CSE 1201 : Pointers

Md. Fahim Arefin
Lecturer, CSE,
University of Dhaka

Introduction

Think of a pointer as a treasure map.

This treasure map doesn’t have the treasure itself but it shows where the treasure is. So, if you follow the map, you can find the treasure!

Pointer: Tells you where the data is (like a treasure map).

Variable: Holds the actual data (like a box holding a treasure).

Introduction

```
int treasure = 5;        // The treasure is the number 5
int *treasure_map;  // We have a treasure map
treasure_map = &treasure; // The treasure map now
   points to where the number 5 is
```

treasure: This is our box that holds the number 5.

treasure_map: This is our map that will tell us where to find the treasure.

&treasure: This is like the unique number on the box. It tells us where "treasure" is stored.

***treasure_map:** If we look at our map, it tells us what the treasure is, which is 5.

Concept of Variable, Value & Address

```
int i = 3;
```

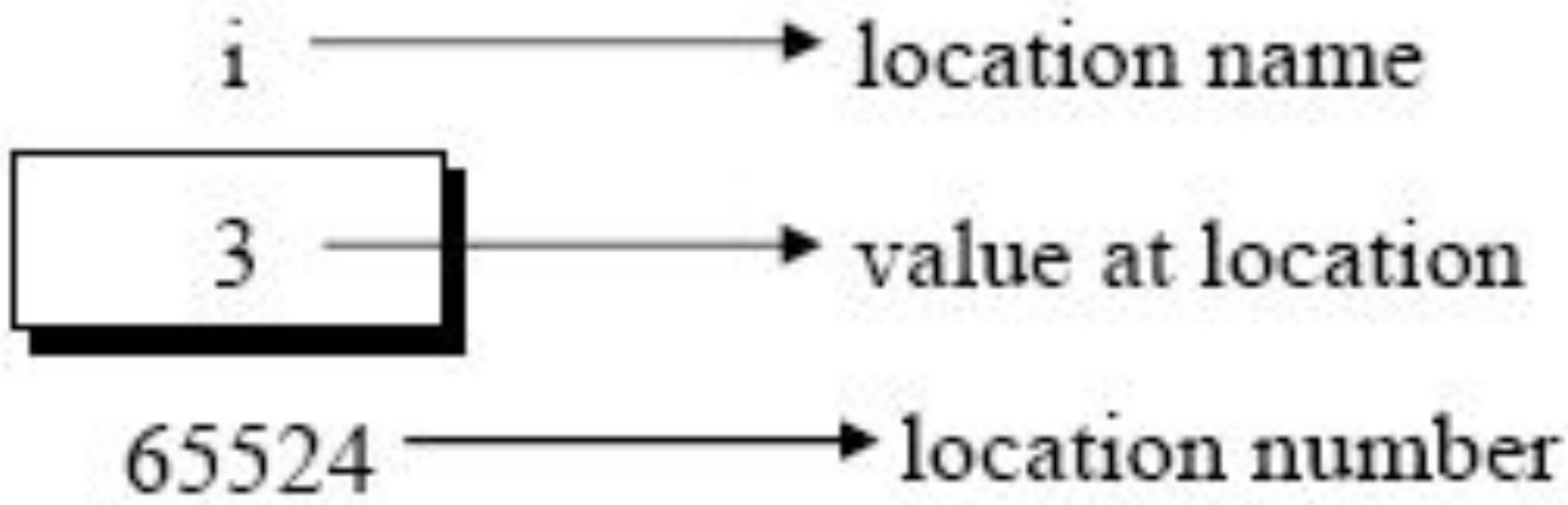
This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 3 at this location.

Concept of Variable, Value & Address



- The important point is, **i**'s address in memory is a number.

Concept of Variable, Value & Address

```
int main(){
    int i =3;
    printf("Address of i = %d\n",&i);
    printf("Address of i = %u\n",&i);
    printf("Address of i = %llu\n",&i);
    printf("Address of i = %p\n",&i);
    printf("Value of i = %d\n",i);
}
```

Output :

```
Address of i = 1798960636
Address of i = 1798960636
Address of i = 6093927932
Address of i = 0x16b39f5fc
Value of i = 3
```

Concept of Variable, Value & Address

- The other pointer operator available in C is ‘*’
- called ‘value at address’ operator
- It gives the value stored at a particular address
- The ‘value at address’ operator is also called ‘indirection’ operator

