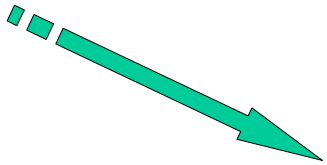
Lecture 10

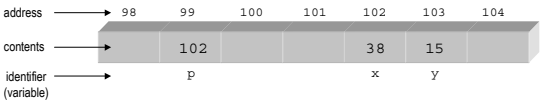


1. Introduction
2. Binary Representation
3. Hardware and Software
4. High Level Languages
5. Standard input and output
6. Operators, expression and statements
7. Making Decisions
8. Looping
9. Arrays
10. Basics of pointers
11. Strings
12. Basics of functions
13. More about functions
14. Files
14. Data Structures
16. Case study: lottery number generator

Basics of Pointers

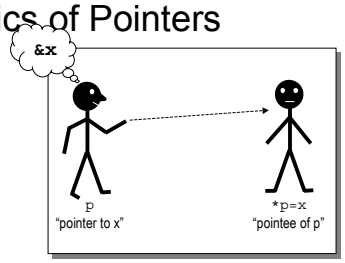
- Pointers are a very important part of C
- In C we need them for many tasks
  e.g. arrays, functions, strings etc
- People tend to find pointers hard to grasp at first!
- First lets go back and look at how variable are stored in memory

Basics of Pointers



- Each cell is numbered with a sequential address
- We can also refer to them by name
- The contents of each is a collection of 0s & 1s which can be interpreted in different ways
- Since an address is a number, it too can be stored in memory
- Here cell 99 contains 102 which is the address of X
- We can say that "p points to x"

Basics of Pointers



- A pointer is a variable that holds the address of another variable
  - We can say that p is a pointer to x
  - We could also say that x is the "pointee" of p

Declaring Pointer Variables

- We have seen declarations such as int x, y which means "x and y are variables of type int"
- We can also declare pointers to all data types e.g.

```
int* p;      p is a pointer to an int variable
float* zz;   zz is a pointer to a float
```

- Note that here '*' means pointer and NOT multiplication. Like many operators in C their usage depends upon the context

Address of Operator &

- Since pointer values are really addresses, we need a way to find the address of a variable.
- This is done by the & operator.
- So `&x` is the "address of x",102 in our example.
- This can then be assigned to p using

```
p=&x;
```

In Summary

```
int x,y;  /* declares 2 int variables */
int* p;   /* pointer to an int */

p=&x;     /* p now points to x */
*p=13;    /* x is now 13 */
```

- The last line means “set the value of the thing pointed to by p to 13”
- Since x is what p points to this is equivalent to saying `x=13;`
- NOTE: `p=13;` sets p to point at address 13, quite different and dangerous

pointer1.c

```
/* Example: basics of pointers, & and * operators */

#include <stdio.h>

main()
{
  int i = 57, j = 19;   /* declare two ints */
  int *ptr;             /* and a pointer to an int */

  ptr = NULL;           /* constant defined in <stdio.h> as 0
                           Means "not pointing to anything" */

  printf("ptr = %p\n", ptr);

  ptr = &i;             /* set ptr to the address of i.
                           i.e. ptr is a pointer to i,
                           and i is the "pointee" of ptr */

  printf("ptr = %p\n", ptr);

  /* We now have two ways of accessing the same variable: */

  printf("*i = %i, j = %i, *ptr = %i\n", i, j, *ptr);

  *ptr *= 2;            /* same as i = 2 * i; */
  (*ptr)--;             /* same as i--; */

  printf("*i = %i, j = %i, *ptr = %i\n", i, j, *ptr);

  ptr = &j;             /* ptr now points at j */

  printf("*i = %i, j = %i, *ptr = %i\n", i, j, *ptr);

  *ptr *= 2;            /* same as j = 2 * j; */
  (*ptr)--;             /* same as j--; */

  printf("*i = %i, j = %i, *ptr = %i\n\n", i, j, *ptr);

  /* Relationships among the variables: */

  printf("&ptr (the address of ptr) is\t%p\n\n", &ptr);
  printf("ptr (the contents of address %p) is\n%p\n\n", &ptr, ptr);
  printf("*ptr (the contents of address %p) is\n%i\n\n", ptr, *ptr);
  printf("&(*ptr) (the address of *ptr) is\n%p\n\n", &(*ptr));
  printf("&j (the address of j) is\n%p\n\n", &j);
  printf("*&j (the contents of address %p) is\n%i\n\n", &j, j);
  printf("*(&j) (the contents of address %p) is\n%i\n", &j, *(&j));
}
```

