B. Tech. I CSE (Sem-2)
Data Structures
CS102

Pointers

*Some slides are kept with logical or syntax error. So, if you are absent in theory class, please also refer the slides provided at the end of the presentation.

Dr. Dipti P. Rana

Department of Computer Science and Engineering

SVNIT, Surat

Review-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 gone
  - This happens implicitly – the user has no control over it

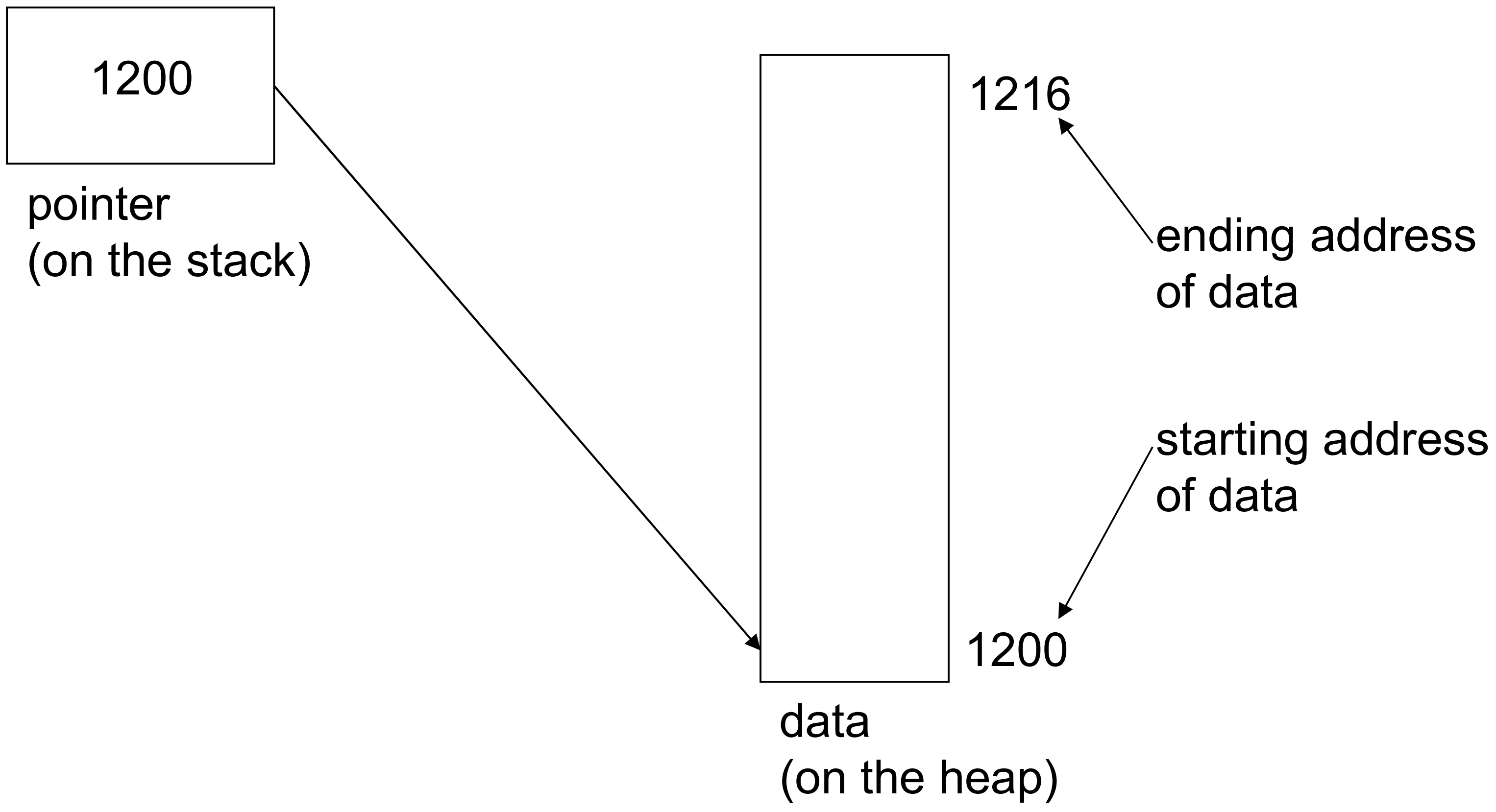
Review-The Heap

- The heap is an area of memory that the user handles explicitly
  - User requests and releases the memory
  - If a user forgets to release memory, it does not get destroyed
    - It just uses up extra memory
- A user maintains a handle on memory allocated in the heap with a *pointer*

Pointers

- A pointer is simply a local variable that refers to a memory location on the heap
- Accessing the pointer, actually references the memory on the heap

Basic Idea


1200
pointer
(on the stack)
1216
ending address of data
starting address of data
1200
data
(on the heap)

Pointers

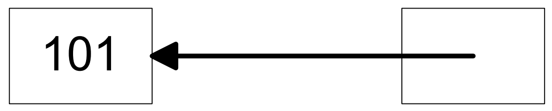
A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Pointer Basics

- Variables are allocated at *addresses* in computer memory (address depends on computer/operating system)
- Name of the variable is a reference to that memory address
- A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):

```
int V = 101;

int *P = &V;
```

Name	**V**	**P**
Address	*v* (some value)	*p* (some value)
Abstract Representation		
Concrete Representation	4 bytes for int value 101	4 bytes for mem address *v*

Pointer Variable Definition

Basic syntax: *Type *Name*

Examples:

```
int *P;              /* P is var that can point to an int var */
float *Q;            /* Q is a float pointer */
char *R;             /* R is a char pointer */
```

Complex example:

```
int *AP[5];          /* AP is an array of 5 pointers to ints */
```

– More on how to read complex declarations later

Address (&) Operator

The address (&) operator can be used in front of any variable object in C -- the result of the operation is the location in memory of the variable

Syntax: &*VariableReference*

Examples:

```
int V;
int *P;
int A[5];
&V - memory location of integer variable V
&(A[2]) - memory location of array element 2 in array A
&P - memory location of pointer variable P
```

Pointer Variable Initialization/Assignment

NULL

- pointer constant to non-existent address
- used to indicate pointer points to nothing

- Can initialize/assign pointer vars to NULL or use the address (&) op to get address of a variable
  - Variable in the address operator must be of the right type for the pointer (an integer pointer points only at integer variables)

Examples:

```
int V;
int *P = &V;
int A[5];
P = &(A[2]);
```

Indirection (*) Operator

- A pointer variable contains a memory address
- To refer to the *contents* of the variable that the pointer points to, indirection operator is used
- Syntax: **PointerVariable*
- Example:
  - int V = 101;
  - int *P = &V;
  - /* Then *P would refer to the contents of the variable V (in this case, the integer 101) */

printf("%d",*P); /* Prints 101 */

Pointer Example1

```
int A = 3;
int B;
int *P = &A;
int *Q = P;
int *R = &B;

printf(“Enter value:“);
scanf(“%d”, R);
printf(“%d %d\n”,A,B);
printf(“%d %d %d\n”,
  *P,*Q,*R);
```

```
Q = &B;
if (P == Q)
  printf(“1\n”);
if (Q == R)
  printf(“2\n”);
if (*P == *Q)
  printf(“3\n”);
if (*Q == *R)
  printf(“4\n”);
if (*P == *R)
  printf(“5\n”);
```

Reference Parameters

- To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)
- Then we use indirection operator inside the function to change the value the parameter points to:

```
void changeVar(float *cvar) {
  *cvar = *cvar + 10.0;
}

float X = 5.0;

changeVar(&X);
printf("%.1f\n",X);
```

Pointer as Return Value

- A function can also return a pointer value:

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%.1f %.1f\n",*maxA,A[4]);
}
```

Pointers to Pointers

- A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;         /* P points to int V */
int **Q = &P;        /* Q points to int pointer P */

printf(“%d %d %d\n”,V,*P,**Q); /* prints 101 3 times */
```

Declaring Pointers

- 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 will is a pointer
  - Use *malloc()* to actually request memory on the heap

