

COMP 8567 Advanced Systems Programming

Advanced C Programming Techniques (Pointers and Functions)

Summer 2023

Outline

- Introduction
- Pointers
- Manipulation of pointers
- The special pointer NULL
- Pointer arithmetic + and –
- Pointers and arrays
- The pointer-to void type
- Dynamic Memory Allocation
- Passing Parameters by value and reference
- Passing Functions as Parameters
- Summary and Conclusion

Introduction

Pointers are crucial to C, and they are mainly used to:

- Provide the means by which functions can modify their **calling arguments**.
- Support the **dynamic allocation** of memory.
- Refer to **a large data structure** in a simple manner.
- Support data structures such as **linked lists**.

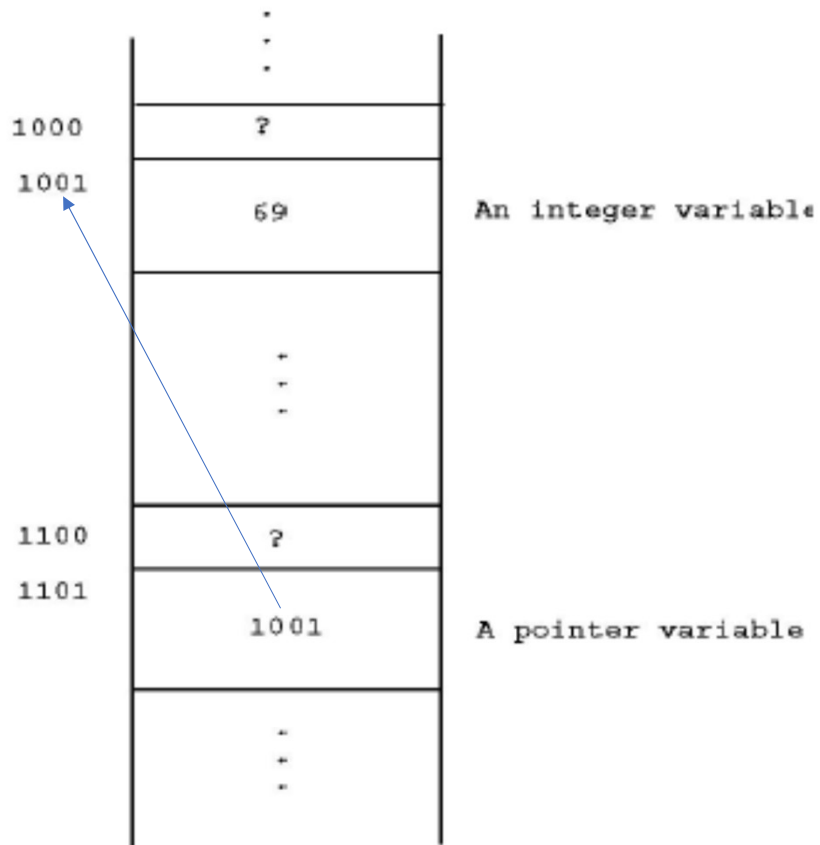
Pointers are Addresses

- A pointer is a **variable** whose value is a memory addresses.
- A pointer variable is similar to other variables
 - It requires memory and stores a value.
- However, the purpose of a pointer is special.
- A data variable stores data (e.g. age).
- A pointer variable stores memory addresses (of variables)



A pointer contains/stores the **memory address** of another variable (hence it **points** to another variable)

Memory of the computer



- Syntax: ***base type *pointer name;***
- where base type defines the type of variable the pointer can point to.
- Example : ***int *ptr; float * p1; int *p2;***
 - ptr is a pointer to an integer variable.
 - p1 is a pointer to a float variable

Pointers- Simple Example //prelim.c

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

```
int main(void)
```

```
{
```

```
int *a; //a is a pointer variable to an integer
```

```
int b=100;
```

```
a=&b; // pointer variable a is assigned the address of b
```

```
*a=200; //dereferencing
```

```
printf("\nThe address of b is %d", &b);
```

```
printf("\nThe value of a is %d", a);
```

```
printf("\nThe value of b is %d", *a);
```

```
}
```

