CS1100 – Introduction to Programming

Instructor: Shweta Agrawal
Lecture 28

CS1100 – Introduction to Programming

Instructor: Shweta Agrawal
Lecture 28

- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs `if-else`, `switch`, loops : `while`, `do-while`, `for`.
- `Arrays` and `Strings` in C.
- Functions & modular programming.
- Recursion.

$\Bigg\}$ So far...

CS1100 – Introduction to Programming

Instructor: Shweta Agrawal
Lecture 28

- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs `if-else`, `switch`, loops : `while`, `do-while`, `for`.
- `Arrays` and `Strings` in C.
- Functions & modular programming.
- Recursion.

$\Big\}$ So far...

- Pointers in C, Pass by reference
- Dynamic memory allocation
- Structures in C

$\Big\}$ Up Next...

More on pointers : Segmentation Fault

- `int *ptr1;` //ptr1 is a pointer to an integer

More on pointers : Segmentation Fault

- `int *ptr1;` //ptr1 is a pointer to an integer
- What does ptr1 point to before initialization? garbage

More on pointers : Segmentation Fault

- `int *ptr1;` //ptr1 is a pointer to an integer

- What does ptr1 point to before initialization? garbage
- What is the output of this piece of code?

```
#include<stdio.h>
int main() {
    int count;
    int *countPtr;

    count = *countPtr;
    printf("%d\n", count);
}
```

More on pointers : Segmentation Fault

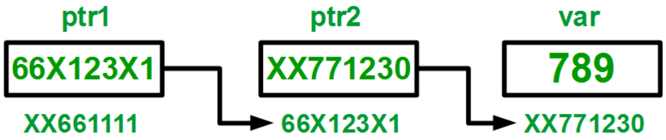
- `int *ptr1;` //ptr1 is a pointer to an integer

- What does ptr1 point to before initialization? garbage
- What is the output of this piece of code?

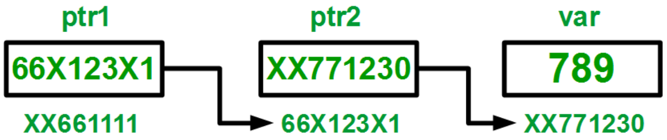
```
#include<stdio.h>
int main() {
    int count;
    int *countPtr;

    count = *countPtr;
    printf("%d\n", count);
}
```

Unpredictable !!

More on Pointers : Pointer to pointers

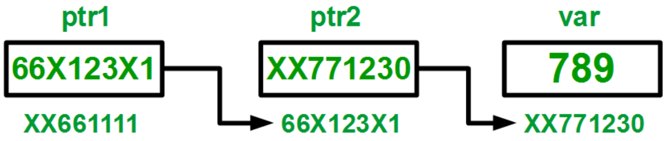


More on Pointers : Pointer to pointers



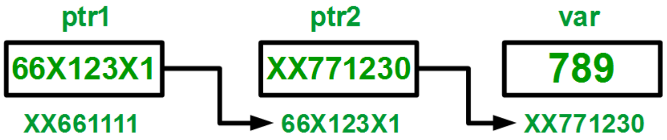
Syntax: `type **ptrname`

More on Pointers : Pointer to pointers



Syntax: `type **ptrname` Example : `int **ptr;`

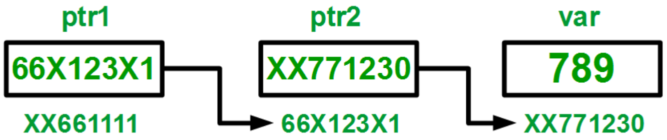
More on Pointers : Pointer to pointers



Syntax: `type **ptrname` Example : `int **ptr;`

```
int var = 789;
int *ptr2;
int **ptr1; // pointer which points to an integer pointer.
```

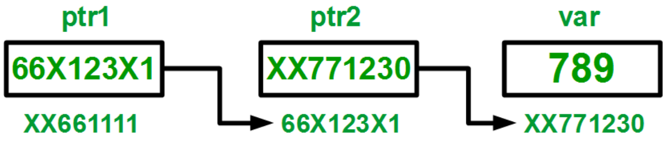
More on Pointers : Pointer to pointers



Syntax: `type **ptrname` Example : `int **ptr;`

```
int var = 789;
int *ptr2;
int **ptr1; // pointer which points to an integer pointer.
ptr2 = &var; // storing address of var in ptr2.
```

