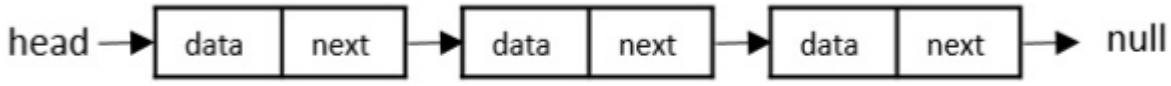
Self-referential Structures in C

What are Self-referential Structures?

A self-referential structure is a struct data type in C, where one or more of its elements are pointer to variables of its own type. Self-referential user-defined types are of immense use in C programming. They are extensively used to build complex and dynamic data structures such as linked lists and trees.

In C programming, an array is allocated the required memory at compile-time and the array size cannot be modified during the runtime. Self-referential structures let you emulate the arrays by handling the size dynamically.



File management systems in Operating Systems are built upon dynamically constructed tree structures, which are manipulated by self-referential structures. Self-referential structures are also employed in many complex algorithms.

Defining a Self-referential Structure

A general syntax of defining a self-referential structure is as follows –

```
strut typename{
   type var1;
   type var2;
   ...
   ...
   struct typename *var3;
}
```

Let us understand how a self-referential structure is used, with the help of the following **example**. We define a struct type called **mystruct**. It has an integer element "**a**" and "**b**" is the pointer to **mystruct** type itself.

We declare three variables of **mystruct** type –

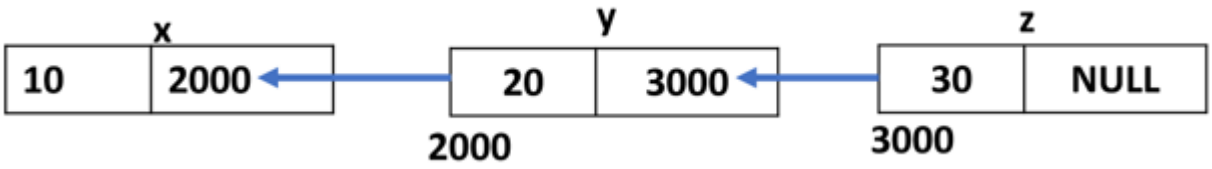
```
struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};
```

Next, we declare three "mystruct" pointers and assign the references **x**, **y** and **z** to them.

```
struct mystruct * p1, *p2, *p3;

p1 = &x;
p2 = &y;
p3 = &z;
```

The variables "x", "y" and "z" are unrelated as they will be located at random locations, unlike the array where all its elements are in adjacent locations.



Examples of Self-referential Structure

Example 1

To explicitly establish a link between the three variable, we can store the address of "y" in "x" and the address of "z" in "y". Let us implement this in the following program –

</> Open Compiler

```
#include <stdio.h>

struct mystruct{
   int a;
   struct mystruct *b;
};

int main(){

   struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};
   struct mystruct * p1, *p2, *p3;

   p1 = &x;
   p2 = &y;
   p3 = &z;
```

```
   x.b = p2;
   y.b = p3;

   printf("Address of x: %d a: %d Address of next: %d\n", p1, x.a, x.b);
   printf("Address of y: %d a: %d Address of next: %d\n", p2, y.a, y.b);
   printf("Address of z: %d a: %d Address of next: %d\n", p3, z.a, z.b);

   return 0;
}
```

Output

Run the code and check its output –

```
Address of x: 659042000 a: 10 Address of next: 659042016
Address of y: 659042016 a: 20 Address of next: 659042032
Address of z: 659042032 a: 30 Address of next: 0
```

Example 2

Let us refine the above program further. Instead of declaring variables and then storing their address in pointers, we shall use the malloc() function to dynamically allocate memory whose address is stored in pointer variables. We then establish links between the three nodes as shown below –

</> Open Compiler

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

struct mystruct{
   int a;
   struct mystruct *b;
};

int main(){

   struct mystruct *p1, *p2, *p3;

   p1 = (struct mystruct *)malloc(sizeof(struct mystruct));
```

```
   p2 = (struct mystruct *)malloc(sizeof(struct mystruct));
   p3 = (struct mystruct *)malloc(sizeof(struct mystruct));

   p1 -> a = 10; p1->b=NULL;
   p2 -> a = 20; p2->b=NULL;
   p3 -> a =30; p3->b=NULL;

   p1 -> b = p2;
   p2 -> b = p3;

   printf("Add of x: %d a: %d add of next: %d\n", p1, p1->a, p1->b);
   printf("add of y: %d a: %d add of next: %d\n", p2, p2->a, p2->b);
   printf("add of z: %d a: %d add of next: %d\n", p3, p3->a, p3->b);

