CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 27

Sorting algorithms

- Selection and Insertion both require roughly n^2 comparisons on "worst-case" inputs.

Sorting algorithms

- Selection and Insertion both require roughly n^2 comparisons on "worst-case" inputs.
- Insertion sort performs significantly better on nearly sorted input data.

  nearly sorted inputs occur in practice many times

Sorting algorithms

- Selection and Insertion both require roughly n^2 comparisons on "worst-case" inputs.
- Insertion sort performs significantly better on nearly sorted input data.

  nearly sorted inputs occur in practice many times
- Several other sorting algorithms (Quick-sort, Bubble-sort, Heap-sort) exist and many are implemented in libraries.

Link

Recursive Thinking : Sorting an Array

Recursive Thinking : Sorting an Array

- **Iterative Thinking**: We talked about selection sort, insertion sort.

Recursive Thinking : Sorting an Array

- **Iterative Thinking**: We talked about selection sort, insertion sort.
- **Recursive Thinking**: Take out the last element, sort the first $n-1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position.

Recursive Thinking : Sorting an Array

- **Iterative Thinking**: We talked about selection sort, insertion sort.
- **Recursive Thinking**: Take out the last element, sort the first $n-1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position. Hey, this is insertion sort !

Recursive Thinking : Sorting an Array

- **Iterative Thinking**: We talked about selection sort, insertion sort.
- **Recursive Thinking**: Take out the last element, sort the first $n-1$ elements recursively, invoking the same function recursively. And then insert this last element in the right position. Hey, this is insertion sort !
- **(Different) Recursive Thinking**: Divide the array into two halves, recursively sort them, merge the resulting arrays into one array keeping the result to be sorted.
  This is new ! - called **merge sort**.

Recursive version of Insertion Sort

```
#include <stdio.h>
void recursiveInsertionSort(int arr[], int n){
   if (n <= 1)
      return;
   recursiveInsertionSort( arr, n-1 );
   int nth = arr[n-1];
   int j = n-2;
   while (j >= 0 && arr[j] > nth){
      arr[j+1] = arr[j];
      j--;
   }
   arr[j+1] = nth;
}
int main(){
   int array[] = {34, 7, 12, 90, 51};
   int n = sizeof(array)/sizeof(array[0]);
   printf("Unsorted Array:\t");
      for (int i=0; i < n; i++)
   printf("%d ",array[i]);
      recursiveInsertionSort(array, n);
   printf("\nSorted Array:\t");
   for (int i=0; i < n; i++)
      printf("%d ",array[i]);
   return 0;
}
```

Links

- Merge Sort Visualizer (Click on this link).

Pointers

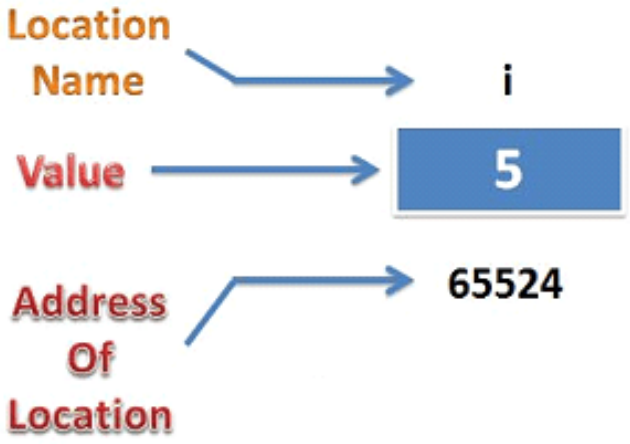
`int x;` : `x` is the name of a variable that holds data which is of type integer.

Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.

Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.



Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.

Pointer variable is a variable whose value is an address.

Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.

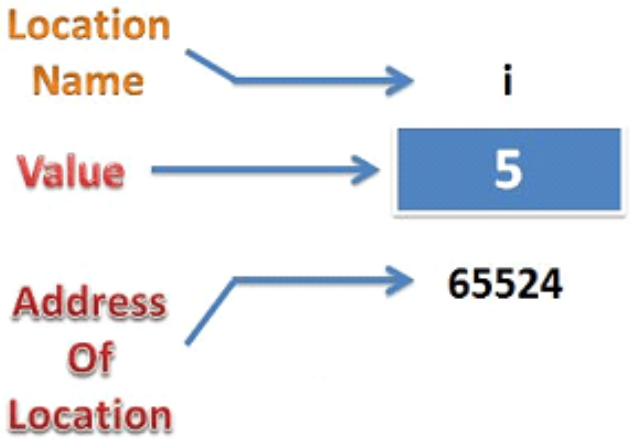
Pointer variable is a variable whose value is an address.