Dynamic Memory Allocation

- Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)
- Previous example: ask the user how many numbers to read, then allocate array of appropriate size
- Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done
  - memory allocated in the *Data Heap*

Memory Management Functions

- malloc - routine used to allocate a single block of memory
- calloc - routine used to allocate arrays of memory
- realloc - routine used to extend the amount of space allocated previously
- free - routine used to tell program a piece of memory no longer needed
  - note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Array Allocation with malloc

prototype: malloc(size_t esize)

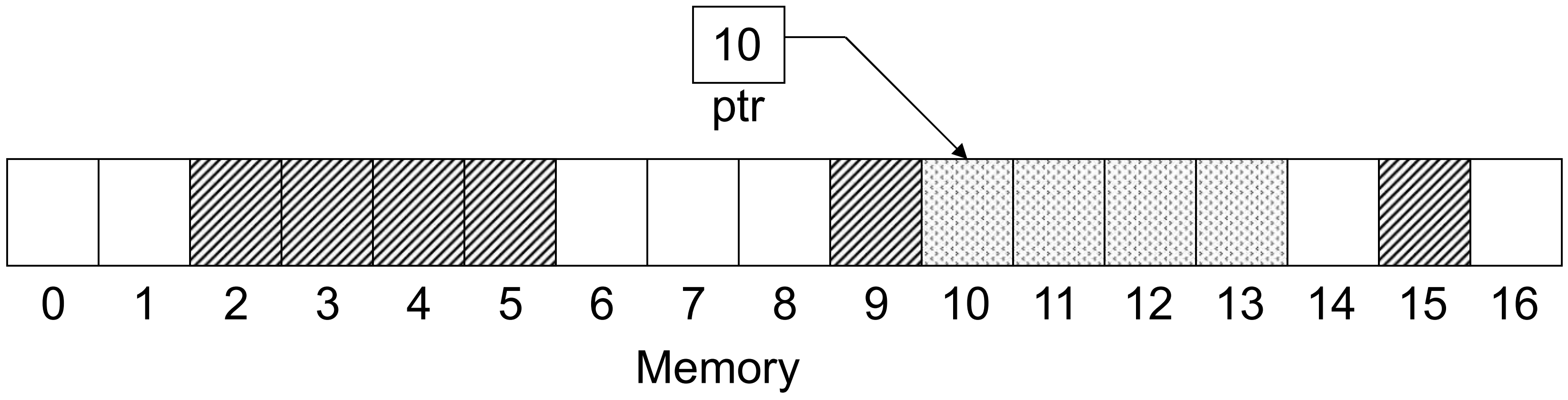
- Use it to allocate a single block of the given size esize by searching heap for size contiguous free bytes
- Memory is allocated from heap
- Returns the address of the first byte
- NULL returned if not enough memory available
- Memory Must be released using free once the user is done
- Programmers responsibility to not lose the pointer

malloc

- Prototype: *int 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:

```
char *ptr;
ptr = malloc(4);  // new allocation
```

Key
▨ previously allocated
░ new allocation



Array Allocation with calloc

prototype: calloc(size_t num, size_t esize)

- size_t is a special type used to indicate sizes, generally an unsigned int
- num is the number of elements to be allocated in the array
- esize is the size of the elements to be allocated
- An amount of memory of size num*esize allocated on heap
- calloc returns the address of the first byte of this memory
- If not enough memory is available, calloc returns NULL
- Generally we cast the result to the appropriate type

Array Allocation with calloc

– Can perform the same function as malloc

```
malloc(N * sizeof(float))
```

is equivalent to

```
(float *) calloc(N, sizeof(float))
```

calloc Example

```
float *nums;
int N;
int I;

printf(“Read how many numbers:”);
scanf(“%d”,&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf(“Please enter number %d: “,I+1);
  scanf(“%f”,&(nums[I]));
}
/* Calculate average, etc. */
```

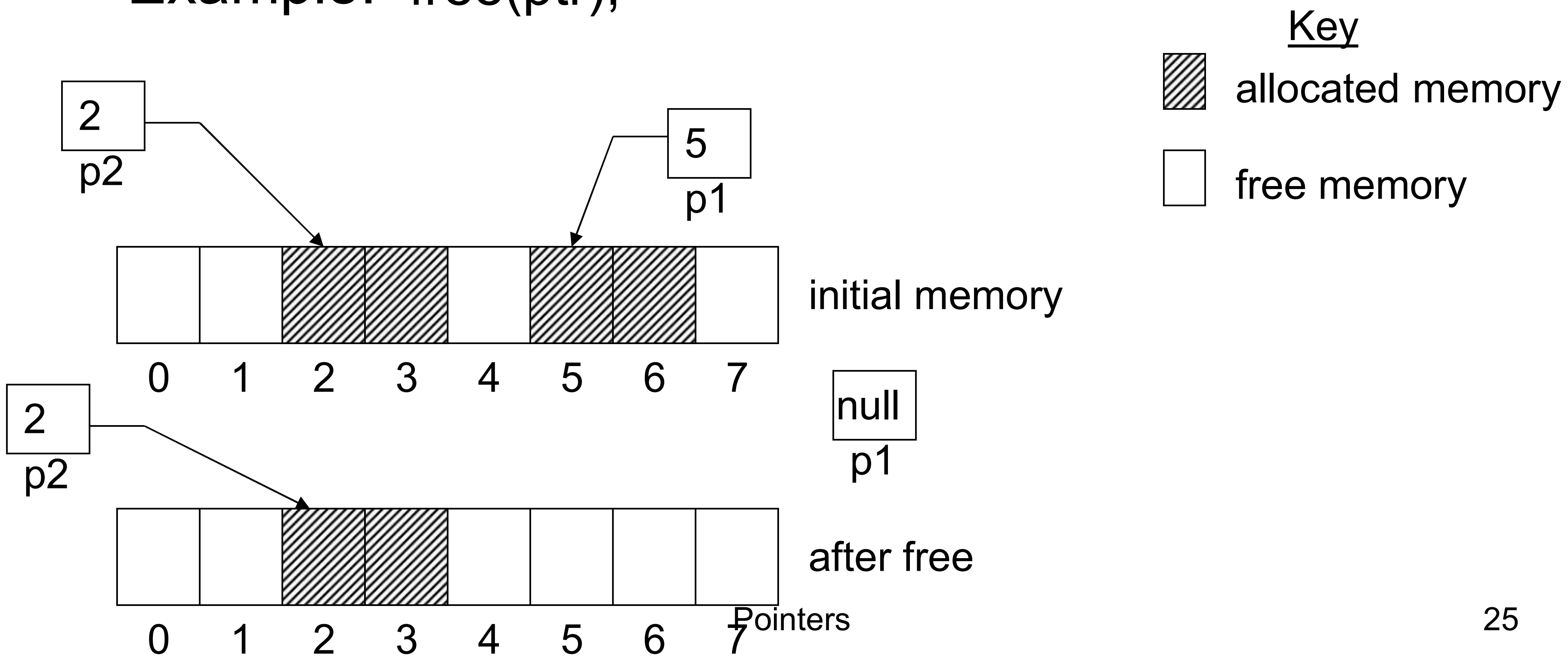
Releasing Memory (free)

prototype: void free(void *ptr)

- Memory at location pointed to by ptr is released (so we could use it again in the future)
- Program keeps track of each piece of memory allocated by where that memory starts
- Results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

free

- Prototype: *int free(int ptr);*
  - Releases the area pointed to by ptr
  - ptr must not be null
    - Trying to free the same area twice will generate an error
- Example: free(ptr);



free Example

```
float *nums;
int N;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));

/* use array nums */

/* when done with nums: */

free(nums);

/* would be an error to say it again - free(nums) */
```

The Importance of free

```
void problem() {
  float *nums;
  int N = 5;

  nums = (float *) calloc(N, sizeof(float));

  /* But no call to free with nums */
} /* problem ends */
```

- When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (which is allocated on the heap) - furthermore, we have no way to figure out where it is)
  - This Problem called ***memory leakage***

Increasing Memory Size with realloc

prototype: void * realloc(void * ptr, size_t esize)

