DYNAMIC MEMORY ALLOCATION

Computer memory available for the program can be divided into four areas:

- **program code**,
- **static** data (ie. constants and global variables),
- **automatic** data
  → variables created and deleted automatically by the compiler,
  on the ***stack***. Example of automatic data are local variables
  in a function:

```
void exemplary_ function (void)
{
  float local_variable;
  local_variable =10;
}
```

- **dynamic** data
  → organized by the dynamic memory manager,
  can be created and deleted at any time during operation of the program,
  is allocated on the *heap*.

Dynamic variables:

→ the **programmer** – is responsible for creating them (reservation of memory) and for their removal (release memory)

→ access to dynamic variable is possible only by its address in memory (stored in the variable indicator)

→ the use of the unallocated area is likely to cause an error!

→ the attempt to re-release of already released area will cause an error!

Example 1: illustration of the "life-time" of variables

static variable ↔ having your own computer (all the time)
local variable ↔ laboratory computer allocated to student (only lab time)
dynamic variable ↔ computer from rental office (for any time)

Example 2: The idea of **accessing to the object** using the **pointer** – **address** - **link**

POINTER / INDICATOR	→	**OBJECT**
phone number	→	phone set
Internet address / link	→	HTML page on the WEB server
postal address	→	apartment, house
PESEL / social security number	→	the person (Jan Kowalski)
room number	→	rented apartment in the hotel
cloakroom number / token	→	**dynamically allocated coat-hanger**

In "C" language, for dynamic memory allocation (the creation of dynamic variables) serve the specialised set of functions from the library: <**alloc**.h> or <**stdlib**.h>

```
void *malloc( size_t  size );                          // allocation of a memory block
void *calloc( size_t  num,  size_t size);              // allocation of zero-initialised array
void *realloc( void* old_pointer,  size_t  new_size);  // change the size
void  free( void* pointer);                            // deallocation – release of memory
```

```
int main( void )
{
   int *ptr = NULL;        // the variable to store the address of allocated memory
   • • •
   ptr = (int*) malloc( sizeof(int) );                  // allocation of single int
   if( ptr == NULL )
      {  printf( "Error allocating memory" );  return;
   • • •
   * ptr = 10;                          // examples of operations on a dynamic number
   * ptr *= 2;
   printf( "%d", * ptr );
   scanf( "%d", ptr );
   • • •
   free( ptr );                         // releasing the memory before the program ends
   return 0;
}
```

Example operations on a dynamic array (the sequence of numbers):

```
int main( void )
{
   int  array_size;
   double  * array_ptr;
   printf( "How many numbers do you want to enter: " );
   scanf( "%d", &array_size );
   if( array_ptr = (double*) calloc( array_size, sizeof(double) ) )
      {
         for( int i = 0; i < array_size, i++ );
            *( array_ptr + i ) = 100;                    // array_ptr[ i ] = 100;
         • • •
      }
   • • •
   if(array_ptr ) free( array_ptr );
   return 0;
}
```

In „**C**++” language, for dynamic memory allocation,
we can continue to use the allocation functions of standard library <stdlib.h>,
but it is recommended to use the „objective oriented” operators: **new** and **delete**

```
    <pointer_to_object> = new  <the_type_of_object>;
    delete  < pointer_to_object > ;
or
    <pointer_to_array> = new  <the_type_of_element> [ size_of_array ] ;
    delete [ ] < pointer_to_ array > ;
```

Example:

```
int*  ptr ;                    // pointer to the integer variable
ptr = new(nothrow) int ;       // creation of a new object (new int variable )
if( ptr != NULL )
   {
      *ptr = 10 ;              // assign a value (access through pointer ptr)
      printf( "%d" , *ptr );   // printing the content of allocated variable.
      • • •
      delete ptr ;             // removing the variable (release the memory)
   }
```