Syntax:

```
data-type *var-name;
```

Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.



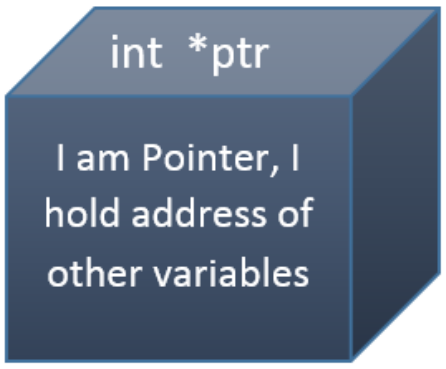
Pointer variable is a variable whose value is an address.

Syntax:

```
data-type *var-name;
```

Example :

```
int *ptr; // ptr is declared to be a variable that can hold the address of an integer variable.
```



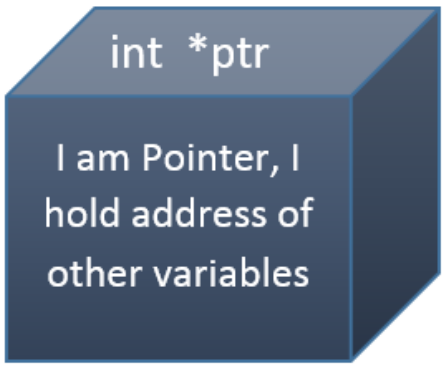
Pointers

`int x;` : `x` is the name of a variable that holds data which is of type integer. Similarly `float f;`, `double d;` and `char c;`.

Pointer variable is a variable whose value is an address.

Syntax:

```
data-type *var-name;
```



Example :

```
int *ptr; // ptr is declared to be a variable that can hold the address of an integer variable.
```

For instance, it can hold the address 65524 which will be the address of the variable `i`.

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
```

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?
Undetermined until pointer is initialized.

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?
Undetermined until pointer is initialized.

- **Assigning an address a pointer** : Using & symbol in front of any variable in C will give the address of the memory location assigned to it.

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?
Undetermined until pointer is initialized.

- **Assigning an address a pointer** : Using & symbol in front of any variable in C will give the address of the memory location assigned to it.
  Example : `ptr1 = &x` assigns the address of the location where variable x is stored to the pointer variable `ptr1`.

Initializing and dereferencing a pointer

```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?
Undetermined until pointer is initialized.

- **Assigning an address a pointer** : Using & symbol in front of any variable in C will give the address of the memory location assigned to it.
  Example : `ptr1 = &x` assigns the address of the location where variable x is stored to the pointer variable `ptr1`.
- **Dereferencing a pointer** : Accessing the variable pointed to by a pointer. Use the * operator.

Initializing and dereferencing a pointer

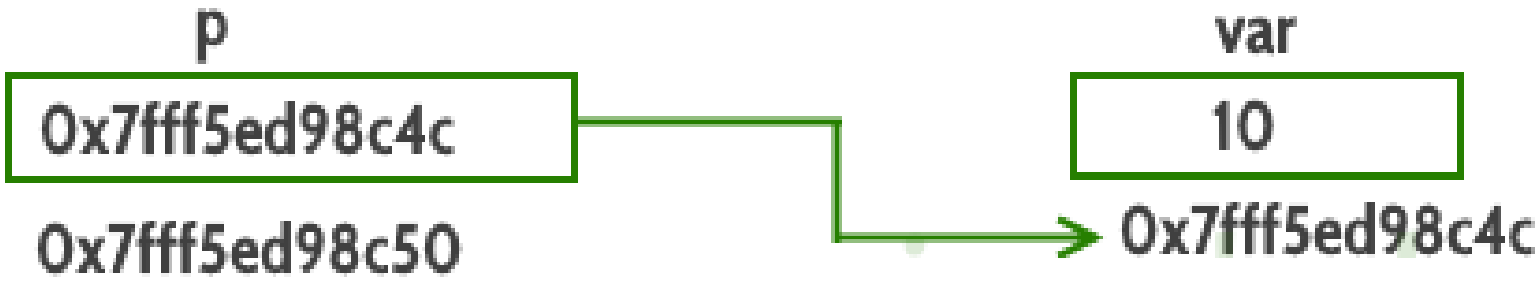
```
int *ptr1; // ptr1 is the address of an integer variable.
char *ptr2; // ptr2 is the address of a character variable.
```

Question : Which integer/character are ptr1/ptr2 pointing to?
Undetermined until pointer is initialized.

