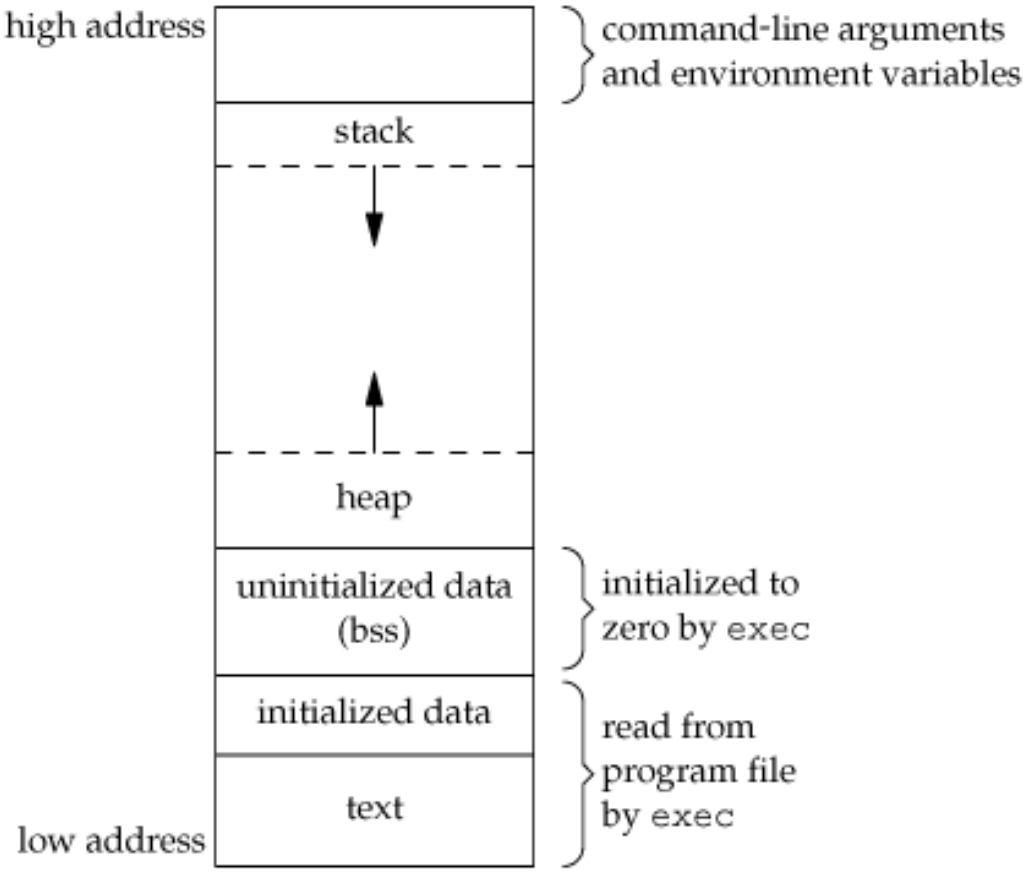
C PROGRAMMING
Lecture 7

1st semester 2023-2024

Program Address Space



The Stack

- The stack is the place where all local variables are stored
  - a local variable is declared in some scope
  - Example

```
int x; //creates the variable x on the stack
```

- As soon as the scope ends, all local variables declared in that scope end
  - the variable name and its space are freed
  - this happens without user control

The Heap

- The heap is an area of memory that the user handles explicitly
  - user requests and releases the memory through system calls
  - if a user forgets to release memory, it doesn’t get destroyed
    - it just uses up extra memory
- A user maintains a handle on memory allocated in the heap with a *pointer*

Declaring Pointers

- Declaring a pointer is easy
  - declared like regular variable except that an asterisk (*) is placed in front of the variable
  - example

```
int *x;
```

  - using this pointer now would be very dangerous
    - x points to some random piece of data
  - declaring a variable does not allocate space on the heap for it
    - it simply creates a local variable (on the stack) that is a pointer
    - use *alloc()* functions family to actually request memory on the heap

Memory Allocation

- In general a user wants a program to handle a variable number of entities (for example elements of an array)
- We might guess the maximum number of entities that might be required.
- Even if we do guess the maximum number, it might be that most of the cases only a few number of array elements is needed.
  - Solution:Allocate storage dynamically.

