Echahid Hamma Lakhdar University of El-Oued
Department of Computer Science
Level: 2nd Year LMD Computer Science
Module: Algorithms and Data Structures
(Memory Allocation / Subprograms)

Static Allocation

In static memory allocation, the size of the allocated memory is fixed before program execution. Once memory is allocated, it cannot be changed. For example, in C language if the programmer writes: `int Tab[10];` Tab is an array that can store a sequence of data of the same type. It can store ten integer elements. It cannot store more than ten, and the reserved size is equal to $4 * 10$ bytes.

Example

int A,B; // reservation of 8 bytes.
float C,D; // reservation of 8 bytes.
char ch[]="good evening!"; // reservation of 10 bytes.
char chaine[][10]={"black","white","red"} // reservation of 30 bytes.

In the case of pointers (a pointer reserves 4 bytes in memory).

double *A; // reservation of 4 bytes.
char *ch; // reservation of 4 bytes.
int *b; // reservation of 4 bytes.
char *c="good evening!" // reservation of 4+10 bytes.
char *chaine[]={"black","white","red"} // reservation of 4*3+5+6+6 bytes.

Dynamic Allocation

In dynamic allocation, memory can be allocated depending on the addition or deletion of variables, and the size of memory can be increased or decreased. The main advantage of dynamic memory allocation is that it saves memory. The programmer can allocate or free memory space when necessary at runtime.

In this case, the data size is only known during execution, so the goal is to find a way to reserve or free memory space as needed during program execution.

The malloc function and the sizeof operator

- The malloc function of the `stdlib` library helps us locate and reserve memory during program execution. This function allocates the required number of bytes and returns a void pointer to the first byte of the allocated memory.
- The malloc function provides the address of a block of available memory of N bytes.

```
int *T = malloc(2000);
```

This provides the address of a block of 2000 available bytes and assigns it to T. If there is not enough memory, T gets the value zero.

- The `sizeof` operator allows us to know the size in bytes of a constant, a variable, or a type.

Example 1

```
#include < stdio.h >
int main()
{
const int MAX=100;
char c='A';
int i=10;
float f;
char * pc=&c;
char t1[]="hello";
int t2[MAX];
printf("Size of MAX = %d",sizeof(MAX));
printf("Size of c = %d",sizeof(c));
printf("Size of char = %d",sizeof(char));
printf("Size of i = %d",sizeof(i));
printf("Size of int = %d",sizeof(int));
printf("Size of f = %d",sizeof(f));
printf("Size of float = %d",sizeof(float));
printf("Size of pc = %d",sizeof(pc));
printf("Size of pointer = %d",sizeof(char*));
printf("Size of t1 = %d",sizeof(t1));
printf("Size of t2 = %d",sizeof(t2));
return 0;
}
```

Result

```
Size of MAX = 4
Size of c = 1
Size of char = 1
Size of i = 4
Size of int = 4
Size of f = 4
Size of float = 4
Size of pc = 8
Size of pointer = 8
Size of t1 = 8
Size of t2 = 400
```

Example 2

```
#include < stdio.h >
int main()
{
int nbre;
int *Pnbre;
printf("Please enter the number of values = ");
scanf("%d",&nbre);
Pnbre=malloc(nbre*sizeof(int));
return 0;
}
```

Releasing memory space

- If we no longer need a block of memory that we reserved with malloc, we can release it using the `free` function of the `stdlib` library.

```
free(pointer);
```

NB:

- Do not try to free memory with `free` that was not allocated by `malloc`.
- The `free` function does not change the pointer content.
- It is recommended to assign zero to the pointer immediately after freeing the memory block it was pointing to.
- If memory is not explicitly freed with `free`, then it is automatically freed at the end of the program execution.

Example

```
#include < stdio.h >
#include < stdlib.h >
int main()
{
long * table1, * table2;
table1 = malloc(1000*sizeof(long));
table2 = calloc(1000,sizeof(long));
if(table1 != NULL) {
printf("Free table1");
free(table1);
}
if(table2 != NULL) {
printf("Free table2");
free(table2);
}
return 0;
}
```

Note:
The `calloc` function allocates a memory block while initializing all bytes to zero. Although relatively close to malloc, there are two differences: - Initialization: `calloc` sets all bytes of the block to zero, while `malloc` does not modify the memory. - Parameters: `calloc` requires two parameters (the number of elements to allocate and the size of each element), while `malloc` expects the total size of the block to allocate.

```
int * pointer = (int *) calloc( INT_NUMBER, sizeof(int) );
```

Subprograms

A subprogram is a well-defined, named sequence of instructions. It can be called from another part of the program. Subprograms allow us to decompose a program into smaller parts to simplify development and make code reusable.

There are two main categories of subprograms in C language: - **Functions**: return a value. - **Procedures (void functions)**: do not return a value.

—

Function Prototype

The prototype of a function allows the compiler to check its usage before its definition. It specifies: - the return type, - the function name, - the list of parameters (types and number).

```
return_type function_name(type1 param1, type2 param2, ...);
```

Example:

```
int add(int a, int b);
```

—

Function Definition

The function definition includes the instructions to be executed when the function is called.

```
return_type function_name(type1 param1, type2 param2, ...) { // function body return value; }
```

Example:

```
int add(int a, int b) { return a + b; }
```

—

Function Call

To execute a function, we use its name followed by parentheses containing arguments.

Example:

```
int main() { int result = add(5, 3); printf("Result = %d", result); return 0; }
```

—

Parameter Passing

There are two main ways to pass parameters in C:

1. **Pass by Value**: - A copy of the argument is given to the function. - Any modification inside the function does not affect the original variable.

Example:

```
void increment(int x) { x = x + 1; }
```

2. **Pass by Reference**: - The memory address of the argument is given to the function. - Any modification inside the function affects the original variable.

Example:

```
void increment(int *x) { *x = *x + 1; }
```

—

Functions and Arrays

- When an array is passed to a function, its address (the pointer to the first element) is passed. - This means the function can modify the content of the array.

Example:

```
void display(int arr[], int size) { for(int i=0; i¡size; i++) printf("%d ", arr[i]); }
```

—

Functions and Pointers

- Pointers are often used to modify variables inside functions or to manage dynamic memory.

Example:

```
void swap(int *a, int *b) { int temp = *a; *a = *b; *b = temp; }
```

—

Recursive Functions

A recursive function is a function that calls itself. It must always have a **base case** to stop the recursion, otherwise it will create an infinite loop.

Example: Factorial Function

```
int factorial(int n) { if (n == 0) return 1; // base case else return n * factorial(n-1); // recursive call }
```

Example: Fibonacci Function

```
int fibonacci(int n) { if (n == 0) return 0; if (n == 1) return 1; return fibonacci(n-1) + fibonacci(n-2); }
```

—

Advantages of Subprograms

- Code reuse. - Program clarity and better structure. - Easier debugging and maintenance. - Reduced redundancy.

Disadvantages of Subprograms

- Slight execution overhead due to function calls. - May increase memory usage if recursion is used intensively.