MORE POINTERS

CS 23200

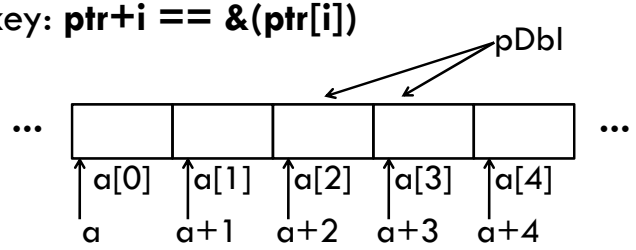
Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- Using const with pointers
- Pointers to pointers
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

Basic Address Arithmetic

- Sometimes we want to...
  - Get a pointer to an item later in an array
  - Move a pointer to the next array item
- The key: **ptr+i == &(ptr[i])**

```
double a[5];

double *pDbl = a+2;

pDbl++;
```



Outline

- Basic address arithmetic
- **Character pointers and strings**
- Using gdb with pointers
- Using const with pointers
- Pointers to pointers
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

Character Pointers and Strings

```
/* copy buffer1 to buffer2 */
char buffer1[1000] = "This is a string.";
char *buffer2 = malloc(sizeof(char) * strlen(buffer1));
buffer2 = buffer1;
buffer1[0] = '\0';
```

- What are the contents of buffer2?
  - The same as buffer1, since they point to the same location
- Assignment of char* works the same as assignment of any other pointer
  - It does not copy strings!

Copying Strings

- Use `strncpy` to copy strings
- `strncpy(char *dest, const char *src, size_t n);`
- Copies at most `n` characters (including a terminating null) from `src` to `dest`
- Will null-terminate `dest` only if `src` has fewer than `n` characters

```
strncpy(dest, src, bufSize);
dest[bufSize-1] = '\0'; /* ensure null-termination */
```

Copying Strings

- Ensure that space is allocated for the copy

```
/* THIS IS INCORRECT */
char* src = "hello";
char* dest;

strncpy(dest, src, strlen(src) + 1);

-------------------------------------
/* THIS IS CORRECT */
char* src = "hello";
char* dest = malloc(sizeof(char) * (strlen(src) + 1));
if(dest != NULL){
  strncpy(dest, src, strlen(src) + 1);
}
```

Character Pointers and Strings

```
/* copy buffer1 to buffer2 */
char buffer1[1000] = "This is a string.";
char *buffer2 = malloc(sizeof(char) * strlen(buffer1));
strncpy(buffer2, buffer1, 1000);
buffer1[0] = '\0';
```

- What are the contents of buffer2?
  - Bugs in the code!
  - buffer2 should be allocated to hold strlen(buffer1) + 1 characters, to account for the terminating null character
  - strncpy should be given the maximum size of the destination (buffer2), not the source (buffer1)
  - We need to ensure null-termination of buffer2

Character Pointers and Strings

```
/* copy buffer1 to buffer2 */
char buffer1[1000] = "This is a string.";
const int n = strlen(buffer1) + 1;
char *buffer2 = malloc(sizeof(char) * n);
strncpy(buffer2, buffer1, n);
buffer2[n-1] = '\0';
buffer1[0] = '\0';
```

- What are the contents of buffer2?
  - "This is a string."

String Constants and Initialization

```
char *strConstant = "This is a string.";
char charArray[] = "This is a string.";
char *heapCharStar = malloc(...);
char *otherCharStar;

strConstant[0] = 't'; /* NOT ALLOWED */
charArray[0] = 't';
heapCharStar[0] = 't'; /* allowed, but heapCharStar
                          may not be null-terminated */
otherCharStar[0] = 't';
                       /* INVALID: dereferencing
                        uninitialized pointer */
```

- String constants are in a read-only section of program memory (not the stack or the heap)
- charArray is just like declaring charArray[18], then copying "This is a string." into that array

String Constants Summary

- String constants like "This is a string." get allocated in read-only memory
- Initializing a **char*** to "This is a string." points to that read-only memory
- Initializing a **char[]** to "This is a string." **copies** the string constant to the char array (which is not read-only)

Practice: Strings and Pointers

Outline

- Basic address arithmetic
- Character pointers and strings
- **Using gdb with pointers**
- Using const with pointers
- Pointers to pointers
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

Using gdb with Pointers

- gdb can print and display expressions involving the indirection and address operators
  - p *plnt
  - display &x
- Printing arrays
  - If you print an array, gdb will print the whole array
  - If you print a pointer, you get the actual address
    - To get the contents of an array from a pointer, print *ptr@arrayLength
  - If you print a char*, gdb will print the entire null-terminated string

Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- **Using const with pointers**
- Pointers to pointers
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

```
char myString[100];
strncpy(myString, "This is a string.", 100);

/*  const pointers point to things that
    should not be changed */
const char *dontChangeThis = myString;
/*  *dontChangeThis is const char
    dontChangeThis[10] is const char */
dontChangeThis[0] = 't';            /* ERROR */

/* const pointers do not say that the memory
   is universally read-only. Other pointers to
   the same memory can be used to change it */
myString[0] = 't';                  /* ok */

/* Cannot get char* from const char*  */
char* changeable = dontChangeThis;  /* ERROR */

/* The variable dontChangeThis is NOT const */
dontChangeThis = &(myString[4]);    /* ok */
```

Using const with pointers

- Assigning a pointer of type `const T*` from a pointer of type `T*` is allowed...

```
int x = 7;
int *px = &x;
const int *pConstX = px;

*px += 3;
*pConstX += 3;      /* not allowed */
```

- ... but not vice versa

```
px = pConstX; /* not allowed */
```

Practice: Pointers and const

Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- Using const with pointers
- **Pointers to pointers**
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

Arrays of Strings

```
const int MAX_LINES = 10000, maxLineSize = 1000;

char *lines[MAX_LINES]; /* an array of pointers */

lines[0] = "This is a header line.";
lines[1] = ""; /* a blank line */

lines[2] = malloc(sizeof(char) * maxLineSize);
lines[2][0] = 'H';
lines[2][1] = 'i';
lines[2][2] = '\0';

/* lines[2] is of type char*
   lines[2][0] is of type char    */
```

What is Wrong Here?

```
const int MAX_LINES = 10000, maxLineSize = 1000;
char *lines[MAX_LINES];
char *nextLine;

lines[0] = "This is a header line.";
lines[1] = ""; /* a blank line */

nextLine = lines[2];
nextLine[0] = 'H';
nextLine[1] = 'i';
nextLine[2] = '\0';

/* lines[2] is uninitialized */
```

What is Wrong Here?

```
const int MAX_LINES = 10000, maxLineSize = 1000;
char *lines[MAX_LINES];
char *nextLine;

lines[0] = "This is a header line.";
lines[1] = ""; /* a blank line */

nextLine = lines[2];
nextLine = malloc(sizeof(char) * maxLineSize);
nextLine[0] = 'H';
nextLine[1] = 'i';
nextLine[2] = '\0';

/* lines[2] is still uninitialized:
even if two pointers are equal (nextLine and lines[2]),
assigning to one does not change the value of the other */
```

Back to the Correct Version

```
const int MAX_LINES = 10000, maxLineSize = 1000;

char *lines[MAX_LINES]; /* an array of pointers */

lines[0] = "This is a header line.";
lines[1] = ""; /* a blank line */

lines[2] = malloc(sizeof(char) * maxLineSize);
lines[2][0] = 'H';
lines[2][1] = 'i';
lines[2][2] = '\0';

/* What if no good estimate of MAX_LINES is known
      until run-time (e.g., input from a file)?  */
```

Pointers to Pointers

```
const int maxLineSize = 1000;

char **lines; /* a pointer to a char*   */
int i, numLines = -1;

/* get the number of lines */
...

lines = malloc(sizeof(char*) * numLines);
for(i=0; i<numLines; i++){
  lines[i] = malloc(sizeof(char) * (maxLineSize + 1));
}

lines[0][0] = 'H';
lines[0][1] = 'i';
...
```

Pointers to Pointers

- You can declare pointers to pointers (to pointers to pointers ...)
- They work just like normal pointers, but you have to be careful ...
  - to keep track of levels of indirection

```
int **thisThing;
*thisThing = 7; /* illegal */
```

  - to allocate (and free) memory at all the different levels

```
int **thisThing;
thisThing[0] = malloc(sizeof(int*) * n);
```

    - thisThing is not initialized, so thisThing[0] is an invalid memory access

Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- Using const with pointers
- Pointers to pointers
- **Initialization of pointer arrays**
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- Command line arguments
- Address arithmetic (optional)

Initialization of Pointer Arrays

```
int *pointersToVars[] = {&x, &y, &z};

char *studentNames[] = {"Bob", "Fred", "Susan"};
```

- Initialization uses a list in braces (just as with any other array), but the items in the list are pointers

Pointer Practice

Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- Using const with pointers
- Pointers to pointers
- Initialization of pointer arrays
- **Multi-dimensional arrays**
- **Pointers vs. multi-dimensional arrays**
- Command line arguments
- Address arithmetic (optional)