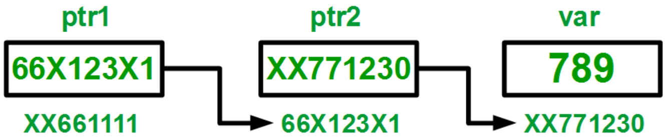
More on Pointers : Pointer to pointers



Syntax: `type **ptrname` Example : `int **ptr;`

```
int var = 789;
int *ptr2;
int **ptr1; // pointer which points to an integer pointer.
ptr2 = &var; // storing address of var in ptr2.
ptr1 = &ptr2; // storing the address of ptr2 in ptr1.
```

More on Pointers : Pointer to pointers



Syntax: `type **ptrname` Example : `int **ptr;`

```
int var = 789;
int *ptr2;
int **ptr1; // pointer which points to an integer pointer.
ptr2 = &var; // storing address of var in ptr2.
ptr1 = &ptr2; // storing the address of ptr2 in ptr1.
```

What are the values of `var`, `*ptr2`, `**ptr1`?

Pointers and Arrays

- In C-language, the name of the array is always a pointer to the beginning of the array.

Pointers and Arrays

- In C-language, the name of the array is always a pointer to the beginning of the array.
- When we declare the array, this pointer is also declared and initialized automatically.

Pointers and Arrays

- In C-language, the name of the array is always a pointer to the beginning of the array.
- When we declare the array, this pointer is also declared and initialized automatically.
- That is, if we declare an array `char board[10];`.

Pointers and Arrays

- In C-language, the name of the array is always a pointer to the beginning of the array.
- When we declare the array, this pointer is also declared and initialized automatically.
- That is, if we declare an array `char board[10];`.
- The derefercing `*board` will gives us the array element `board[0];`

Pointers and Arrays

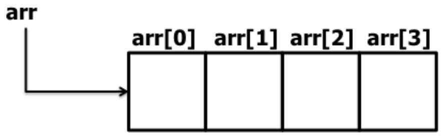
- In C-language, the name of the array is always a pointer to the beginning of the array.
- When we declare the array, this pointer is also declared and initialized automatically.
- That is, if we declare an array `char board[10];`.
- The dereferncing `*board` will gives us the array element `board[0];`
- That is, `&board[0]` is equivalent to `board`.

Pointers and Arrays

- In C-language, the name of the array is always a pointer to the beginning of the array.
- When we declare the array, this pointer is also declared and initialized automatically.
- That is, if we declare an array `char board[10];`.
- The dereferncing `*board` will gives us the array element `board[0];`
- That is, `&board[0]` is equivalent to `board`.
- This pointer `board` can only point to this array and cannot be reassigned.

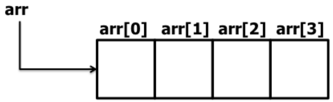
Pointers and Arrays

```
int arr[4];
```



Array access using pointers

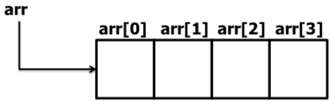
```
int arr[4];
```



- `&arr[0]` is same as `arr`.

Array access using pointers

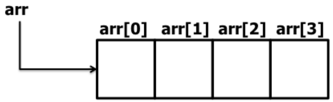
```
int arr[4];
```



- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.

Array access using pointers

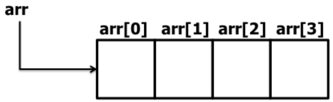
```
int arr[4];
```



- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.

Array access using pointers

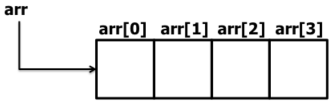
```
int arr[4];
```



- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.
- `&arr[i]` is same as `(arr+i)`.

Array access using pointers

```
int arr[4];
```

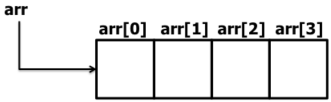


- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.
- `&arr[i]` is same as `(arr+i)`.

- `arr[0]` is same as `*arr`.

