Pointers & Memory Management in C

- *Learning Goals:*

  * Motivation

  * Pointer as an Abstract Data Type
  - Attributes and value domains
  - Operators (malloc, free, calloc, realloc)

  * Visualizing pointers w/ box-pointer diagrams
  - More Operators: Assignment, Comparison, Initialization
  - Yet More Operators (pointer arithmetic)

  * What are Pointers used for in C?
  - Dynamic Data-Structures, array, string,
  - result parameters

  * Common errors and how to handle those?
  - dangling pointers, memory leaks, ...
  - signal handlers for SIGSEGV, SIGBUS
  - malloc.h library

Pointers ADT

- *Pointer as an Abstract Data Type*
- *Attributes: value (unsigned integer)*
  * Domain: memory address
  * of base type T
- *Operations:*
  * address (&)
  * dereference (*)
  * assignment
  * relational, e.g. ==, !=
  * pointer arithmetic
  * memory allocation, deallocation

What can pointers point to?

- *Reference (&) and Dereference (*) operators*

```
int *point1;
int data1;
int main()
{       int *my_point = &data1;
        int *your_point;
        int more_data;

        point1 = &more_data;
        your_point = point1;
        *your_point = 17;
        *my_point = 33;
}
```

- *Draw memory diagram to explain the code.*

Memory allocation/deallocation

- *Simple Example - malloc(), free()*

```
#include <stdlib.h>
int main() {
        double *point;
        point = (double*) malloc(sizeof(double)) ;
        /* code to use the variable */
        free(point);
}
```

- *Access via . and -> operators*

```
struct date {
int month, day, year;
};
int main() {
        struct date *my_date;
        my_date = (struct date *)malloc(sizeof(struct date));
        (*my_date).year = 1776;
        my_date->month = 7;
        my_date->day = 4;
        /* code using my_date */
        free(my_date);
}
```

Memory allocation/deallocation routines

- *Q? How do we allocate/deallocate memory in C?*
- *void *malloc(numBytes)*
  * argument = number of bytes requested
  * returns pointer to allocated space
    int *p; p = malloc(sizeof(int));
- *void free(p)*
  * Recycle the space pointed to by pointer p
- *void *calloc(numItems, itemSize)*
  * Allocates space for an array of items
  * Returns pointer to beggingin of allocated space
  * Argument 1 = Number of items
  * Argument 2 = Size of an item
  * Items Initialized to 0
- *void *realloc(*oldSpace, sizeNewSpace) --> pointer to new space*

- copies oldSpace to newSpace, deallocates oldSpace.

Memory allocation/deallocation

- *Exercise: What will printf print?*

```
float *p, *q, *r;
 p = (float*) malloc(sizeof(float));
 q = (float*) malloc(sizeof(float));
  *p = 1.0; *q = *p;  r = q ;
 printf("%g, %g, %g ", *p,  *q, *r) ;
```

- *Exercise: What will printf print?*

```
student *s, *t;
s = (struct student*) malloc( sizeof(student) );
(*s).id = 1111;
strcpy(s->name, "Mary");
printf( "%d %s ", s->id,  (*s).name );
```

- *Q? Contrast the following pairs:*

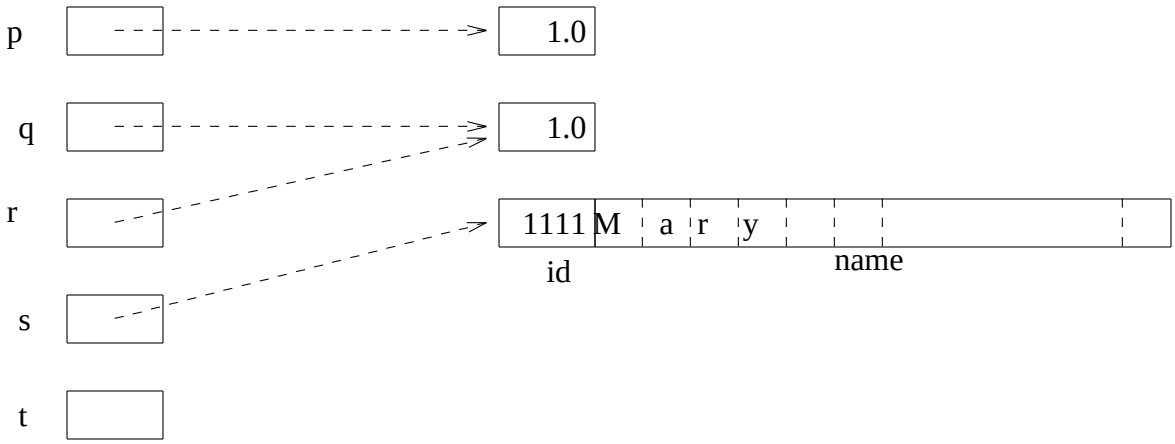
```
p,  *p;
(p == q), ((*p) == (*q))
(r == q),  (p == q)
```