Concept of Variable, Value & Address

```
main( )
{
        int  i = 3 ;

        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

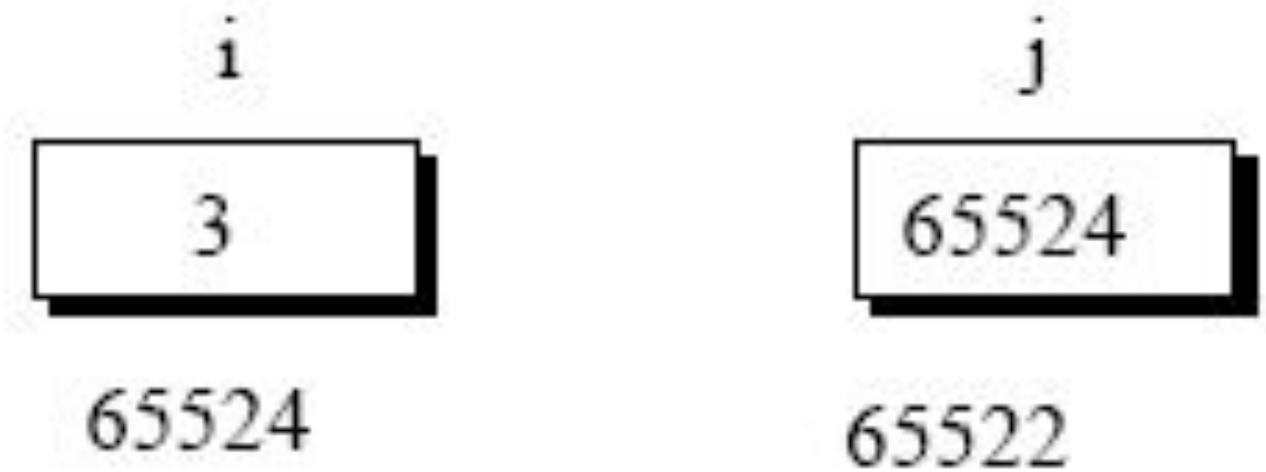
Concept of Pointer

- The expression **&i** gives the address of the variable **i**
- This address can be stored in a variable

  j = &i ;

- But remember that **j** is not an ordinary variable like any other integer variable
- It is a variable that contains the address of other variable

Concept of Pointer



- As you can see, **i**'s value is 3 and **j**'s value is **i**'s address

Concept of Pointer

- We can't use **j** in a program without declaring it
- since **j** is a variable that contains the address of **i**, it is declared as,

  int *j ;

- This declaration tells the compiler that **j** will be used to store the address of an integer value
- In other words **j** points to an integer

Concept of Pointer

- **int *j** would mean, the value at the address contained in **j** is an **int**

Concept of Pointer

```
main( )
{
        int  i = 3 ;
        int  *j ;

        j = &i ;
        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nAddress of i = %u", j ) ;
        printf ( "\nAddress of j = %u", &j ) ;
        printf ( "\nValue of j = %u", j ) ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nValue of i = %d", *( &i ) ) ;
        printf ( "\nValue of i = %d", *j ) ;
}
```

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Concept of Pointer

Look at the following declarations,

1. int *alpha ;
2. char *ch ;
3. float *s ;

- The declaration **float *s** does not mean that **s** is going to contain a floating-point value
- **s** is going to contain the address of a floating-point value

Concept of Pointer

- Pointer, we know is a variable that contains address of another variable
- Now this variable itself might be another pointer
- Thus, we now have a pointer that contains another pointer's address

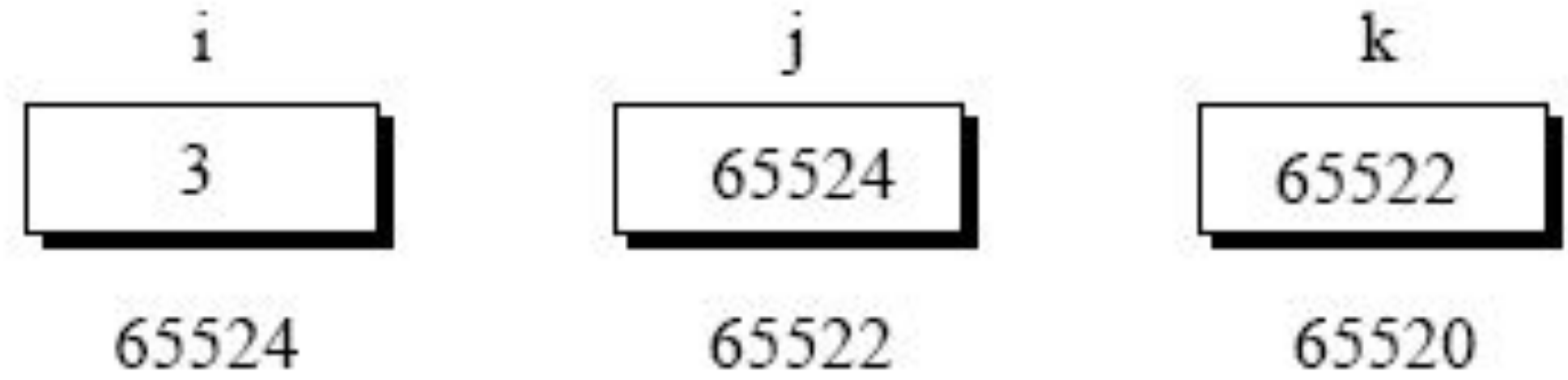
Concept of Pointer