Comparing the creation of a regular/static array and dynamic array:

```
// creation of a standard array (here for illustration only, to visualize the difference)
const  int  ARRAY_SIZE = 100;
double  standard_array[ ARRAY_SIZE ];
```

```
// creation (and releasing) of dynamically allocated array
int  array_size;
cout << "How many numbers do you want to enter: " ;
cin >> array_size ;

double *dynamic_array = NULL;
dynamic_array = new(nothrow) double [ array_size ];

• • •

for(int i=0; i<array_size; i++)
   dynamic_array [ i ] = 10.5;

• • •

for(int i=0; i<array_size; i++)
   cout << endl << "Array[" << i+1 << "] = " << dynamic_array [ i ];

• • •

if( dynamic_array ) delete [ ] dynamic_array;
```

```
double* ptr;
try {
   ptr = new double [100 ];
} catch( bad_alloc& err ) {
   cout << "Error: "<< err.what() << endl;
}
```

Example 1 – single reallocation (change the size) of one-dimensional array

```
int main( )
{
   // Create the 10-element array containing a numbers in range  [-50÷50]
   int size=10;
   long*  array = new(nothrow) long[ size ];
   for(int i=0; i< size; i++)
      array[ i ] = rand()%101 - 50;

   cout<<endl<<"The content of the array after drawing elements: "<<endl;
   for (int i=0; i< size ; i++)
      cout << endl <<"Array[" << i << "] = " << array[ i ];
   cout<<endl<<"Array size: "<< size << endl;

   // count how many of the drawn numbers have a positive value
   int  count_positive=0;
   for(int i=0; i< size; i++)
      if( array[i]>0 )
         count_positive++;

   // remove all the negative numbers → and reduce the amount of used memory
   long* temporary_array = new(nothrow) long [count_positive];
   if( temporary_array ==NULL )
      cout<<" CAUTION - error while creating the temporary array ";
   else
      {
         int j=0;
         for(int i=0;i< size;i++)
            if( array[i] > 0 )
               {
                  temporary_array [ j ] = array[ i ];
                  j++;
               }

         delete [ ] array;
         array = temporary_array;
         size = count_positive;
      }

   cout<<endl<<"The content of the array after removing negative numbers:"<<endl;
   for (int i=0; i< size ; i++)
      cout << endl <<"Array[" << i << "] = " << array[ i ];
   cout<<endl<<"Array size: " << size << endl;

   cin.get();
   if(array) delete [ ] array; // final release of all used memory, before program exits
   return 0;
}
```

Example (2) – alternative version of example (1) → using the functions

```
bool  REMOVE_NEGATIVE(long* &array_ptr, int &array_size)
{
   int count_positive=0;
   for(int i=0; i<array_size; i++)
      if( array_ptr[i]>0 )
         count_positive++;

   long* temporary_array = new(nothrow) long [count_positive];
   if( temporary_array==NULL )
      return false;

   int j=0;
   for(int i=0; i<array_size; i++)
      if( array_ptr[i]>0 )
         {
            temporary_array[ j ] = array_ptr[ i ];
            j++;
         }

   delete [ ] array_ptr;
   array_ptr = temporary_array;
   array_size = count_positive;
   return true;
}


long* GENERATE_POSITIVE_AND_NEGATIVE(int  count);
void   DISPLAY_ARRAY(long* array_ptr, int array_size);
bool   REMOVE_NEGATIVE(long* &array_ptr, int &array_size);


int main( )
   {
      int n=10;
      long *array = GENERATE_POSITIVE_AND_NEGATIVE(n);
      DISPLAY_ARRAY( array,n);
      cin.get();

      if( REMOVE_NEGATIVE( array,n )==false )
         cout<<"Attention - error while deleting negative values";

      cout<<endl<<endl<<"After calling function  REMOVE_NEGATIVE:"<<endl;
      DISPLAY_ARRAY( array,n );
      cin.get();
      if( array ) delete [ ] array;
      return 0;
   }
```

continuation of Example (2)

```
void DISPLAY_ARRAY(long* array, int size)
{
   for (int i=0; i<size ; i++)
      cout << endl <<"Array[" << i << "] = " << array[ i ];
   cout<<endl<<"Array size: "<< size << endl;
}


long* GENERATE_POSITIVE_AND_NEGATIVE(int size)
{
   long* new_array = new(nothrow) long[ size ];
   for(int i=0; i<size ; i++)
      new_array[ i ] = rand ()%101 - 50;
   return new_array;
}
```