Pointers

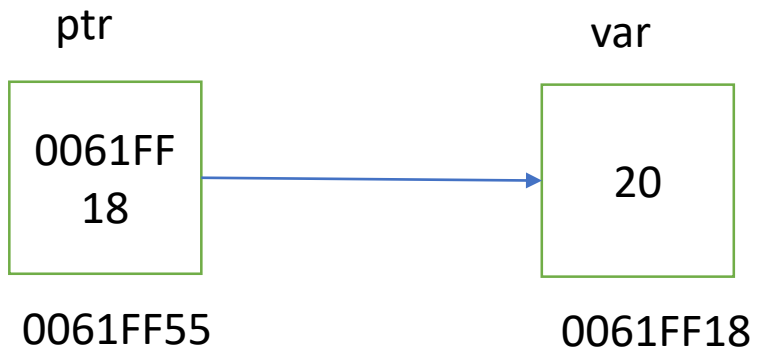
Program to print the address of a variable

```
#include <stdio.h>
void main(void)
{
	int a;
	float b;
	char c;
	printf("Address of a: %p\n", &a);
	printf("Address of b: %p\n", &b);
	printf("Address of c: %p\n", &c);
}
```

Dictionary Definitions

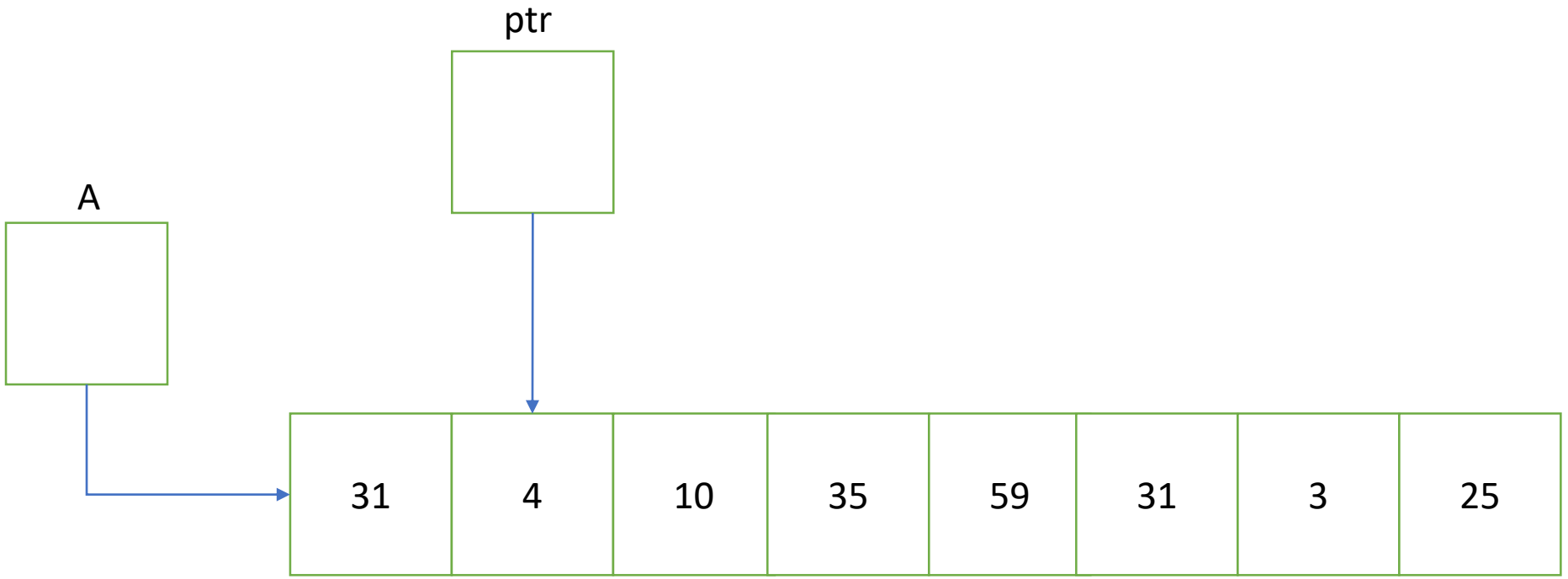
- A special type of variable that contains an address of a memory location.
- Think of pointer as a new data type that holds memory addresses.
- It is associated with the type of data that is contained in the memory location.

```
int var = 20;
int * ptr;
ptr = &var;
```



