CS - 114 : Computer Workshop

Prof. Chamakuri Nagaiah
Mahindra-École Centrale, Hyderabad
nagaiah.chamakuri@mechyd.ac.in

Introduction : Pointers

- A pointer is a variable that represents the location (rather than the value) of a data item.

Introduction : Pointers

- A pointer is a variable that represents the location (rather than the value) of a data item.
- They have a number of useful applications.
  - Enables us to access a variable that is defined outside the function.
  - Can be used to pass information back and forth between a function and its reference point.
  - More efficient in handling data tables.
  - Reduces the length and complexity of a program.
  - Sometimes also increases the execution speed.

Basic Concept

- In memory, every stored data item occupies one or more contiguous memory cells (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.

  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Basic Concept

- In memory, every stored data item occupies one or more contiguous memory cells (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.

  - Since every byte in memory has a unique address, this location will also have its own (unique) address.
- Consider the statement

  `int xyz = 15;`

  – This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 15 in that location.

  – Suppose that the address location chosen is 5830.

xyz	$\rightarrow$ variable
15	$\rightarrow$ value
5830	$\rightarrow$ address

Basic Concept

- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
    - Such variables that hold memory addresses are called pointers.
    - Since a pointer is a variable, its value is also stored in some memory location.

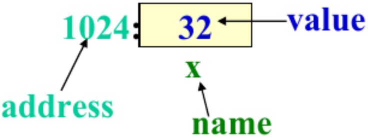
Basic Concept

- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
  - Such variables that hold memory addresses are called pointers.
  - Since a pointer is a variable, its value is also stored in some memory location.
- Suppose we assign the address of xyz to a variable p.
  - p is said to point to the variable xyz.

Variable	Value	Address
xyz	15	5830
p	5830	4565

Values vs. Locations

- Variables name memory locations, which hold values.



New Type : Pointer

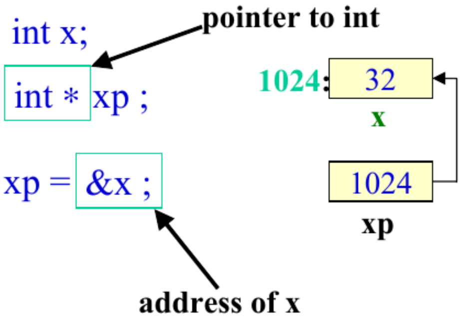
Pointers

- After declaring a pointer:

  `int *ptr;`

  ptr doesn't actually point to anything yet. We can either:
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to... (next time)

Pointer




```
*xp = 0;         /* Assign 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```

Pointers Abstractly

```
int x;
int * p;
p=&x;
...
(x == *p)    True
(p == &x)    True
```

Pointers

- Declaring a pointer just allocates space to hold the pointer - it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.

Pointers

- Declaring a pointer just allocates space to hold the pointer - it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.
- & is called reference operator. It gives you the address of a variable.
- Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).

Pointer Usage Example

```
#include <stdio.h>
int main(){
 int* pc; int c; c=22;
 printf("Address of c:%u\n",&c);
 printf("Value of c:%d\n\n",c);
 pc=&c;
 printf("Address of pointer pc:%u\n",pc);
 printf("Content of pointer pc:%d\n\n",*pc);
 c=11;
 printf("Address of pointer pc:%u\n",pc);
 printf("Content of pointer pc:%d\n\n",*pc);
 *pc=2;
 printf("Address of c:%u\n",&c);
 printf("Value of c:%d\n\n",c);
}
```

Common mistakes when working with pointers

```
int c, *pc;

// Wrong! pc is address whereas, c is not an address.
pc = c;

// Wrong! *pc is the value pointed by address whereas,
// %amp;c is an address.
*pc = &c;

//Correct! pc is an address and, %amp;pc is also an address.
pc = &c;

