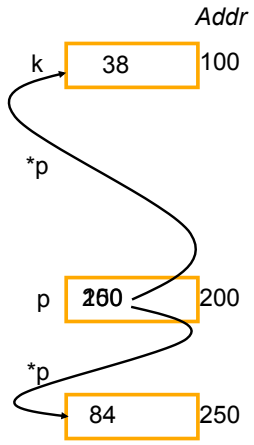
IC150
Pointers

Lecture 16

Timothy A. Gonsalves

What is a pointer?

- *Recap:* a variable int k
  - Names a memory location that can hold one value at a time
  - Memory is allocated statically at compile time
  - One name ⇔ one value
- **A pointer variable int *p**
  - Contains the address of a memory location that contains the actual value
  - Memory can be allocated dynamically at runtime
  - One name ⇔ many values



l-value and r-value

- Given a variable k
- its l-value refers to the *address* of the memory location
  - l-value is used on the left side of an assignment

    k = expression
- its r-value refers to the *value* stored in the memory location
  - r-value is used in the right hand side of an assignment

    var = k + …
- pointers allow one to manipulate the l-value!

pointers

- pointers are variables that store the *address* of a memory location
- the memory required by a pointer (variable) depends upon the size of the memory in the machine
  - one byte could address a memory of 256 locations
  - two bytes can address a memory of 64K locations
  - four bytes can address a memory of 4G locations
  - modern machines have RAM of 1GB or more…
- The task of allocating this memory is best left to the system

declaring pointers

- pointer variable – precede its name with an asterisk
- pointer type - the type of data stored at the address we will be storing in our pointer. For example,

```
int *p;
```

- **p** is the *name* of the variable. The '*' informs the compiler that we want a *pointer variable*, i.e. to set aside however many bytes is required to store an address in memory.
- The **int** says that we intend to use our pointer to point to an integer value

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Contents of pointer variables

- In ANSI C if a pointer is declared outside any function, it is *initialized* to a "null" pointer

```
int k;
int *p;
p = &k              // assigns the address of int k to p
if (p == NULL)      // tests for a null pointer
p = malloc(sizeof(int));
```

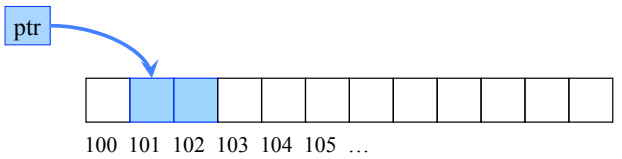
- dynamic allocation, creates an *anonymous* int in memory at runtime

dereferencing operator

- The "dereferencing operator" is the asterisk and it is used as follows:
- `*ptr = 7;`
- will copy 7 to the address pointed to by `ptr`. Thus if `ptr` "points to" `k`, the above statement will set the *value of* `k` *to 7*.
- Using `*` is a way of referring to the value of that which `ptr` is pointing to, not the value of the pointer itself
- `printf("%d\n", *ptr);` --- prints 7

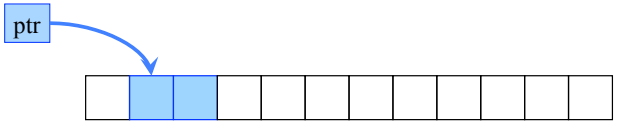
short int pointer

- short *ptr; – says that ptr is the address of a short integer type
- short – allocates two bytes of memory

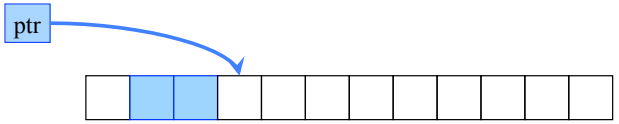


- *ptr = 2; -- will store the value 2 in the above two bytes
- if we had said "int *ptr" it would have allocated 4 bytes of memory for the integer

pointer arithmetic



- ptr = ptr +1;
  - says to point to the *next* data item after this one



Makes sense only for same type data – eg an array of integers

Memory needed for a pointer

A pointer requires two chunks of memory to be allocated:

1. Memory to hold the pointer (address)
   - Allocated statically by the pointer declaration
2. Memory to hold the value pointed to
   - Allocated statically by a variable declaration
   - OR allocated dynamically by `malloc()`