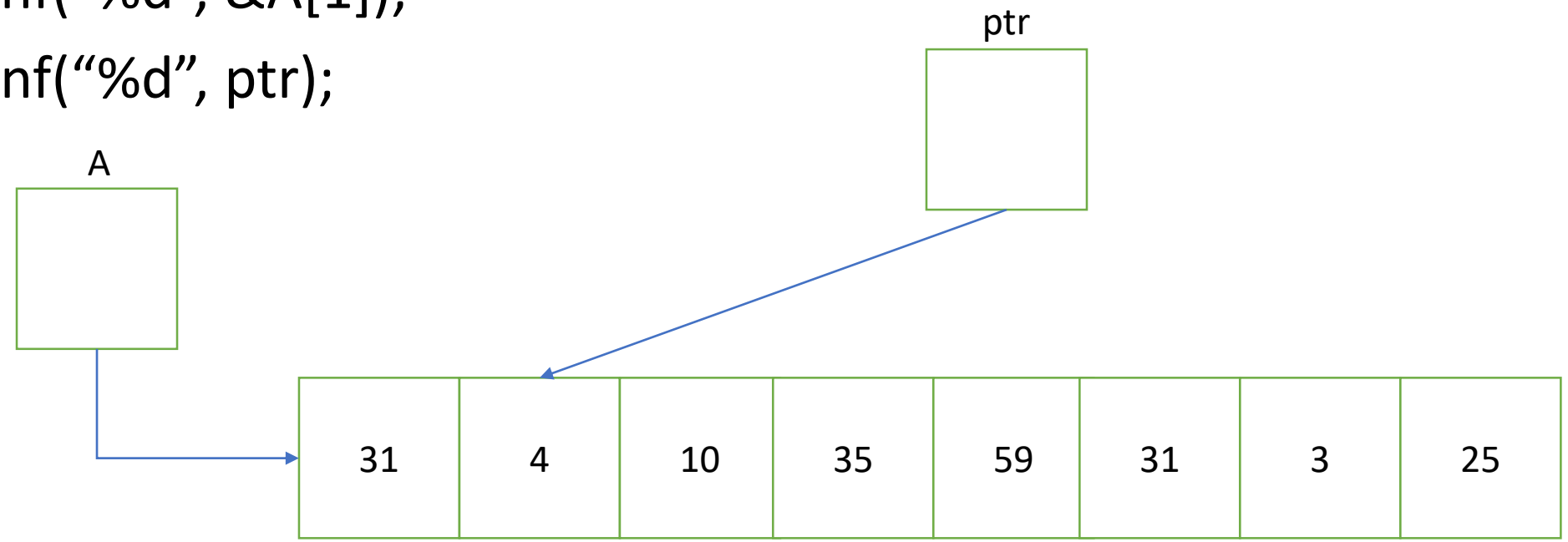
Example I

```
int A[] = {31, 4, 10, 35, 59, 31, 3, 25};
int * ptr = &A[1];
```



Example II

```
int A[] = {31, 4, 10, 35, 59, 31, 3, 25};
int * ptr = &A[1];
scanf("%d", &A[1]);
scanf("%d", ptr);
```



Interesting

- A is of type int [] (array of integer).
- Internally, A stores the pointer to A[0].
- Thus, in order to access an element of an array, you can use int * wherever you used int[]. As per our previous example, you can use *prt in place of A[1].

```
void main() {
        int A[] = {1, 2, 3};
        int * ptr = A;
        printf("%d %d %d\n", A[0], A[1], A[2]);
        printf("%d %d %d\n", ptr[0], ptr[1], ptr[2]);
}
```

