Constant (keyword const)

- const is a keyword applied on a variable
- This is to tell the compiler that the value will not be changed throughout the program.

Eg1:
```
int main()
{
    const int num = 10;
    num = 20; //error
    return 0;
}
```

Both const int num and int const num are the same. Read it as num is an integer constant.

Eg2:
```
int main()
{
    int num = 10;
    const int *ptr = &num;
    *ptr = 10 //error
    (*ptr)++; //error
    return 0;
}
```

Both const int *ptr and int const *ptr are the same. Ptr is the pointer to a constant integer. Changing *ptr is not allowed.

Eg3:
```
int main()
{
    int num1 = 10, num2 = 20;
    int * const ptr = &num1;
    ptr = &num2; //error
    ptr++; //error
    return 0;
}
```

ptr is a constant pointer to an integer. Changing ptr is not allowed.

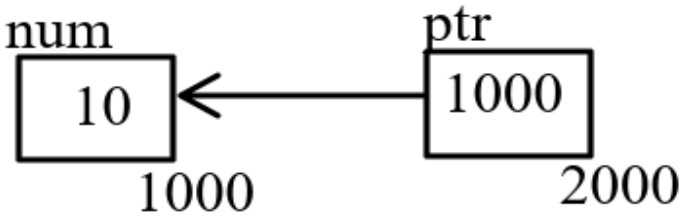
Eg4:
```
int main()
{
    int num1 = 10, num2 = 20;
    const int *const ptr = &num1;
```

```
        ptr++; //error
        (*ptr)++; //error
        ptr = &num2; //error
        *ptr = 20 ; //error
        return 0;
}
```

ptr is a constant pointer to a constant integer. Changing *ptr and ptr are not allowed.

Dos and Don'ts of pointers

- Eg:
```
int main()
{
        int num = 10;
        int *ptr = &num;
        return 0;
}
```



- Adding, subtracting the pointer with a constant is allowed.
  ptr = ptr + 1; // ptr + 1 * sizeof(int)
  ptr = ptr - 1; // ptr - 1 * sizeof(int)
- Multiplying and dividing the pointer with a constant is an error
  ptr = ptr * 1; //error
  ptr = ptr / 1; //error
- Subtracting a pointer with a pointer is allowed.
  ptr = ptr - ptr;
- Adding, multiplying and dividing a pointer with another pointer is an error.
  ptr = ptr + ptr; //error
  ptr = ptr * ptr; //error
  ptr = ptr / ptr; //error
- Bitwise operators are not allowed

- Logical operators are allowed

Pitfalls of Pointers

1. Segmentation fault:

   This is caused whenever we are accessing the address that is not allowed to be accessed.

   Eg:
```
int main()
{
    int *ptr = NULL;
    printf("%d\n", *ptr); // seg fault as accessing address 0 is not allowed
    return 0;
}
```
2. Dangling pointer:

   Pointer pointing to a freed location is called a dangling pointer.

   Eg:
```
int main()
{
    int *ptr = calloc(5, 1);
    free(ptr);
    printf("%d\n", *ptr); //ptr is a dangling pointer
    return 0;
}
```
   ptr is still having the address which it was pointing to. But the address is no longer with the user as it has already been freed. Dereferencing the dangling pointer leads to undefined behaviour.
3. Wild pointer:

   Uninitialized pointers which are pointing to some random address are called wild pointers. Dereferencing wild pointers also leads to undefined behaviour. Good practice is to initialise it with NULL when the pointer is being declared.

   Eg:
```
int main()
{
```

```
        int *ptr; //wild pointer
        static int *ptr1; //not a wild pointer because static
pointers are initialised with NULL by default.
        return 0;
    }
```

4. Memory leak:

    Failing to free the dynamically allocated memory leads to memory leak.

    Eg:
```
int main()
{
    int *ptr;
    while(1)
    {
        ptr = malloc(100);
        //forgot to free the memory
    }
    return 0;
}
```

5. Bus error:

    This error is caused when the CPU cannot handle the address in the address bus.

    Eg:
```
int main()
{
    char array[4];
    int *ptr = &array[1];
    scanf("%d\n", ptr); //bus error
    return 0;
}
```

    Usually the CPU can handle the address which is a multiple of 4. Trying to read an integer to the address which is not a multiple of 4 leads to bus error.