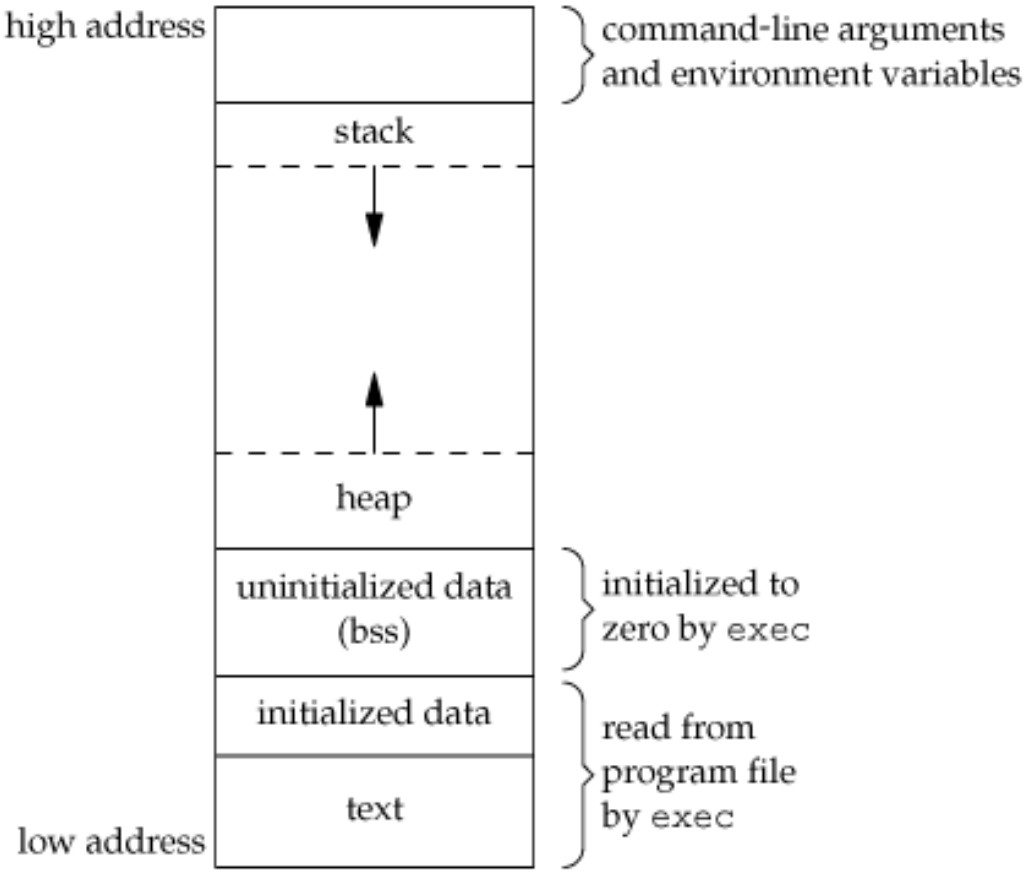
C PROGRAMMING
Lecture 5

1st semester 2022-2023

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

- 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

- 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

==24707== Invalid write of size 4

==24707== at 0x4004E2: f (in /home/info/horea/programare_c/lecture5/ve)

==24707== by 0x4004F2: main (in /home/info/horea/programare_c/lecture5/ve)

==24707== Address 0x4c28058 is 0 bytes after a block of size 40 alloc'd

==24707== at 0x4A0763E: malloc (vg_replace_malloc.c:207)

==24707== by 0x4004D5: f (in /home/info/horea/programare_c/lecture5/ve)

==24707== by 0x4004F2: main (in /home/info/horea/programare_c/lecture5/ve)

Linked Lists

- Linked list: data structure composed by nodes representing a sequence. Each node is composed of useful data and a link to the next node in the sequence. The structure allows operations for insertion or removal of elements from the sequence.
- Compared to arrays, lists are easier to operate when inserting or removing elements as these operations don't need reallocation or reorganization of the list.
- Advantages:
  - Basis for abstract types such as stacks, queues, associative arrays
  - Dynamic structure; memory allocated while the program is running
  - Easy implementation for insert and delete operations
- Disadvantages:
  - Can waste memory because of using pointers that require extra storage space
  - Compared to arrays, nodes are stored at non contiguous locations, increasing access time to specific elements
  - Nodes must be read in order from the beginning as linked lists
  - Difficult to navigate backwards