Manipulation of Pointers

Operators:

& : a unary operator that returns the address of its operand.

```
int a=10;
```

```
int *ptr; //ptr is a pointer to an integer
```

```
ptr=&a; //ptr now contains the address of the integer variable a
```

***** : A unary operator used to **dereference** its operand (a pointer)

```
int a=10;
```

```
int *ptr;
```

```
ptr=&a;
```

***ptr** refers to the **value of the variable** referenced by ptr (i.e variable a)

```
printf("%d",*ptr); //output =10
```



```

#include <stdio.h>
//ex1.c
int main(void) {

int n1=10, n2; // 2 variables of type integer

int *ptr;    //a pointer to integer

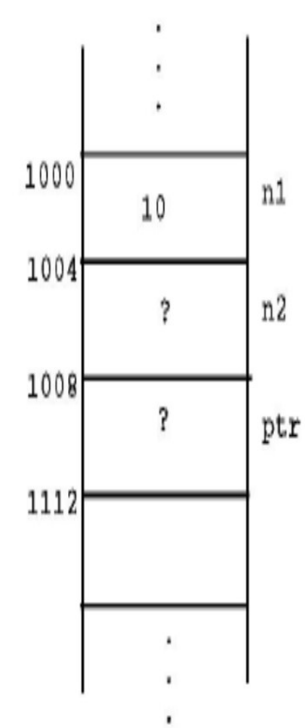
ptr = &n1;    // ptr = address of num1

n2 = *ptr;    //n2 = value stored at the address pointed by ptr, i.e value of n1

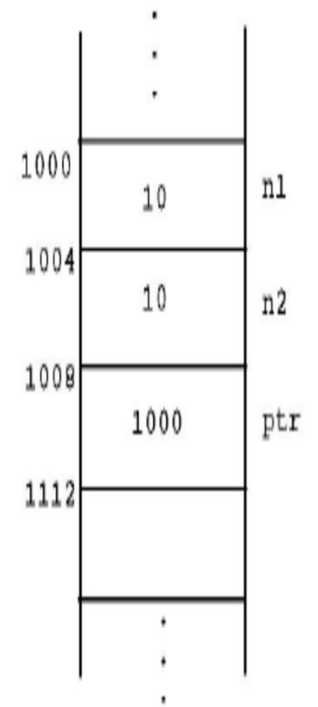
printf("\n n1:%d n2:%d *ptr:%d \n\n", n1, n2, *ptr);

return(0);
}

```



Decalaration of
n1, n2 and ptr



The contents of n1, n2,
and ptr before the end

Pointer Declaration and Assignments: Examples:

```
float z;
```

```
int x, y;
```

```
int *ptr1 = &x // Like any variable,  
pointers can be initialized at declaration  
time with the address of a compatible  
variable, but not with a constant
```

```
int *ptr2 = &y;
```

**The syntax of the following assignments
is correct:**

```
ptr2 = ptr1;
```

```
/* ptr2 receives the contents of  
ptr1 which is the address of x in that case  
*/
```

```
*ptr1 = *ptr2; // this is equivalent to x=y
```

```
*ptr2 = 4; // this is equivalent to y=4  
x = *ptr2; // this is equivalent to x=y
```

**However, the syntax of the following
assignments is not correct:**

```
ptr1 = x; // type mismatch: assigning  
a data value (integer) to a pointer
```

```
y = ptr1; // type mismatch: assigning  
an address to an integer variable
```

```
ptr1=&z; // this is wrong because ptr1  
is supposed to point to an integer variable  
and z is declared as float.
```

Special Pointer NULL //nullp.c

We sometimes need to indicate that a pointer is not pointing to any valid data. For that purpose, the **constant NULL** is defined in **stdlib.h** and can be used in a C program.

A common use of NULL: When a function returning a pointer wants to indicate a failure operation, the function can return NULL to specifically indicate a failure

Example :

```
int *afunction();    /* a function that returns a pointer to an integer */
int *ptr;            /* a pointer to integer */
ptr = afunction();
if (ptr == NULL)     /* afunction() was unsuccessful */
{take-appropriate-action}
```

Pointer Arithmetic

When a pointer is incremented (or decremented), it will point to the memory location of the next (or previous) element of its **base type**.