- ptr is a Pointer to a piece of memory previously dynamically allocated
- Esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already)
- Program allocates memory of size esize
  - Then it copies the contents of the memory at ptr to the first part of the new piece of memory,
  - Finally, the old piece of memory is freed up

realloc Example

```
float *nums;
int I;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (I = 0; I < 5; I++)
  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated,
   the first 5 floats from the old nums are copied as
   the first 5 floats of the new nums, then the old
   nums is released */
```

sizeof() Operator

- The *sizeof()* operator is used to determine the size of any data type
  - Prototype: *int sizeof(data type);*
  - Returns how many bytes the data type needs
    - For example: sizeof(int) = 4, sizeof(char) = 1
  - Works for standard data types and user defined data types (structures)
  - Looks like a function, but it is an unary operator

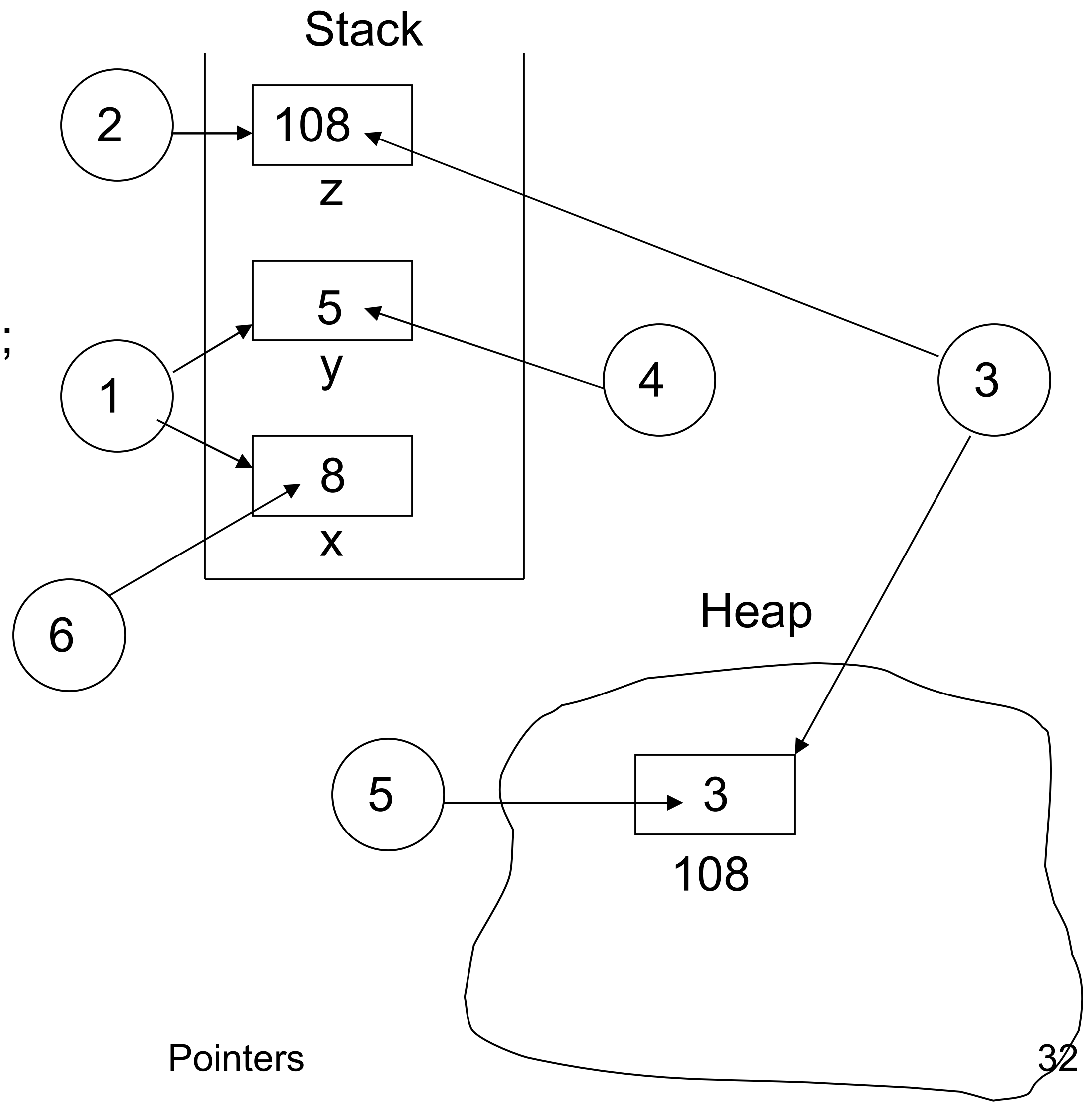
sizeof() Operator

- When we pass a++ to sizeof, the expression a++ is not evaluated
- In case of functions, parameters are first evaluated, then passed to function

```
// C program to demonstrate that sizeof // is an operator
#include<stdio.h>
int main()
{
	int a = 5;
	printf("%d\n", (int)sizeof(++a));
	printf("%d", a);
	return 0;
}
```

Simple Example

```
   int main() {
1      int x, y;
2      int *z;
3      z = malloc(sizeof(int));

4      y = 5;
5      *z = 3;
6     x = *z + y;
7     free(z);

       return 0;
  }
```



Simple Example

1. Declare local variables x and y.
2. Declare local pointer z.
3. Allocate space on the heap for single integer. This step also makes z point to that location (notice the address of the space on the heap is stored in z’s location on the stack.
4. Set the local variable y equal to 5.
5. Follow the pointer referenced by z to the heap and set that location equal to 3.
6. Grab the value stored in the local variable y and follow the pointer z to grab the value stored in the heap. Add these two together and store the result in the local variable x.
7. Releases the memory on the heap (so another process can use it) and sets the value in the z pointer variable equal to NULL. (this step is not shown on the diagram)

Common Mistakes

- Using a pointer before allocating heap space

```
int *ptr;
*ptr = 5;
```

- Changing the pointer, not the value it references

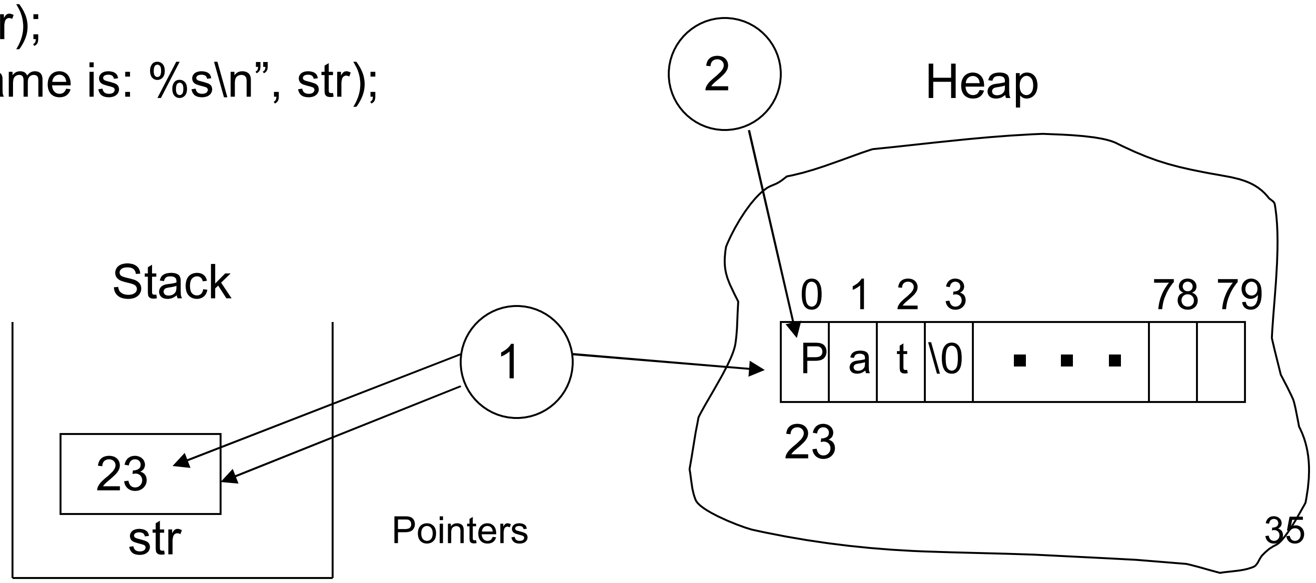
```
int *ptr = malloc(sizeof(int));
ptr = 10;   // sets value on stack to 10, not value on the heap
```