Multi-dimensional Arrays

```
const int numStudents = 15;
const int numAssignments = 20;

double grades[numStudents][numAssignments];
grades[studentID][assignNum] = 100.0;

OR

double grades[numAssignments][numStudents];
grades[assignNum][studentID] = 100.0;
```

Multi-dimensional Arrays

```
const int numStudents = 3;
const int numAssignments = 2;
double grades[numStudents][numAssignments]
   = {{0.0, 0.0}, {0.0, 0.0}, {0.0, 0.0}};

int studentID, assignNum;
for(studentID=0; studentID<numStudents; studentID++){
   for(assignNum=0; assignNum<numAssignments;
            assignNum++){
      grades[studentID][assignNum] = 0.0;
   }
}
```

- Initialization can be done with nested braces (but it is usually easier to write as nested loops)

Review: Pointers and Arrays

- Can use indexing with both pointers and arrays

```
int *ptr = malloc(sizeof(int) * 20);
int arr[20];
ptr[0] = 7;
arr[0] = 7;
```

- Can use indirection (dereferencing) operator and pointer arithmetic with both pointers and arrays

```
int *ptr = malloc(sizeof(int) * 20);
int arr[20];
*(ptr + 7) = 15;
*(arr + 7) = 15;
free(ptr);
ptr = arr + 10;
```

Pointers vs. Arrays

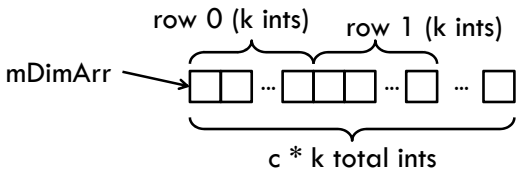
- Remember: an array declaration allocates space for several items, but a pointer declaration does not
- The basic principles for multi-dimensional arrays and pointers to pointers are the same as for simple arrays and pointers

Pointers vs. Multi-dimensional Arrays

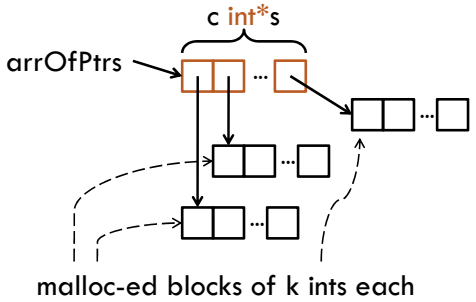
- Multi-dimensional arrays used to store matrices, images, arrays of strings, etc.
- Allocating a `c` by `k` block of ints

```
int mDimArr[c][k];


int *arrOfPtrs[c];
for(i=0; i<c; i++)
   arrOfPtrs[i] = malloc(sizeof(int) * k);


int **ptrToPtrs;
ptrToPtrs = malloc(sizeof(int*) * c);
for(i=0; i<c; i++)
   ptrToPtrs[i] = malloc(sizeof(int) * k);
```

Pointers vs. Multi-dimensional Arrays




```
int mDimArr[c][k];
int *arrOfPtrs[c];
int **ptrToPtrs;
```



ptrToPtrs:
looks the same as arrOfPtrs except the block of int*s is malloc-ed (instead of a statically allocated array)

Pointers vs. Multi-dimensional Arrays

```
int mDimArr[c][k];
int *arrOfPtrs[c];
int **ptrToPtrs;
```

```
int *pInt;
...

/* SIMILARITIES */
mDimArr[i][j] = 7;
arrOfPtrs[i][j] = 7;
ptrToPtrs[i][j] = 7;

pInt = &(mDimArr[i][j]);
pInt = &(arrOfPtrs[i][j]);
pInt = &(ptrToPtrs[i][j]);
```

```
/* PRIMARY DIFFERENCE */
pInt = malloc(...);

/* invalid */
mDimArr[i] = pInt;

/* okay */
arrOfPtrs[i] = pInt;
ptrToPtrs[i] = pInt;

/* related to the allocation
   differences we just saw */
```

Pointers vs. Multi-dimensional Arrays

- Multi-dimensional arrays are rectangular (i.e., each row has the same length)

```
int mDimArr[c][k];
```

- Arrays of pointers and pointers to pointers do not have this restriction

```
int *arrOfPtrs[c];
for(i=0; i<c; i++)
   arrOfPtrs[i] = malloc(sizeof(int) * (i+1));
```

```
int **ptrToPtrs;
ptrToPtrs = malloc(sizeof(int*) * c);
for(i=0; i<c; i++)
   ptrToPtrs[i] = malloc(sizeof(int) * (i+1));
```