Linked Lists

- Consider the problem of building a linked list (of, let's say, positive integers) in the order they are read (Consider 0 terminates reading)
- For each number that is read:
  - allocate storage for node
  - put the value in the node
  - make the node the last in the list
- For the first number we must set head to point the new node and for the rest, current node next to point each time to new node
- Also use a last node pointer to the last element of the list

Linked Lists

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
        int num;
        struct node *next;
        } node, *nodeptr;

void print_list(nodeptr head){
        while(head!=NULL){
                printf("%d ",head->num);
                head=head->next;
        }
        printf("\n");
}
```

Linked Lists

```
int main(){
        int n;
        nodeptr head, new, last;
        head=NULL;
        if(scanf("%d", &n)!=1) n=0;
        while(n!=0){
                new=malloc(sizeof(node));
                new->num=n;
                new->next=NULL;
                if(head==NULL)
                        head=last=new;
                else{
                        last->next=new;
                        last=new;
                }
                if(scanf("%d", &n)!=1) n=0;
        }
        print_list(head);
        return 0;
}
```

Insert Ordered List

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
        int num;
        struct node *next;
        } node, *nodeptr;

nodeptr insertlist(nodeptr head, int n){
        nodeptr new, p, pp;
        new=malloc(sizeof(node));
        new->num=n;
        if(head==NULL){  // first number
                new->next=NULL;
                return new;  // first node in list
        }
```

Insert Ordered List

```
 else{ // list has at least 1 number
                if(n<=head->num){ //insert in front
                        new->next=head;
                        return new;  // first node in list
                }
                else{
                        pp=head;
                        p=head->next;
                        while(p!=NULL&&n>p->num){
                                pp=p;
                                p=p->next;
                        }
                        // insert after pp
                        pp->next=new;
                        new->next=p;
                        return head; // head remained the same
                }
        }
}
```

Insert Ordered List

```
void print_list(nodeptr head){
        while(head!=NULL){
                printf("%d ",head->num);
                head=head->next;
        }
        printf("\n");
}
int main(){
        int n;
        nodeptr head;
        head=NULL;
        if(scanf("%d", &n)!=1) n=0;
        while(n!=0){
                head=insertlist(head,n);
                if(scanf("%d", &n)!=1) n=0;
        }
        if(head==NULL)
                printf("\n No value was read \n");
        else{
                printf("\n The sorted list is");
                print_list(head);
        }
        return 0;
}
```

Delete Linked List

- We need to know the head, node to delete and previous node
- Several cases:
  - Head is NULL
  - Node to delete is the first node
  - Arbitrary node from the list (different from first)

Delete Linked List

```
nodeptr delete(nodeptr head, int value)
{
        if (head == NULL)
                return NULL;
        if (head->num == value) {
                nodeptr tempNextP;
                tempNextP = head->next;
                free(head);
                return tempNextP;
        }
        head->next = delete(head->next, value);
        return head;
}

 scanf("%d",&val);
 printf("\n");
 head=delete(head,val);
```

Other Dynamic Structures

- Trees
- Hash Tables
- Directed Acyclic Graphs
- …

Command Line Arguments

- In C a user can pass value to a program using command line arguments.
- Command-line arguments are given after the name of a program in command-line; they are passed in to the program from the operating system.
- To use command line arguments in the program, the main function declaration should be:

```
int main(int argc, char *argv[]);
```

or

```
int main(int argc, char **argv);
```

- Main can accept two (even 3) arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

Command Line Arguments

- When main is called, argc represents the count of the command-line arguments
- argv is an array composed by the arguments themselves.
- Each argument is considered to be a string which is represented as a pointer to char; thus argv is an array of pointers to char.
- Besides the command line arguments, we can use as input (not explicitly given as arguments) the environment variables: envp[]. In this case, the call is:

```
int main(int argc,char *argv[],char *envp[]);
```

Command Line Arguments

```
#include <stdio.h>
int main(int argc, char *argv[], char *envp[])
{
      int i;
      for(i = 0; i < argc; i++)
            printf("arg %d: %s\n", i, argv[i]);
      i=0;
      do{
            printf("env %d: %s\n", i,
envp[i++]);
      }while (envp[i]!=NULL);
      return 0;
}
```

Command Line Arguments

- Compute the sum of 2 integers given as command line arguments

Command Line Arguments

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

int main(int argc, char *argv[])
{
    int sum;
    if(argc!=3){
        printf("Usage: ./cmdsum n1 n2\n");
                return -1;
         }
         sum=atoi(argv[1])+atoi(argv[2]);
         printf("The sum is: %d\n",sum);
         return 0;
}
```