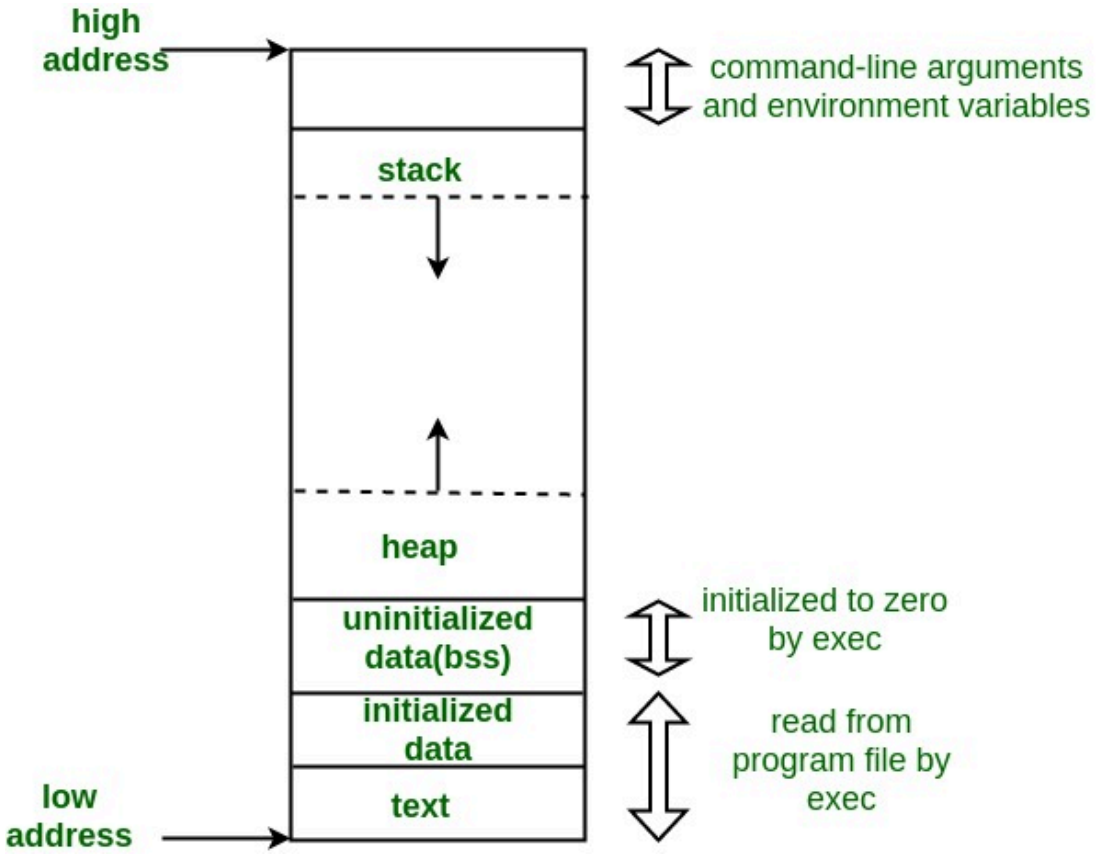
# Unit 1
# Pointers & Dynamic memory allocation

Memory Layout of C Programs




Ref: https://www.geeksforgeeks.org/memory-layout-of-c-program/

**1.1 Generic pointers**

```
/*
A generic pointer can point to any type of data variable.
But it can not be dereferenced to  access the value at the destination.
Generic pointers must be type-casted (E.g. (int*)gp )to any specific type for dereferencing.
Ref: https://www.geeksforgeeks.org/void-pointer-c-cpp/  */

#include <stdio.h>
int main()
{
   int x=10;
   char ch = 'A';
   void *gp;
   gp = &x;
```

```
    printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
    gp = &ch;
    printf("\n Generic pointer now points to the character= %c", *(char*)gp);
    return 0;
}
```

**1.2 Application of generic pointers**

```
/* Different types of data can be passed to a function using same argument with the help of
Generic pointers. Handling of each type of data can be done separately in the called function
With the help of an additional argument that indicates the type of data passed.
Ref: https://www.geeksforgeeks.org/void-pointer-c-cpp/   */
#include <stdio.h>
void process_data(void *ptr, int type);

int main()
{
    int x=10;
    char ch = 'A';
    float pi = 3.14;

    process_data(&x,1); // Passing integer by reference
    process_data(&ch,2);  // Passing character by reference
    process_data(&pi,3);  //  // Passing float by reference
    return 0;
}

void process_data(void *ptr, int dtype)
{
    switch(dtype)
    {
    case 1:
        {
        printf("Integer type data. Type-cast to integer pointer for dereferencing.\n");
        int d = *(int*)ptr;
        printf("Data is %d\n", d);
        break;
        }
    case 2:
        {
        printf("Character type data. Type-cast character pointer for dereferencing.\n");
        char d = *(char*)ptr;
        printf("Data is %c\n", d);
        break;
        }
```

```
    case 3:
        {
        printf("Float type data. Type-cast float pointer for dereferencing.\n");
        float d = *(float*)ptr;
        printf("Data is %f\n", d);
        break;
        }
    }
}
```

