

Topic 4

Dynamic Memory

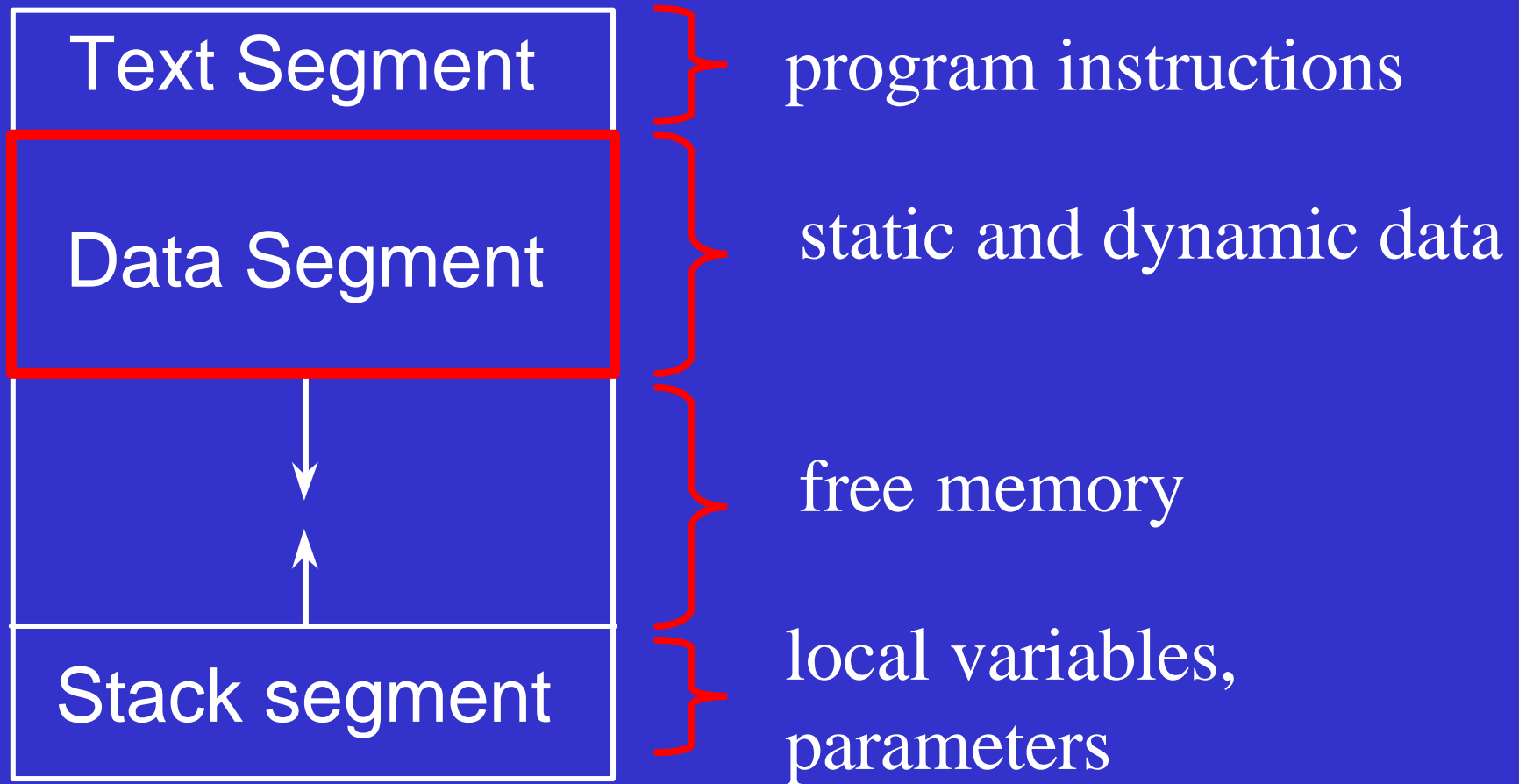
CSE1303 Part A

Data Structures and Algorithms

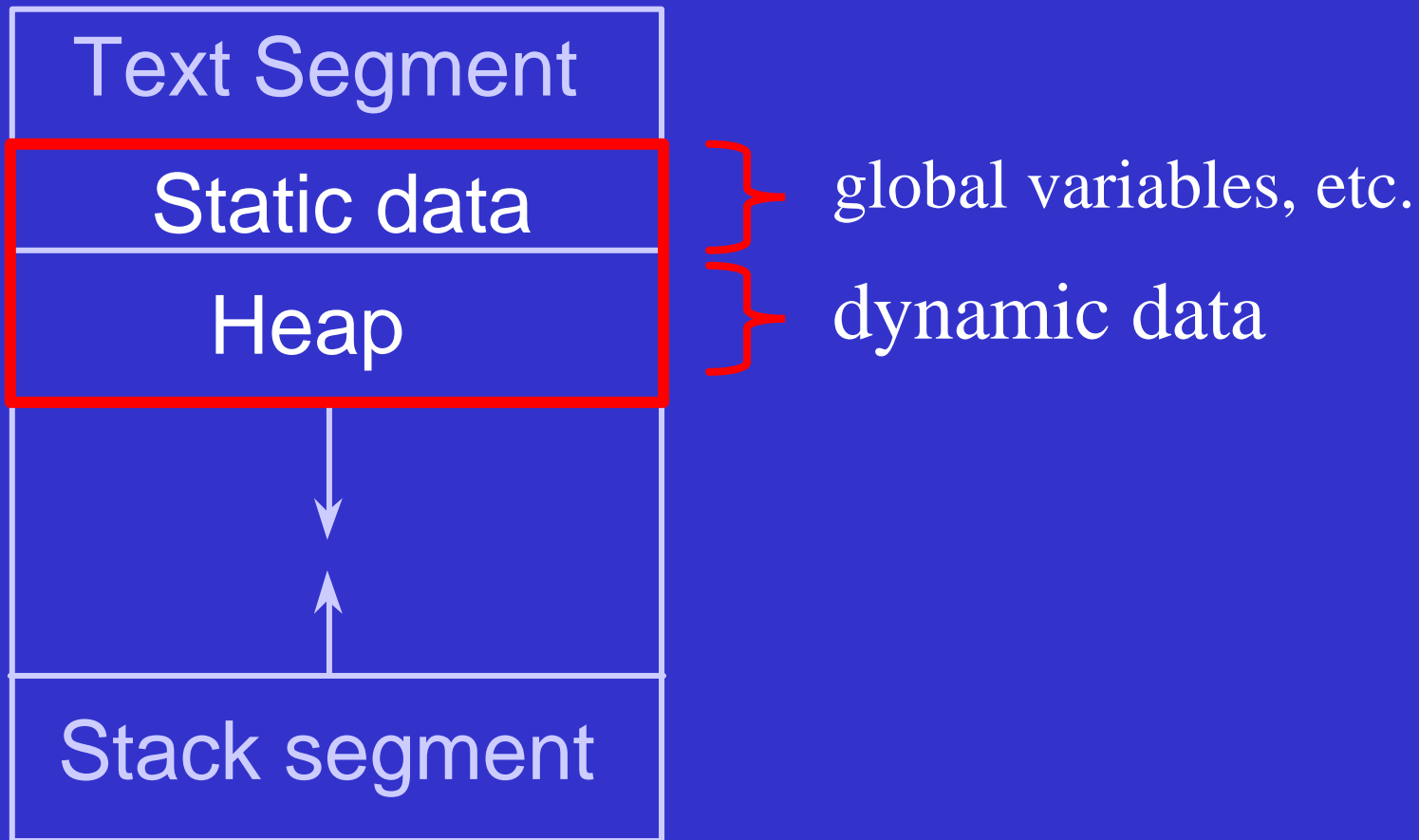
Overview