- *Compare (*s).name vs. *(s.name) vs. *s.name*

  * Hint: priority(.) > priority(*))

Box Pointer Diagrams

- *Pointer Semantics: Box-pointer diagrams*
  - * Trace (declaration, memory allocation, assignment, ...)
  - * Ex. Draw box pointer diagram after each statement of program



What are Pointers used for in C?

- *C is a low level language*
  * relative to C++, Java, ...
  * Compiler support for many features missing
  * Programmers implement high level concepts
  * Self-discipline from Programmers is crucial
- *Do it yourself w/ Pointers*
  * Arrays
  * Strings
  * Parameter passing
    - result parameters
    - functions as parameters
  * Dynamic Data Structures

Pointers & Arrays

```
int index;
double ar[4];
double *pt;

int main()
{
        pt = ar + 2; /* pt point to ar[2] */
        for (index = 0; index < 4; index++) [
                ar[index] = (double)( 3 * index );
        }
}
```

- *Arrays are implemented with Pointers*
  - Elements are contiguous in memory
  - ar = const pointer to first element (i.e.ar[0])
  - ar[index] computes ar + index * size(element)
  - same as *(ar + index)

Pointers arithmetic with Strings

- *String = array of characters*
- *Q? What will the following print?*

```
char str[] = "ABCDEFG";
char *PC = str, *PC2 = PC + 1;
short X = 33; short *PX = &X;
printf("%c ", *PC ) ;
/* Pointer comparison (==, !=) */
if (PC != PC2) printf ("PC and PC2 are different") ;
/*pointer arithmetic */
/*pointer + number -> pointer */
PC =+ 4; printf("%c ", *PC ) ;
PC--; printf("%c ", *PC ) ;
/* pointer - pointer -> number */
printf("%d ", (PC2 - PC) ) ;
```

Pointers & Parameter Passing

- *Parameter Types*
  - * Input to function, or value
  - * Output from function or result
- *C only support Input parameters*
  - * Output parameters are implemented by pointers
  - * Example: swap() function

```
#include <stdio.h>
void swap(int *i, int *j)
{
        int t;
        t = *i;
        *i = *j;
        *j = t;
}
{
        int a,b;
        a=5;
        b=10;
        printf("%d %d0,a,b);
        swap(&a,&b);
        printf("%d %d0,a,b);
}
```

Multiple usage of pointers

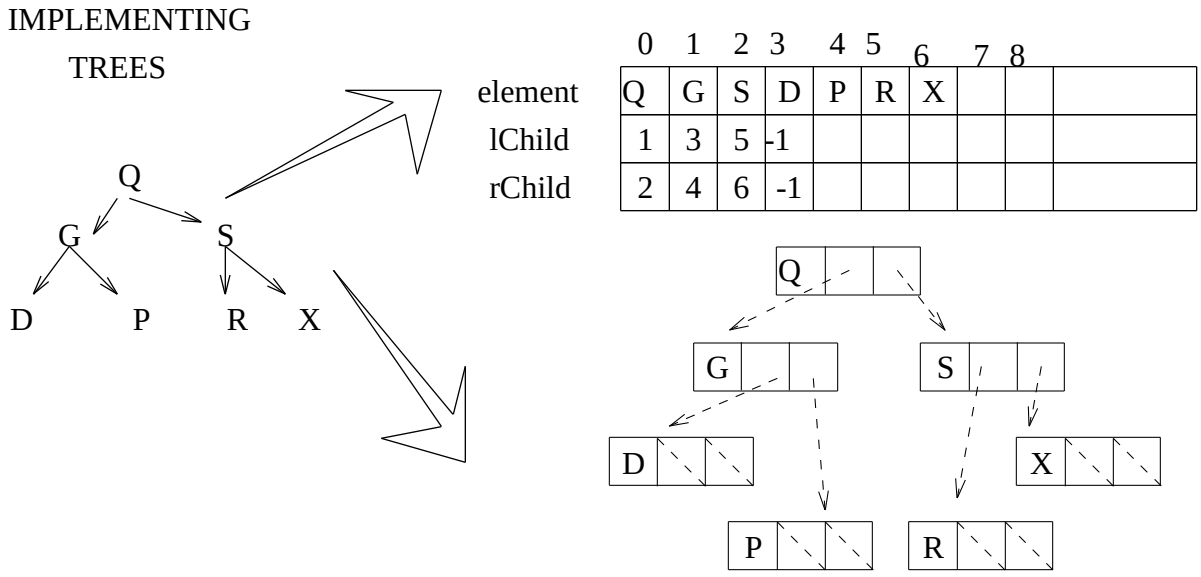
- *1.4 Programming in UNIX*
  * Extended example - argument arrays!
- *Strings = array of characters*
  char *s = "abc" ;
  char s[] = "abc" ;
- *Arrays of strings*
  int main(int argc, char *argv[]); /*Example 1.7, pp. 17 */
  char ** makeargv(char *s); /*Example 1.8, pp. 18 */
  char **myargv; /*Example 1.9, pp. 18 */
- *Parameter Passing*
  * Passing an array of string as result parameter
  * Example 1.12, page 19
    int makeargv(char *s, char *** argvp)
  * Program 1.2, page 22-23
    int makeargv(char *s, char *delimiters, char *** argvp)
- *Ex. Review Program 1.1 and 1.2 to answer the following:*
  * What are argv[] and argc used for?
  * What is the parameter passing mode in C?
  * What are the data types of arguments to makeargv()?

