

**CSE1303 Part A**  
**Data Structures and Algorithms**  
**Summer Semester 2003**

**Lecture A6 – Dynamic Memory**

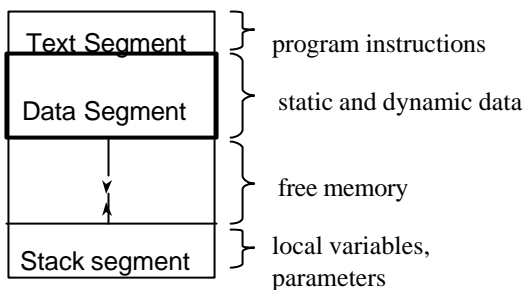
Kymerly Fergusson

**Overview**

- Virtual Memory
- What is Dynamic Memory ?
- How to find the size of objects.
- Allocating memory.
- Deallocating memory.

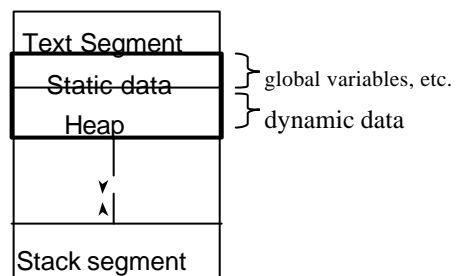
2

**Virtual Memory**



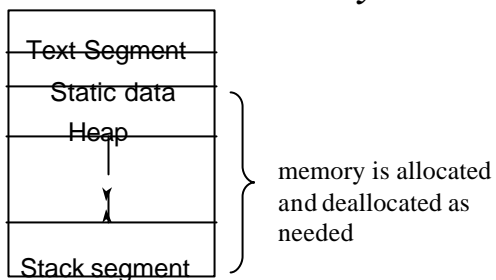
3

**Virtual Memory**



4

**Virtual Memory**



5

**What is Dynamic Memory?**

- Memory which is allocated and deallocated during the execution of the program.
- Types:
  - data in run-time stack
  - dynamic data in the heap

6

### Example: Run-Time Stack

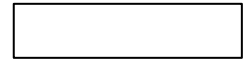
- Memory is allocated when a program calls a function.
  - parameters
  - local variables
  - where to go upon return
- Memory is deallocated when a program returns from a function.

7

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
    
```

stack

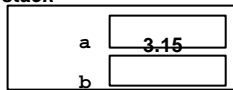


8

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
    
```

stack

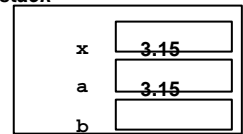


9

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
    
```

stack

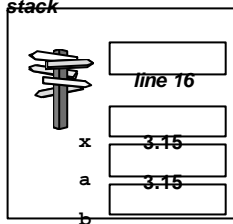


10

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
    
```

stack

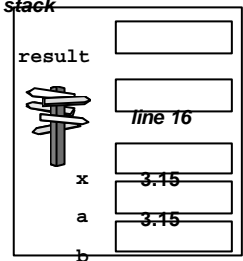


11

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
    
```

stack



12

## Computer Science Monash University

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```

**stack**

result	9.9225
line 16	
x	3.15
a	3.15
b	

13

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```

**stack**

result	9.9225
line 16	
x	3.15
a	3.15
b	

14

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```

**stack**

a	3.15
b	9.9225

15

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```

**stack**

a	3.15
b	9.9225

16

```

01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```

**stack**

--	--

17

```

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

int factorial (int x)
{
    if (x == 0)
    {
        return 1;
    }
    return x * factorial (x -1);
}

void main()
{
    int n;
    printf("Enter a number: \n");
    scanf("%d", &n);

    printf("Factorial: %d\n", factorial(n));
}

```

18

## How much memory to allocate?

•The **sizeof** operator returns the size of an object, or type, in bytes.

•Usage:

**sizeof( Type )**

**sizeof Object**

19

### Example 1:

```
int    n;
char   str[25];
float  x;
double numbers[36];

printf("%d\n", sizeof(int));
printf("%d\n", sizeof n);

n = sizeof str;
n = sizeof x;
n = sizeof(double);
n = sizeof numbers;
```

20

## Notes on sizeof

•Do not assume the size of an object, or type; use **sizeof** instead.

Example: `int n;`

- In DOS: 2 bytes (16 bits)
- In GCC/Linux: 4 bytes (32 bits)
- In MIPS: 4 bytes (32 bits)

21

### Example 2:

```
#include <stdio.h>

#define MAXNAME 80
#define MAXCLASS 100

struct StudentRec
{
    char   name[MAXNAME];
    float  mark;
};

typedef struct StudentRec Student;
```

22

### Example 2 (cont.):

```
int main()
{
    int    n;
    Student class[MAXCLASS];

    n = sizeof(int);
    printf("Size of int = %d\n", n);

    n = sizeof(Student);
    printf("Size of Student = %d\n", n);

    n = sizeof class;
    printf("Size of array class = %d\n", n);

    return 0;
}
```

23

## Notes on sizeof (cont.)

•The size of a structure is not necessarily the sum of the sizes of its members.

