to is unknown
- The difference of pointers addressing elements of the same array can be calculated

pointers2.c

```
4   int a[] = { 100, 200, 300 };
5   int* pi = a;
6   printf("Value (address) of the 1st element:\t%d (%p)\n", pi[0], a);
7   pi++;
8   printf("Value (address) of the 2nd element:\t%d (%p)\n", *pi, pi);
9   printf("Value (address) of the 3rd element:\t%d (%p)\n", *(a+2), a+2);
```

Output

```
Value (address) of the 1st element: 100 (0x7ffd290eb7dc)
Value (address) of the 2nd element: 200 (0x7ffd290eb7e0)
Value (address) of the 3rd element: 300 (0x7ffd290eb7e4)
```

Bubble sort

bubble4.c

```
15 void printArray(const int* a, int n) {
16   for(const int* end=a+n; a<end; a++) {
17     printf("%d\t", *a);
18   }
19   printf("\n");
20 }
```

Substitution of substrings

Task:

- Create a *new* string based on an *old* one by replacing a specific substring with something else!
- Eg. "*Jane* cooks, *Jane* bakes, *Jane* does the washing up" $\rightarrow$ "*Emily* cooks, *Emily* bakes, *Emily* does the washing up"

Solution:

1. Search for the first occurrence of *Jane* in the *old* string
2. Append everything in front of it to the end of the initially empty *new* string
3. Next, append *Emily* to *new*
4. Search for the next occurrance of *Jane*, then repeat steps 2 and 3 until we find all occurrences of *Jane*
5. Append the remaining characters after the last occurrence of *Jane* to *new*

Substitution of substrings

What do we need?

- `size_t strlen(const char *s);`
  Provides the length of `s` excluding the terminating `'\0'` character
- `char *strstr(const char *haystack, const char *needle);`
  Returns the first occurrence of `needle` in `haystack`, or a `NULL` pointer if it is not found
- `char *strcat(char *dest, const char *src);`
  `char *strncat(char *dest, const char *src, size_t n);`
  Appends `src` to the end of `dest`, then puts the terminating `'\0'` appropriately at the end of the string
  `strncat()` appends at most `n` characters to the end of `dest`
- `size_t` is generally an `unsigned int`
- There must be sufficient (`strlen(dest)+strlen(src)+1` or `strlen(dest)+n+1`) space starting from `dest`

Substitution of substrings

names1.c

```
8  void names(const char* old, char* new,
9             const char* from, const char* to) {
10   // Start of the substring NOT containing the name looked for
11   const char *begin = old;
12   char *end; // End of the same substring
13   int fromLength = strlen(from);
14   while((end = strstr(begin, from)) != NULL) {
15     // Jane found somewhere
16     // Copy everything in begin of her
17     strncat(new, begin, end-begin);
18     // Append Emily
19     strcat(new, to);
20     // Continue searching after Jane
21     begin = end + fromLength;
22   }
23   // Copying the characters after the last occurrence of Jane
24   strcat(new, begin);
25 }
```

Substitution of substrings

Determining length

```
8  size_t strlen(const char* s) {
9    const char* save = s;
10   while(*s != '\0') s++;
11   return s-save;
12 }
```

Looking for the *needle* in the *haystack*

```
14 // Returns the address of the 1st occurrence of needle in haystack
15 // or a NULL pointer if needle not found
16 char *strstr(const char *haystack, const char *needle){
17         const char *h, *n;
18   for(; *haystack; ++haystack) {
19     for(h=haystack,n=needle; *n!='\0' and *h==*n; ++h,++n);
20     if(*n=='\0') return((char*)haystack);
21   }
22   return NULL;
23 }
```

Substitution of substrings

Appending at most *n* characters

```
25 char* strncat(char *dest, const char *src, size_t n) {
26   char *save = dest;
27   // Positioning to the terminating '\0' of dest
28   while(*dest != '\0') ++dest;
29   // Copying src to the end of dest until the terminating '\0' comes or
30   // at most n characters
31   while(n-- and *src!='\0') {
32     *dest = *src;
33     dest++; src++;
34   }
35   *dest = '\0';
36   // Returning the combined string
37   return save;
38 }
```

Substitution of substrings

Merging strings

```
39 char* strcat(char *dest, const char *src) {
40   char *save = dest;
41   // Positioning to the terminating '\0' of dest
42   while(*dest) ++dest;
43   // Copying src to the end of dest until the terminating '\0' comes
44   while((*dest++ = *src++));
45   // Returning the combined string
46   return save;
47 }
```

Determining vowels and consonants

Task:

Determine from all the letters of a string whether they are vowels or consonants.