Declaration and Dereferencing

```
#include<stdio.h>
void main() {
        int temp = 10;
        int * ptr = &temp;  //declaration (with datatype)
        printf("%d", *ptr);     //de-referencing (anywhere else in the program)
}
```

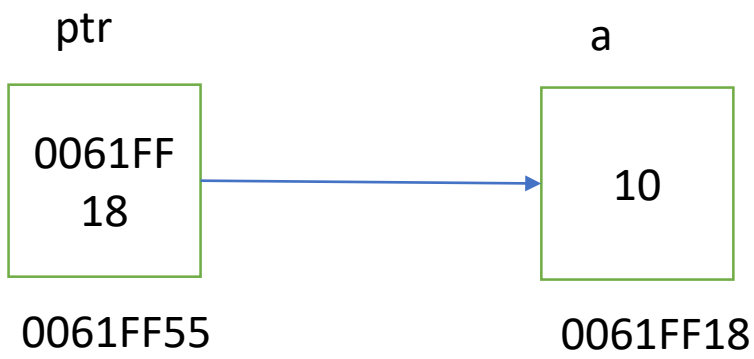
Output:

10

Pointer Arithmetic

- What will be the output of the following:

```
int a = 10;
int * ptr = &a;
*ptr = *ptr + 5;
```



Contd.

```
int A[] = {31, 4, 10, 35, 59, 31, 3, 25};
int * ptr = &A[1];
A[2] = *A + *ptr;
```

Contd.

int num[] = { 1, 22, 16, -1, 23};

num is a pointer pointing to first element num[0].
similarly, num + 1 points to num[1]
and, num + 2 points to num[2]
and, so on.

What will be the output of the following:
printf(“%d %d %d”, *(num+1), *(num+2), *(num+3));

Question

```
void main() {
        char str[] = "Welcome to CSVTU";
        char *ptr = str + 11;          //initialization
        printf("%s\n", ptr);
        printf("%s\n", ptr - 3);
}
```

Array and Pointer

- In C, array names are pointers and can be used interchangeably in most cases.
- However, array names can not be assigned, but pointer variables can be.

```
int ar[10], *b;
b = ar;
b = b + 1;      //This is okay
b++;            //This is okay as well
ar = ar + 2;    //Error
ar = b;         //Error
```

Swap Two Variables

```
void swap(int a, int b) {
        int t;
        t = a; a=b; b=t;
        printf("%d %d\n", a, b);
}
void main() {
        int a = 1, b = 2;
        swap(a, b);
        printf("%d %d\n", a, b);
}
```

```
void swap(int * a, int * b) {
        int t;
        t = *a; *a=*b; *b=t;
        printf(“%d %d”, *a, *b);
}
void main() {
        int a = 1, b = 2;
        swap(&a, &b);
        printf(“%d %d”, a, b);
}
```

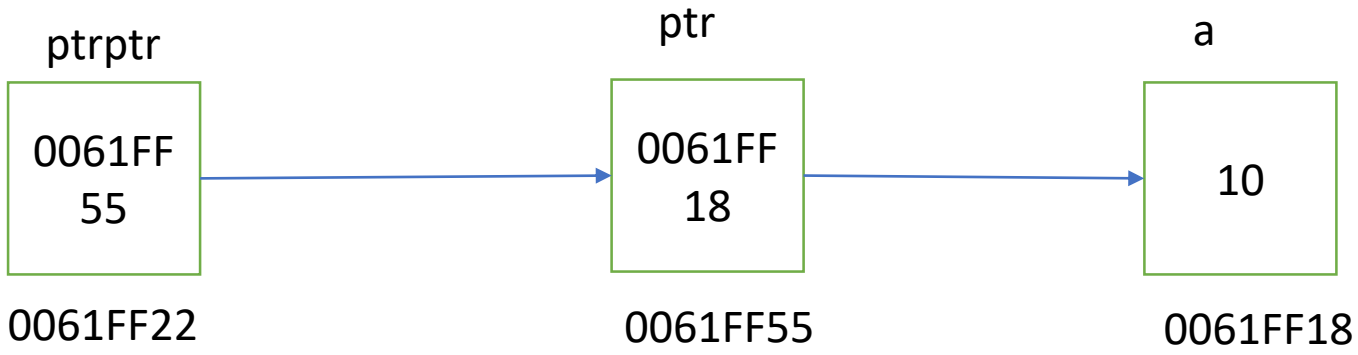
Will the following code work?

```
void swap(int * ptra, int * ptrb) {
        int * ptrt;
        ptrt = ptra;
        ptra = ptrb;
        ptrb = ptrt;
        printf("%d %d", *ptra, *ptrb);
}
void main() {
        int a = 1, b = 2;
        swap(&a, &b);
        printf("%d %d", a, b);
}
```