- **Assigning an address a pointer** : Using & symbol in front of any variable in C will give the address of the memory location assigned to it.
  Example : `ptr1 = &x` assigns the address of the location where variable x is stored to the pointer variable `ptr1`.
- **Dereferencing a pointer** : Accessing the variable pointed to by a pointer. Use the * operator.
  Example : `*ptr1` gives the value stored in the memory location that the `ptr1` points to (if it does !).

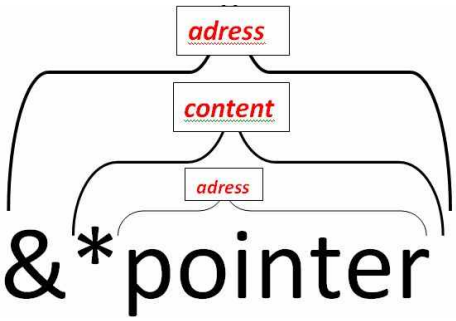
Demo 1 : Initializing and dereferencing

```
int var = 10;
int *p;
p = &var;
```



P is a pointer that stores the address of variable var.
The data type of pointer p and variable var should match because an integer pointer can only hold the address of integer variable.

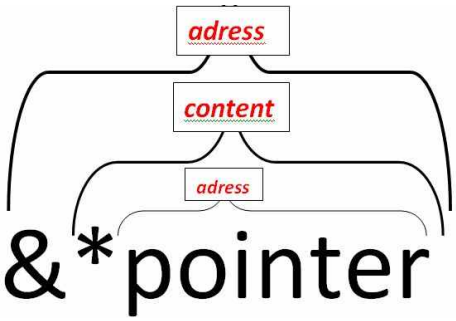
Example 1 : Initializing and dereferencing



- Example :

```
int x, y, *ptr1;
ptr1 = &x; Initializing ptr1
```

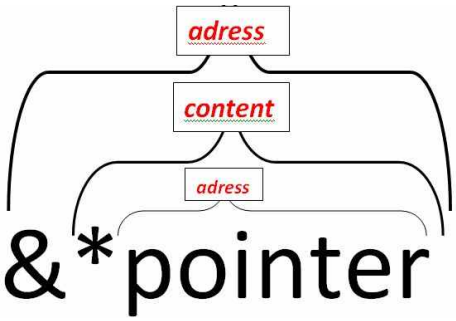
Example 1 : Initializing and dereferencing



- Example :

```
int x, y, *ptr1;
ptr1 = &x; Initializing ptr1
x = 10;
```

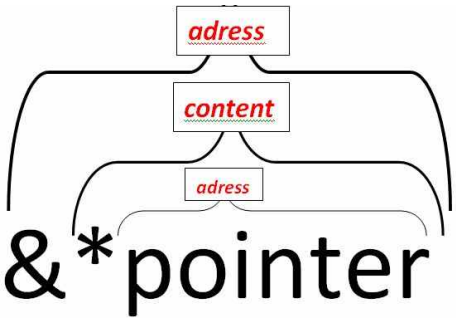
Example 1 : Initializing and dereferencing



- Example :

```
int x, y, *ptr1;
ptr1 = &x; Initializing ptr1
x = 10;
y = *ptr1; Dereferencing ptr1
x++;
```

Example 1 : Initializing and dereferencing

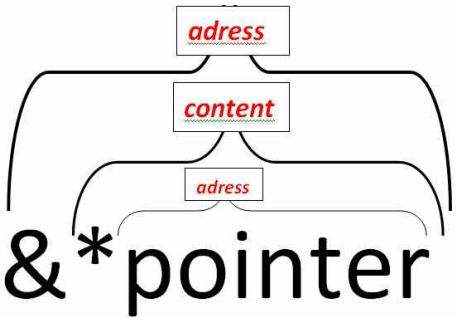


- Example :

```
int x, y, *ptr1;
ptr1 = &x; Initializing ptr1
x = 10;
y = *ptr1; Dereferencing ptr1
x++;
```

- What are the values of `x, y, *ptr1`?

Example 1 : Initializing and dereferencing



- Example :

```
int x, y, *ptr1;
ptr1 = &x; Initializing ptr1
x = 10;
y = *ptr1; Dereferencing ptr1
x++;
```

- What are the values of `x, y, *ptr1`?
- after this, in the same program ...
  suppose we add :

```
ptr1 = &y;
(*ptr1)++;
```

- What are the values of `x, y, *ptr1`?