// Correct! *pc is the value pointed by address and,
//c is also a value.
*pc = c;
```

Accessing the Address of a Variable

- The '&' operator can be used only with a simple variable or an array element.

  &distance

  &x[0]

  &x[i-2]

Accessing the Address of a Variable

- The ’&’ operator can be used only with a simple variable or an array element.

  &distance

  &x[0]

  &x[i-2]

- Following usages are illegal:
  - &235

    - Pointing at constant.
  - int arr[20];

    :

    &arr;

    - Pointing at array name.
  - &(a+b)

    - Pointing at expression.

Example

```
#include <stdio.h>
main()
{
 int
 a;
 float b, c;
 double d;
 char ch;
 a = 10; b = 2.5; c = 12.36; d = 12345.66; ch = 'A';
 printf ("%d is stored in location %u \n", a, &a) ;
 printf ("%f is stored in location %u \n", b, &b) ;
 printf ("%f is stored in location %u \n", c, &c) ;
 printf ("%ld is stored in location %u \n", d, &d) ;
 printf ("%c is stored in location %u \n", ch, &ch) ;
}
```

Output

```
10 is stored in location 3822597804
2.500000 is stored in location 3822597800
12.360000 is stored in location 3822597796
140724131083928 is stored in location 3822597784
A is stored in location 3822597783
```

Things to Remember

- Pointer variables must always point to a data item of the same type.
  - Following code will result in erroneous output

```
float x;
int *p;
:
p = &x;
```

Things to Remember

- Pointer variables must always point to a data item of the same type.
  - Following code will result in erroneous output

```
float x;
int *p;
:
p = &x;
```

- Assigning an absolute address to a pointer variable is prohibited.

```
int *count;
:
count = 1268;
```

Accessing a Variable Through its Pointer

- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the indirection operator (*).

```
int   a, b;
int   *p;
:
p = &a;
b = *p;
```

Equivalent to

```
b = a;
```

Example

```
#include  <stdio.h>
main()
{
    int    a, b;
    int    c = 5;
    int    *p;

    a  =  4  *  (c  +  5) ;

    p  =  &c;
    b  =  4  *  (*p  +  5) ;
    printf  ("a=%d  b=%d \n",  a, b);
}
```

Equivalent

```
a=40 b=40
```

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;

Pointer Expressions: What are allowed in C?

- Add an integer to a pointer.
- Subtract an integer from a pointer.

Pointer Expressions: 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, then p2-p1 gives the number of elements between p1 and p2.

Pointer Expressions: 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 fact, it is not the integer value which is added/subtracted, but rather the scale factor times the value.

Scale Factor

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.

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called call-by-reference (or by address or by location).
- Normally, arguments are passed to a function by value.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.
- swap of two number???

Pointers and Arrays: When an array is declared

- The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a constant pointer to the first element.

Example

- Consider the declaration:

  int x[5] = {1, 2, 3, 4, 5};

  - Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

  - Both x and &x[0] have the value 2500.
  - p = x; and p = &x[0]; are equivalent.
  - We can access successive values of x by using p++ or p - - to move from one element to another.

Example

- Relationship between p and x:

$$\begin{array}{lll} p & = \&x[0] & = 2500 \\ p+1 & = \&x[1] & = 2504 \\ p+2 & = \&x[2] & = 2508 \\ p+3 & = \&x[3] & = 2512 \\ p+4 & = \&x[4] & = 2516 \end{array}$$

- *(p+i) gives the value of x[i]

Example: function to find average

```
#include <stdio.h>
main()
{
  int x[100], k, n;

  scanf ("%d", &n);

  for (k=0; k<n; k++)
     scanf ("%d", &x[k]);

  printf  ("\nAverage is %f",
                 avg (x, n));
}
```

```
float avg (array, size)
int array[], size;
{
  int  *p, i , sum = 0;

  p = array;

  for (i=0; i<size; i++)
      sum = sum + *(p+i);

  return ((float) sum / size);
}
```

Arrays

- Consequences:
  - ar is a pointer
  - ar[0] is the same as *ar
  - ar[2] is the same as *(ar+2)
  - We can use pointer arithmetic to access arrays more conveniently.
  - & ar[0] is equivalent to ar
  - & ar[1] is equivalent to (ar + 1) AND, ar[1] is equivalent to *(ar + 1).
- Declared arrays are only allocated while the scope is valid.