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

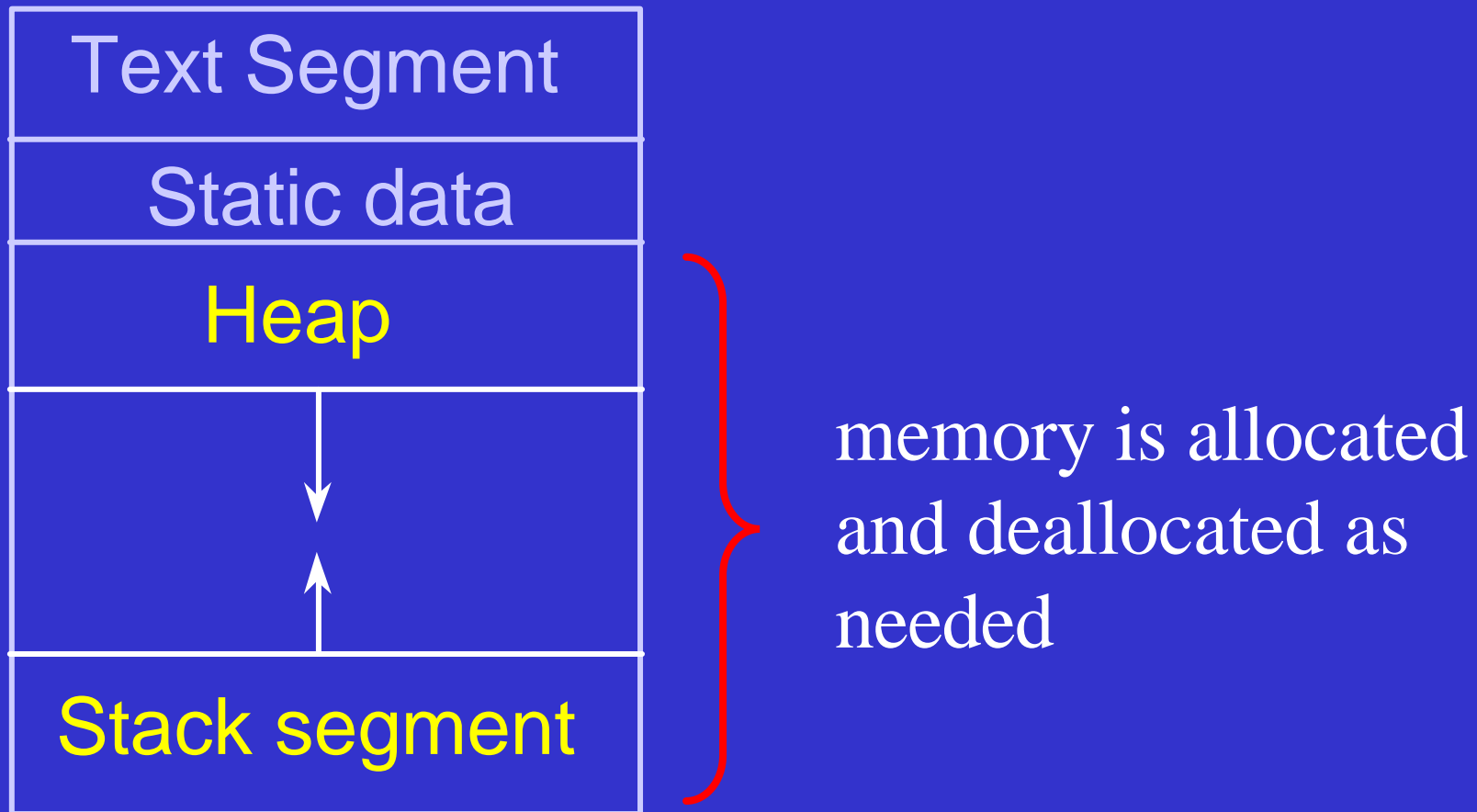
Virtual Memory



Virtual Memory



Virtual Memory



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**

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.

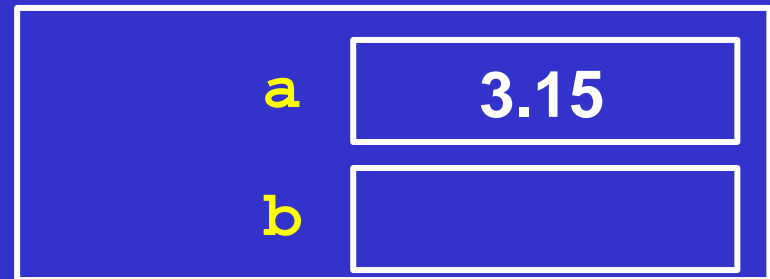
```
01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
```