Example 2 : First programs using pointer manipulation

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

    count = 10;
    countPtr = &count;
    printf("count = %d\n", count);
    printf("count via countPtr = %d\n",  *countPtr);
    printf("address of count = %p\n", &count);
    printf("value of countPtr = %x\n", countPtr);
}
```

Example 3 : Second program using pointers

```
#include <stdio.h>
int main(){
   int* pc;
   int c;
   c=22;
   printf("Address of c:%p\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address stored in the pointer pc:%p\n",pc);
   printf("Content of location pointed to by pc:%d\n\n",*pc);
   c=11;
   printf("Address stored in the pointer pc:%p\n",pc);
   printf("Content of location pointed to by pc:%d\n\n",*pc);
   *pc=2;
   printf("Address of c:%p\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

An Application : Passing Parameters to Functions

Predict the output of the following program:

An Application : Passing Parameters to Functions

Predict the output of the following program:

```
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

An Application : Passing Parameters to Functions

Predict the output of the following program:

```
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

- **Error** : No swapping happens.
- **Our usual solution :** Make `a` and `b` global variables.
- This is convenient but dangerous

An Application : Passing Parameters to Functions

Predict the output of the following program:

```
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

- **Error** : No swapping happens.
- **Our usual solution :** Make `a` and `b` global variables.
- This is convenient but dangerous - because there may be other functions which uses these variables.

An Application : Passing Parameters to Functions

Predict the output of the following program:

```
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

- **Error** : No swapping happens.
- **Our usual solution :** Make a and b global variables.
- This is convenient but dangerous - because there may be other functions which uses these variables.
- Here is a more elegant solution : pass pointers holding the addresses of a and b to the function.

An Application : Passing Parameters to Functions (a review of Swap)

Fixing the error:

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
     int t;
     t = *p1;
     *p1 = *p2;
     *p2 = t;
}
int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
    printf("%d %d",a,b);
}
```

An Application : Passing Parameters to Functions

Fixing the error :

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
     int t;
     t = *p1;
     *p1 = *p2;
     *p2 = t;
}
int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
    printf("%d %d",a,b);
}
```

Original Version :

```
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

What just happened : **Pass by reference**

- In C all arguments are passed by value.

What just happened : **Pass by reference**

- In C all arguments are passed by value. That is, when the function is invoked the values of the arguments are copied to the new variables in the function definition and they are used inside the function.

What just happened : **Pass by reference**

- In C all arguments are passed by value. That is, when the function is invoked the values of the arguments are copied to the new variables in the function definition and they are used inside the function.
- However, some functions need to modify the arguments sent to them.

What just happened : **Pass by reference**

- In C all arguments are passed by value. That is, when the function is invoked the values of the arguments are copied to the new variables in the function definition and they are used inside the function.
- However, some functions need to modify the arguments sent to them.
- This is achieved by passing the address of the variable.

What just happened : **Pass by reference**

- In C all arguments are passed by value. That is, when the function is invoked the values of the arguments are copied to the new variables in the function definition and they are used inside the function.
- However, some functions need to modify the arguments sent to them.
- This is achieved by passing the address of the variable.
  - The caller uses &var-name to pass it to the function.

What just happened : **Pass by reference**

- In C all arguments are passed by value. That is, when the function is invoked the values of the arguments are copied to the new variables in the function definition and they are used inside the function.
- However, some functions need to modify the arguments sent to them.
- This is achieved by passing the address of the variable.
  - The caller uses &var-name to pass it to the function.
  - The function prototype must accept a pointer to the appropriate data type.

Another Swap function

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
     int *temp;
     printf ("before (in function) %p %p\n", p1, p2);
     temp = p1;
     p1 = p2;
     p2 = temp;
     printf ("(after (in function) %p %p\n", p1, p2);
}

int main()
{
    int a = 10, b = 20;
    printf ("in main (before swap) %d %d %p %p\n", a, b, &a, &b);
    swap(&a, &b);
    printf("%d %d\n",a,b);
    printf ("in main (after swap) %d %d %p %p\n", a, b, &a, &b);
}
```

Another Swap function

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
     int *temp;
     printf ("before (in function) %p %p\n", p1, p2);
     temp = p1;
     p1 = p2;
     p2 = temp;
     printf ("(after (in function) %p %p\n", p1, p2);
}

int main()
{
    int a = 10, b = 20;
    printf ("in main (before swap) %d %d %p %p\n", a, b, &a, &b);
    swap(&a, &b);
    printf("%d %d\n",a,b);
    printf ("in main (after swap) %d %d %p %p\n", a, b, &a, &b);
}
```

Why does this function not achieve the desired swap?