Basics of Pointers

- Note that the * and & are complementary
- So *(&x) is x
  &(*p) is p

  (Parentheses arent actually needed, evaluation is R to L)

- Weve already seen an example of &, in our use of scanf

  `scanf(“%i”, &x);  /* input a value for x */`
- `&x` is a pointer to `x` and could be rewritten as

  ```
  int*p=&x;
  scanf(“%i”,p);
  ```

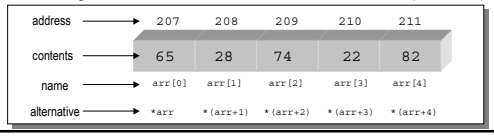
scanfptr.c

```
/* Example: scanf and pointers */

#include <stdio.h>

main()
{
  float number;
  float *number_p;

  number_p = &number;  /* number_p points to number */

  /* One way to do it: */

  puts("Enter a floating point number:");
  scanf("%f", &number);
  printf("You entered %G.\n\n", number);

  /* Another way to do the same thing: */

  puts("Enter a floating point number:");
  scanf("%f", number_p);
  printf("You entered %G.\n", *number_p);
}
```

Arrays and Pointers

- Last time we looked at arrays. Now a secret is revealed

  ARRAYS ARE REALLY POINTERS
- Thus the declaration `int arr[5]={65,28,74,22,82};`
  - reserves storage for 5 ints (here 207 to 211)
  - gives them values as in the list
  - creates a pointer to int called arr
  - gives it the value of the address of the first int (here 207)



Arrays and Pointers

- Because arr is actually an address (type `int*`) we can assign its value to a pointer (of type `int*`)

  ```
  int *ptr;
  ptr=arr;
  ```
  or `ptr=&arr[0];` `/* same thing*/`
- We can refer to the 3rd `int` as `arr[2]` or `*(arr+2)` or `*(ptr+2)` or `ptr[2]`
- We can now see why array indexes always start from zero, they are the offset from the nominal address of the array i.e. its first element

```c
/* BUG ZONE!!!
   Example: some common pointer errors */

#include <stdio.h>

main()
{
  int i = 57;
  float ztran4;
  int track[] = {1, 2, 3, 4, 5, 6}, stick[2][2];
  int *nsave;

  /* Let's try using *nsave as an int variable, and set it to 38 */

  *nsave = 38;  /* BUG */

  nsave = NULL;
  *nsave = 38;  /* BUG */

  nsave = 38;   /* BUG */

  &nsave = 38;  /* BUG */

  nsave = &i;
  *nsave = 38;

  nsave = &ztran4;  /* BUG */

  nsave = track;  /* nsave points at track[0] */

  /* Increase track[0] by 1: */
  *nsave++;  /* BUG */

  /* Now point at stick: */
  nsave = stick;       /* BUG */
  nsave = &stick;      /* BUG */
  nsave = stick[0][0]; /* BUG */
  nsave = *stick;      /* BUG */
  nsave = **stick;     /* BUG */
  nsave = stick[0];
  nsave = &stick[0][0];
  nsave = &**stick;
}
```

pointer.bug

```c
/* Example: pointers and two-dimensional arrays */

#include <stdio.h>

main()
{
  int arr[2][3] = {{11, 22, 33},
                   {44, 55, 66}};  /* an array of ints */
  int *ptr;                        /* a pointer to int */
  int i;                           /* row index */
  int j;                           /* column index */

  ptr = &arr[0][0];  /* set pointer ptr to point at the start of the
                        array, i.e. ptr contains the address of arr[0][0],
                        so arr[0][0] is the "pointee" of ptr.

  We could also write ptr = *arr, because this means &(**arr). Now,
  **arr means *(*arr), which means *(arr[0]), which means arr[0][0].

  We could also write ptr = arr[0]; because ptr is a pointer
  to int and so is arr[0]: it points to arr[0][0].

  We could also write ptr = *arr; because ptr is a pointer
  to int (type int *) and arr is a pointer to a pointer to int
  (type int **). */

  /* We now have several ways of accessing the array elements. */

  /* As a two-dimensional array: */
  for (i = 0; i < 2; i++)
  {
    for (j = 0; j < 3; j++)
      printf("%i ", arr[i][j]);  /* the most natural */
    putchar('\n');
  }
  putchar('\n');

  /* As two one-dimensional arrays: */
  for (j = 0; j < 3; j++)
    printf("%i ", *(arr[0] + j));
  putchar('\n');

  for (j = 0; j < 3; j++)
    printf("%i ", *(arr[1] + j));
  putchar('\n');
  putchar('\n');
```