- Forgetting to free space on the heap (memory leak)

```
int *p1 = malloc(sizeof(int));
int *p2 = malloc(sizeof(int));
p1 = p2;  // making p1 point to p2 is fine, but now you cannot free
          // the space originally allocated to p1
```

One More Example

```
  #include <stdio.h>

  #define MAX_LINE 80

  int main() {
1    char *str = malloc(MAX_LINE * sizeof(char));

     printf(“Enter your name: “);
2    scanf(“%s”, str);
     printf(“Your name is: %s\n”, str);
3    free(str);

     return 0;
  }
```



One More Example

1. In one line, declare the pointer variable (gets placed on the stack), allocate memory on the heap, and set the value of the pointer variable equal to the starting address on the heap.
2. Read a value from the user into the space on the heap. This is why scanf takes pointers as the parameters passed in.
3. Release all the space on the stack pointed to by str and set the value of the str pointer on the stack equal to null. (step not shown)

Dereferencing

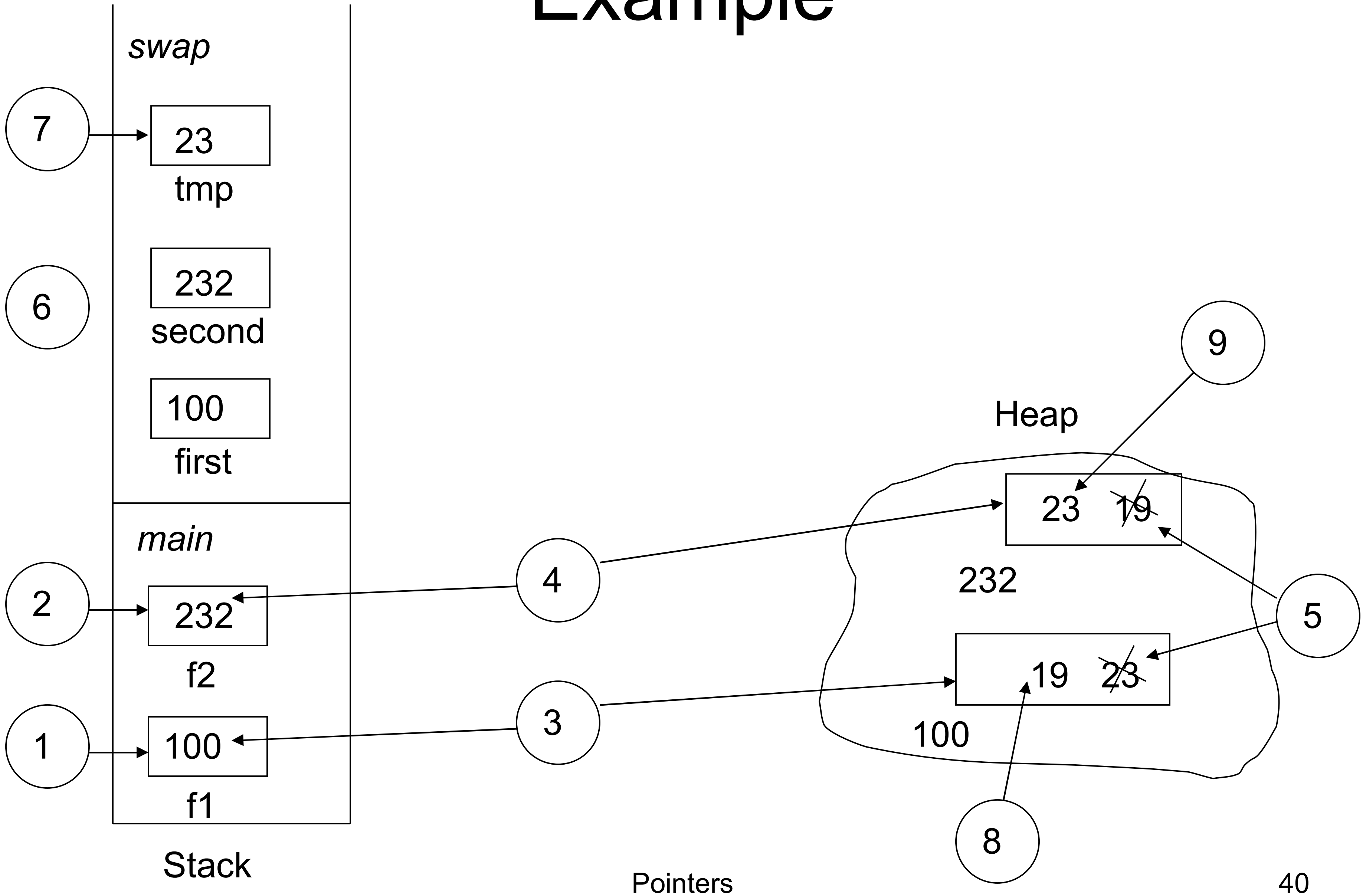
- Pointers work Because they deal with addresses – not value
  - An operator performs an action at the value indicated by the pointer
  - The value in the pointer is an address
- We can find the value of any variable by dereferencing it
  - Simply put an ampersand (&) in front of the variable and you now have the address of the variable

Pointers and Functions

- One Limitation of functions is that they only return a single value
- So how to change multiple values in a single function?
  - Pass in pointers
  - Now any changes that are made to the address being referred to
  - This changes the value for the calling function as well as the called function

```
  #include <stdio.h>

  void swap(float*, float*);

  int main() {
1     float *f1, *f2;
2     f1 = malloc(sizeof(float));
3     f2 = malloc(sizeof(float));

      printf("Enter two numbers: ");
5     scanf("%f%f", f1, f2);  // assume the user types 23 and 19
      printf("f1 = %f\tf2 = %f\n", *f1, *f2);
6     swap(f1, f2);
      printf("After swap: f1 = %f\tf2 = %f\n", *f1, *f2);
      free(f1); free(f2);

      return 0;
  }

  void swap(float* first, float* second) {
7     float tmp = *first;
8     *first = *second;
9     *second = tmp;
  }
```

Example



swap
7
23
tmp
6
232
second
100
first
main
2
232
f2
1
100
f1
Stack
4
3
Heap
9
23 ~~19~~
232
5
19 ~~23~~
100
8

Example

1. Declare a pointer, f1, on stack.
2. Declare a pointer, f2, on stack.
3. Allocate space on the heap for a float and place the address in the pointer variable f1.
4. Allocate space on the heap for a float and place the address in the pointer variable f2.
5. Read values from the user. Hand scanf() the pointers f1 and f2 and the data gets put on the heap.
6. Call the swap function. This pushes a new entry in the stack. Copy the value of the pointers f1 and f2 into first and second.
7. Create a new local variable tmp. Follow the pointer of first and place its value into temp.
8. Follow the pointer of second, grab the value, follow the pointer of first, place grabbed value there.
9. Grab the value from tmp, follow the pointer of second, place the grabbed value there.