Pointer to a pointer

- If we have a pointer P to some memory call, P is also stored somewhere in the memory.
- So, we can also talk about the address that stores P.

```
int a = 10;
int * ptr = &a;
int ** ptrptr = &ptr;
```



Static vs Dynamic Memory Allocation

```
printf(“int: %lu”, sizeof(int));
printf(“int: %lu”, sizeof(float));
printf(“int: %lu”, sizeof(double));

printf(“int: %lu”, sizeof(int *));
printf(“int: %lu”, sizeof(float *));
printf(“int: %lu”, sizeof(double *));
```

Since a pointer variable holds the address, thus, the size of pointer variable is same irrespective of data type.

Static Memory Allocation

- When we declare an array, size has to be specified before hand.
- During compilation, the C compiler knows how much space to allocate to the program.
  - space for each variable.
  - space for an array depending on the size.
- This memory is allocated in a part of the memory known as the stack.
- Need to assume worst case scenario
  - May result in wastage of memory

Dynamic Memory Allocation

- There is a way of allocating memory to a program during runtime.
- This is known as dynamic memory allocation.
- Dynamic allocation is done in a part of the memory called the heap.
- You can control the memory allocated depending on the actual input(s).
  - less wastage

malloc() function

- The malloc function is declared in stdlib.h
- Takes as argument an integer (say n, typically > 0)
- Allocates n consecutive bytes of memory space, and
- return the address of the first cell of this memory space.
- The return type is void *

Syntax

(void *) malloc(byte-size)
(void *) malloc(number of blocks X size of a block)

void * does not mean pointer to nothing!

- malloc knows nothing about the memory blocks it has allocated.
- void * means it is pointer to something about which nothing is known.
- The blocks allocated by malloc can be used to store anything provided we allocate enough of them.

Integer Array

```
int * ptr;
ptr = (int *) malloc(10 * sizeof(int));
```

The above code will reserve a memory big enough to store 10 integers. Since, we are going to use it to store integers, we explicitly type caste it to integer pointer.

The sizeof operator makes the code machine independent.

malloc evaluates its arguments at runtime to allocate space. Returns a void *, pointer to first address of allocated space.

Problem

- Write a program that reads two integers, n and m, and stores powers of n from 0 up to m (n^0, n^1, n^2, ..., n^m);

```
#include<stdio.h>
#include<stdlib.h>
int main() {
        int * pow, i, n, m;
        scanf("%d %d", &n, &m);
        pow = (int *) malloc ((m+1) * sizeof(int));
        pow[0] = 1;
        for (i=1; i<=m; i++)
                pow[i] = pow[i-1] * n;
         for (i=0; i<=m; i++)
                printf("%d\n", pow[i]);
}
```

NULL

- A special pointer value to denote “points-to-nothing.”
- C uses the value 0 or name NULL.
- A malloc call can return NULL if it is not possible to satisfy memory request.
  - negative or zero size argument
  - too big size argument

- Uninitialized pointer has GARBAGE value, NOT NULL.
- Memory returned by malloc is not initialized.

calloc() function

- Similar to malloc(), allocates memory for n=element array of size bytes each. Memory is initialized to 0.