Example:

```
struct cardRec {
    char suit;
    int  number;
};

typedef struct cardRec Card;

Card hand[5];

printf("%d\n", sizeof(Card));
printf("%d\n", sizeof hand);
```

“alignment” and “padding”

5 \* sizeof(Card)

24

### Dynamic Memory: Heap

- Memory can be allocated for new objects.
- Steps:
  - determine how many bytes are needed
  - allocate enough bytes in the heap
  - take note of where it is (memory address)

25

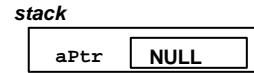
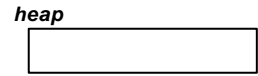
#### Example 1:

```
#include <stdlib.h>

main()
{
    int* aPtr = NULL;

    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```



26

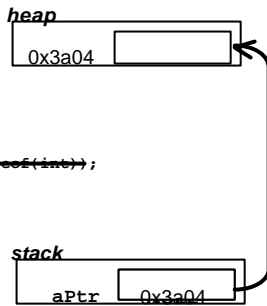
#### Example 1:

```
#include <stdlib.h>

main()
{
    int* aPtr = NULL;

    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```



27

#### Example 1:

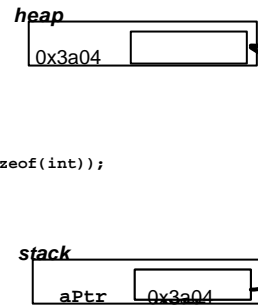
```
#include <stdlib.h>

main()
{
    int* aPtr = NULL;

    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```

“type cast”



28

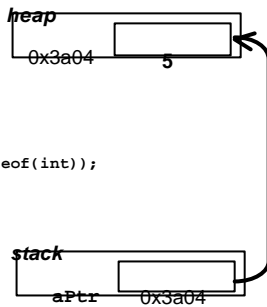
#### Example 1:

```
#include <stdlib.h>

main()
{
    int* aPtr = NULL;

    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```



29

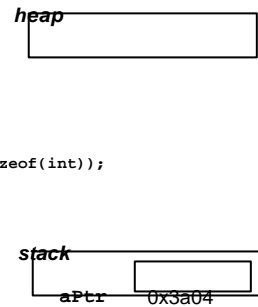
#### Example 1:

```
#include <stdlib.h>

main()
{
    int* aPtr = NULL;

    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```



30

**Example 1:**

```
#include <stdlib.h>

main()
{
  int* a;
  aPtr = (int*)malloc(sizeof(int));
  *aPtr = 5;
  free(aPtr);
}
```

**deallocates memory**

**heap**

**stack**

aPtr 0x3a04

31

**Example 2:**

```
#include <stdlib.h>

main()
{
  int* aPtr;
  int* bPtr;

  aPtr = (int*)malloc(sizeof(int));
  *aPtr = 5;

  bPtr = (int*)malloc(sizeof(int));
  *bPtr = 8;

  free(aPtr);

  aPtr = bPtr;
  bPtr = (int*)malloc(sizeof(int));
  *bPtr = 6;
}
```

32

## Allocating Memory

- Need to include `stdlib.h`
- `malloc(n)` returns a pointer to `n` bytes of memory.
- Always check if `malloc` has returned the `NULL` pointer.
- Apply a type cast to the pointer returned by `malloc`.

33

## Deallocating Memory

- `free(pointer)` deallocates the memory pointed to by a **pointer**.
- It does nothing if `pointer == NULL`.
- **pointer must** point to memory previously allocated by `malloc`.
- Should free memory no longer being used.

34

### Example 3:

```
main()
{
  Student* studentPtr = NULL;

  studentPtr = (Student*)malloc(sizeof(Student));

  if (studentPtr == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(1);
  }

  *studentPtr = readStudent();
  printStudent(*studentPtr);

  free(studentPtr);
}
```

35

### Example 4:

```
main()
{
  Student* class = NULL;
  int n, i, best = 0;

  printf("Enter number of Students: ");
  scanf("%d", &n);

  class = (Student*)malloc(n * sizeof(Student));
  if (class != NULL) {
    for (i = 0; i < n; i++) {
      class[i] = readStudent();
      if (class[best].mark < class[i].mark) {
        best = i;
      }
    }
    printf("Best student: ");
    printStudent(class[best]);
  }
}
```

36

### *Common Errors*

- Assuming that the size of a structure is the sum of the sizes of its members.
- Referring to memory already freed.
- Not freeing memory which is no longer required.
- Freeing memory not allocated by malloc.

37

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

### *Example 5:*

```
float** makeMatrix(int n, int m){
    float* memoryPtr;
    float** matrixPtr;
    int i;

    memoryPtr = (float*)malloc(n*m*sizeof(float));
    matrixPtr = (float**)malloc(n*sizeof(float*));

    if (memoryPtr == NULL || matrixPtr == NULL) {
        fprintf(stderr, "Not enough memory\n");
        exit(1);
    }

    for (i = 0; i < n; i++, memoryPtr += m){
        matrixPtr[i] = memoryPtr;
    }
    return matrixPtr;
}
```

38

### *Revision*

- sizeof
- malloc
- free
- Common errors.

### *Preparation*

- Read Kruse et al. Chapter 4, Section 4.5

39

### *Revision: Reading*

- Kruse 4.5
- Deitel and Deitel 12.3
- Standish 8.6
- King 17

### *Preparation*

*Next lecture: Nodes and Linked Structures*

- Read Kruse et al. Chapter 4, Section 4.5

40