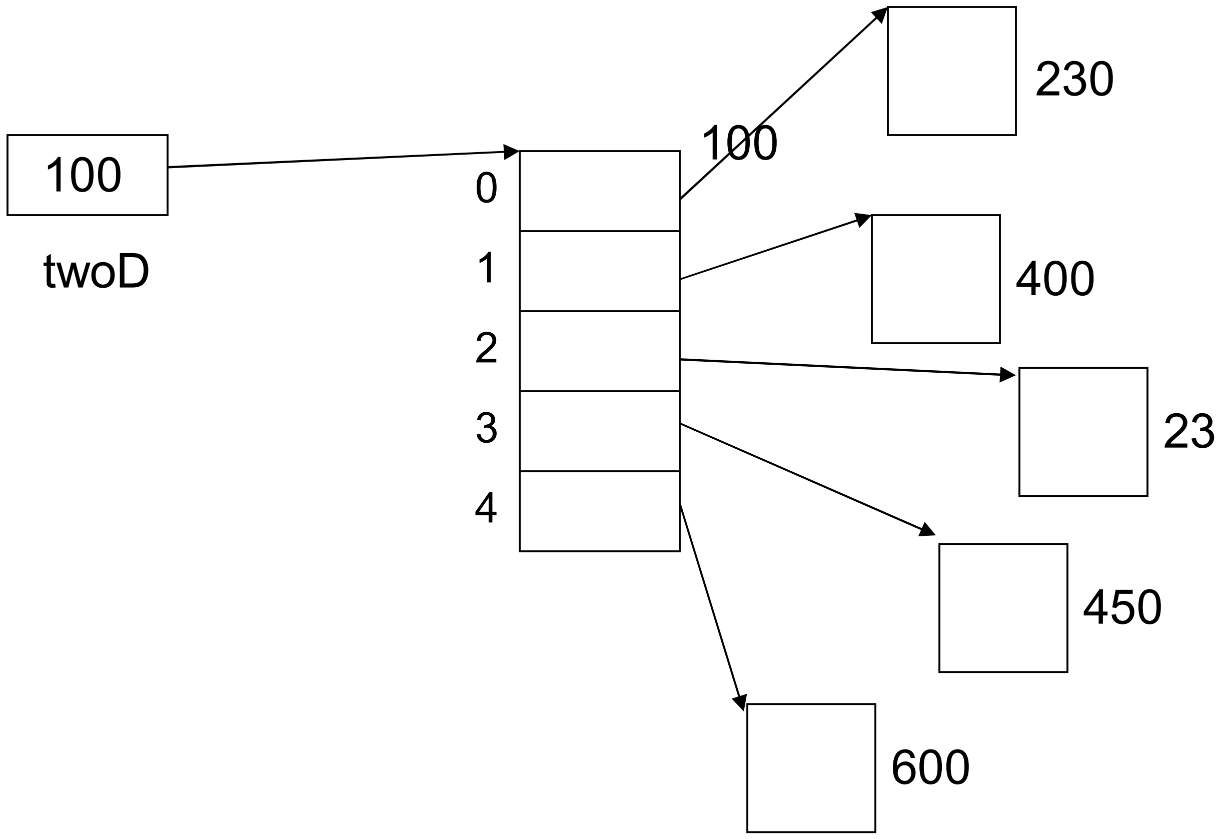
2-D Pointers

- To really make things confusing, you can have pointers to pointers
  - And pointers to pointers to pointers …
- This comes in handy whenever a 2-D array is needed
  - You can also declare 2-D arrays, but these go on the stack
  - If dynamic memory is needed, must use pointers
- Declaring a pointer to a pointer
  - just Put 2 asterisks (*) in front of the variable
  - Example

```
char **names;
```

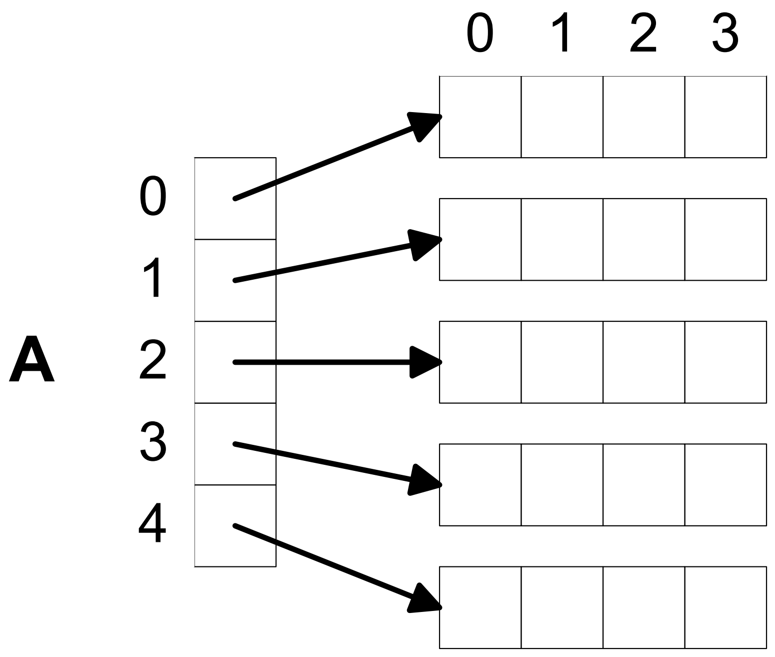
2-D Pointers

- Basic idea



Dynamically Allocating 2D Arrays

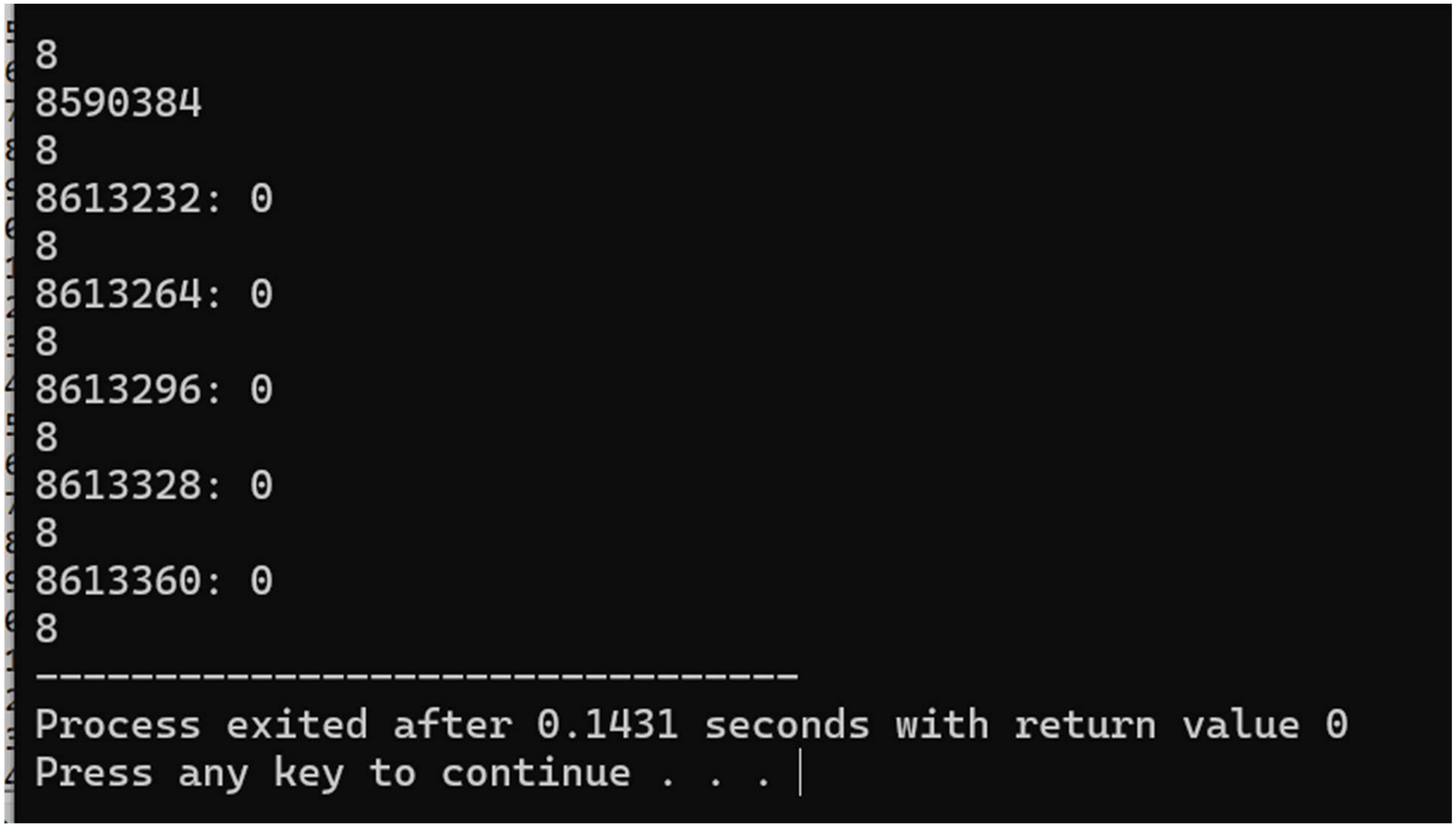
- Can not simply dynamically allocate 2D (or higher) array
- Idea - allocate an array of pointers (first dimension), make each pointer point to a 1D array of the appropriate size
- Can treat result as 2D array