Example :

Assume that ptr, a pointer to int, contains the address 200 and n is an integer.

Also assume that the **size of an integer is 4 bytes**.

ptr+1 will contain the address $200 + 4 = 204$.

ptr- 1 will contain the address $200 - 4 = 196$.

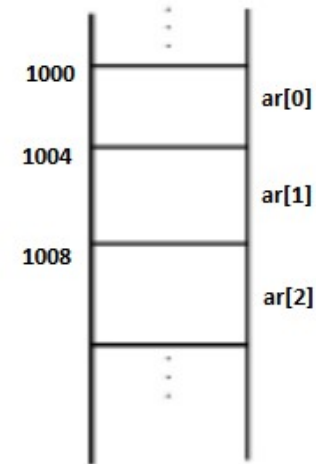
ptr = ptr + n ptr will contain the address $200 + nx4$. //ptr=ptr+2 = ptr+2x4= 208

ptr = ptr - n ptr will contain the address $200 - nx4$. //ptr=ptr-2=ptr-2x4=192

```
//ex3.c
#include <stdio.h>
int main(void)
{
    int a=100;
    int *p=&a;
    printf("\nThe address of p1 %d\n",p);
    p=p+1;
    printf("\nThe address of p1+1 is %d\n",p);
    p=p-1;
    printf("\nThe address of p1-1 is %d\n",p);
    p=p+2;
    printf("\nThe address of p1+2 is %d\n",p);
    p=p-2;
    printf("\nThe address of p1-2 is %d\n",p);
}
```

Pointers and Arrays

- Pointers and arrays are closely related.
- In particular, the name of an array is the address of the array's first element.
- For example, the declaration **int ar[3]** reserves 3 consecutive units of memory. Each unit has the same size, size of an integer (e.g., 4 bytes).



$\&\text{ar}[0] = 1000$, $\&\text{ar}[1] = 1004$,
 $\&\text{ar}[2] = 1008$.

In general, $\&\text{ar}[i] = \text{ar} + i \cdot 4$ (size of integer).

ar corresponds to the address 1000.

ar is

a pointer except that it is a constant.

//ex9.c

Pointers and arrays are interchangeable!

Consider the example :

```
main(){  
    int ar[4] = {5, 10, 15, 20};  
    int *ptr;  
    int i;  
    for (i=0; i<4; i++)  
        printf("`%d `", ar[i]);
```

```
    ptr=ar; /* Equivalent to ptr=&ar[0] */
```

```
    for (i=0; i<4; i++)  
        printf("`%d `", ptr[i]);  
}
```

Important : Because the name of an array is a constant pointer, it is illegal to make the following assignment
ar = ptr (ptr=ar is fine!)

```

#include <stdio.h> //ex10.c
int main(void){
int ar[4] = {5, 10, 15, 20};
int *ptr; int i;

for (i=0; i<4; i++) {
printf("Using dereferencing %d\n", *(ar+i)); //Print the value of each array element using
dereferencing
printf("Using array index %d\n", ar[i]);
}

//Print the values of the array by assigning ptr=ar using dereferencing
ptr=ar; /* Equivalent to ptr=&ar[0] */
for (i=0; i<4; i++){
printf("Using dereferencing %d\n", *(ptr+i));
printf("Using array index %d\n", ptr[i]);
}}

```

Limitations of Pointer Arithmetic

- `*`, `/`, and `%` cannot be used with pointers.
- `+` and `-` are restricted : only **integer offsets** can be added to or subtracted from a pointer.
 - `ptr1=ptr1+4;`
 - `ptr2=ptr2-3`
- Two pointers **cannot** be added together
 - `ptr3=ptr2+ptr2`
- A pointer `p1` can be subtracted from another pointer `p2`. The result is an integer, **the number of elements** (not bytes or addresses) between `p1` and `p2` (**offset**)
 - `p1=p2-p3`

Valid:

`p1=p1-4`

`p2=p2-3`

`p3=p1-p2`

```
//arrdiff.c
#include <stdio.h>
int main(void){
    int ar[100]; int *p1, *p2;
    p1=ar; p2=&ar[60];
    printf("p2-p1(offset): %d\n", p2 - p1);
    p1=&ar[30]; p2=&ar[80];
    printf("p2-p1(offset): %d\n", p2 - p1);
}
```

Note: If `p1` and `p2` are pointers to integers containing addresses 300 and 400 respectively, `p2 - p1` will be equal to $(400 - 300)/4 = 25$ assuming 4 bytes for the size of an integer.