Array access using pointers

```
int arr[4];
```

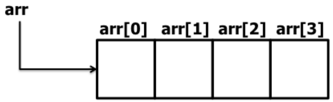


- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.
- `&arr[i]` is same as `(arr+i)`.

- `arr[0]` is same as `*arr`.
- `arr[1]` is same as `*(arr+1)`.

Array access using pointers

```
int arr[4];
```

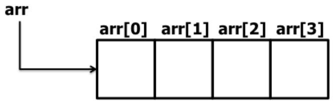


- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.
- `&arr[i]` is same as `(arr+i)`.

- `arr[0]` is same as `*arr`.
- `arr[1]` is same as `*(arr+1)`.
- `arr[2]` is same as `*(arr+2)`.
- `arr[3]` is same as `*(arr+3)`.

Array access using pointers

```
int arr[4];
```



- `&arr[0]` is same as `arr`.
- `&arr[1]` is same as `(arr+1)`.
- `&arr[2]` is same as `(arr+2)`.
- `&arr[3]` is same as `(arr+3)`.
- `&arr[i]` is same as `(arr+i)`.

- `arr[0]` is same as `*arr`.
- `arr[1]` is same as `*(arr+1)`.
- `arr[2]` is same as `*(arr+2)`.
- `arr[3]` is same as `*(arr+3)`.
- `arr[i]` is same as `*(arr+i)`.

Array access using pointers

```c
#include<stdio.h>

int main()
{
    int A[10] = {12, 3, 4, 5, 8, 16, 7, 88, 19, 10};
    int *ptr = &A[0];
    int i;

    for (i=0; i<10; i++) {
        printf("%d\t", A[i]);
        printf("%d\t", *(ptr+i));
        printf("%d\n", *ptr+i);
    }
}
```

Arrays and pointers

```
#include<stdio.h>

int main()
{
    int A[10] = {12, 3, 4, 5, 8, 16, 7, 88, 19, 10};
    int *ptr = &A[0];
    int i;

    for (i=0; i<10; i++) {
        printf("%d\t", A[i]);
        printf("%d\t", *(ptr+i));
        printf("%d\n", *ptr+i);
    }
}
```

string copy using pointers

```
#include<stdio.h>
#include<string.h>
void mystrcpy(char *source, char *dest) {
     int len = strlen(source);
     int i;
     for (i = 0; i < len; i++) {
         dest[i] = source[i];
     }
     dest[i] = '\0';
}

void main() {
    char s1[20] = "This is a string";
    char s2[20];

    mystrcpy(s1, s2);
    printf("%s\n", s2);
}
```

Another string copy using pointers

```
#include<stdio.h>
#include<string.h>
void mystrcpy(char *source, char *dest) {
     while(*source) {
        *dest = *source;
        dest++;
        source++;
     }
     *dest = '\0';
}

void main() {
    char s1[20] = "This is a string";
    char s2[20];

    mystrcpy(s1, s2);
    printf("%s\n", s2);
}
```

Reading input using pointers

```
#include <stdio.h>
int main() {
  int i, x[6], sum = 0;
  printf("Enter 6 numbers: ");
  for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
      scanf("%d", x+i);

  // Equivalent to sum += x[i]
      sum += *(x+i);
  }
  printf("Sum = %d", sum);
  return 0;
}
```

Array of pointers

Goal: We wish to store the names of three students in our class – “Sai”, “Narasimhan”, “Lakshmi” in some appropriate data-type.

Array of pointers

Goal: We wish to store the names of three students in our class – "Sai", "Narasimhan", "Lakshmi" in some appropriate data-type.

- What data-structure will you use?

Array of pointers

Goal: We wish to store the names of three students in our class – "Sai", "Narasimhan", "Lakshmi" in some appropriate data-type.

- What `data-structure` will you use?
  How about `char Names[3][11]`?

Array of pointers

Goal: We wish to store the names of three students in our class – "Sai", "Narasimhan", "Lakshmi" in some appropriate data-type.

- What data-structure will you use?
  How about `char Names[3][11]`?
- Use `char* Names[3]`

Array of pointers

Goal: We wish to store the names of three students in our class – "Sai", "Narasimhan", "Lakshmi" in some appropriate data-type.