Dynamically Allocating 2D Array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int I;

A = (float **) calloc(5, sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (I = 0; I < 5; I++)
  A[I] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4
   float variables */

/* A[I][J] is the Jth entry in the array that the
   Ith member of A points to */
```

Dynamically Allocating 2D Array



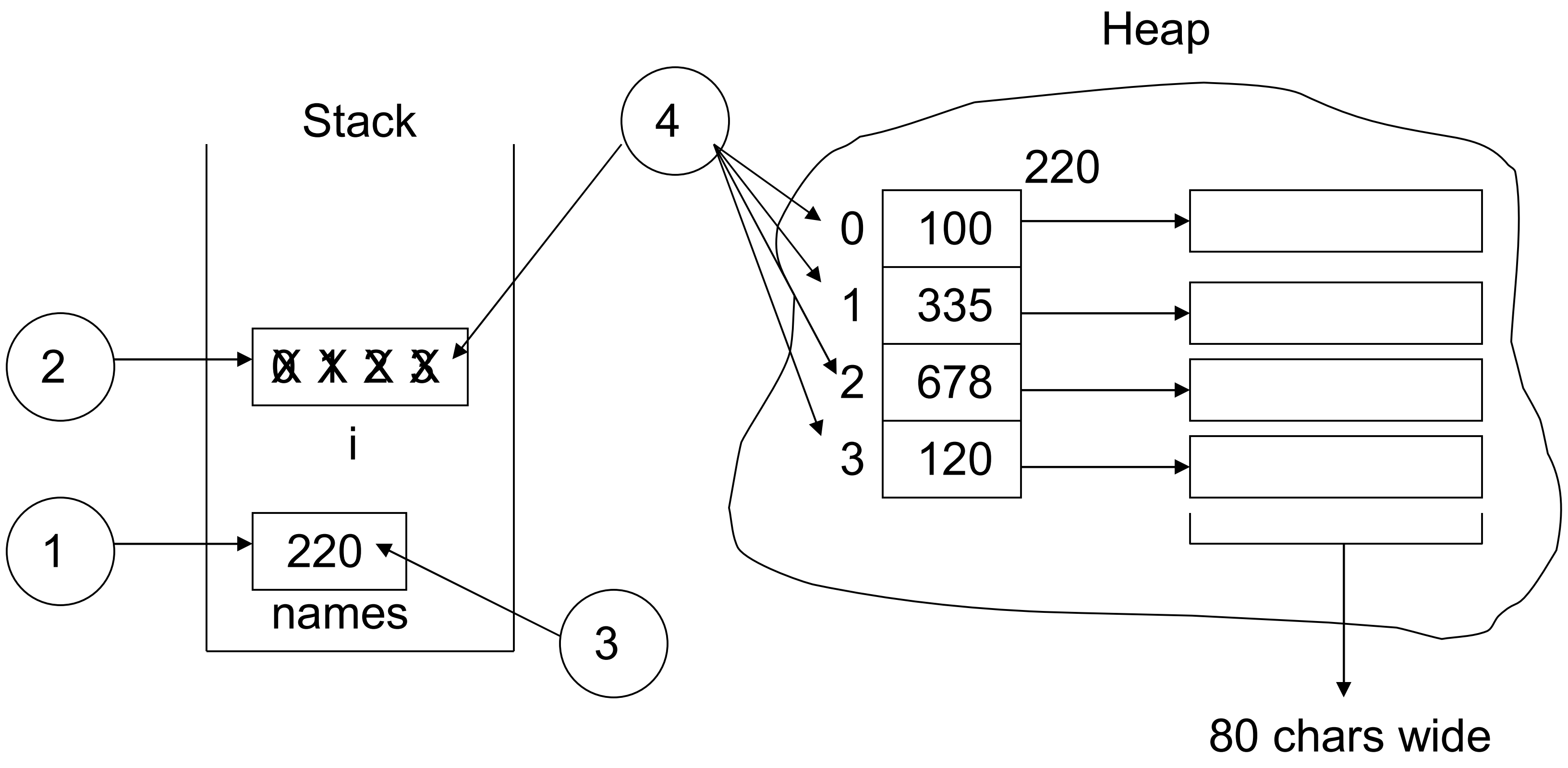
```
8
8590384
8
8613232: 0
8
8613264: 0
8
8613296: 0
8
8613328: 0
8
8613360: 0
8
--------------------------------
Process exited after 0.1431 seconds with return value 0
Press any key to continue . . .
```

Creating a 2-D Array

- Assume a 2-D array of characters is needed
  - This is basically an array of strings
- Assume 4 strings with a max of 80 chars

```
   int main() {
1    char** names;
2    int i;

3    names = (char**)malloc(4 * sizeof(char*));
4    for(i=0; i<4; i++)
       names[i] = (char*)malloc(80 * sizeof(char));

5    for(i=0; i<4; i++)
       free(names[i]);
6    free(names);

     return 0;
   }
```

2-D Arrays



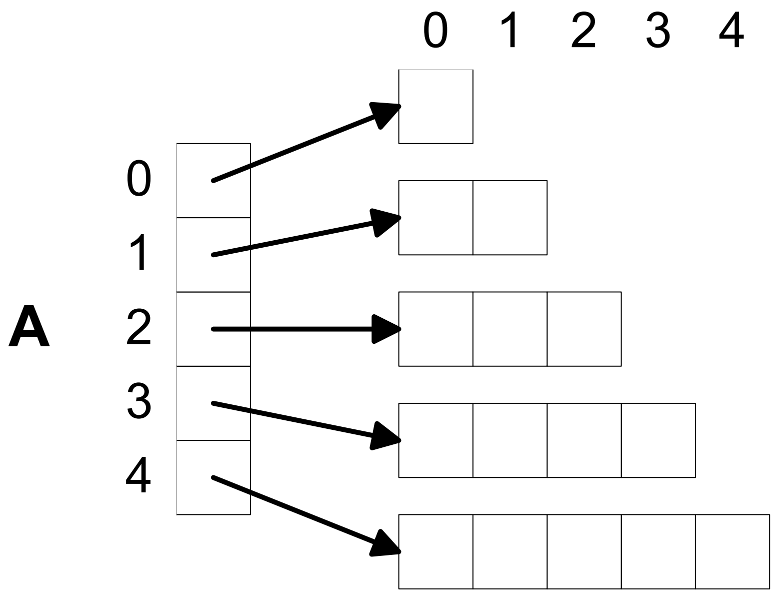
Heap
Stack
4
220
0
100
1
335
2
678
3
120
2
~~0~~ ~~1~~ ~~2~~ ~~3~~ 4
i
1
220
names
3
80 chars wide

2-D Arrays

1. Create a pointer on the stack that will point to a group of pointers
2. Create a local variable on the stack
3. Make names point to an array of 5 pointers to characters. This array is located on the heap.
4. Go through each pointer in the array and make it point at an 80 character array. Each of these 80 character arrays is also located on the heap
5. Freeing each of the 80 character arrays. (not shown on diagram).
6. Free the array of pointers. (not shown on the diagram)

Non-Square 2D Arrays

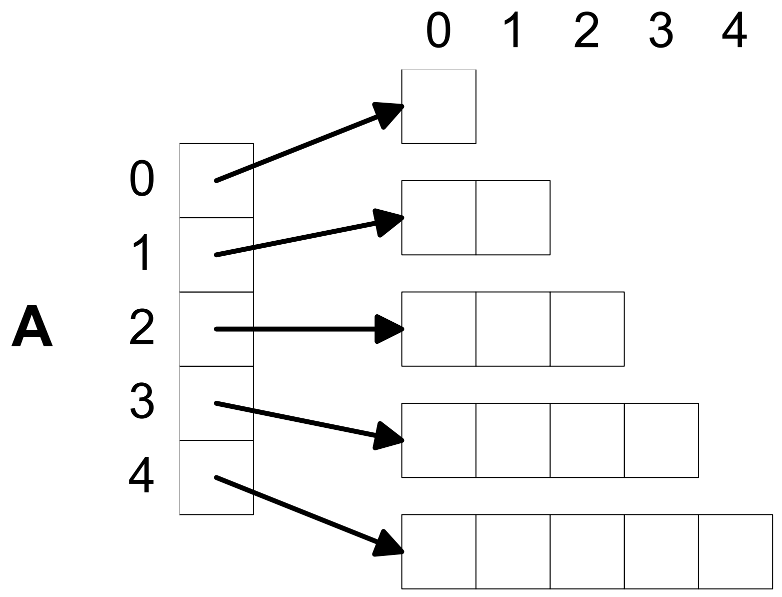
- Can we do like this?



Non-Square 2D Arrays

No need to allocate square 2D arrays:

