Union

allows to store different data types in the same memory location.

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

```
union Data {
   int i;
   float f;
   char str[20];
};
                    int main( ) {

                       union Data data;

                       printf( "Memory size occupied by data : %d\n", sizeof(data));

                       return 0;
                    }
```

```
int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

Pointers

- Pointers are the variables that are used to store the location of value present in the memory.
- A pointer to a location stores its memory address.
- The process of obtaining the value stored at a location being referenced by a pointer is known as dereferencing.
- It is the same as the index for a textbook where each page is referred by its page number present in the index.
- One can easily find the page using the location referred to there.
- Such pointers usage helps in the dynamic implementation of various data structures such as stack or list.

Why are Pointers Used ?

- To return more than one value from a function
- To pass arrays & strings more conveniently from one function o another
- To manipulate arrays more easily by moving pointers to them, Instead of moving the arrays themselves
- To allocate memory and access it (Dynamic Memory Allocation)
- To create complex data structures such as Linked List, Where one data structure must contain references to other data structures

int *ptr1 // ptr1 references to a memory location that holds data of int datatype.

```
int var = 30;
int *ptr1 = &var; // pointer to var
*ptr1=50;
int **ptr2 = & ptr1; // pointer to pointer variable ptr1
print(“%d”, *ptr1)  // prints 30
print(“%d”,**ptr2)  // prints 30
```

```
void pointerDemo( ) {
int var1 = 30;                    int *ptr1;          int **ptr2;        ptr1 = &var1;
ptr2 = &ptr1;
printf("Value at ptr1 = %p \n",ptr1);
printf("Value at var1 = %d \n",var1);
printf("Value of variable using *ptr1 = %d \n", *ptr1);
printf("Value at ptr2 = %p \n",ptr2);
printf("Value stored at *ptr2 = %d \n", *ptr2);
printf("Value of variable using  **ptr2 = %d \n", **ptr2);    }
```

NULL Pointer: Such type of pointer is used to indicate that this points to an invalid object. This type of pointer is often used to represent various conditions such as the end of a list.

VOID Pointer: This type of pointer can be used to point to the address of any type of variable but the only limitation is that it cannot be dereferenced easily.

Dangling Pointer: The type of pointers that don’t refer to a valid object and are not specifically initialized to point a particular memory. For ex: int *ptr1 = malloc(sizeof(char))

Function pointer: This is a type of pointer to reference an executable code. It is mostly used in the case of the recursive procedure to holds the address of the code that needs to be executed later.

```
#include <stdio.h>

int main() {

int *p= NULL;

printf("The value of pointer is

%u",p);

  return 0;

}
```

a) To initialize a pointer variable when that pointer variable isn’t assigned any valid memory address yet.

b) To pass a null pointer to a function argument when we don’t want to pass any valid memory address.

c) To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it’s not NULL.

```
int main() {
  int a = 7;
  float b = 7.6;
  void *p;
  p = &a;
  printf("Integer variable is = %d", *( (int*) p) );
  p = &b;
  printf("\nFloat variable is = %f", *( (float*) p) );
  return 0; }
```

function_return_type(*Pointer_name)(function argument list)

```
int subtraction (int a, int b) {
   return a-b;  }
int main() {
   int (*fp) (int, int)=subtraction;
   //Calling function using function pointer
   int result = fp(5, 4);
   printf(" Using function pointer we get the result: %d",result);
   return 0; }
```

```
int main()
{
    char **strPtr;
    char *str = "Hello!";
    strPtr = &str;
    free(str);
    //strPtr now becomes a dangling pointer
    printf("%s", *strPtr);
}
```

Pointers used in Arithmetic Operations

