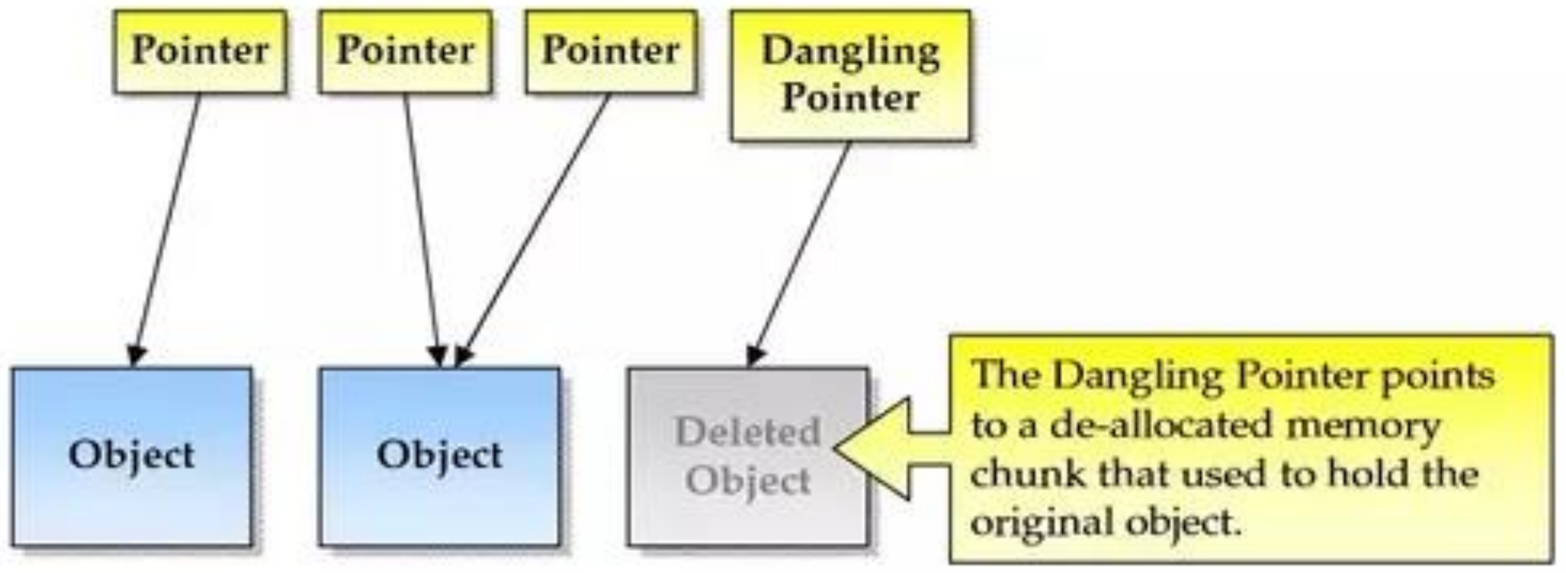
18ES611 Embedded System Programming

Sarath tv

Dangling pointers

- Daggling pointers arise **when the referencing object is deleted or deallocated, without changing the value of the pointers**.
- It creates the problem because the pointer is still pointing the memory that is not available. When the user tries to dereference the daggling pointers than it shows the undefined behavior and can be the cause of the segmentation fault.
- Dangling pointer is a pointer that not pointing a valid object of the appropriate type and it can be the cause of the undefined behavior.
- **Important causes of the dangling pointers in C language**
  - *When variable goes out of the scope*
  - *After destroying the stack frame*
  - *Deleting the memory explicitly*


Pointer
Pointer
Pointer
Dangling Pointer
Object
Object
Deleted Object
The Dangling Pointer points to a de-allocated memory chunk that used to hold the original object.

When variable goes out of the scope

A local variable's scope and lifetime belong to their block where it is declared. Whenever control comes to the block than memory is allocated to the local variable and freed automatically upon exit from the block.