One variable or pointer declaration ⇒ allocation of one chunk of memory

IC150
… Pointers

Lecture 17

Timothy A. Gonsalves

Memory needed for a pointer

A pointer requires two chunks of memory to be allocated:

1. Memory to hold the pointer (address)
   - Allocated statically by the pointer declaration
2. Memory to hold the value pointed to
   - Allocated statically by a variable declaration
   - OR allocated dynamically by `malloc()`

One variable or pointer declaration ⇒ allocation of one chunk of memory

Accessing Arrays with Pointers

```
int main()
{
  int i, *ptr;
  int myArray[] = {1,23,17,4,-5,100};

  ptr = &myArray[0];      // point ptr to the first element of the array
  printf("\n\n");
  for (i = 0; i < 6; i++)
    {                     // print myArray using array indexing
      printf("myArray[%d] = %d ", i, myArray[i]);
                          // print myArray using ptr
      printf("ptr + %d = %d\n", i, *(ptr + i));  // B
    }
  return 0;
}
```


ptr++ and ++ptr

- **++ptr** and **ptr++** are both equivalent to **ptr + 1**
  - though they are "incremented" at different times
- Change line B to:

  printf("ptr + %d = %d\n",i, *ptr++);
- and run it again... then change it to:

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

Arrays

- the name of the array is the address of the first element in the array
- In C

  ptr = &myArray[0];

  is equivalent to

  ptr = myArray;
- Many texts state that the name of an array is a pointer

Arrays names are *not* pointers

While we can write

ptr = myArray;

we cannot write

myArray = ptr;

The reason: while **ptr** is a variable, **myArray** is a constant. That is, the location at which the first element of **myArray** will be stored cannot be changed once **myArray[]** has been declared.

Pointer types

- C provides for a pointer of type void. We can declare such a pointer by writing:

  void *vptr;

- A void pointer is a *generic* pointer. For example, a pointer to *any* type can be compared to a void pointer
- Casts can be used to convert from one type of pointer to another under the proper circumstances

Trying out pointers

```
#include <stdio.h>
int j, k; int *ptr;
int main(void) {
  j = 1; k = 2;
  ptr = &k;
  printf("\n");
  printf("j has the value %d and is stored at %p\n", j, (void *)&j);
  printf("k has the value %d and is stored at %p\n", k, (void *)&k);
  printf("ptr has the value %p stored at %p\n", ptr, (void *)&ptr);
  printf("The value of the integer pointed to by ptr is %d\n", *ptr);
  return 0;
}
```

Generic address of j

Dereferencing – will print r-value of k

Pointers and strings

- C does not have a string type
  - languages like Pascal, Fortran have…
- In C a string is an array of characters terminated with a binary zero character (written as **'\0'**).
  - remember this is not the character '0'
- One can manipulate strings as character arrays
- Character arrays can also be accessed by pointers

A character array

- One could create a string as follows,

```
char s[40];
s[0] = 'T';
s[1] = 'e';
s[2] = 'd':
s[3] = '\0';
```

- Note - terminated with a *nul* character
- nul (or '\0') ≠ NULL (pointer to nothing)
- Obviously this is very tedious

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

"Strings"

- One might write:

```
char s[40] = {'T','e','d','\0'};
```

- This is tedious. So, C permits:

```
char s[40] = "Ted";
```

  – Note that C automatically inserts '\0'

- Compiler sets aside a contiguous block of memory 40 bytes long
- The first four bytes contain Ted\0

strings: input and output

- The function *gets()* accepts the name of the string as a parameter, and fills the string with characters that are input from the keyboard till newline character is encountered. At the end function gets appends a null terminator. *Not a popular function* because there are no built in checks
- char ***gets**(char *s);
- gets (str) – reads from standard input into str
- puts (str) – writes to standard output from str
  – Where str is a string variable.

gets may overwrite memory!

```
char b1[] = "ABCDE";
char b2[] = "LMNOF";
char b3[] = "ZYXWV";
puts(b1);
puts(b2);
puts(b3);
puts("Input:");
gets(b2);
puts(b1);
puts(b2);
puts(b3);
```