Recap

- Introduction ✓
- Pointers ✓
- Manipulation of Pointers ✓
- The Special pointer NULL ✓
- Pointer arithmetic + and – ✓
- Pointers and Arrays ✓
- The pointer-to void type
- Dynamic Memory Allocation
- Passing Parameters by value and Reference
- Passing Functions as Parameters

The Pointer to Void Type

- A pointer variable ptr defined as **void * ptr** is a generic pointer variable (it can point to any type)
- Advantage : a pointer to void may be converted, without a cast operation, to a pointer to another data type.
- **void *Malloc() //Pointer to Void**

```
void *myfunction(); /* a fn that returns a generic ptr  
int n1;  
int *ptr1;  
float n2;  
float *ptr2;  
ptr2 = &n1; /* Error */  
ptr1 = &n1; /* OK */  
ptr2 = &n2; /* OK */  
ptr1 = myfunction(); // OK 'void *' is casted to 'int *' */  
ptr2 = myfunction(); // OK, 'void *' is casted to 'float *'
```

Important : Pointers to void
cannot be dereferenced

Example :

```
int n;  
void *ptr=&n;  
*ptr = 25; /* Error */
```

```
void *ptr;  
int a=200;  
ptr=&a;  
int *b;  
b=ptr;
```

```

#include <stdio.h>
//void.c
main()
{
void * ptr1;
int a=100;
ptr1=&a;
printf("\n The current address of the generic pointer ptr1 is %d\n", ptr1);

float b=20.32;

ptr1= &b;
printf("\n The current address of the generic pointer ptr1 is %d\n", ptr1);

//printf("%f",*ptr1); //generic pointers cannot be dereferenced

/*
float *temp;
temp =ptr1;
printf("%f",*temp);
*/
}

```

Dynamic Memory Allocation

There are 3 types of memory allocation in C:

Static allocation : a variable's memory is allocated and persists throughout the entire life of the program. This is the case of **global variables**.

Automatic allocation : When local variables are **declared inside a function**, the space for these variables is allocated when the function is called (starts) and is freed when the function terminates. This is also the case of parametric variables.

Dynamic allocation :

Allows a program at the execution time to allocate memory **when needed** and to **free it** when it is no longer needed. Advantage : it is often impossible to know, prior to the execution time, the size of memory needed in many cases.

For example, the size of an array based on any input size n

- Through its standard library (include `stdlib.h`), C provides functions for allocating new memory from the heap (available unused storage).
- The most commonly used functions for managing dynamic memory are :
 - **`void * malloc(int size)`** : to allocate a block (number of bytes) of memory of a given **size** and returns a pointer to this newly allocated block.
 - **`void free(void *ptr)`** : to free a previously allocated block of memory.
- Note: **`sizeof`** is an operator often used with `malloc`. It returns the size in bytes of its operand(a data type name or a variable name). For instance, `sizeof(char)=1 //so.c`

Example : ex8.c

```
main(){  
    int *ptr1;  
    float *ptr2;  
  
    ptr1 = malloc (sizeof(int)); /* allocate space for an integer */  
  
    ptr2 = malloc (sizeof(float)); /* allocate space for a float */  
  
    *ptr1 = 20;  
    *ptr2 = 13.5;  
    free(ptr1); /* free previously allocated space */  
    free(ptr2);  
}
```

Dynamic Arrays

When the size of an array is not known before the execution time, allocating arrays dynamically is a good solution.