stack



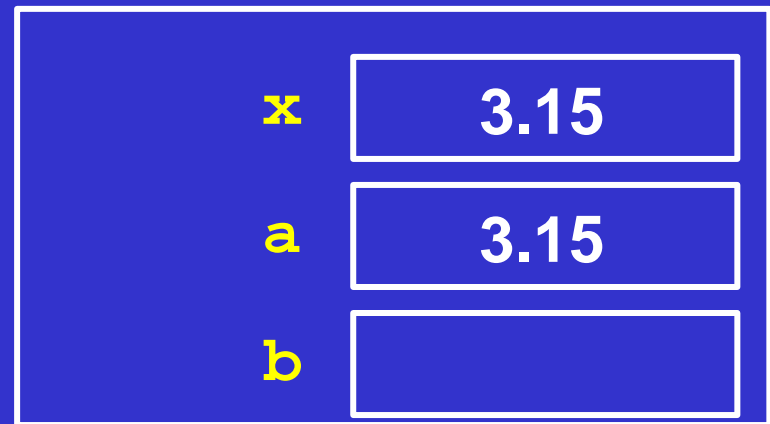
```
01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
```

stack



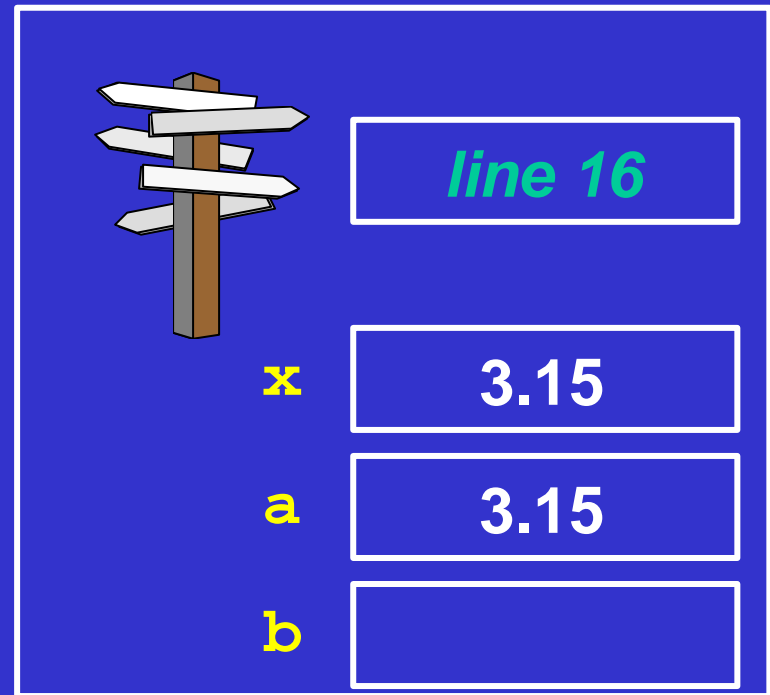
```
01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
```

stack



```
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

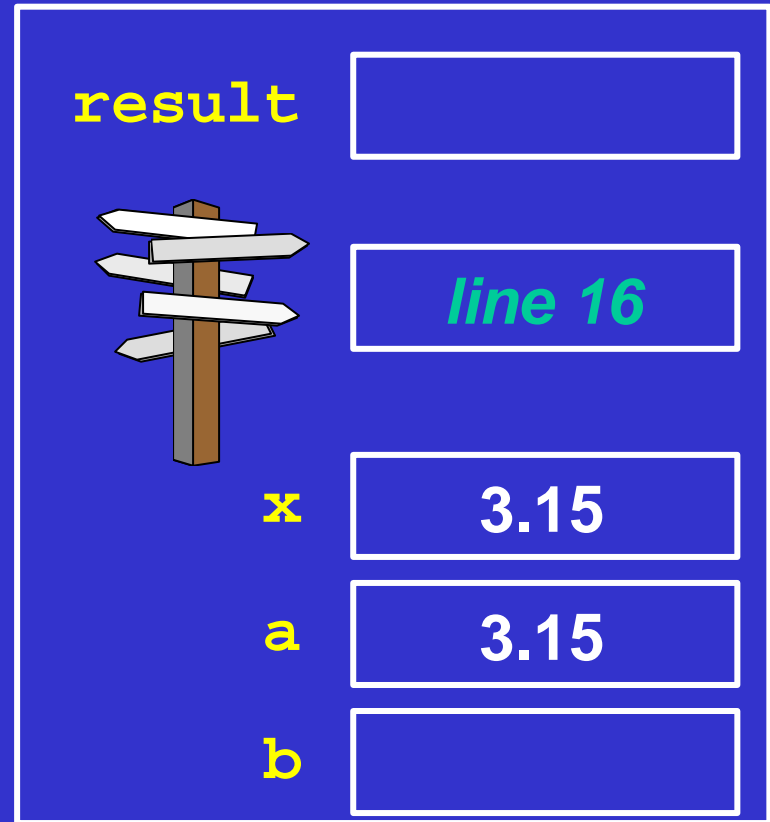