```
float **A;
int I;

A = (float **) calloc(5,
         sizeof(float *));

for (I = 0; I < 5; I++)
  A[I] = (float **)
          calloc(I+1,
           sizeof(float));
```



argv

- Up until now, main has been written
  - *int main() { … }*
- This is okay, but it is usually written
  - *int main(int argc, char** argv) { … }*
- argc
  - Number of command line arguments being passed in
    - This counts the name of the program
- argv
  - An array of pointers to characters
  - Each string represents one command line argument
  - End of the pointer is a null-terminated string
    - Each of these strings is one of the white space separated command line arguments

Example

```
#include <stdio.h>

int main(int argc, char** argv) {
    int i;

    printf(“Number of arguments: %d\n”, argc);
    for(i=0; i<argc; i++)
      printf(“argument %d: %s”, i, argv[i]);

    return 0;
}
```

Example

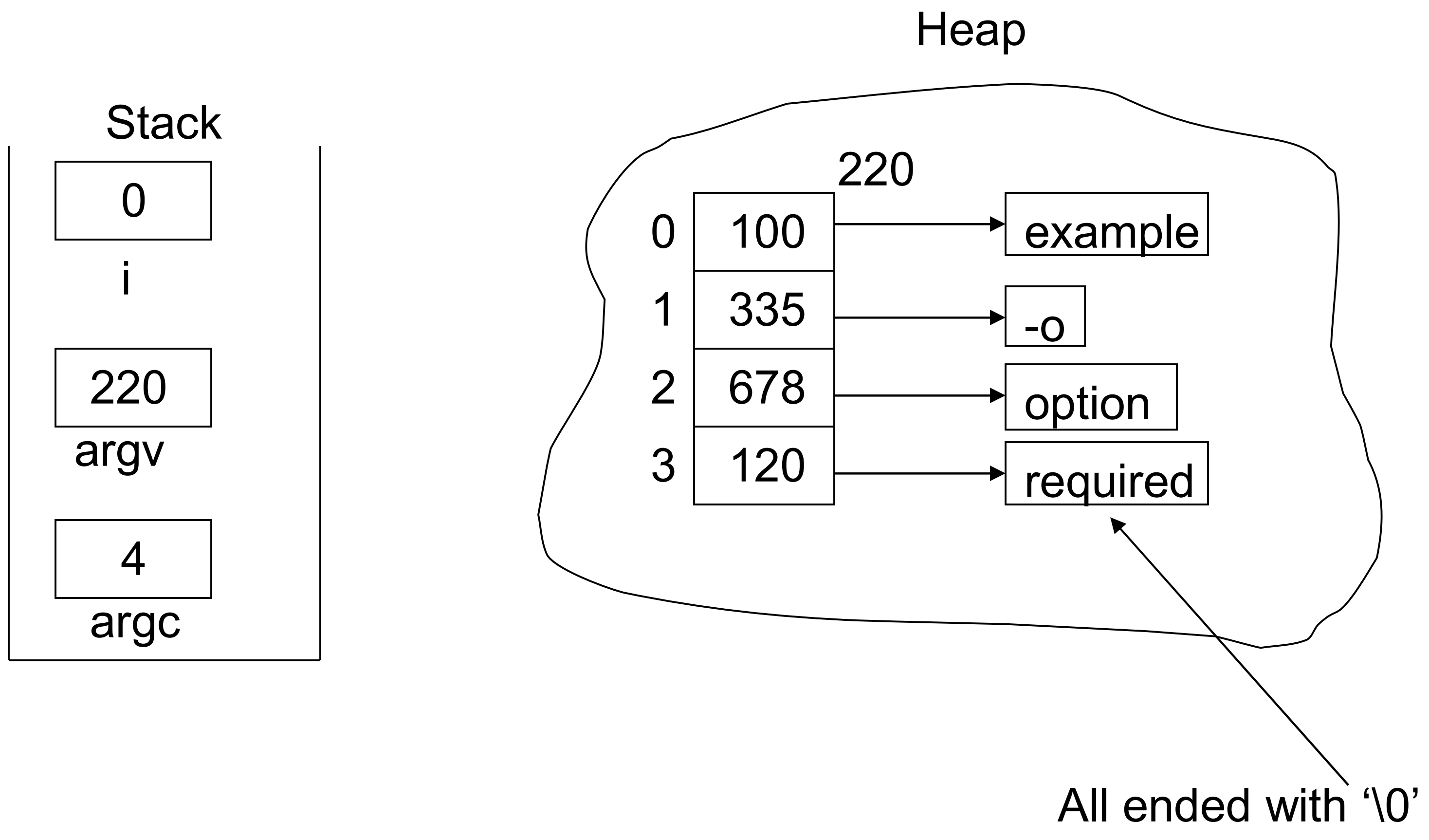
- Given the following command line

```
prompt> example –o option required
```

- The output of the sample program

```
Number of arguments: 4
argument 0: example  //Name of the program
argument 1: ‘-”o”\0’     //First white spaced character separated arg
argument 2: option
argument 3: required
```

Example


Heap
Stack
0
i
220
argv
4
argc
220
0
100
1
335
2
678
3
120
example
-o
option
required
All ended with '\0'

NULL Pointer

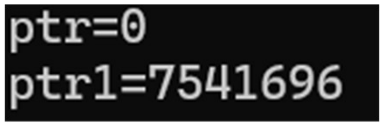
- In C, a null pointer represents a pointer that does not point to any memory address
- It signifies absence of a valid location, commonly used for initialization, error handling, and termination conditions in algorithms
- Null pointers simplify memory management, aiding in identifying uninitialized or deallocated memory access
- They enhance program robustness by enabling explicit detection and handling of invalid memory operations, contributing to code reliability and stability

NULL Pointer: Usages

1. To Avoid Crashing a Program
   - Whenever we declare a pointer in program, the pointer points to some random location in the memory
   - And when we try to retrieve the information at that location, we get some garbage values
   - If you use this garbage value in the program or pass it to any function, your program may crash
   - To avoid the program crash problem, we can use a Null pointer
2. For Freeing(deallocating) Memory
   - Suppose we have a pointer for some data and we do not need that data anymore, so we wanted to free the memory
   - But even after freeing the data, the pointer still points to the same memory location
   - This pointer is called a dangling pointer
   - To avoid this dangling pointer, we can set the pointer to NULL

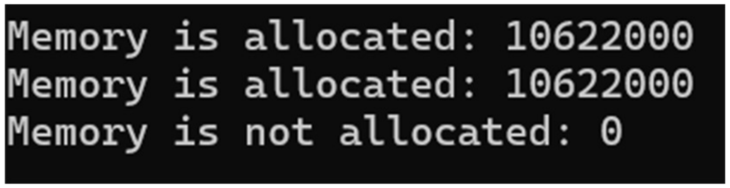
Examples