Pointer Practice

Outline

- Basic address arithmetic
- Character pointers and strings
- Using gdb with pointers
- Using const with pointers
- Pointers to pointers
- Initialization of pointer arrays
- Multi-dimensional arrays
- Pointers vs. multi-dimensional arrays
- **Command line arguments**
- Address arithmetic (optional)

Command-line Arguments

```
> ./program arg1 arg2
---------------------------
int main(int argc, char* argv[]){
   ...
```

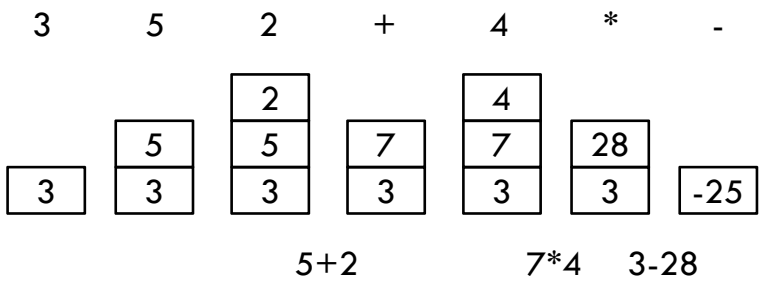
- argc : the argument count
  - argc = 3
  - The program name is the first argument
- argv : the argument vector
  - argv[0] = "./program"
  - argv[1] = "arg1"
  - argv[2] = "arg2"
  - argv[3] = NULL

Pointer Practice

An Exercise

- Exercise 5-10 in Section 5.10 of K&R
- Reverse Polish notation (postfix notation)
  - Use a stack to evaluate an expression like
    3 5 2 + 4 * -
  - Numbers get pushed on the stack
  - For an operator, pop the top two numbers and apply the operator (nextToTop operator veryTop), then push the result back on the stack

Reverse Polish Notation Example



- Build a program that takes a postfix expression as its command line arguments and returns the result
  - int main(int argc, char* argv[])
  - Use atoi(char*) to convert a string to an integer

An Exercise

```
int main(){
   const int bufferSize = 200;
   char wordBuffer[bufferSize];

   /* a list of DISTINCT words
      (i.e., no two words in this list should be equal */
   char **words;
   int numWords;

   /* read words from standard input until
      a newline or EOF */
   /* assume that a word is separated by spaces
      from adjacent words */
   /* use strcmp to test if words are equal */

   /* Exercise: complete the code */
}
```

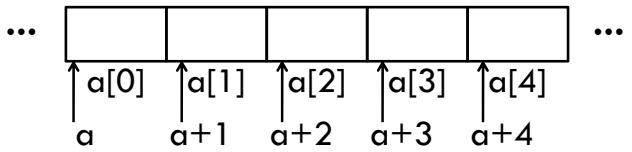
Optional Material: Address Arithmetic

Pointer + Int = Pointer

```
double a[5];
/* a  is a pointer to the first element in the array */
/* a == &a[0] */

double *pDbl = a;  /* pDbl == &a[0] */
pDbl = a + 2;      /* pDbl == &a[2] */
*(a + 3) = 7;    /* a[3] == 7   */
```

a+i == &a[i]

***(a+i) == a[i]**



Address Arithmetic

```
int i, n = 1000;
type *arr = malloc(sizeof(type) * n);
type *p = arr + 20, *p2;
/* p points 20 items after arr; p == &(arr[20]) */

p++;         /* p == &(arr[21]) */
p += 10;     /* p == &(arr[31]) */
p2 = p - 7; /* p2 == &(arr[24]) */
p -= 20;     /* p == &(arr[11]) */
/* p += arr;    NOT ALLOWED */

i = p2 - p; /* i == number of items from p through p2
                   minus one == 13 */
i = p - p2; /* i == -13 */
if(p2 <= p)  /* false */
p2 == p + (p2 - p) ???
```

Address Arithmetic

- Allowed:
  - Assigning pointers (of the same type)
  - Adding/subtracting a pointer and an integer
  - Subtracting or comparing two pointers
  - Assigning or comparing with NULL
- Anything else is not allowed, including...
  - Adding two pointers
  - Multiplication, division, shifting of pointers
  - Adding pointers with floating point numbers

An Exercise

- Write a function that takes three arguments:
  - A pointer that corresponds to an array of ints
  - The length of the array
  - Another int*
- The function should return 1 if the other pointer points to a valid element of the array and 0 otherwise

A Solution

```
int isValidMember(const int* array,
   const int length, const int* ptr){

   if(ptr <= array + length - 1 &&
      ptr >= array){
      return 1;
   }
   return 0;
}
```