```
int main() {
   int val = 28;
   int *pt;
   pt = &val;
   printf("Address of pointer : %u\n",pt);
   pt = pt + 5;
   printf("Addition to pointer : %u\n",pt);
   pt = pt - 5;
   printf("Subtraction from pointer : %u\n",pt);
   pt = pt + 1;
   printf("Increment to pointer : %u\n",pt);
   pt = pt - 1;
   printf("Decrement to pointer : %u\n",pt);
   return 0; }
```

MEMORY ALLOCATION

- The blocks of information in a memory system is called **memory allocation.**
- To allocate memory it is necessary to keep in information of available memory in the system. If memory management system finds sufficient free memory, it allocates only as much memory as needed, keeping the rest available to satisfy future request.
- In memory allocation has two types. They are **static and dynamic**
  - memory allocation.

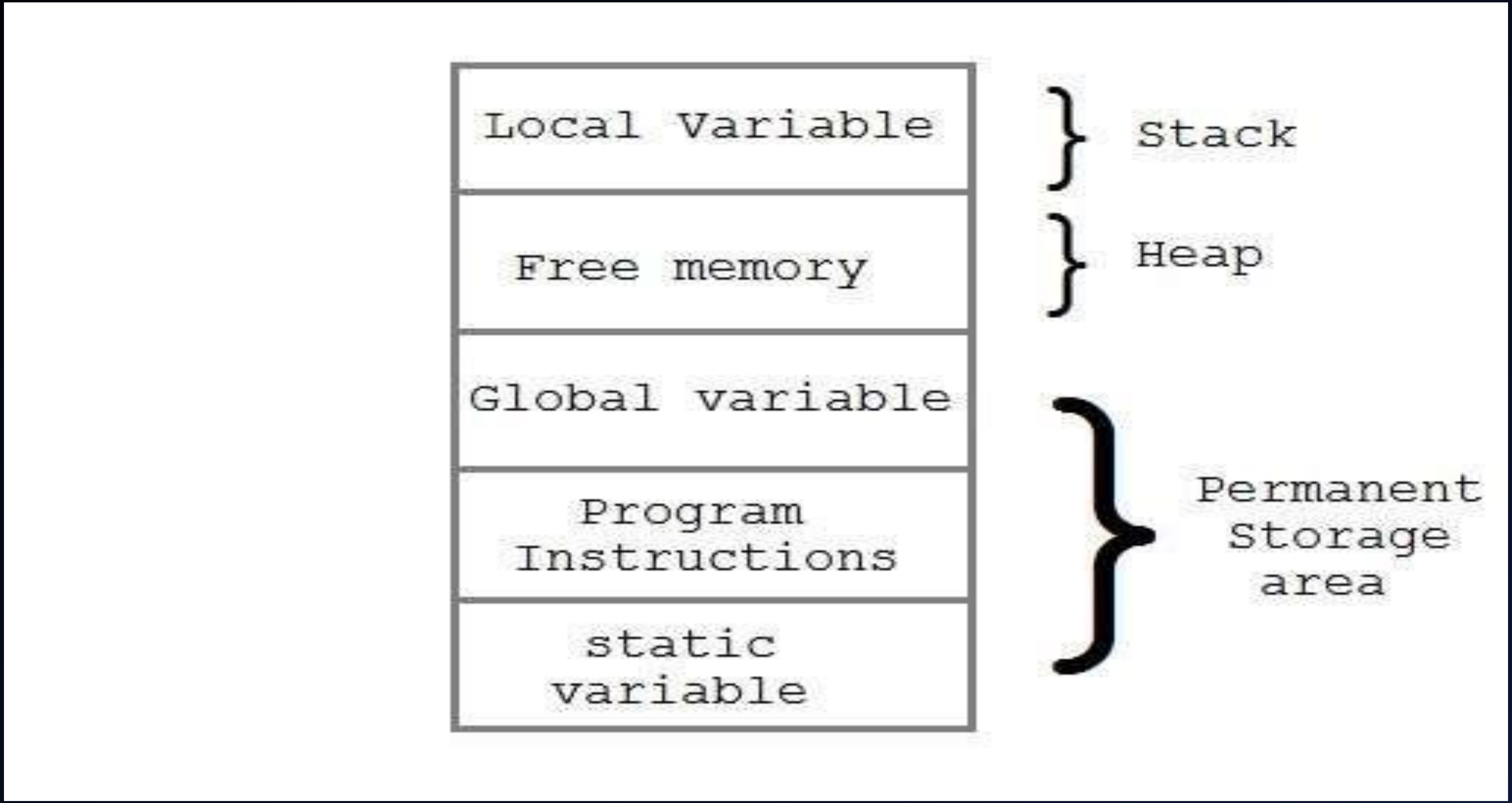
STATIC MEMORY ALLOCATION

- In static memory allocation, size of the memory may be required for the that must be define before loading and executing the program.

DYNAMIC MEMORY ALLOCATION

- In the dynamic memory allocation, the memory is allocated to a variable or program at the run time.
- The only way to access this dynamically allocated memory is through pointer.

MEMORY ALLOCATION PROCESS



Malloc:

1.The name **malloc** stands for *memory allocation*.