```
int i =3;
int *j,**k;
j = &i;
k = &j;
printf("\nAddress of i = %u",&i);
printf("\nAddress of i = %u",j);
printf("\nAddress of i = %u",*k);
printf("\nAddress of j = %u",&j);
printf("\nAddress of j = %u",k);
printf("\nAddress of k = %u",&k);

printf("\nValue of i = %d",i);
printf("\nValue of j = %u",j);
printf("\nValue of k = %u",k);

printf("\nValue of i = %d",*(&i));
printf("\nValue of i = %d",*j);
printf("\nValue of i = %d\n",**k);
```

Output :

```
Address of i = 1841100284
Address of i = 1841100284
Address of i = 1841100284
Address of j = 1841100272
Address of j = 1841100272
Address of k = 1841100264
Value of i = 3
Value of j = 1841100284
Value of k = 1841100272
Value of i = 3
Value of i = 3
Value of i = 3
```

Concept of Pointer



- i is an ordinary **int**
- **j** is a pointer to an **int**
- **k** is a pointer to an integer pointer

Methods to pass arguments

Arguments can generally be passed to functions in one of the two ways:

(a) sending the values of the arguments

(b) sending the addresses of the arguments

Call by value

- ‘value’ of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function
- changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function

Example : Sharing Photos of Treasure

Call by value

```
main( )
{
      int  a = 10, b = 20 ;

      swapv ( a, b ) ;
      printf ( "\na = %d b = %d", a, b ) ;
}

swapv ( int  x, int  y )
{
      int  t ;

      t = x ;
      x = y ;
      y = t ;