The steps for creating a dynamic array are :

1. Declare a pointer with an appropriate base type.
2. Call malloc to allocate memory for the elements of the array. The argument of malloc is equal to the **desired size of the array multiplied by the size in bytes of each element of the array**.
3. Assign the result of malloc to the pointer variable.


```

#include <stdio.h>
#include <stdlib.h> //dynarr.c
int main(void) {
    int *ar,n;
    printf("\nEnter the number of elements in the array\n");
    scanf("%d",&n);
    ar=malloc(n*sizeof(int));

        for(int i=0;i<n;i++)
        {
            printf("Enter element %d\n", i);
            scanf("%d",&ar[i]);
        }
        printf("\nThe elements of the array are\n");
        for(int i=0;i<n;i++)
        {
            printf("\n%d",ar[i]);
        }

    free(ar);
    //End dynarr.c
}

```

Declared Array vs. Dynamic Array

Declared arrays vs. dynamic arrays

- The memory associated with a declared array is allocated automatically when the function containing the declaration is called whereas, the memory associated with a dynamic array is not allocated until malloc is called.
- The size of a declared array is a constant and must be known prior to the execution time whereas, the size of a dynamic array can be of any size and need not to be known in advance.

IMPORTANT: If the heap runs out of space, malloc will return a NULL instead of a pointer to void

```
ar = malloc(n * sizeof(float));
```

```
if (ar == NULL ) // malloc has failed take-appropriate-action and suitable message has to be displayed
```

Passing Parameters by Address (Reference)

- One of the most common application of pointers in C is their use as function parameters, making it possible for a function to get the **address of actual parameters**.
- When an argument (parameter) is passed by address to a function, the latter receives the **address** of the actual argument which is passed as a **pointer from the calling function**.
- Passing by values creates **copies of the variables** and results in wastage of space and processing power
- **Pointers can be directly dereferenced and manipulated as required**
- Passing by reference is more efficient in terms of memory and speed.

Example: Passing by Reference

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

```
//pbyref.c
```

```
void swapnum(int *i, int *j) {
```

```
    int temp = *i;
```

```
    *i = *j;
```

```
    *j = temp;
```

```
}
```

```
int main(void) {
```

```
    int a = 10;
```

```
    int b = 20;
```

```
    swapnum(&a, &b);
```

```
    printf("\nPassing by reference\n");
```

```
    printf("After swapping, a is %d and b is %d\n", a, b);
```

```
    return 0;
```

```
}
```

Passing Functions as Parameters

- C allows functions to be passed as arguments.
- How to declare a pointer to functions?

Some Examples:

- Pointer to a function with no parameters and not returning anything.
 - `void (*ptrFunc)();`
- Pointer to a function with one integer parameter and returning a value **double**
 - `double (*ptrFunc)(int)`
 - `ptrFunc=abc;`
- Pointer to a function with two parameters (int, float) and returning a value **int**
 - `int (*ptrFunc)(int,float);`

```
#include <stdio.h> //ptof.c
```

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

```
int subtract(int a,int b)
{
return a-b;
}
```

```
int main(void)
{
int (*f1)(int, int);
```

```
f1=add; //the address of the add function is assigned to pointer f1
int retvalue=f1(10,20);
```

```
printf("\nThe return value is %d\n",retvalue);
```

```
f1=subtract; //the address of the add function is assigned to pointer f1
```

```
retvalue=f1(10,20);
printf("\nThe return value is %d\n",retvalue);
}
```

```

#include <stdio.h>

int print1()
{
    printf("\nHello World from print1!\n");
    return 0;
}

int print2()
{
    printf("\nHello World from print2!\n");
    return 0;
}

void helloworld(int (*fn)(),int (*fn1>())
{
    fn();
    fn1();
}

```

```

//fap.c
int main(void)
{
    helloworld(print1,print2);
    return (0);
}

```

- `// fap9.c`

Recap and Summary

- Introduction ✓
- Pointers ✓
- Manipulation of Pointers ✓
- The Special pointer NULL ✓
- Pointer arithmetic + and – ✓
- Pointers and Arrays ✓
- The pointer-to void type ✓
- Dynamic Memory Allocation ✓
- Passing Parameters by value and Reference ✓
- Passing Functions as Parameters ✓
- Conclusion

THANK YOU