Solution:

1. Define a string of vowels (because there are much more consonants)
2. Check all the characters of the string and if the current one is a letter look for it among vowels
3. If it is found somewhere $\rightarrow$ vowel, else $\rightarrow$ consonant

What do we need?

- `char *strchr(const char *s, int c);`
  Returns the address of the 1st occurrence of `c` in `s`, or `NULL` if it is not found
  It works even if `c=='\0'`
- `char *strrchr(const char *s, int c);`
  Like `strchr()`, but returns the address of the last occurrence

Determining vowels and consonants

vowel1.c

```
5  int main(void) {
6    char* text = "Commodore 64 Basic V2";
7    char* vowels = "euioa";
8    printf("Determining vowels (V), consonants (C) and others (-) "
9      "\n%s\n", text);
10   for(; *text; text++) {
11     if(isalpha(*text)) { // Binary <--> linear search?
12       if(strchr(vowels, tolower(*text))) printf("V");
13       else printf("C");
14     } else printf("-");
15   }
16   printf("\n");
17   return 0;
18 }
```

Determining vowels and consonants

Looking for a character in a string

```
5 const char *strchr(const char *s, int c) {
6   while(*s!=c and *s!='\0') s++;
7   if(*s) return s;
8   else return NULL;
9 }
```

Remark: `printf` is too complex (slow) to print a sole character

- `int putchar(int c);`
  Puts character `c` to the standard output
  Return value: the printed character or `EOF` in case of an error

Determining vowels and consonants

vowel2.c

```
11 int main(void) {
12   char* text = "Commodore 64 Basic V2";
13   char* vowels = "euioa";
14   printf("Determining vowels (V), consonants (C) and others (-) "
15     "\n%s\n", text);
16   for(; *text; text++) {
17     if(isalpha(*text)) { // Binary <--> linear search?
18       if(strchr(vowels, tolower(*text))) putchar('V');
19       else putchar('C');
20     } else putchar('-');
21   }
22   printf("\n");
23   return 0;
24 }
```

Comparing names

Task:

Read two names and determine their order according to the alphabet.

What do we need?

- `int strcmp(const char *s1, const char *s2);`
  `int strncmp(const char *s1, const char *s2, size_t n);`
  They compare parameters `s1` and `s2` interpreted as sequences of `unsigned chars`.
  The return value is
  - negative, if `*s1 < *s2`,
  - positive, if `*s1 > *s2`,
  - zero otherwise.

  `strncmp()` carries out the comparison at most with the first `n` characters
- Does it have any sense to directly compare `s1` and `s2` with relational operators?

Comparing names

compare1.c

```
6  int main(void) {
7    printf("Enter two names and we determine which one comes first "
8           "according to the alphabet\n");
9    char n1[MAX], n2[MAX];
10   printf("First name: "); scanf("%s", n1);
11   printf("Second name: "); scanf("%s", n2);
12   int order = strcmp(n1, n2);
13   if(order < 0)
14     printf("%s is ahead of %s\n", n1, n2);
15   else if(order > 0)
16     printf("%s is ahead of %s\n", n2, n1);
17   else
18     printf("The two names are the same\n");
19   return 0;
20 }
```

Comparing names

Comparing strings

```
5  int strcmp(const char *s1, const char *s2 ) {
6    for(; *s1 == *s2; ++s1, ++s2)
7      if(!(*s1)) return 0; // s1 is the same as s2
8    return(*s1 - *s2); } // s1 < s2 or s1 > s2
```

Problem: `scanf` cannot read a string containing white spaces
Possible solutions:

- `char *gets(char *s);`
  Reads a line from standard input until *new line* or `EOF` arrives. It does not store the new line character itself. No buffer overrun protection, deprecated.
- `char *fgets(char *s, int size, FILE *stream);`
  Reads a line from `stream` and stores at most `size-1` characters until *new line* or `EOF` arrives. It stores the new line character as well.

Reading lines

Reading with gets

```
10    printf("First name: "); gets(n1);
11    printf("Second name: "); gets(n2);
```

Reading with fgets

```
10    char* end;
11    printf("First name: "); fgets(n1, MAX, stdin);
12    if((end=strrchr(n1, '\n')) != NULL)
13      *end = '\0';
14    printf("Second name: "); fgets(n2, MAX, stdin);
15    if((end=strrchr(n2, '\n')))
16      *end = '\0';
```

Reading lines

Combination of good properties: `sze_getline` (Be careful! Some libraries already contain the function `getline` and that may cause a name collision. To prevent this, the prefix `sze_` was applied.)