```
#include<stdio.h>
void fun(int *ptr)
{
    if(ptr==NULL)
        return;
    else
        // function code
}
void main()
{
    int *ptr=NULL;
    printf("%u",ptr);
    int *ptr1;
    printf("\n%u",ptr1);
    fun(ptr);
}
```



```
ptr=0
ptr1=7541696
```

```
#include <stdio.h>
void check(int *ptr)
{   if(ptr==NULL)
            printf("Memory is not allocated\n");
    else
            printf("Memory is allocated\n");
}
int main()
{
    int *ptr=NULL;
    ptr=malloc(4*sizeof(int));
    check(ptr);
    free(ptr);
    check(ptr);
    ptr=NULL;
    check(ptr);
    return 0;
}
```



```
Memory is allocated: 10622000
Memory is allocated: 10622000
Memory is not allocated: 0
```

NULL Pointer: Applications

- Initialization/Default Pointers
  - Use null pointers to initialize pointers when no valid memory address is available.
- Error Handling
  - Employ null pointers to indicate errors, especially in functions returning pointers, such as memory allocation functions.
- Termination Conditions
  - Utilize null pointers as termination conditions in recursive algorithms or linked list traversal.
- Dynamic Memory Allocation
  - Check for null pointers after dynamic memory allocation functions like malloc() or calloc() to handle memory allocation failures gracefully.
- error handling scenarios.

NULL Pointer: Applications

- Sentinel Values
  - Employ null pointers as sentinel values in data structures like trees or graphs to represent the absence of a node or edge.
- Function Pointers
  - Utilize null function pointers to indicate uninitialized or unused function pointers.
- Resource Release
  - Use null pointers to avoid releasing resources like file handles or database connections multiple times in error handling scenarios.

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

```
sum  = *p1 + *p2 ;
prod = *p1 * *p2 ;
prod = (*p1) * (*p2) ;
*p1 = *p1 + 2;
x = *p1 / *p2 + 5 ;
```

Contd.

- What are allowed in C?
  - Add an integer to a pointer.
  - Subtract an integer from a pointer.
  - Subtract one pointer from another (related).
    - If p1 and p2 are both pointers to the same array, them p2–p1 gives the number of elements between p1 and p2.
- What are not allowed?
  - Add two pointers.

    ```
    p1  =  p1 + p2 ;
    ```
  - Multiply / divide a pointer in an expression.

    ```
    p1  =  p2 / 5 ;
    p1  =  p1 – p2 * 10 ;
    ```

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int   *p1,  *p2 ;
int   i,  j;
:
p1  =  p1  +  1 ;
p2  =  p1  +  j ;
p2++ ;
p2  =  p2  –  (i + j) ;
```

- In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.

Contd.

Data Type	Scale Factor
char	1
int	4
float	4
double	8

– If p1 is an integer pointer, then

    p1++

  will increment the value of p1 by 4.

E

Returns no. of bytes required for data type representation

```
#include  <stdio.h>
main()
{
    printf (“Number of bytes occupied by int is %d \n”, sizeof(int));
    printf (“Number of bytes occupied by float is %d \n”, sizeof(float));
    printf (“Number of bytes occupied by double is %d \n”, sizeof(double));
    printf (“Number of bytes occupied by char is %d \n”, sizeof(char));
}
```

Output:

```
Number of bytes occupied by int is  4
Number of bytes occupied by float is  4
Number of bytes occupied by double is  8
Number of bytes occupied by char is  1
```

Structures Revisited

- Recall that a structure can be declared as:

```
struct   stud   {
                    int    roll;
                    char  dept_code[25];
                    float  cgpa;
                };
struct  stud  a, b, c;
```

- And the individual structure elements can be accessed as:

```
a.roll ,  b.roll ,  c.cgpa , etc.
```

Arrays of Structures

- We can define an array of structure records as

```
struct   stud   class[100] ;
```

- The structure elements of the individual records can be accessed as:

```
class[i].roll
class[20].dept_code
class[k++].cgpa
```

Example: Sorting by Roll Numbers

```
#include <stdio.h>
struct   stud
{
      int    roll;
      char  dept_code[25];
      float  cgpa;
 };

main()
{
      struc  stud  class[100], t;
      int  j, k, n;

      scanf  (“%d”, &n);
                        /* no. of students */
```

```
for  (k=0; k<n; k++)
         scanf (“%d %s %f”, &class[k].roll,
               class[k].dept_code,
&class[k].cgpa);
     for  (j=0; j<n-1; j++)
         for  (k=j+1; k<n; k++)
            {
                 if  (class[j].roll > class[k].roll)
                      {
                           t = class[j] ;
                           class[j] = class[k] ;
                           class[k] = t
                       }
              }
    <<<< PRINT THE RECORDS >>>>
}
```

Pointers and Structures

Pointers and Structures

- Consider the declaration:

```
struct   stud   {
                        int    roll;
                        char  dept_code[25];
                        float  cgpa;
                    }    class[100],  *ptr ;
```

  - The name class represents the address of the zero-th element of the structure array.
  - ptr is a pointer to data objects of the type struct stud.
- The assignment

```
ptr  =  class ;
```

  will assign the address of class[0] to ptr.

Pointers and Structures

- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(stud).
  - It is made to point to the next record.
- Once ptr points to a structure variable, the members can be accessed as:

```
ptr –> roll ;
ptr –> dept_code ;
ptr –> cgpa ;
```

  - The symbol “–>” is called the arrow operator.

Example

```
#include <stdio.h>

typedef struct {
            float real;
            float imag;
           } _COMPLEX;
```

```
swap_ref(_COMPLEX *a, _COMPLEX *b)
{
 _COMPLEX tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
}
```

```
print(_COMPLEX *a)
{
  printf("(%f,%f)\n",a->real,a->imag);
}
```

```
main()
{
 _COMPLEX x={10.0,3.0}, y={-20.0,4.0};

  print(&x); print(&y);
  swap_ref(&x,&y);
  print(&x); print(&y);
}
```

```
(10.000000,3.000000)
(-20.000000,4.000000)
(-20.000000,4.000000)
(10.000000,3.000000)
```

A Warning

- When using structure pointers, we should take care of operator precedence.
  - Member operator "." has higher precedence than "*".
    - ptr –> roll and (*ptr).roll mean the same thing.
    - *ptr.roll will lead to error.
  - The operator "–>" enjoys the highest priority among operators.
    - ++ptr –> roll will increment roll, not ptr.
    - (++ptr) –> roll will do the intended thing.

Structures and Functions

- A structure can be passed as argument to a function.
- A function can also return a structure.
- The process shall be illustrated with the help of an example.
  - A function to add two complex numbers.

Example: complex number addition

```
#include  <stdio.h>
struct  complex  {
                        float  re;
                        float  im;
                      };

main()
{
    struct  complex  a, b, c;
    scanf  (“%f %f”, &a.re, &a.im);
    scanf  (“%f %f”, &b.re, &b.im);
    c  =  add (a, b) ;
    printf  (“\n %f %f”, c.re, c.im);
}
```

```
struct  complex  add  (x, y)
struct complex  x, y;
{
     struct  complex  t;

     t.re = x.re + y.re ;
     t.im = x.im + y.im ;
     return (t) ;
}
```

Example: Alternative way using pointers

```
#include  <stdio.h>
struct  complex  {
                        float  re;
                        float  im;
                      };

main()
{
   struct  complex  a, b, c;
   scanf  ("%f %f", &a.re, &a.im);
   scanf  ("%f %f", &b.re, &b.im);
   add (&a, &b, &c) ;
   printf  ("\n %f %f", c,re, c.im);
}
```

```
void  add  (x, y, t)
struct complex  *x, *y, *t;
{
    t->re = x->re + y->re ;
    t->im = x->im + y->im ;
}
```

Common Mistakes

- When using structure pointers, we should take care of operator precedence.
  - Member operator "." has higher precedence than "*".
    - ptr –> roll and (*ptr).roll mean the same thing.
    - *ptr.roll will lead to error.
  - The operator "–>" enjoys the highest priority among operators.
    - ++ptr –> roll will increment roll, not ptr.
    - (++ptr) –> roll will do the intended thing.