2.**malloc()** takes one argument that is, *number of bytes*.

Calloc

1.The name **Calloc** stands for *contiguous allocation*.

2.calloc() take two arguments those are: *number of blocks* and *size of each block*.

Function:

calloc allocates a region of memory large enough to hold "n elements" of "size" bytes each. Also initializes contents of memory to zeroes.

Malloc allocates "size" bytes of memory.

Number of arguments:

2 (calloc)

1 (malloc)

Syntax: void *calloc (number_of_blocks, size_of_each_block_in_bytes);

void *malloc (size_in_bytes);

Contents of allocated memory:

Calloc allocated region is initialized to zero.

In malloc the contents of allocated memory are not changed. i.e., the memory contains unpredictable or garbage values. This presents a risk.

Speed of execution:

calloc is a tiny bit slower than malloc because of the extra step of initializing the memory region allocated. However, in practice the difference in speed is very small and can be ignored.

ALLOCATION A BLOCK OF MEMORY : MALLOC

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void.

Ptr=(cast-type*) **malloc** (byte-size);

ptr = (float*) malloc(100 * sizeof(float));

Example 2 : char A=(char*) malloc(l O*20*sizeof(char));

Note : If the memory allocation is success it returns the starting address else if returns NULL

EXAMPLE PROGRAM