- What data-structure will you use?
  How about `char Names[3][11]`?
- Use `char* Names[3]`
  - "Names" is an array of pointers to characters.

Array of pointers

Goal: We wish to store the names of three students in our class – “Sai”, “Narasimhan”, “Lakshmi” in some appropriate data-type.

- What data-structure will you use?
  How about `char Names[3][11]`?
- Use `char* Names[3]`
  - “Names” is an array of pointers to characters.

```
#include<stdio.h>
main() {
    char *Names[3]={"Sai", "Narasimhan", "Lakshmi"};
    int i;
    for (i=0; i<3; i++) {
       printf("%s\n",Names[i]);
    }
}
```

An array of pointers

Goal: Read the three names from standard input.

An array of pointers

Goal: Read the three names from standard input.

```
#include<stdio.h>
main() {
    char *Names[3];
    int i;

    for (i=0; i<3; i++) {
        printf("Enter Name %d\t", i+1);
        scanf("%s", Names[i]);
    }
}
```

An array of pointers

Goal: Read the three names from standard input.

```
#include<stdio.h>
main() {
    char *Names[3];
    int i;

    for (i=0; i<3; i++) {
        printf("Enter Name %d\t", i+1);
        scanf("%s", Names[i]);
    }
}
```

This program is incorrect! There is no memory allocated for `Names[i]`. The program most likely gives a core dump.

An array of pointers – Another program

Goal: Read the three names from standard input.

An array of pointers – Another program

Goal: Read the three names from standard input.

```
#include<stdio.h>
int main() {
    char *Names[3]; char temp[100]; int i;

    for (i=0; i<3; i++) {
        scanf("%s", temp);
        Names[i] = temp;
        printf("String input %s\n",Names[i]);
    }
    for (i=0; i<3; i++) {
        printf("String output %s\n",Names[i]);
    }
}
```

An array of pointers – Another program

Goal: Read the three names from standard input.

```
#include<stdio.h>
int main() {
    char *Names[3]; char temp[100]; int i;

    for (i=0; i<3; i++) {
        scanf("%s", temp);
        Names[i] = temp;
        printf("String input %s\n",Names[i]);
    }
    for (i=0; i<3; i++) {
        printf("String output %s\n",Names[i]);
    }
}
```

This program is still incorrect! All 3 array locations point to the same array temp.

Allocating memory using malloc

- malloc – memory allocator – is a function that allocates memory to the program and returns a pointer to that memory.

Allocating memory using malloc

- malloc – memory allocator – is a function that allocates memory to the program and returns a pointer to that memory.

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

Allocating memory using malloc

- malloc – memory allocator – is a function that allocates memory to the program and returns a pointer to that memory.

-
  ```
  int *ptr;
  ptr = (int *) malloc(sizeof(int));
  ```
- The input to malloc is size of the memory required.
- malloc returns a pointer to the memory allocated – the type of the pointer is `(void *)`.

Allocating memory using malloc

- malloc – memory allocator – is a function that allocates memory to the program and returns a pointer to that memory.

-
  ```
  int *ptr;
  ptr = (int *) malloc(sizeof(int));
  ```
- The input to malloc is size of the memory required.
- malloc returns a pointer to the memory allocated – the type of the pointer is `(void *)`.
- Note the typecasting into `(int *)`.

Allocating memory using malloc

- malloc – memory allocator – is a function that allocates memory to the program and returns a pointer to that memory.

-
  ```
  int *ptr;
  ptr = (int *) malloc(sizeof(int));
  ```
- The input to malloc is size of the memory required.
- malloc returns a pointer to the memory allocated – the type of the pointer is `(void *)`.
- Note the typecasting into `(int *)`.
- Memory obtained using malloc is destroyed only when it is explicitly freed or the program terminates.
- This is unlike variables which are unavailable outside their scope.

An array of pointers – a correct program

Goal: Read the three names from standard input.