   return 0;
}
```

Output

Run the code and check its output –

```
Add of x: 10032160 a: 10 add of next: 10032192
add of y: 10032192 a: 20 add of next: 10032224
add of z: 10032224 a: 30 add of next: 0
```

Example 3

We can reach the next element in the link from its address stored in the earlier element, as "p1 → b" points to the address of "p2". We can use a **while** loop to display the linked list, as shown in this example –

</> Open Compiler

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

struct mystruct{
   int a;
   struct mystruct *b;
};
```

```c
int main(){

   struct mystruct *p1, *p2, *p3;

   p1=(struct mystruct *)malloc(sizeof(struct mystruct));
   p2=(struct mystruct *)malloc(sizeof(struct mystruct));
   p3=(struct mystruct *)malloc(sizeof(struct mystruct));

   p1 -> a = 10; p1 -> b = NULL;
   p2 -> a = 20; p2 -> b = NULL;
   p3 -> a = 30; p3 -> b = NULL;

   p1 -> b = p2;
   p2 -> b = p3;

   while (p1 != NULL){
      printf("Add of current: %d a: %d add of next: %d\n", p1, p1->a, p1->b);
      p1 = p1 -> b;
   }

   return 0;
}
```

Output

Run the code and check its output –

```
Add of current: 10032160 a: 10 add of next: 10032192
Add of current: 10032192 a: 20 add of next: 10032224
Add of current: 10032224 a: 30 add of next: 0
```

Creating a Linked List with Self-referential Structure

In the above examples, the dynamically constructed list has three discrete elements linked with pointers. We can use a **for** loop to set up required number of elements by allocating memory dynamically, and store the address of next element in the previous node.

Example

The following example shows how you can create a linked list using a self-referential structure –

</> Open Compiler

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

struct mystruct{
   int a;
   struct mystruct *b;
};

int main(){

   struct mystruct *p1, *p2, *start;
   int i;

   p1 = (struct mystruct *)malloc(sizeof(struct mystruct));
   p1 -> a = 10; p1 -> b = NULL;

   start = p1;

   for(i = 1; i <= 5; i++){
      p2 = (struct mystruct *)malloc(sizeof(struct mystruct));
      p2 -> a = i*2;
      p2 -> b = NULL;
      p1 -> b = p2;
      p1 = p2;
   }

   p1 = start;
   while(p1 != NULL){
      printf("Add of current: %d a: %d add of next: %d\n", p1, p1 -> a, p1 ->
b);
      p1 = p1 -> b;
   }

   return 0;
}
```

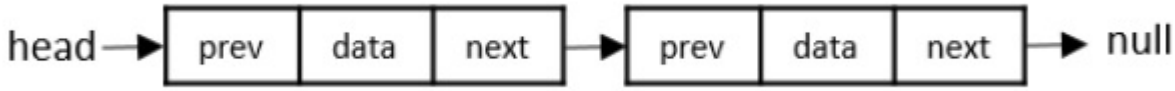
Output

Run the code and check its output –

```
Add of current: 11408416 a: 10 add of next: 11408448
Add of current: 11408448 a: 2 add of next: 11408480
Add of current: 11408480 a: 4 add of next: 11408512
Add of current: 11408512 a: 6 add of next: 11408544
Add of current: 11408544 a: 8 add of next: 11408576
Add of current: 11408576 a: 10 add of next: 0
```

Creating a Doubly Linked List with Self-referential Structure

A linked list is traversed from beginning till it reaches NULL. You can also construct a doubly linked list, where the structure has two pointers, each referring to the address of previous and next element.

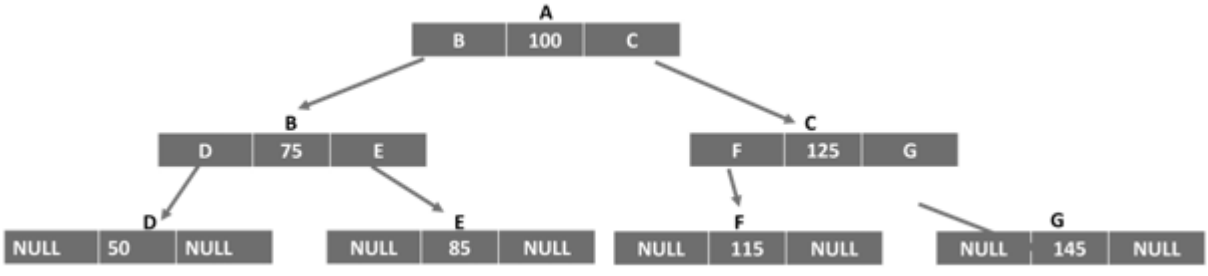


The struct definition for this purpose should be as below –

```
struct node {
   int data;
   int key;
   struct node *next;
   struct node *prev;
};
```

Creating a Tree with Self-referential Structure

Self-referential structures are also used to construct non-linear data structures such as **trees**. A binary search tree is logically represented by the following figure –



The struct definition for the implementing a tree is as follows –

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

To learn these complex data structure in detail, you can visit the DSA tutorial – Data Structures Algorithms