      printf ( "\nx = %d y = %d", x, y ) ;
}
```

```
x = 20 y = 10
a = 10 b = 20
```

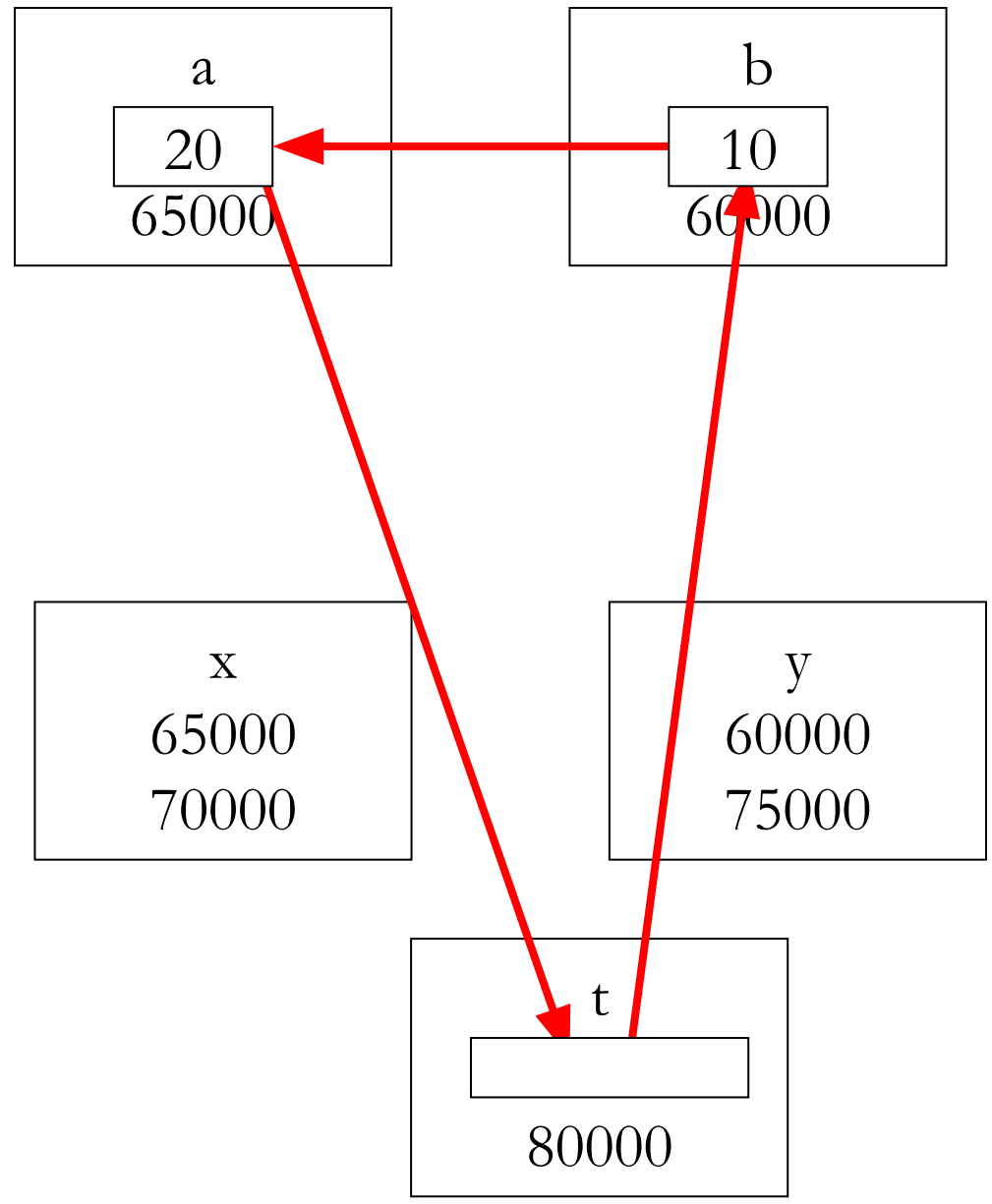
Call by Reference

- the addresses of actual arguments in the calling function are copied into formal arguments of the called function
- using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them

Example : Sharing the Treasure Map

Call by Reference

```
main( )
{
      int  a = 10, b = 20 ;

      swapr ( &a, &b ) ;
      printf ( "\na = %d b = %d", a, b ) ;
}

swapr( int  *x, int  *y )
{
      int  t ;

      t = *x ;
      *x = *y ;
      *y = t ;
}
```



Call by Reference

- Using a call by reference intelligently we can make a function return more than one value at a time, which is not possible ordinarily

Call by Reference

```
main( )
{
      int  radius ;
      float  area, perimeter ;

      printf ( "\nEnter radius of a circle " ) ;
      scanf ( "%d", &radius ) ;
      areaperi ( radius, &area, &perimeter ) ;

      printf ( "Area = %f", area ) ;
      printf ( "\nPerimeter = %f", perimeter ) ;
}

areaperi ( int  r, float  *a, float  *p )
{
      *a = 3.14 * r * r ;
      *p = 2 * 3.14 * r ;
}
```

```
Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000
```

What is the output?

```
void demo(int i,int j){
    i = i * i;
    j = j * j;
}

int main(){
   int i = 5, j = 2;
   demo(i,j);
   printf("%d %d\n",i,j);
}
```

What will be the output?

```
void demo(int *i,int *j){
    *i = *i * *i;
    *j = *j * *j;
}

int main(){
   int i = 4, j = 2;
   demo(&i,&j);
   printf("%d %d\n",i,j);
}
```

What will be the output?

```
void demo(int *i,int j){
    *i = *i * *i;
    j = j *j;
}

int main(){
   int i = 4, j = 2;
   demo(&i,j);
   printf("%d %d\n",i,j);
}
```

What will be the output?