An array of pointers – a correct program

Goal: Read the three names from standard input.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main() {
    char *Names[3]; char temp[100]; int i;
    for (i=0; i<3; i++) {
      scanf("%s", temp);
      Names[i]=(char *)malloc(sizeof(strlen(temp)));
      strcpy(Names[i], temp);
      printf("String input %s\n",Names[i]);
    }
    for (i=0; i<3; i++)
      printf("String output %s\n",Names[i]);
    return 0;
}
```

An array of pointers – a correct program

Goal: Read the three names from standard input.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main() {
    char *Names[3]; char temp[100]; int i;
    for (i=0; i<3; i++) {
      scanf("%s", temp);
      Names[i]=(char *)malloc(sizeof(strlen(temp)));
      strcpy(Names[i], temp);
      printf("String input %s\n",Names[i]);
    }
    for (i=0; i<3; i++)
      printf("String output %s\n",Names[i]);
    return 0;
}
```

Note the use of malloc and also the stdlib.h

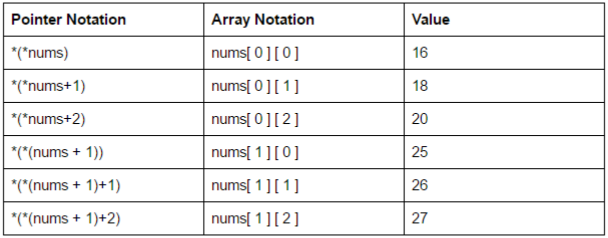
2D Arrays using pointers

Consider the following declaration:
int nums[2][3] $= \{\{16, 18, 20\}, \{25, 26, 27\}\}$;
How to reference these elements using pointers?

2D Arrays using pointers

Consider the following declaration:
int nums[2][3] = {{16, 18, 20}, {25, 26, 27}};
How to reference these elements using pointers?

In general, nums[i][j] is equivalent to *(*(nums+i)+j)

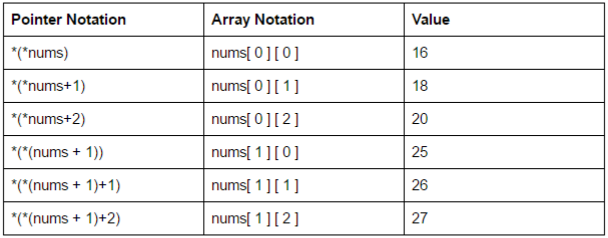


Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

2D Arrays using pointers

Consider the following declaration:
int nums[2][3] $= \{\{16, 18, 20\}, \{25, 26, 27\}\}$;
How to reference these elements using pointers?

In general, nums[i][j] is equivalent to *(*(nums+i)+j)



Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;
- What is char x = *(ptr+3); ?

Some more practice

- Consider the following declaration:
  char * ptr = "geek";
- What is char x = *(ptr+3); ?
- Null Pointer: We can create a null pointer by assigning null value during the pointer declaration.

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;
- What is char x = *(ptr+3); ?
- Null Pointer: We can create a null pointer by assigning null value during the pointer declaration.
- This method is useful when you do not have any address assigned to the pointer.

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;
- What is char x = *(ptr+3); ?
- Null Pointer: We can create a null pointer by assigning null value during the pointer declaration.
- This method is useful when you do not have any address assigned to the pointer.
- Declaration: int *p = NULL

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;
- What is char x = *(ptr+3); ?
- Null Pointer: We can create a null pointer by assigning null value during the pointer declaration.
- This method is useful when you do not have any address assigned to the pointer.
- Declaration: int *p = NULL
- if(ptr) : succeeds if p is not null

Some more practice

- Consider the following declaration:
  char * ptr = “geek”;
- What is char x = *(ptr+3); ?
- Null Pointer: We can create a null pointer by assigning null value during the pointer declaration.
- This method is useful when you do not have any address assigned to the pointer.
- Declaration: int *p = NULL
- if(ptr) : succeeds if p is not null
- if(!ptr) : succeeds if p is null

More practice: Pointers and strings

```
#include <stdio.h>
#include <string.h>
int main()
{
char str[]="Hello Guru99!";
char *p;
p=str;
printf("First character is:%c\n",*p);
p =p+1;
printf("Next character is:%c\n",*p);
printf("Printing all the characters in a string\n");
p=str;  //reset the pointer
for(int i=0;i<strlen(str);i++)
{
printf("%c\n",*p);
p++;
}
return 0;
}
```