Another run

```
puts(b1); ABCDE
puts(b2); LMNOP
puts(b3); ZYXWV
puts(…); Input: 1234567890
gets(b2);
puts(b1); 90BCDE
puts(b2); 1234567890
puts(b3); ZYXWV
```


IC150
String Pointers

Lecture 18

Timothy A. Gonsalves

character pointers

```
#include <stdio.h>
char strA[80] = "A string to be used for demonstration";
char strB[80];
int main(void)
{
char *pA;          /* a pointer to type character */
char *pB;          /* another pointer to type character */
puts(strA);        /* show string A */
pA = strA;         /* point pA at string A */
puts(pA);          /* show what pA is pointing to */
```

--continued →

copying strings…

```
pB = strB;                    /* point pB at string B */
putchar('\n');        /* move down one line on the screen */
while(*pA != '\0')            /* while string  */
   {
    *pB++ = *pA++;       /* copy and increment pointer */
   }
*pB = '\0';                   /* insert end-of-string */
puts(strB);                   /* show strB on screen */
return 0;
}
```

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

a version of strcpy

```
char *myStrcpy(char *dest, char *src)
{
  char *p = dest;
  while (*src != '\0')
    {
       *p++ = *src++;
    }
  *p = '\0';
  return dest;
}
```

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

*ptr++

- ***ptr++** is to be interpreted as returning the value pointed to by **ptr** and then incrementing the pointer value.
- This has to do with the precedence of the operators.
- **(*ptr)++** would increment, not the pointer, but that which the pointer points to!
- i.e. if used on the first character of the example string "IIT" the 'I' would be incremented to a 'J'.

Copying arrays using pointers

- Exercise – define a function to copy part of an integer array into another. Access the elements using pointers.
- function prototype

  void intCopy(int *ptrA, int *ptrB, int num);
  - where **num** is the number of integers to be copied.

Equivalent definition using arrays

```
char *myStrcpy(char dest[], char source[])
{                       char *myStrcpy(char *destination, char *source)
 int i = 0;             char *p = destination;
 while (source[i] != '\0')
  {                     while (*source != '\0')
    dest[i] = source[i];        *p++ = *source++;
    i++;
  }
 dest[i] = '\0';                *p = '\0';
 return dest;                   return destination;
}
```


pointer arithmetic ≡ array indexing

- Both work identically
- Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what *actually gets passed* is the address of the first element of each array.
- The numerical value of the parameter passed is the same. This would tend to imply that somehow **source[i]** is the same as ***(p+i)**.
- **a[i]** it can be replaced with ***(a + i)** without any problems. In fact, the compiler will create the same code in either case.

indexes are converted to pointer addresses

- We could write ***(i + a)** just as easily as ***(a + i)**.
- But ***(i + a)** could have come from **i[a]** ! From all of this comes the curious truth that if:

```
char a[20];
int i;
```

writing

```
a[3] = 'x';
```

is the same as writing

```
3[a] = 'x';                    !
```

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Equivalent statements

```
dest[i] = source[i];
```

- due to the fact that array indexing and pointer arithmetic yield identical results, we can write this as:

```
*(dest + i) = *(source + i);
```

- Also we could write

```
while (*source != '\0')
```

- as

```
while (*source)
```

- since the code for '\0' is 0 = false

Declaring "IITM"

1. char myName[40] = "IITM";
   - allocates space for a 40 byte array and puts the string in the first 5 bytes
2. char myName[] = "IITM";
   - the compiler allocates 5+1 characters (for IITM and '\0') starting from the memory location named by **myName**
3. char *myName = "IITM";
   - Allocates 5 bytes for the string, **plus** 4 bytes to store the pointer variable **myName**

CS101
Multi-dimension Arrays

Lecture 19

Timothy A. Gonsalves

Multidimensional arrays

- char multi[5][10];
  - multi[1][10] – an array of ten characters
  - multi -- 5 such arrays

```
multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

- individual elements are addressable:

```
multi[0][3] = '3'
multi[1][7] = 'h'
```

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Linear contiguous memory

multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}

0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
^

|_____ starting at the address &multi[0][0]

address pointers

multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}

- multi is the address of '0'
- multi+1 is the address of 'a"
  - It adds 10, *the number of columns*, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add **10*sizeof(int)** which would be usually 20.
- To get to the content of the 2nd element in the 4th row we add 1 to this address and dereference the result as in
  - *(*(multi + 3) + 1)

address computation

- With a little thought we can see that:

((multi + row) + col) and
multi[row][col] yield the same results.

- Because of the *double de-referencing* required in the pointer version, the name of a 2 dimensional array is often said to be equivalent to a *pointer to a pointer*.

multi[row][col] = *(*(multi + row) + col)

- To understand more fully what is going on, let us replace
  *(multi + row) with **X**
  i.e. *(*(multi+row)+col) = *(*X + col)
- **X** is like a pointer since the expression is de-referenced and **col** is an integer
- Here "pointer arithmetic" is used
- That is, the address pointed to (i.e. value of) **X + col = X + col * sizeof(int)**.

- Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression **multi + row** as used above, **multi + row + 1** must increase by value an amount equal to that needed to "point to" the *next row*,
- which for integers would be an amount equal to **COLS * sizeof(int)**.
- That says that if the expression ***(*(multi + row) + col)** is to be evaluated correctly at run time, the compiler must generate code which takes into consideration the value of **COLS**, i.e. the 2nd dimension.

remember passing arrays as parameters?

- Thus, to evaluate either expression, a total of 5 values must be known:

1. The address of the first element of the array, which is returned by the expression **multi**, i.e., the name of the array.
2. The size of the type of the elements of the array, in this case **sizeof(int)**.
3. The 2nd dimension of the array
4. The specific index value for the first dimension, **row** in this case.
5. The specific index value for the second dimension, **col** in this case.

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

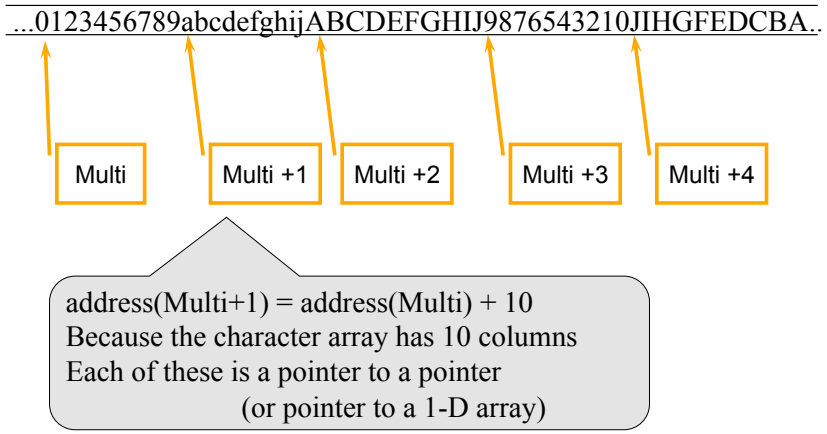
Revisiting functions on arrays

- Initializing a 2-dimensional array