- Never reads more characters than can be stored in the available space
- Does not store the *new line* character
- Clears the character buffer before next reading

Reading with own getline function

```
 7  int sze_getline(char s[], int lim) {
 8    int c, i;
 9    for(i=0; i<lim and (c=getchar())!=EOF and c!='\n'; ++i)
10      s[i] = c;
11    s[i] = '\0';
12    while(c!=EOF and c!='\n') c = getchar();
13    return i;
14  }

20    printf("First name: "); sze_getline(n1, MAX);
21    printf("Second name: "); sze_getline(n2, MAX);
```

Reading lines

Task:

Write a program that reads names from the standard input until EOF or an empty line arrives. Determine and print the longest name!

longest1.c

```
15 int main(void) {
16   int length;          // Length of the current line
17   int max = 0;         // Current maximum length
18   char line[MAX+1];    // Current line
19   char save[MAX+1];    // Currently longest line
20   printf("Determining the longest line\n"
21          "Finish the lines by pressing ENTER\n"
22          "Exit with empty line or by pressing Ctrl+Z\n\n");
23   while((length=sze_getline(line, MAX)) > 0) {
24     if (length > max) {
25       max = length;
26       strcpy(save, line); } }
27   printf("The longest line:\n");
28   if(max > 0) printf("\n%s\n", save); // At least one line was entered
29   return 0;
30 }
```

Reading lines

- `char *strcpy(char *dest, const char *src);`
  `char *strncpy(char *dest, const char *src, size_t n);`
  Copying characters from `src` to `dest`
  They return the address of the result (`dest`)
  `strncpy()` copies at most `n` characters. If `src` is shorter then `n` it stores even the terminating `'\0'` as well
  They do not check that the result fits at `dest` or not!

Copying strings

```
5  char *strcpy(char *dest, const char *src) {
6    char *save = dest;
7    // copying src to dest including the terminating '\0'
8    while((*dest++ = *src++));
9    // Returns the copy
10   return save; }
```

Tokenizing a string

Task:

- Determine the key–value pairs of an URL query string.
- `http://it.sze.hu/index.php?tart=hirek&old=2`

What do we need?

- `char *strtok(char *str, const char *delim);`
  If `str` is not `NULL` it returns the first token found in it. Tokens may be separated by any of the characters in `delim`. In order to find the further tokens `str` must be `NULL`. If no more tokens `strtok` returns `NULL`. Terminating null characters are inserted in `str` during searcing!

Tokenizing a string

keyvalue1.c

```
4  #define SEP "&"
5  int main(void) {
6    char url[] = "http://it.sze.hu/index.php?tart=hirek&old=2";
7    char* pk = strchr(url, '?');
8    char* pv;
9    if(pk && (pk=strtok(pk+1, SEP))) {
10     do {
11       pv = strchr(pk, '=');
12       if(pv) *pv = '\0';
13       else continue;
14       printf("Key: %s, value: %s\n", pk, pv+1);
15     } while((pk=strtok(NULL, SEP)));
16   } else {
17     printf("The URL does not contain query string.\n");
18   }
19   return 0; }
```

Generating random numbers

- Generating pseudo-random numbers (PseudoRandom Number Generator, PRNG)
- Required header: `stdlib.h`
- Initial state: `void srand(unsigned int seed);`, where seed initializes the generator
- Random numbers: $0 \leq$ `int rand(void);` $\leq$ `RAND_MAX`

Examples:

- `x = (double)rand()/RAND_MAX` ahol $\{x | x \in \mathbb{R}, 0 \leq x \leq 1\}$
- `x = MIN + rand()%(MAX-MIN+1)` ahol $\{x | x \in \mathbb{Z}, MIN \leq x \leq MAX\}$

Generating random numbers

Problem: same seed → same number sequences
Solution:

- seed must be different at every program start → current time
- Required header: `time.h`
- `time_t time(time_t *t);`
- Return value: expressed with type `time_t` (`long`); seconds elapsed since "the epoch", 1970-01-01 00:00:00 +0000 (UTC). It also stores it at `t` except that is `NULL`.

The C++ language has much more sophistacated capabilities.
Further information regarding generating random numbers.

Generating random numbers

guess.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MIN 1
#define MAX 100

int main(void) {
  srand(time(NULL));
  int guess, number = MIN + rand()%(MAX-MIN+1);
  printf("Guess a number between %d and %d.\n", MIN, MAX);
  do {
    printf("Guess: "); scanf("%d", &guess);
    if(guess < number) printf("The number is greater.\n");
    else if(guess > number) printf("The number is less.\n");
  } while(guess != number);
  printf("You guessed it!\n");
  return 0; }
```