```
int main(){
    int a, *b, **c, ***d, ****e;
    a = 10;
    b = &a;
    c = &b;
    d = &c;
    e = &d;
    printf("%d %d %d",a,a+*b, **c+***d+****e);
}
```

Passing Array Elements to Functions

- An array of pointers can even contain the addresses of other arrays
- Array elements can be passed to a function by calling the function by value, or by reference
- In the call by value we pass values of array elements to the function
- in the call by reference we pass addresses of array elements to the function

Passing Array Elements to Functions: Call by value

```
void disp(int m){
    printf("%d ",m);
}

int main(){
    int i;
    int marks[]={55,65,75,56, 78, 78,90};
    for (i=0;i<=6;i++){
        disp(marks[i]);
    }
}
```

```
55 65 75 56 78 78 90
```

Passing Array Elements to Functions: Call by reference

```
void disp(int *n){
    printf("%d ",*n);
}

int main(){
    int i;
    int marks[]={55,65,75,56, 78, 78,90};
    for (i=0;i<=6;i++){
        disp(&marks[i]);
    }
}
```

Pointer Arithmetic

```
int main() {
    int i = 3, *x;
    float j = 1.5, *y;
    char k = 'c',*z;

    printf("Value of i = %d\n",i);
    printf("Value of j = %f\n", j);
    printf("Value of k = %c\n",k);

    x = &i;
    y = &j;
    z = &k;

    printf("Original address in x = %u\n",x);
    printf("Original address in y = %u\n",y);
    printf("Original address in z = %u\n",z);

    x++;
    y++;
    z++;

    printf("New address in x = %u\n",x);
    printf("New address in y = %u\n",y);
    printf("New address in z = %u\n",z);
}
```

Output :

```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 1842230780
Original address in y = 1842230764
Original address in z = 1842230751
New address in x = 1842230784
New address in y = 1842230768
New address in z = 1842230752
```

Pointer Arithmetic

- Addition of a number to a pointer

```
int i =4, *j,*k;
j = &i;
j = j +1;
j = j+ 9;
k = j +3;
printf("%d %d %d\n",&i,j,k);
```

Output :

1829910012
1829910052
1829910064

Pointer Arithmetic

- Subtraction of a number from a pointer

```
int main() {
    int i =4, *j,*k;
    j = &i;
    j = j -2;
    j = j - 5;
    k = j - 6;
    printf("%d %d %d\n",&i,j,k);
}
```

Output :

1802253820
1802253792
1802253768

Pointers and Arrays

```
#include <stdio.h>

int main() {
    int num[] = {24,34, 12, 44, 56, 17};
    int i ;
    for (i = 0; i<=5;i++){
        printf("\n Element No. %d",i);
        printf("\n Address = %u",&num[i]);
    }
}
```

Output :

```
Element No. 0
 Address = 1806841312
Element No. 1
Address = 1806841316
Element No. 2
Address = 1806841320
Element No. 3
Address = 1806841324
Element No. 4
Address = 1806841328
Element No. 5
Address = 1806841332
```

What do we know so far?

- Array elements are always stored in contiguous memory locations.
- A pointer when incremented always points to an immediately next location of its type.

Pointer and Array

- Our next two programs show ways in which we can access the elements of this array.

Pointer and Array

```
int main() {
    int num[] = {24,34, 12, 44, 56, 17};
    int i ;
    for (i = 0; i<=5;i++){
        printf("\n Address = %llu",&num[i]);
        printf("\n Element %d",num[i]);
    }
}
```

Output :

```
Address = 6129300960
 Element 24
 Address = 6129300964
 Element 34
 Address = 6129300968
 Element 12
Address = 6129300972
Element 44
Address = 6129300976
Element 56
Address = 6129300980
Element 17
```

Pointer and Array

```
int main() {
    int num[] = {24,34, 12, 44, 56, 17};
    int i, *j ;
    j = &num[0];
    for (i = 0; i<=5;i++){
        printf("\n Address = %llu",j);
        printf("\n Element %d",*j);
        j++;
    }
}
```

Output :

```
Address = 6093977056
 Element 24
 Address = 6093977060
 Element 34
 Address = 6093977064
 Element 12
Address = 6093977068
Element 44
Address = 6093977072
Element 56
Address = 6093977076
Element 17
```

Passing entire array to function

```
void display(int *j, int n){
    int i;
    for(i=0;i<n-1;i++){
        printf("element = %d\n",*j);
        j++;
    }
}
```

```
int main() {
    int num[] = {24,34, 12, 44, 56, 17};
    display(&num[0],6);
}
```

Output :

element = 24
element = 34
element = 12
element = 44
element = 56

Similar Statements!

```
display(&num[0],6);
display(num,6);
```

Any Questions?

Thank You