Memory Allocation

- Once you have allocated a variable such as an array on the stack, it is fixed in its size. You cannot make it longer or shorter. If you use malloc() or calloc() to allocate an array on the heap, you can use realloc() to resize it at some later time. To use these functions you need to

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

- The built-in functions malloc(), calloc(), realloc() and free() can be used to manage dynamically allocated data structures on the heap. The life cycle of a heap variable involves three stages:

1. allocating the heap variable using malloc() or calloc()
2. (optionally) resizing the heap variable using realloc()
3. releasing the memory from the heap using free()

Dynamic Memory Allocation

- `malloc()`
- Prototype: `void *malloc(int size);`
  - function searches heap for *size* contiguous free bytes
  - function returns the address of the first byte
  - programmers responsibility to not lose the pointer
  - programmers responsibility to not write into area past the last byte allocated
- Example:

Static allocation:

```
int array[10];
```

Dynamic allocation:

```
int * array=malloc(10*sizeof(int));
```

Dynamic Memory Allocation

- Also, type casting can be considered:

```
int *ptr;
ptr=malloc(10*sizeof(*ptr)); /*without cast*/
ptr=(int*)malloc(10*sizeof(*ptr)); /*with
  cast*/
```

- The calloc() function is like malloc() except that it also initializes all elements to zero. The calloc() function takes two input arguments, the number of elements and the size of each element.

Dynamic Memory Allocation

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

int main() {
    int n;
    int *arr;

    // Read the size of the array from the user
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Dynamically allocate memory for an array of integers
    int *arr = (int *)malloc(n * sizeof(int));

  // Allocate memory for an integer array of size elements and initialize to zero
    arr = (int *)calloc(n, sizeof(int));
```

Dynamic Memory Allocation

```
// Input values into the dynamically allocated array
    printf("Enter %d integer values:\n", n);
    for (int i = 0; i < n; ++i) {
        printf("Value %d: ", i + 1);
        scanf("%d", &arr[i]);
    }

    // Display the values stored in the dynamically allocated array
    printf("Values stored in the array:\n");
    for (int i = 0; i < n; ++i) {
        printf("%d ", arr[i]);
    }

    // Free the dynamically allocated memory
    free(arr);

    return 0;
}
```

Dynamic Memory Allocation

```
// Resize the array using realloc
 int newSize;
 printf("Enter the new size of the array: ");
 scanf("%d", &newSize);

 arr = (int *)realloc(arr, newSize * sizeof(int));
```

Dynamic Memory Allocation

- To use malloc, we need to know how many bytes to allocate. The sizeof() operator is used to calculate the size of a particular type.

```
points=malloc(n*sizeof(Point));
```

- Because malloc returns (void *), we need to change the type to the corresponding type of pointer – done by casting.

```
points=(Point*)malloc(n*sizeof(Point));
```

Dynamic Memory Allocation

```
typedef struct {
  int year;
  int month;
  int day;
} date;
date *eventlist=malloc(sizeof(date)*10);
int i;
for (i=0; i<10; i++) {
  eventlist[i].year = 2016;
  eventlist[i].month = 11;
  eventlist[i].day = 29;
}
```

Dynamic Memory Allocation

```
// Define a custom structure
typedef struct {
    int x;
    int y;
} Point;

int main() {
    int n;

    // Get the number of points from the user
      scanf("%d", &n);

    // Allocate memory for an array of Point structures
    Point *points = (Point *)malloc(n * sizeof(Point));
```

Dynamic Memory Allocation

- Once the data is no longer needed, it should be released back into the heap for later use.
- This is done using the free function, passing it the same address that was returned by malloc.