```
#define COLS 25
void SetValue(int mArray[][COLS])
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            mArray[row][col] = 1;
        }
    }
}
```


- Question:
  - When we say `value = *ptr;` the pointer ptr is dereferenced to get the data stored
  - What happens in *(*(array + row) + column)?
  - Why is *(array + row) not dereferenced to give, say, an integer?
- Answer:
  - It *is* dereferenced
  - Remember a 2-D array is *a pointer to a pointer*
  - *(array + row) dereferences to a pointer to a 1-D array
  - *(array + row) + 1 would do a pointer increment

Array Multi[5][10]



CS101
Searching

Lecture 20

Timothy A. Gonsalves

Searching for elements

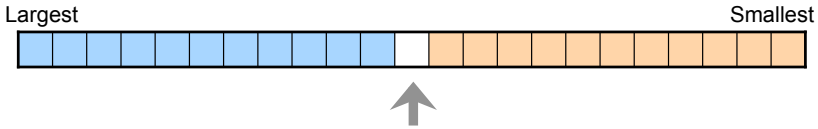
- Given an array of marks and names is there someone who got X marks?
- If X occurs in the Marks array return the index of the position where it occurs.
- If the numbers are randomly arranged then there is no option but to scan the entire array to be sure.
- Would the task would be simpler if the marks were arranged in sorted order?
  - Like finding a word in a dictionary

Searching in a sorted array

- Given an array of marks and names sorted in descending order of marks is there someone who got X marks?
- If X is high (say 92/100) one could start scanning from the left.
- If X is low (say 47/100) one could scan the array right to left.
- But what if we do not know whether X is high or low?

Divide and Conquer

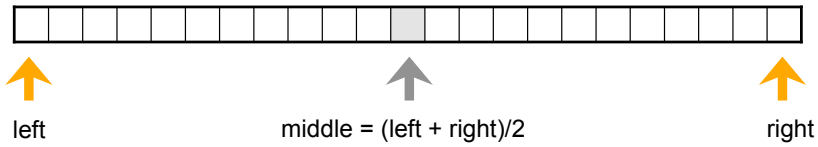
- Look at the middle element



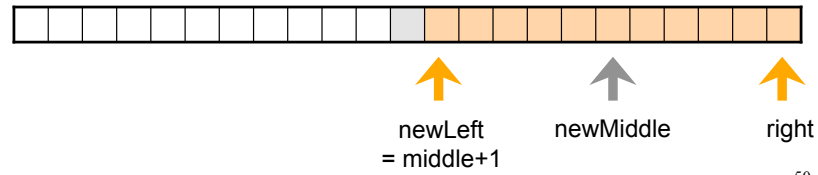
- If array[middle] = = X, done
- If array[middle] > X, look *only in the right part*
- Else look for the number *only in the left part*
- The problem is reduced into a smaller problem
  - new problem is half the size of the original one
- *Recursively apply this strategy*

Divide and Conquer

- Two indexes define the range of searching



- If array[middle] > X look *only in the right part*



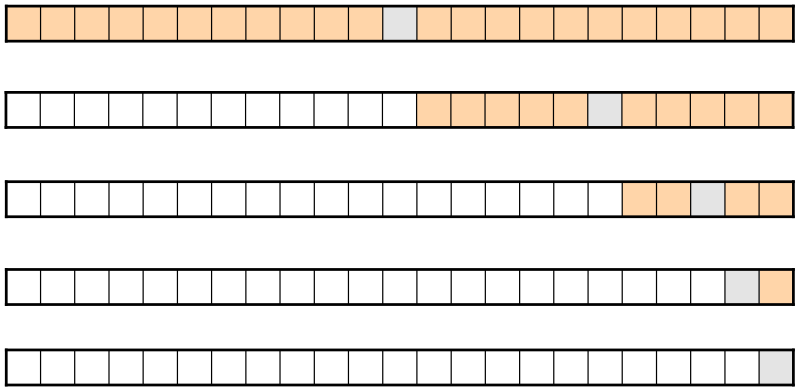
Binary Search (also called Binary Chop)

- Starts with the full sorted array
  - left = 0 and right = N-1
- The range of search are the elements between left and right including array[left] and array[right]
- Search terminates if right < left
- Otherwise
  - If (array[middle] == X) return middle
  - If (array[middle] < X) left = middle +1
  - Else right = middle -1

Binary Search (bsearch in stdlib.h – elements increasing order)

```
int BinarySearch(int value, int array[], int n)
{
    int left = 0, int right = n-1;
    while (left <= right)
       {
         middle = (left+right)/2;
         if (array[middle] == value) return middle;
         if (array[middle] < value) left = middle +1;
         else right = middle -1;
       }
    return INFINITY;     /* calling function must interpret this correctly! */
    }
```

Complexity of Binary Search



After each inspection the array reduces by half. For an array of size N there are about $\log_2 N$ inspections *in the worst case*.

Complexity: Recurrence equation

- Let n be the size of the array
- Let the function T(n) represent the time complexity of solving the problem of size n
- In each call the problem is reduced by half
  - one comparison in each call
- T(n) = 1 + T(n/2)
- Solving this gives us the complexity $\log_2(n)$

T(n) = 1 + T(n/2)
T(n/2) = 1 + T(n/4)
T(n/4) = 1 + T(n/8)
.
.
T(2) = 1 + T(1)
= 1 + 1

log(n) equations

Finding student with X marks

```
char *findStudent (int x, char names[][COL], marks[], n)
{
int index = BinarySearch(int x, int marks[], int n);
if (index == n) return "no such student";
else return names[index];
}
```

return NULL;

Above function assumes that names are corresponding to marks, and marks are in decreasing order.