pointer3.c

```c
/* Example: pointers and arrays */

#include <stdio.h>

main()
{
  int arr[5] = {65, 28, 74, 22, 82};  /* declare an array of ints */
  int *ptr1, *ptr2;                   /* and two pointers to an int */
  int i;

  ptr1 = arr;       /* set pointer ptr1 to point at the array,
                       i.e. ptr1 contains the address of arr[0],
                       and arr[0] is the "pointee" of ptr1.
                       (Note that arr is itself a pointer.)
                       We could also write ptr1 = &arr[0]; */

  ptr2 = &arr[4];   /* set pointer ptr2 to point at the last
                       element in the array, i.e. ptr2 contains the
                       address of arr[4], and arr[4] is the "pointee"
                       of ptr2. */

  /* We now have several ways of accessing the same array elements: */

  printf("arr[0] is %i, arr[4] is %i\n", arr[0], arr[4]);
  printf("*arr is %i, *(arr + 4) is %i\n", *arr, *(arr + 4));
  printf("*&arr[0] is %i, *&arr[4] is %i\n", *&arr[0], *&arr[4]);
  printf("*ptr1 is %i, *ptr2 is %i\n", *ptr1, *ptr2);
  printf("*(ptr2 - 4) is %i, *(ptr1 + 4) is %i\n", *(ptr2 - 4), *(ptr1 + 4));
  printf("ptr1[0] is %i, ptr1[4] is %i\n", ptr1[0], ptr1[4]);
  printf("ptr2[-4] is %i, ptr2[0] is %i\n\n", ptr2[-4], ptr2[0]);

  /* We can use them in loops, e.g.: */

  for (i = 0; i < 5; i++)
    printf("%i ", arr[i]);
  putchar('\n');

  for (i = 0; i < 5; i++)
    printf("%i ", *(arr + i));
  putchar('\n');

  for (i = 0; i < 5; i++)
    printf("%i ", ptr1[i]);
  putchar('\n');

  for (i = 0; i < 5; i++)
    printf("%i ", ptr2[i-4]);
  putchar('\n');
}
```

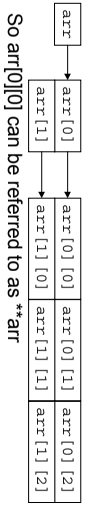
pointer2.c

Two Dimensional Arrays and Pointers

- When we declare

  `int arr[2][3]={{11,22,33},{44,55,66}};`

  - Storage is reserved for 6 ints
  - values are given to them as in the list of lists
  - a pointer to a pointer to an int is created called arr
  - its value is set to the address of the first int



So arr[0][0] can be referred to as **arr

arr[1][2] can be referred to as *(arr[1]+2) or *(arr[0]+5)

```c
/* As a single one-dimensional array: */
  for (j = 0; j < 6; j++)
    printf("%i ", *(*arr + j));
  putchar('\n');
  putchar('\n');

  /* Or, equivalently: */
  for (j = 0; j < 6; j++)
    printf("%i ", *(arr[0] + j));
  putchar('\n');
  putchar('\n');

  /* Suppose we know only the number of elements (6) and have a
  pointer to the first element (ptr). We have to make an assumption
  about the array shape. */

  /* Assuming a one-dimensional array (1 row by 6 columns): */
  for (j = 0; j < 6; j++)
    printf("%i ", ptr[j]);
  putchar('\n');
  putchar('\n');

  /* Assuming a 2 x 3 array (2 rows by 3 columns): */
  for (i = 0; i < 2; i++)
  {
    for (j = 0; j < 3; j++)
      printf("%i ", *(ptr + 3*i + j));
    putchar('\n');
  }
  putchar('\n');

  /* Assuming a 3 x 2 array (3 rows by 2 columns): */
  for (i = 0; i < 3; i++)
  {
    for (j = 0; j < 2; j++)
      printf("%i ", *(ptr + 2*i + j));
    putchar('\n');
  }
  putchar('\n');

  /* Assuming a 6 x 1 array (6 rows by 1 column): */
  for (i = 0; i < 6; i++)
  {
    printf("%i ", *(ptr + i));
    putchar('\n');
  }
  putchar('\n');
}
```

pointer2.c cont