People never make mistakes are people who do nothing. And the biggest mistake in life is to think yourself perfect.

~Y. Emre
Call by Value and Call by Reference

- Normally a value of parameter sent to a function does not change. And modifications in function does not effect original variable.

- The case in which the original variable is not changed but its copy is sent to a function is called "call by value" or "pass by value".

- Sometimes we need to return more than one value from a function or we need the original variable changed by the function.
For this purposes we use "call by reference" or "pass by reference".

In call by reference, arguments are not passed with their values, but with their addresses. Thus, all modifications on arguments effect the original variable.
Call by Value

```c
#include <stdio.h>
void arttir(int);

int main14(void)
{
    int i;
    i = 5;
    printf("oncesi %d\n", i);
    arttir(i);
    printf("sonrasi %d\n", i);
    getchar();
    return 0;
}

void arttir(int k)
{
    k++;
}
```
Call by Reference

```c
#include <stdio.h>
void increment(int *);
int main(void)
{
    int i;
    i = 5;
    printf("oncesi %d\n", i);
    increment(&i);
    printf("sonrasi %d\n", i);
    getchar();
    return 0;
}

void increment(int *k)
{
    (*k)++;
}
```
If your function has to return more than one value, pass by reference usage is necessary.

Because return keyword can only return one value from function.

For example, we want to write a division function that gives division result and remainder.

In this case, divided number and divisor is sent to function and remainder and division should be returned back from function.

As return keyword can only return one value, second value must be returned by reference method.
Call by Reference

```c
#include<stdio.h>

int bolme_isleemi( int, int, int * );

int main( void )
{
    int bolunen, bolen;
    int bolum, kalan;
    bolunen = 13;
    bolen = 4;
    bolum = bolme_isleemi( bolunen, bolen, &kalan );
    printf( "Bolum: %d Kalan: %d\n", bolum, kalan );
    getchar();
    return 0;
}

int bolme_isleemi( int bolunen, int bolen, int *kalan )
{
    *kalan = bolunen % bolen;
    return bolunen / bolen;
}
```
Dynamic Memory Allocation

- When a program executes, the operating system reserves space to run the program (stack and heap).

- The stack is memory space where functions and their locally defined variables reside.

- The heap is reserved for the program and it is an empty section to use for allocating memory at runtime.
Stack and Heap

► Stack and heap are the logical parts of memory.
► Stack works in LIFO (Last in First Out) principal. If considered as a box: one of the books that you put in the box is placed on top of the other. Latest added book is accessed first.
► Heap is like a farm of programmer and usage of it is in responsibility of programmer.
Stack and Heap

- While we store value type variables, pointer variables and code addresses in the stack.
- Stack is faster than heap. Because working principal of stack is easy and spaces that we want to reach are placed one after the other.
Stack and Heap

- Memory spaces that is shown by pointers are stored in the heap space.
- Heap is slower than stack. Because to reach an object in the heap we should perform a complex search as we put an object into any empty space in heap.
Dynamic Memory Allocation

- We may need an array whose number of elements may vary according to needs.

- For such kind of need, creating a large array to solve the problem may consume memory in vain.

- More effective solution is usage of dynamic memory allocation.
In dynamic memory allocation, amount of memory needed is determined during the execution of program.

- *malloc, calloc, or realloc* are the three functions used to manipulate memory.

- These commonly used functions are available through the *stdlib* library, so you must include this library in order to use them.

```c
#include<stdlib.h>
```
Malloc() Function

► Malloc function is used to allocate a block of memory for one variable.

► If there is not enough memory available, malloc will return NULL.

```c
int *ptr;
ptr = (int *) malloc(n*sizeof(int));
```
Calloc() Function

- Calloc function is also used to allocate a block of memory.

- If there is not enough memory available, calloc will return NULL.

- Unlike malloc function, it takes two arguments.

  ```c
  char *ptr;
  ptr = (char *)calloc(10, sizeof(char));
  ```
Realloc() Function

- Realloc is used to resize an allocated memory space.

- A pointer that will point the starting address of resized memory space and new size are passed to realloc function as parameter.

```c
void *realloc(void *ptr, size_t size);
```
Free() Function

- In high level programming languages such as (C#, Java) removing unused objects from memory is achieved automatically by Garbage Collector.

- Unfortunately, there is no garbage collector for C language and bad and good programmer is separated easily with this issue.
Free() Function

- How important an effective memory management is may be understood when we write large programs.

- We should avoid consuming unnecessary memory.

- Every call to an malloc or calloc function you must have a corresponding call to free.

```c
int *ptr;
ptr = (int *) malloc(n*sizeof(int));
free(ptr);
```
Example 1

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

int main(void)
{
    int n,i,*ptr,sum=0;
    printf("Eleman sayısını girin\n");
    scanf("%d",&n);

    ptr = (int *)malloc(n*sizeof(int));
    if(ptr==NULL)
    {
        printf("Yeterli hafıza yok");
    }
    printf("Dizi elemanlarını girin\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",ptr+i);
        sum += *(ptr+i);
    }
    printf("Toplam = %d",sum);
    getchar();
    getchar();
    return 0;
}
```
#include <stdio.h>
#include <stdlib.h>

int *dizileri_birlestir( int [], int, int [], int );

int main( void )
{
    int i;
    int liste_1[5] = { 6, 7, 8, 9, 10 };  
    int liste_2[7] = {13, 7, 12, 9, 7, 1, 14 };  
    // sonucun dondurulmesi icin pointer tanimliyoruz
    int *ptr;

    ptr = dizileri_birlestir( liste_1, 5, liste_2, 7 );

    // ptr isimli pointer'i bir dizi olarak dusunebiliriz
    for( i = 0; i < 12; i++ )
        printf("%d ", ptr[i] );
    printf("\n");
    return 0;
}
Example 2

```c
int *dizileri_birlestir( int dizi_1[], int boyut_1,
                        int dizi_2[], int boyut_2 )
{
    int *sonuc = (int *)calloc( boyut_1+boyut_2, sizeof(int) );
    int i, k;
// Birinci dizinin değerleri ataniyor.
    for( i = 0; i < boyut_1; i++ )
        sonuc[i] = dizi_1[i];

// İkinci dizinin değerleri ataniyor.
    for( k = 0; k < boyut_2; i++, k++ ) {
        sonuc[i] = dizi_2[k];
    }
// Geriye sonuc dizisi gönderiliyor.
    return sonuc;
}
```
Next Week

► Examples with Pointers
References


► Paul J. Deitel, “C How to Program”, Harvey Deitel.

► “A book on C”, All Kelley, İra Pohl
Thanks for listening

Any Questions?
The first step is to establish that something is possible; then probability will occur.