```
#include <stdio.h>  #include <stdlib.h>
struct emp
{
int eno;  char name;  float esal; };
 void main()
{
struct emp *ptr;
ptr = (struct emp *) malloc(sizeof(struct emp));
if(ptr == null){
pintf(“out of memory”); }
else
{
printf(“Enter the emp deitals”);
scanf(“%d%s%f,&ptr-> eno,ptr-> name,&ptr-> esal”);  return 0;
}
```

Output:

```
Enter the emp details  eno 1
ename priya  esal
10,000
```

ALLOCATION A BLOCK OF MEMORY : CALLOC

calloc() is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**.

Ptr=(cast-type*)**calloc**(n,elem-size);

Function calloc() used to allocate multiple blocks of contiguous memory in bytes. All the blocks are of same size.

```
Ptr =(int*) calloc(5,10);
```

EXAMPLE PROGRAM

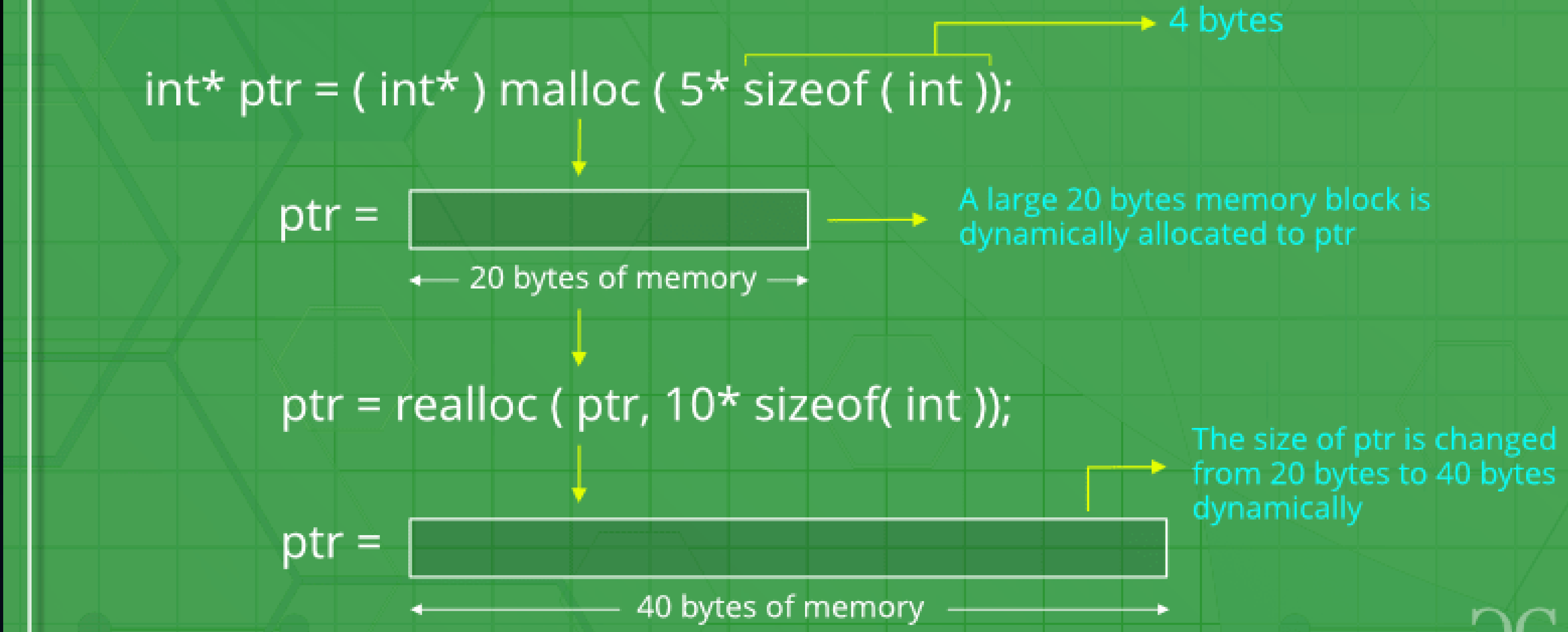
```
#include <stdio.h>  #include
<stdlib.h>  int main()
{
int i, n;
 int *a;
 printf("Number of elements to be entered:");
 scanf("%d",&n);
 a = (int*)calloc(n, sizeof(int));
 printf("Enter %d numbers:\n",n);
for( i=0 ; i < n ; i++ )
{
scanf("%d",&a[i]);
}
printf("The numbers entered ar
for( i=0 ; i < n ; i++ )
{
printf("%d ",a[i]);
}
free( a );  return(0);}
```

ALTERING THE SIZE OF A BLOCK : REALLOC

realloc() changes memory size that is already allocated dynamically to a variable.

ptr=**REALLOC**(ptr,new size);

Realloc()


4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
ptr =
The size of ptr is changed from 20 bytes to 40 bytes dynamically
40 bytes of memory

EXAMPLE PROGRAM

```
#include <stdio.h>  #include <stdlib.h>
int main()
{
int *ptr = (int *)malloc(sizeof(int)*2);
int i;
int *ptr_new;                *ptr = 10;
*(ptr + 1) = 20;
ptr_new = (int *)realloc(ptr, sizeof(int)*3);
*(ptr_new + 2) = 30;
for(i = 0; i < 3; i++)
printf("%d ", *(ptr_new + i));
return 0;   }
```

Output:

```
10 20 30
```

RELEASING THE USED SPACE: FREE

Free() function should be called on a pointer that was used either with ”calloc()” or “malloc()”,otherwise the function will destroy the memory management making a system to crash.

free (ptr)

```
Int main()
{
Int *p;
*p=12;
Printf("%d", *p);
```

```
Int main()
{
int a=12;
Int *p= &a;
Printf("%d", *p)
*p=20;
Printf("%d", *p);
```