Programming Languages

Bibhas Adhikari

IIT Kharagpur

November 5, 2020

ADT:Passing arrays of structure to function

```
typedef struct employee {
int id;
char name[10];
float salary;
} e;
main( )
{
e emp1[3] = {0, 0, 0};
int x;
processEmp( emp1 ); //pass array name, which is a pointer
for (x = 0; x < 3; x++){
printf("\n ID: %d\n", emp1[x].id);
printf("Name: %s\n", emp1[x].name);
printf("Salary: $%.2f\n\n", emp1[x].salary);
}
}
```

ADT:Passing arrays of structure to function

```
void processEmp( e * emp ) //function receives a pointer
{
emp[0].id = 123;
strcpy(emp[0].name, "Ankush");
emp[0].salary = 65000.00;
emp[1].id = 234;
strcpy(emp[1].name, "Bibhas");
emp[1].salary = 28000.00;
emp[2].id = 456;
strcpy(emp[2].name, "Chandan");
emp[2].salary = 48000.00;
}
```

Dynamic memory allocation

- allocating, reallocating, and freeing memory using different functions
  - RAM (Random Access Memory) is considered as a volatile memory which is used for allocating, storing, and retrieving data
  - virtual memory - a reserved part of hard disk in which the operating system can swap memory segments

Stack

Software programs use their own area of memory, which is a combination of RAM and virtual memory, is called a stack. When a function is called in a program, the variables and parameters of the function are pushed onto the program's memory stack and then pushed off or "popped" when the function has completed or returned.

Dynamic memory allocation

Heap

Once a software program has terminated, the memory is returned for reuse for other software and system programs, and the operating system is responsible for managing this unallocated memory, which is called the heap

sizeof operator

The sizeof operator considers a variable name or data type as an argument and returns the number of bytes required to store the data in memory

Dynamic memory allocation

```
main()
{
int x;
float f;
double d;
char c;
typedef struct employee {
int id;
char *name;
float salary;
}e;
printf("\nSize of integer: %d bytes \n", sizeof(x));
printf("Size of float: %d bytes\n", sizeof(f));
printf("Size of double %d bytes\n", sizeof(d));
printf("Size of char %d byte\n", sizeof(c));
printf("Size of employee structure: %d bytes\n", sizeof(e));
}
```

Dynamic memory allocation

The sizeof operator can be used to calculatee the memory requirements of arrays:

```
main( )
{
int array[10];
printf("\nSize of array: %d bytes\n", sizeof(array));
printf("Number of elements in array ");
printf("%d\n", sizeof(array) / sizeof(int));
}
```

Dynamic memory allocation

malloc( ) function

malloc( ) is a function in the standard library *<stdlib.h>* and takes a number as an argument. *malloc( )* attempts to retrieve designated memory segments from the heap and it returns a pointer which is the starting point for the memory reserved

```
main( )
{
char *name;
name = (char *) malloc(80 * sizeof(char));
if ( name == NULL )
printf("\nOut of memory!\n");
else
printf("\nMemory allocated.\n");
}
```

Dynamic memory allocation

Managing strings using *malloc( )* function

```
main( )
{
char *name;
name = (char *) malloc(5*sizeof(char));
if ( name != NULL ) {
printf("\nEnter your name:  ");
gets(name);
printf("\nHi %s\n", name);
}
}
```

- Individual memory segments acquired by *malloc( )* can be treated much like array members

Dynamic memory allocation

Freeing memory

free( ) function - takes a pointer as an argument and frees the memory the pointer refers to

```
main( )
{
char *name;
name = (char *) malloc(5*sizeof(char));
if ( name != NULL ) {
printf("\nEnter your name:  ");
gets(name);
printf("\nHi %s\n", name);
free(name);
}
}
```

Dynamic memory allocation

calloc( )

calloc( ) function attempts to take hold of contiguous segments of memory from the heap. It takes two arguments: the first determines the number of memory segments needed and the second is the size of the data type

```
main( )
{
int *numbers;
numbers = (int *) calloc(10, sizeof(int));
if ( numbers == NULL )
return; // return if calloc is not successful
}
```

Dynamic memory allocation

realloc( )

realloc( ) function provides a way to expand contiguous blocks of memory while preserving the original contents. It takes two arguments for parameters and returns a pointer as output

```
newPointer = realloc(oldPointer, 10 * sizeof(int));
```

realloc( )'s first argument takes the original pointer set by *malloc( )* or *calloc( )*. The second argument describes the total amount of memory you want to allocate.

Scenario	Outcome
Successful without move	Same pointer returned
Successful with move	New pointer returned
Not successful	NULL pointer returned