```
char* foo() {
   char string[32]; ...;
   return string;
 } is incorrect
```

Arrays

- Array size n; want to access from 0 to $n-1$, so you should use counter AND utilize a constant for declaration & incr
  - Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++) ...
```

  - Right

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++) ...
```

- why?

Arrays

- Array size n; want to access from 0 to $n-1$, so you should use counter AND utilize a constant for declaration & incr
  - Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++) ...
```

  - Right

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++) ...
```

- why? SINGLE SOURCE OF TRUTH

  – You're utilizing indirection and avoiding maintaining two copies of the number 10

- Pitfall: An array in C does not know its own length, & bounds not checked!

Arrays in functions

- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.
  - Can be incremented

int strlen(char s[])	int strlen(char *s)
{	{
...	...
}	}

Arrays and pointers

- #define N 20 int a[2N], i, *p, sum;
- p = a; is equivalent to p = &a[0];
- p is assigned 300. (in next slide)
- Pointer arithmetic provides an alternative to array indexing.
- p=a+1; is equivalent to p=&a[1]; (p is assigned 304)
- illegal : a=p; ++a; a+=2;

```
for (p=a; p<&a[N]; ++p)
    sum += *p ;
```

```
for (i=0; i<N; ++i)
    sum += *(a+i) ;
```

```
p=a;
for (i=0; i<N; ++i)
    sum += p[i] ;
```

How to assign pointer address manually in C?

- void * p = (void *)0x28ff44;
  int needs to be the type of the object that you are referencing.
- Structures example

```
#include<stdio.h>
typedef struct A{
    int a;
  }
main () {
    A *a = (A *)2000; a = a+1;
    printf("%u",a);
 }
```

Pointer arithmetic

- Since a pointer is just a mem address, we can add to it to traverse an array.
- p+1 returns a ptr to the next array element.
- (*p)+1 vs *p++ vs *(p+1) vs *(p)++ ?

  x = *p++ $\rightarrow$ x = *p; p = p + 1;

  x = (*p)++ $\rightarrow$ x = *p; *p = *p + 1;
- What happens if we have an array of large structs (objects)?

  – C takes care of it: In reality, p+1 doesn't add 1 to the memory address, it adds the size of the array element.

Pointer Arithmetic

- We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {
 int i;
 for (i=0; i<n; i++) {
     *to++ = *from++;
   }
}
```

- C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a char, 4 bytes for an int, etc.)

Pointer Arithmetic

- C knows the size of the thing a pointer points to - every addition or subtraction moves that many bytes.
- So the following are equivalent:

```
int get(int array[], int n)
 {
   return (array[n]);
   /* OR */
   return *(array + n);
 }
```

Pointer Arithmetic

- Array size n; want to access from 0 to n-1

  - test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = ar; q = &(ar[10]);
while (p != q)
 /* sum = sum + *p; p = p + 1; */
 sum += *p++;
```

  - Is this legal?

Pointer Arithmetic

- Array size n; want to access from 0 to n-1
  - test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = ar; q = &(ar[10]);
while (p != q)
  /* sum = sum + *p; p = p + 1; */
  sum += *p++;
```

  - Is this legal?

- C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error

Example with 2-D array

TO BE DISCUSSED LATER

Structures : Nesting of structure:

```
#include <stdio.h>
void main()
{
    struct world
    {
        int a;
        char b;
        struct india
        {
            char c;
            float d;
        }p;
    };
    struct world st ={1,'A','i',1.8};
    printf("%d\t%c\t%c\t%f",st.a,st.b,st.p.c,st.p.d);
}
```

```c
void main()
{
    struct india
    {
        char c;
        float d;
    };
    struct world
    {
        int a[3];
        char b;
        struct india in;
    };
    struct world st ={{1,2,3},'P','q',1.4};
    printf("%d\t%c\t%c\t%f",st.a[1],st.b,st.in.c,st.in.d);
}
```