```

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

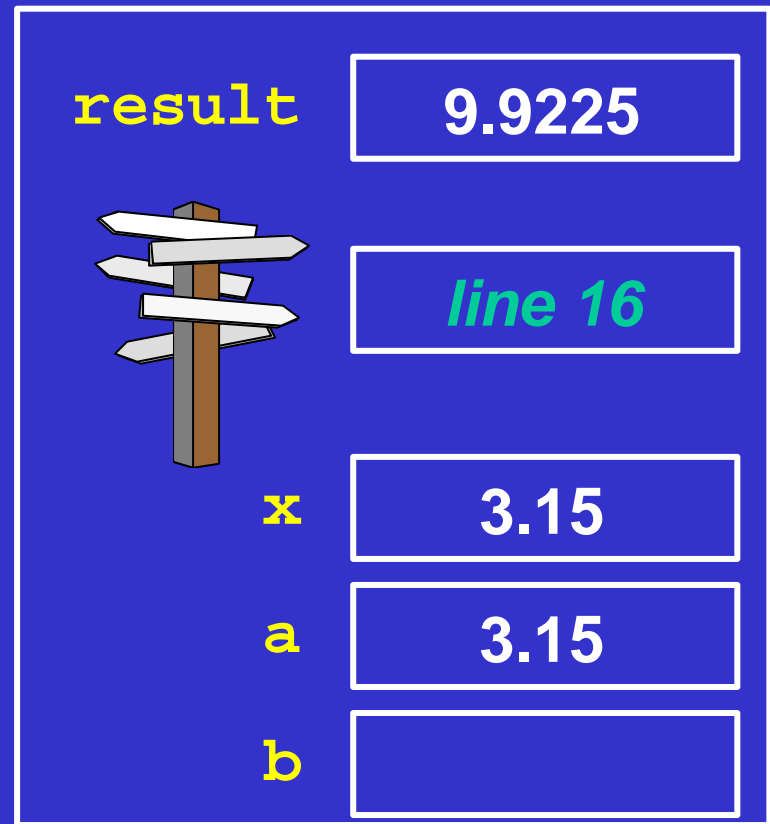


```

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: }