Why use Pointers and Dynamic data structures?

- *What are Dynamic data structures?*
  - * Collections which expand and contract as program executes.
  - * Different from Arrays, whose sizes are fixed at creation
- *Why do we use Dynamic data structures?*
  - * 1. Flexibility - conceptually closer to many data-structures
  - - e.g. Unix directory, roadmaps, electrical circuits, ...
  - * 2. Simplify programming - do not need to decide max_size
  - * 3. Performance - may save memory
  - - e.g. interpreter for Lisp, Magic (VLSI design editor), AutoCAD
- *Three areas of memory during program execution*
  - * Static Area - holds global variables
  - * Program Stack - holds local variables from functions/blocks
  - * Heap (free memory) - for dynamic use by program via pointers
- *How is Dynamic data structures implemented?*
  - * Declare Pointers
  - * Allocate memory at run-time as elements come
  - * Deallocate memory if elements are deleted
  - * A garbage collector recycles memory (Green thing!)

Comparing Implementations

- *Two Major Choices for Implementing Data Structures*
  - * Array Based OR Pointer Based
  - * Ex. Binary Tree - which implementation is preferred?



Common Errors With Pointers

- *Match list of errors with Code fragments:*
  - A. Never reference a node after it is deallocated
  - B. Do not return pointer to local var of functions
  - C. Never reference a pointer before it is allocated
  - D. Avoid Memory allocation in infinite loop
  - E. Use dereferencing operator (*, ->) whenever needed.
  - F. Use malloc(), free() with non-pointer arguments
- *Code Fragments*

```
/* fragment 1 */
 Tree *t1; printf("%d", t1->freq); ;
/* fragment 2 */
do { t1->left = new Tree; t1 = t1->left } while (TRUE);
/* fragment 3 */
 free(t1); printf("%d", t1->frequency()) ;
/* fragment 4 */
 Tree t1; t1 = malloc(sizeof(Tree)); free(t1);
 Tree *t1; *t1 = malloc(sizeof(Tree)); free(t1*); //error
/* fragment 5 */
Tree *t1, *t2; /* ...  */
printf("%d", t1.frequency()); /* error */
```

Revisit 1.4 Programming in UNIX

- *Extended example - argument arrays!*
  * Review pointers, argv[], argc, parameter passing
- *Ex. Review Program 1.1 and 1.2 to answer the following:*
  * What are argv[] and argc used for?
  * What is the parameter passing mode in C?
  * What are the data types of arguments to makeargv()?
  * What does makeargv() return?
  * List a few possible error situations for makeargv().
  - How does makeargv() respond to those errors?
  * Is it possible to rewrite makeargv() with following header?
  - Headers from Example 1.8, Example 1.10

    int makeargv(char *s, char *delimiters, char **argvp)
  * What is maximum number of arguments allowed?
  * Is there any memory leak? Justify your answer.
  - Consider memory allocated to ’t’ and ’*argvp’
- *Q? What the following loop do?*

```
for (i=1; i< numtokens + 1; i++)
   *((*argvp) + i) = strtok(NULL, delimiters);
```

  * Why is the above loop not followed by free(t)?