Problem 1: Array Element Access

Write a program in C that demonstrates the use of a pointer to a const array of integers. The program should do the following:

1. Define an integer array with fixed values (e.g., {1, 2, 3, 4, 5}).

2. Create a pointer to this array that uses the const qualifier to ensure that the elements cannot be modified through the pointer.

3. Implement a function printArray(const int *arr, int size) to print the elements of the array using the const pointer.

4. Attempt to modify an element of the array through the pointer (this should produce a compilation error, demonstrating the behaviour of const).

Requirements:

a. Use a pointer of type const int* to access the array.

b. The function should not modify the array elements.

```
#include<stdio.h>

void printArray(const int *arr, int size);

int main(){

    int array[5]={1,2,3,4,5};

    const int *ptr =&array[0];

    printf("Array is::\n");

    printArray(ptr,5);

    return 0;

}

void printArray(const int *arr, int size){

    for (int i=0;i<size;i++){

        printf("%d ",arr[i]);

    }

    //arr[0]=12;

    printf("\n");
}
```

Problem 2: Protecting a Value

Write a program in C that demonstrates the use of a pointer to a const integer and a const pointer to an integer. The program should:

1. Define an integer variable and initialise it with a value (e.g., int value = 10;).

2. Create a pointer to a const integer and demonstrate that the value cannot be modified through the pointer.

3. Create a const pointer to the integer and demonstrate that the pointer itself cannot be changed to point to another variable.

4. Print the value of the integer and the pointer address in each case.

Requirements:

a. Use the type qualifiers const int* and int* const appropriately.

b. Attempt to modify the value or the pointer in an invalid way to show how the compiler enforces the constraints.

```c
#include<stdio.h>
int main(){
    int a=10;
    int b=20;
    const int *ptr1=&a;
    printf("\nValue of ptr1::%d",*ptr1);
    //*ptr1=30;
    int *const ptr2=&b;

    printf("\nValue of ptr2::%d",*ptr2);
    //ptr2=&a;
    return 0;


}
```

3.Length of String without strlen or sizeof operators.

```c
#include<stdio.h>
int main(void){
    char str1[]="Hello";
    char str2[]="HelloWorld";
    int count=0;
    while(str1[count] != '\0'){
        count++;
    }
    printf("Length of String1 is::%d",count);
    count=0;
    while(str2[count] != '\0'){
        count++;
    }
    printf("\nLength of String2 is::%d",count);

}
```

4.Problem: Universal Data Printer
You are tasked with creating a universal data printing function in C that can handle different types of data (int, float, and char*). The function should use void pointers to accept any type of data and print it appropriately based on a provided type specifier.
Specifications
Implement a function print_data with the following signature:
    void print_data(void* data, char type);

Parameters:

data: A void* pointer that points to the data to be printed.

type: A character indicating the type of data:
    'i' for int
    'f' for float
    's' for char* (string)

Behaviour:
    If type is 'i', interpret data as a pointer to int and print the integer.
    If type is 'f', interpret data as a pointer to float and print the floating-point value.
    If type is 's', interpret data as a pointer to a char* and print the string.

In the main function:
    Declare variables of types int, float, and char*.
    Call print_data with these variables using the appropriate type specifier.

Example output:
Input data: 42 (int), 3.14 (float), "Hello, world!" (string)
Output:
Integer: 42
Float: 3.14
String: Hello, world!

Constraints

1. Use void* to handle the input data.
2. Ensure that type casting from void* to the correct type is performed within the print_data function.
3. Print an error message if an unsupported type specifier is passed (e.g., 'x').

```
#include<stdio.h>
void print_data(void* data, char type);
int main() {
    int i;
    float f;
    char s[100];
    printf("Enter an integer , float,string(press enter each time)::");
    scanf("%d", &i);
    scanf("%f", &f);
    scanf("%s", s);
    print_data(&i, 'i');
    print_data(&f, 'f');
    print_data(s, 's');

    return 0;
}
void print_data(void* data, char type) {
    switch (type) {
        case 'i':
            printf("Integer: %d\n", *(int*)data);
            break;
        case 'f':
            printf("Float: %.2f\n", *(float*)data);
            break;
        case 's':
            printf("String: %s\n", (char*)data);
            break;
        default:
            printf("INVALID");
            break;
    }
}
```

5.Concat Strings.

```
#include<stdio.h>
void concat(char result[], const char str1[], const char str2[]);
int main(void) {
    char result[100];
    const char str1[] = "Hello";
    const char str2[] = "World";
    concat(result, str1, str2);
    printf("Concatenated String is: %s", result);
    return 0;
}
void concat(char result[], const char str1[], const char str2[]) {
    int i = 0, j = 0;
    while (str1[i] != '\0') {
        result[i] = str1[i];
        i++;
    }
    while (str2[j] != '\0') {
        result[i] = str2[j];
        i++;
        j++;
    }
    result[i] = '\0';
}
```

6.Check Strings are equal.

```
#include<stdio.h>
int equal(const char str1[], const char str2[]);
int main(void) {
    const char str1[] = "Hello";
    const char str2[] = "Hello";
    if (equal(str1, str2)) {
        printf("str1&str2 are equal.\n");
    } else {
        printf("str1&str2 are not equal.\n");
    }
    return 0;
}
int equal(const char str1[], const char str2[]) {
    int i = 0;
    while (str1[i] != '\0' || str2[i] != '\0') {
        if (str1[i] != str2[i]) {
            return 0;
        }
        i++;
    }

    return 1;
}
```