If a local variable is referred to outside of its lifetime, the behavior is undefined. **The value of a pointer becomes indeterminate** when the variable it points to reaches the end of its lifetime.

```
#include <stdio.h>
int main(void)
{
int * piData;
{   //block
int Data = 27;
piData = &Data;
}
printf("piData = %d\n", *piData);
//piData is //dangling pointer
return 0;
}
```

After destroying the stack frame

The stack frame that is allocated to a function is destroyed after returning the control from the function. The common mistake performed by the developer is that to return the address of the stack allocated variable from the function. If you tried to access the returning address from the pointer, **you will get the unpredictable result or might get the same value** but it is very dangerous and need to avoid

```
#include<stdio.h>
int *Fun()
{
int Data = 5; //Local variable
return &Data; //Address of local
variable
}
int main()
{
int *piData = Fun(); //Returning
address of the local variable
printf("%d", *piData);
return 0;
}
```

Data has no scope beyond the function. If you try to read the value of Data after calling the Fun() using the pointer you may get the correct value (5), but any functions called thereafter will overwrite the stack storage allocated for Data with other values and the pointer would no longer work correctly.

Deleting the memory explicitly

The *compiler handles static and auto allocated memory* but if the user allocates the memory manually than it is the responsibility of the user to free the manually allocated memory.

In the C language malloc, calloc and realloc library function are used to allocate the memory at runtime and free is used to deallocate the allocated memory. **The jumbled combination of malloc (calloc, realloc) and free is born the dangling pointers.**

A problem is invoking, whenever programmer has freed the allocated memory and try to access the freed memory. You will be lucky if you will not get the segmentation fault after using the freed memory. It is a very common mistake that is generally done by the developers.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *piData = NULL;
piData = malloc(sizeof(int)* 10);
//creating integer of size 10.
free(piData); //free the allocated
memory
*piData = 10; //piData is dangling
pointer
return 0;
}
```

How to avoid the dangling pointers errors

- The behavior of the dangling pointer is undefined, so it is very important to avoid the born of dangling pointers. *The common mistake that is done by many programmers is that not assigning the NULL explicitly after freeing the dynamic memory*. So It is very good habits to assign the NULL after deallocation of the dynamically allocated memory.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
char *pcData = NULL;
pcData = malloc(sizeof(char)* 10); //creating integer
of size 10.
free(pcData); /* piData can be becomes a dangling
pointer */
pcData = NULL; //piData is no longer dangling pointer
return 0;
}
```

- Apart from that, *another mistake is to return the address of the local variable (stack variable) from the function*, it is also a cause to create a dangling pointer. **Using the static variable** we can resolve the problem because the lifetime of the static variable is entire run of the program.

```c
#include<stdio.h>
int *foo()
{
static int Data = 6;
return &Data;
}
int main()
{
int *piData = NULL;
piData = foo();
// Now piData is Not a dangling pointer as it points
// to static variable.
printf("%d",*piData);
return 0;
}
```

What is the wild pointer?

- *A pointer that is not initialized properly prior to its first use is known as the wild pointer.* Uninitialized pointers behavior is totally undefined because it may point some arbitrary location that can be the cause of the program crash, that’s is the reason it is called a wild pointer.
- int *piData; //piData is wild pointer
- an integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.
- int *piData = NULL; *// piData is a null pointer*

What is void or Generic pointers in C?

- A void pointer is a generic pointer, it has no associated data type. It can store the address of any type of object and it **can be type-casted to any types**.
- According to C standard, the pointer to void shall have the same representation and alignment requirements as a pointer to a character type.
- A void pointer declaration is similar to the normal pointer, but the difference is that instead of data types we use the void keyword.

```
void *vptr;
```

Type casting

- Way to convert a variable from one data type to another data type
- It is best practice **to convert lower data type to higher data type to avoid data loss.**
- Data will be truncated when the higher data type is converted to lower. For example, if a float is converted to int, data which is present after the decimal point will be lost

IMPLICIT CONVERSION

- Implicit conversions do not require any operator for converted. They are **automatically** performed when a value is copied to a compatible type in the program.

```
#include<stdio.h>
#include<conio.h>
 void main()
    {
         int i=20;
         double p;
      clrscr();
       p=i; // implicit conversion
         printf("implicit value is %d",p);
         getch();
    }
```

```
#include<stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}
```

EXPLICIT CONVERSION

Explicit Type Conversion– This process is user defined. Here the **user can type cast the result to make it of a particular data type.**

The syntax in C:

(type) expression

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a,b;
  float c,d;
  clrscr();
  printf("Enter 2 numbers : ");
  scanf("%d%d",&a,&b);
  c=a/b;
  d=(float)a/b;
  printf("c=%.2f",c);
  printf("\nd=%.2f",d);
  getch();
}
```

THANK YOU!!!!!