Syntax:

(void *) calloc(n, element-size);

```
int * ptr;
ptr = (int *) calloc(n, sizeof(int));
```

realloc() function

- Changes the size of the memory block pointed to by ptr to size bytes.
- Reallocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

realloc(ptr, new-size);

```
int * ptr;
ptr = (int *)calloc(n, sizeof(int));
ptr = realloc(ptr, n*sizeof(int));
```

free()

- Power to allocate memory when needed must be complimented by the responsibility to de-allocate memory when no longer needed!

Syntax:

free(pointer variable);

```
int * ptr;
ptr = (int *) malloc(10 * sizeof(int));
free(ptr);
```

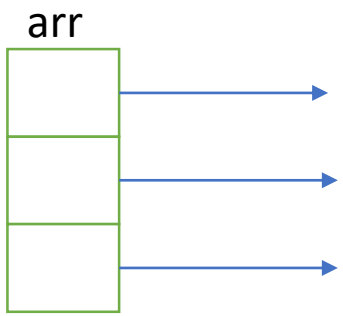
Array of Pointers

Consider the following declaration

```
int * arr[3];
```

How do we read it?

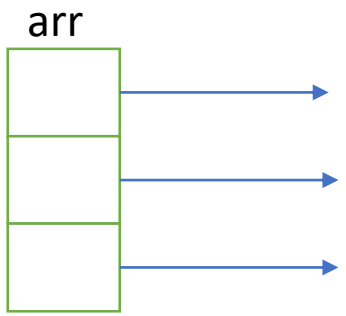
It is an array of pointers to integers.



Array of Pointers (contd)

Dynamic memory equivalent:

```
int **arr;
arr = (int **) malloc ( 3 * sizeof(int *));
```



Things to remember!

- Individual elements in the array arr (arr[0], arr[1], ..., arr[9]) are NOT allocated any space. Uninitialized.
- We need to do it (directly or indirectly) before using them.

```
int j;
for( j=0; j<10; j++)
        arr[j] = (int *) malloc (sizeof(int));
```

Matrix

- Take two matrices as input and find their sum.

```
mat1[3][3], mat2[3][3], sumMat[3][3];
for(int i=0; i<3; i++) {
        for(int j=0; j<3; j++) {
                sumMat[i][j] = mat1[i][j] + mat2[i][j];
        }
}
```

Function returning pointer

char * strdup(const char * s);

- It creates a copy of the string passed as argument.
- Copy is created in dynamically allocated memory block of sufficient size.
- Returns a pointer to the copy created.
- C does not allow returning an array of any type from a function. But we can use pointer to simulate the same.

Code: Alert

```
#include<stdio.h>
int * fun();
int main() {
        printf(“%d”, *func());
}
int * fun() {
        int *p , i;
        p = &i;
        i = 10;
        return p;
}
//This program will give garbage as fun will be removed on completion and thus, i to which pis pointing will be destroyed.
```

Code

```
#include<stdio.h>
int * fun();
int main() {
        printf(“%d”, *func());
}
int * fun() {
        int *p;
        p = (int *)malloc(sizeof(int));
        *p = 10;
        return p;
}
```

Memory Leaks

```
int * a;
a = (int *) malloc(5*sizeof(int));
a = NULL;
```

- Memory is allocated but is never freed. This leads to leakage in memory.
- We can neither free it nor access it.

Dangling Pointers

```
char * ptr = (char *) malloc(sizeof(char));
char * ptr1 = ptr;
free(ptr);
```

- Pointer is pointing to a location that no longer exist.

```
void main() {
        char * dp = NULL;
        {
                char c;
                dp = &c;
        }
}
```

Multi Dimensional Array vs Multi Level Pointer

```
int a[2][3]


int **b;
b = (int **) malloc(2*sizeof(int *));
b[0] = (int *) malloc(3*sizeof(int *));
b[1] = (int *) malloc(3*sizeof(int *));
```