```

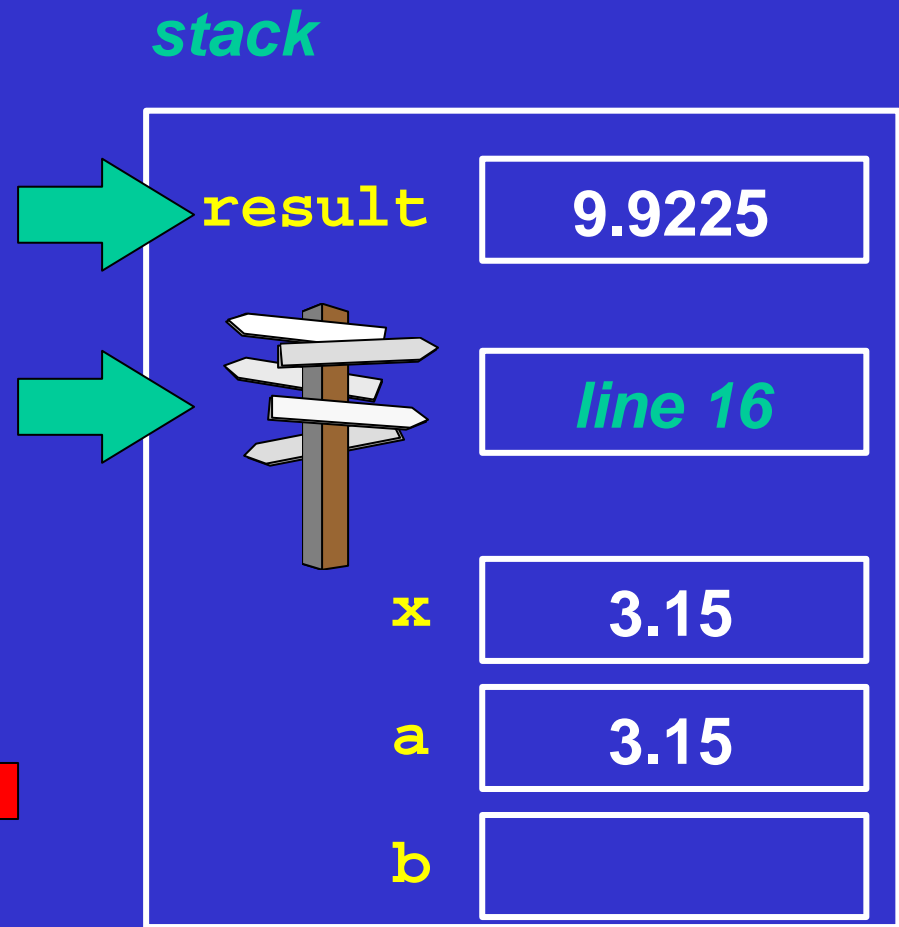
stack



```

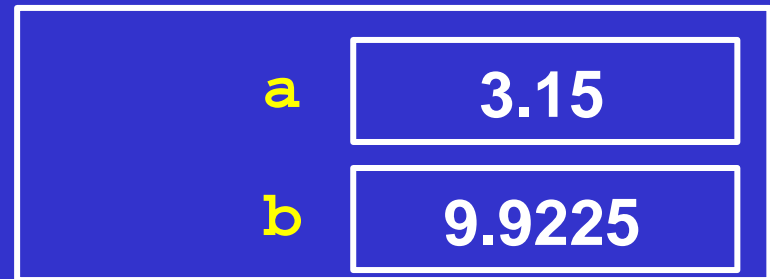
01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }

```



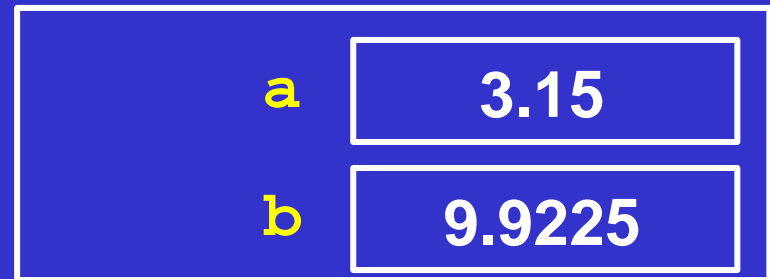
```
01: #include <stdio.h>
02:
03: float square(float x)
04: {
05:     float result;
06:
07:     result = x * x;
08:     return result;
09: }
10:
11: main()
12: {
13:     float a = 3.15;
14:     float b;
15:
16:     b = square(a);
17:     printf("%f\n", b);
18: }
```

stack



```
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



```
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



```
#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);
}
}
```

How much memory to allocate?

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

sizeof (*Type*)

sizeof *Object*

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;
```

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)

Example 2:

```
#include <stdio.h>

#define MAXNAME    80
#define MAXCLASS   100

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

typedef struct StudentRec Student;
```

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;
}
```

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];
```

**“alignment”
and “padding”**

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

5*sizeof(Card)

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)

Example 1:

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

```
main()
```

```
{
```

```
    int* aPtr = NULL;
```

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

```
    *aPtr = 5;
```

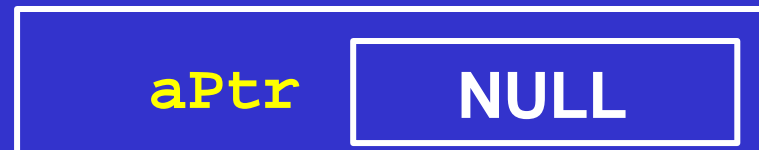
```
    free(aPtr);
```