```
void free(void*);
```

- If allocated data is not freed, the program might run out of heap memory and will be unable to continue. Example:

```
free(ptr);
```

sizeof() Function

- The *sizeof()* function is used to determine the size of any data type
  - prototype: `int sizeof(data_type);`
  - returns how many bytes the data type needs
    - example: sizeof(int) = 4, sizeof(char) = 1
  - works for standard data types and user defined data types (structures)

Example

- 2D array allocation:

```
int** mat=(int**)malloc(rows*sizeof(int*))
for(index=0;index<rows;++index){
   mat[index]=(int*)malloc(col*sizeof(int));
}
// compute something with mat
for(index=0;index<rows;index++){
   free(mat[index]);
}
free(mat);
```

Example

- Imagine a 4D object for which you need to read all 4 dimensions and allocate memory for all its elements. Read values for all these elements and then display their sum. At the end, deallocate (free) the memory used.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(){
        int i,j,k,l,d1,d2,d3,d4;
        int sum=0;
        printf("Enter 4 dims: \n");
        scanf("%d",&d1);
        scanf("%d",&d2);
        scanf("%d",&d3);
        scanf("%d",&d4);
```

Example

```
int ****a4d=malloc(d1*sizeof(int ***));
for(i=0;i<d1;i++){
  a4d[i]=malloc(d2*sizeof(int **));
  for(j=0;j<d2;j++){
    a4d[i][j]=malloc(d3*sizeof(int*));
    for(k=0;k<d3;k++){
      a4d[i][j][k]=malloc(d4*sizeof(int));
    }
  }
}
```

Example

```
printf("Reading matrix elements\n");
  for(i=0;i<d1;i++)
    for(j=0;j<d2;j++)
      for(k=0;k<d3;k++)
        for(l=0;l<d4;l++){
 printf("elem a[%d][%d][%d][%d]=",i,j,k,l);
 scanf("%d",&a4d[i][j][k][l]);
 sum+=a4d[i][j][k][l];
        printf("sum is: %d\n",sum);
```

Example

```
printf("Deallocating memory\n");
for(i=0;i<d1;i++) {
  for(j=0;j<d2;j++){
    for(k=0;k<d3;k++){
       free(a4d[i][j][k]);
           }
           free(a4d[i][j]);
         }
         free(a4d[i]);
       }
       free(a4d);
       printf("program ends\n");
       return 0;}
```

Valgrind

- debugging and profiling tools that help you make your programs faster and more correct.
- tools that can automatically detect many memory management and threading bugs, and profile your programs in detail

Valgrind

```
#include <stdlib.h>
void f(void)
{
  int* x=malloc(10 * sizeof(int));
  x[10]=0;    // problem 1: heap block overrun
  }   // problem 2: memory leak -- x not freed

int main(void)
{
  f();
  return 0;
}
```

Valgrind

==14031== Memcheck, a memory error detector
==14031== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14031== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14031== Command: ./a.out
==14031==
==14031== Invalid write of size 4
==14031==   at 0x108668: f (in /home/dpop/Dany/C_code/cm7/a.out)
==14031==   by 0x108679: main (in /home/dpop/Dany/C_code/cm7/a.out)
==14031== Address 0x522f068 is 0 bytes after a block of size 40 alloc'd
==14031==   at 0x4C31B0F: malloc (in

Valgrind

==14031== HEAP SUMMARY:
==14031== in use at exit: 40 bytes in 1 blocks
==14031== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==14031==
==14031== **LEAK SUMMARY:**
==14031== definitely lost: 40 bytes in 1 blocks
==14031== indirectly lost: 0 bytes in 0 blocks
==14031== possibly lost: 0 bytes in 0 blocks
==14031== still reachable: 0 bytes in 0 blocks
==14031== suppressed: 0 bytes in 0 blocks
==14031== Rerun with --leak-check=full to see details of leaked memory

Valgrind

Install Valgrind:

You can install Valgrind on Linux using your distribution's package manager.
For example, on Debian/Ubuntu, you can use:

bash

sudo apt-get install valgrind

Valgrind

Compile Your C Program with Debug Symbols:

gcc -g your_program.c -o your_program

Or

gcc your_program.c

Run Your Program with Valgrind:

Use the valgrind command to run your compiled program:

bash
Copy code
valgrind ./your_program