Lecture 8 - dynamic memory allocation and self-referencing structures

Dynamic memory allocation

Standard functions **malloc** i **calloc** dynamicallly reserve from the system the required blocks of memory.

```
void *malloc(size_t n);
```

Returns:

- pointer to n bytes of a noninitialized memory in the case of success
- NULL, if the requirement cannot be satisfied.

Example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>

int main(void)
{
  char *str;
  /* allocate memory for string */
  if ((str = (char *) malloc(10)) == NULL)
  {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if out of memory */
  }
  /* copy "Hello" into string */
  strcpy(str, "Hello");
  /* display string */
  printf("String is %s\n", str);
  /* free memory */
  free(str);
  return 0;
}
```

```
void *calloc(size_t obj, size_t size);
```

The **calloc** function returns the pointer to the memory area foreseen for an array composed of *nobj* elements each of *size* size.

Function returns **NULL**, if this command could not be realized. The area is initialized with zeros.

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

int main(void)
{
   char *str = NULL;
   /* allocate memory for string */
   str = (char *) calloc(10, sizeof(char));
   /* copy "Hello" into string */
   strcpy(str, "Hello");
   /* display string */
   printf("String is %s\n", str);
   /* free memory */
   free(str);
   return 0;
}
```

Attention Pointer values returned by both functions should be casted to an appropriate type.

Changing the size of the assigned memory

```
void *realloc(void *p, size_t size);
```

The *realloc* function changes the size of the object pointed by **p** to the value determined by **size**. The content of the object is not changed in its initial part of the size equal to the smaller of the two sizes: the old one and the new one.

If the new size is larger, then the extra area of memory is not initialized.

The function returns the pointer to the new area or NULL, if the order could not be realized. Then pointer **p** is not changed.

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

int main(void)
{
  char *str;
  /* allocate memory for string */
  str = (char *) malloc(10);
  /* copy "Hello" into string */
  strcpy(str, "Hello");
  printf("String is %s\n  Address is %p\n", str, str);
  str = (char *) realloc(str, 20);
  printf("String is %s\n  New address is %p\n",
          str, str);
  /* free memory */
  free(str);
  return 0;
}
```

Release of the reserved memory

```
void free(void *p);
```

The *free* function releases the memory area pointed by *p*; it does nothing if *p* is equal to NULL.

Argument *p* must be a pointer to an area previously assigned by one of the functions: *malloc, calloc, realloc*.

Difference between standard array declaration and its dynamic allocation

```
1)    char hello[5];

2)    char *hello=(char *) malloc((size_t) 5);
```

1. memory assigned to the stack memory, which is again used after finishing the execution of the function,
2. allocation to the heap memory - heap of dynamic variables in the operational memory.

Heap memory - the whole memory except the stack buffer (and the safety buffer around stack).

Attention:

- it's an error to release something that was not allocated by means of the *malloc* or *calloc* function.
- it is also an error to use something that was previously released.

Typical and not correct piece of code is the following loop, that releases memory blocks connected in string:

```
for (p=head; p != NULL; p=p->next)  /* Error  */
  free(p);
```

Correct way is to store everything, that may be further needed before the release of memory:

```
for (p=head; p != NULL; p=q) {
  q=p->next;
  free(p);
}
```

Solution assumed by the authors of **Numerical Recipes**:

```
#include <stdlib.h>
#include <stdio.h>
#define TYPFLOAT long double
#define MSQRT sqrtl
#define MFABS fabsl

void nrerror(char error_text[])
{
 fprintf(stderr,
        "Numerical Recipes run-time error...\n");
 fprintf(stderr,"%s\n",error_text);
 fprintf(stderr,"...now exiting to system...\n");
 exit(1);
}

TYPFLOAT **dmatrix(int nrl,int nrh,int ncl,int nch)
{
 int i;
 TYPFLOAT **m;

 m=(TYPFLOAT **)
   malloc((unsigned) (nrh-nrl+1)*sizeof(TYPFLOAT*));
 if (!m) nrerror("allocation failure 1 in dmatrix()");
 m -= nrl;

 for(i=nrl;i<=nrh;i++) {
   m[i]=(TYPFLOAT *)
      malloc((unsigned) (nch-ncl+1)*sizeof(TYPFLOAT));
   if (!m[i])
     nrerror("allocation failure 2 in dmatrix()");
   m[i] -= ncl;
 }
 return m;
}

void free_dmatrix(TYPFLOAT **m,
   int nrl,int nrh,int ncl,int nch)
{
 int i;

 for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
 free((char*) (m+nrl));
}
```

Calls to function *dmatrix* may look as follows:

```
TYPFLOAT **Q;

/* .......  */
Q=dmatrix(-10,N,-5,N);
```

Structures referencing to themselves (self - referencing structures - list creation

List - organized sequence of elements, each of which contains pointer to the next element.
Tree - organized sequence of elements, each of which contains pointer to the next and previous element.

Example (program creating the list of data read from the keyboard and printing it):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DL_NAZW 20  /* max. length of name  */

typedef struct element /* def. of type element */
   {  char nazw[DL_NAZW+1]; /*last name */
      int age;             /* age */
      struct element *nast; /* pointer to the
                               next element */
   } t_element;

void main()
{  void reading(t_element **); /* creation function  */
   void writing(t_element *);   /* writing function  */
   t_element *poczatek; /* pointer to
                           the beginning of the list  */
   reading(&poczatek);
   writing(poczatek);
}
/* Function for reading and creation of the list */
void reading(t_element **adpocz)
{  char nazwwe[DL_NAZW+1]; /* the name read */
   t_element *temp;        /* to exchange
                              the pointer */
   *adpocz=NULL;    /* at the beginning
                       an empty list */
   while (1) {
      printf("Last name: ");
      gets(nazwwe);
      if (strlen(nazwwe)) {
         temp=(t_element *) malloc(sizeof(t_element));
         strcpy(temp->nazw,nazwwe);
         printf("age: ");
         scanf("%d", &temp->age);
         getchar(); /* jump over the character \n */
         temp->nast=*adpocz;
         *adpocz=temp;
      }
      else break; /* exit, if the name is empty  */
   }
}
/* Function printing out the list content */
void writing(t_element *poc)
{  printf("\n\n   Last name            Age \n\n");
   while (poc) {
      printf("%20s  %3d\n",poc->nazw, poc->age);
      poc=poc->nast;
   }
}
```

User-defined data types

The *typedef* feature allows users to define new data types that are equivalent to existing data types.

In general terms a new data type is defined as

```
typedef  type  new-type;
```

where type refers to an existing type (either a standard data type or a previous user-defined data type), and new-type refers to the new user-defined data type. It should be understood that the new data type will be new in name only. In reality, the new data type will not be fundamentally different from one of the standard data types.

Examples

```
typedef int age;

then declaration:
age male, female;

is equivalent to:
int male, female;
```

Example 2

```
typedef float height[100];
height men, women;
```

defines height as a 100-element, floating-point type array. Hence men and women are 100-element, floating-point arrays.

A user-defined structure type:

```
typedef struct {
   member 1;
   member 2;
   ....
   member m;
}  new-type;
```

Example

```
typedef struct {
   int acct_no;
   char acct_type;
   char name[25];
   float balance;
}  record;

record oldcustomer, newcustomer;
```

Example (nested use of the typedef declaration)

```
typedef struct {
   int month;
   int day;
   int year;
}  date;

typedef struct {
   int acct_no;
   int acct_type;
   char name[25];
   float balance;
   date lastpayment;
}  record;

record customer[100];
```