**1.3 Call by reference using pointers**

```
/*
Call by reference using pointers: Passing address to a function.
```
**Ref:** https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/
```
*/
#include <stdio.h>
void square(int *ptr);

int main()
{
    int x=10;
    // Call by value
    square1(x);
    printf("After square1() call: x = %d\n",x);

    // Call by reference
    square2(&x);
    printf("After square2() call: x = %d\n",x);
    return 0;
}

void square1(int d)
{
    d = d * d;
}

void square2(int *px)
{
    *px = *px * *px;
}
```

**1.4 Pointer to a structure**

```
/* Pointer to a structure variable, and accessing
members through the pointer.
Ref:  https://www.geeksforgeeks.org/structure-pointer-in-c/
  */
#include<stdio.h>
struct stud{
    char name[20];
    int usn;
    float cgpa;
};

int main()
{
    struct stud s = {"Amit",100,8.5};
    struct stud *sptr = &s;

    // Print using structure variable
    printf("Name: %s   USN:%d  %.2f\n", s.name,s.usn,s.cgpa);

    // Print using structure pointer
    printf("Name: %s   USN:%d  %.2f\n", sptr->name,sptr->usn,sptr->cgpa);
    return  0;
}
```

**1.5 Call by reference using structure**

```
/* Pointer to a structure variable, and accessing
members through the pointer.
Ref:  https://www.programiz.com/c-programming/c-structure-function   */
#include<stdio.h>
struct stud{
    char name[20];
    int usn;
    float cgpa;
};

// Defining alternate name for student using typedef
typedef struct stud  student;

void process_data(student *sptr);

int main()
{
    student s = {"Amit",100,8.5};
```

```
    process_data(&s);  // Call by reference
    return  0;
}

void process_data(student *sptr)
{
    // Print using structure pointer
    printf("Name: %s   USN:%d  %.2f\n", sptr->name, sptr->usn, sptr->cgpa);
}
```

**1.6 Dynamic memory allocation for 1 integer**

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

int main()
{
int *px;
// Dynamically allocating memory for 1 integer
px = (int *)malloc(sizeof(int));

*px = 100; // Storing data in allocated memory
printf("Values is %d Address allocated block is: %x\n",*px, px);

// Free the allocated memory??

return 0;
}
```

**1.7 Dynamic memory allocation for an array of integers using malloc().**

```
/*
Ref:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
 */

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

int main()
{

        // This pointer will hold the
```

```
        // base address of the block created
        int* ptr;
        int n, i;

        // Get the number of elements for the array
        printf("Enter number of elements:");
        scanf("%d",&n);
        printf("Entered number of elements: %d\n", n);

        // Dynamically allocate memory using malloc()
        ptr = (int*)malloc(n * sizeof(int));

        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }

    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");

    // Store the elements in the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 10;
        // *(ptr+i) = i  + 10;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
        // printf("%d, ", *(ptr+i));
    }

    // Freeing the memory
    free(ptr);
return 0;
}
```

**1.8 Dynamic memory allocation for an array of integers using calloc().**

```
/*
```

Ref:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
*/

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

int main()
{
        int* ptr;
        int n, i;

        printf("Enter number of elements:");
        scanf("%d",&n);
        printf("Entered number of elements: %d\n", n);

        // Dynamically allocate memory using malloc()
        ptr = (int*)calloc(n, sizeof(int));

        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }

    // Store the elements in the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 10;
        // *(ptr+i) = i  + 10;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
        // printf("%d, ", *(ptr+i));
    }

    // Freeing the memory
    free(ptr);
        return 0;
}
```

**9. Demonstration of realloc() function.**

```
/*
Dynamic memory allocation for an array of integers
Ref:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
 */

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

int main()
{
        int* ptr;
        int n, i;

        printf("Enter number of elements:");
        scanf("%d",&n);
        printf("Entered number of elements: %d\n", n);

        // Dynamically allocate memory using malloc()
        ptr = (int*)malloc(n*sizeof(int));

    // Store the elements in the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 10;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d ", ptr[i]);
        // printf("%d ", *(ptr+i));
    }

    // Enlarge array to add 5 more elements
    int  n1 = n + 5;
    int *ptr2 = (int*)realloc(ptr, n1*sizeof(int));

    // Add 5 more elements to the array.
    for (i = n; i < n1; ++i) {
        ptr[i] = i + 10;
    }
```

```
     // New array size
        // Print the elements of the array
    printf("\nThe elements of the enlarged array are: ");
    for (i = 0; i < n1; ++i) {
        printf("%d ", ptr2[i]);
    }

    // Freeing the memory
    free(ptr2);
    return 0;
}
```