```
}
```

heap



stack



Example 1:

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

```
main()
```

```
{
```

```
    int* aPtr = NULL;
```

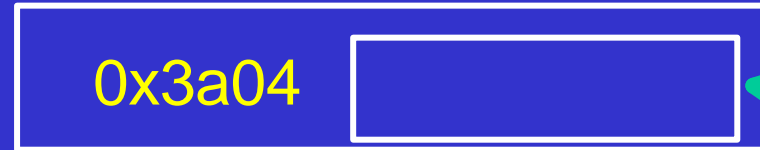
```
    aPtr = (int*)malloc(sizeof(int));
```

```
    *aPtr = 5;
```

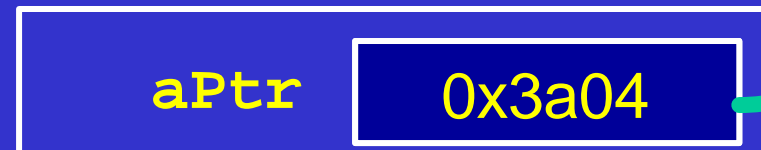
```
    free(aPtr);
```

```
}
```

heap



stack



Example 1:

“type cast”

```
#include <stdlib.h>
main()
{
    int* aPtr = NULL;

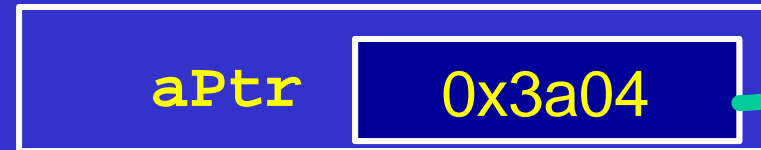
    aPtr = (int*)malloc(sizeof(int));
    *aPtr = 5;

    free(aPtr);
}
```

heap



stack



Example 1:

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

```
main()
```

```
{
```

```
    int* aPtr = NULL;
```

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

```
    *aPtr = 5;
```

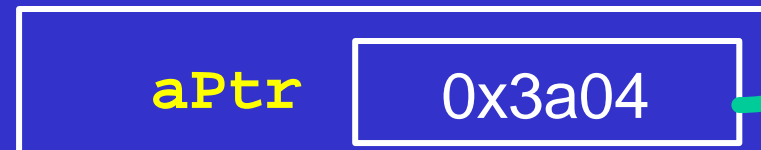
```
    free(aPtr);
```

```
}
```

heap



stack



Example 1:

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

```
main()
```

```
{
```

```
    int* aPtr = NULL;
```

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

```
    *aPtr = 5;
```

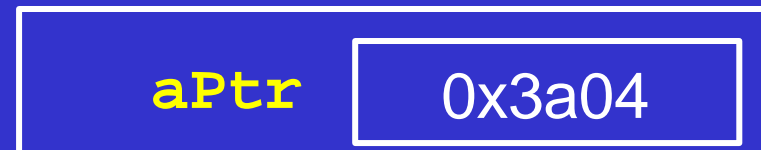
```
    free(aPtr);
```

```
}
```

heap



stack



Example 1:

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

```
main()
```

```
{
```

```
    int* a
```

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

```
    *aPtr = 5;
```

```
    free(aPtr);
```

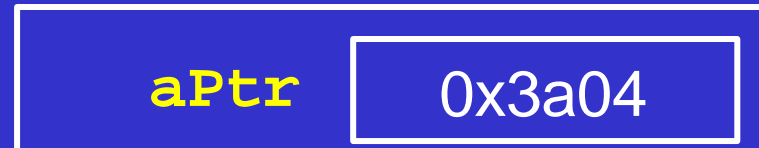
```
}
```

heap



**deallocates
memory**

stack



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;
}
```

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**.

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.

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);
}
```

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]);
    }
}
```

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.

```
#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;
}
```

Revision

- sizeof
- malloc
- free
- Common errors.

Preparation

- Read Kruse et al. Chapter 4, Section 4.5