Review

Review Topics

- Structures
- Typedefs
- Pointers
- Testing
- Strings
- String Parsing
- String to Number Conversion

Structures

- Structure Declaration

```
#define NAME_LEN(50)
struct PERSONNEL_REC
{
    char last[NAME_LEN + 1];
    char first[NAME_LEN + 1];
    char middle[2];
    int ident;
};
```

- Structure Definition

```
struct PERSONNEL_REC new_employee;
```

Structures

- Pointers to Structures

```
struct PERSONNEL_REC new_employee;
  ...
print_emp ( &new_employee );
  ...
void print_emp( struct PERSONNEL_REC *emp )
{
  printf( "%d: %s, %s %s. \n",
            emp->ident,
            emp->last,
            emp->first,
            emp->middle
        );
}
```

Typedefs

- Definition

  a synonym (K&R) or a mnemonic abbreviation (H&S) for a data type and can be used anywhere a type specifier is permitted

- Syntax
  - typedef *as-type new-type*;
  - typedef *as-type *new-pointer-type*;
  - typedef *as-type new-type, new-type*;
  - typedef *as-type new-type*[num];

Typedefs

- Example

```
typedef int EMP_ID_t;
typedef char INITIAL_t[2];
typedef char NAME_t[NAME_LEN + 1];
typedef struct
{
   NAME_t last;
   NAME_t first;
   INITIAL_t middle;
   EMP_ID_t ident;
} EMPLOYEE_t, *EMPLOYEE_p_t;
   - or -
typedef struct PERSONNEL_REC EMPLOYEE_t, *EMPLOYEE_p_t;

EMPLOYEE_t new_employee;
```

Pointers

- Definition

  A pointer is the address of an object of a particular type, or a variable which may be used to represent such an address.

- Declarations
  - Include type and "*"

    ```
    int                    *ptr1;
    struct PERSONAL_REC *emp_rec;
    ```

  - Void Pointer Types (generic pointers)

    ```
    void *generic_ptr;
    ```

Pointers

- Valid Pointer Variable Values
  - NULL from stdio.h

    ```
    #include <stdio.h>
    int * ptr = NULL;
    ```

  - The address of an object of the appropriate type

    ```
    int num = 0;
    ptr = &num;
    ```

  - void pointers may assume the address of any object except the address of a function

    ```
    void * vptr = null;
    vptr = &num;
    ```

Pointers

- Valid Pointer Operations
  - Any pointer may be tested for equality with NULL

```
int *ptr = NULL;
ptr = foo();
if ( ptr == NULL )
   puts( "foo() failed!" );
```

  - Pointers may have integers added to or subtracted from them, as long as the result is another valid pointer

```
int * start, * end;
int array[100];
start = array;
end = start + 99;
```

Pointers

- Valid Pointer Operations
  - Pointers pointing to the same array may be compared

```
if ( end > start )
   puts( "yippee" );
```

  - Pointers pointing to the same array may be subtracted

```
int delta;
delta = end - start; /* 99 */
```

Pointers

- Valid Pointer Operations
  - Special case: an array consisting of *n* elements may have the address of its *n + 1th* element computed

```
int array[90];
int *ptr;
ptr = &array[0];
while ( ptr != &array[90] )
{
    *ptr = -1;
    ++ptr;
}
```

Pointers

- Object Definition

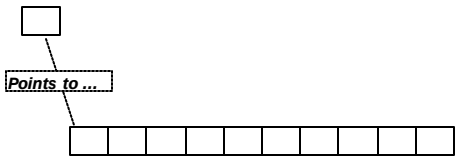
  K&R (Second Edition, p. 195)

  An object, sometimes called a variable, is a location in storage…

  - int * ptr; a pointer object
  - int arr[10]; 10 consecutive or contiguous **objects**
  - ptr = &arr[0];

    --*or*–

    ptr = arr;



Pointers

- Pointer Arithmetic
  - Syntax

    ptr + 1 → points to next element

    ptr + i → points to the ith element after ptr

    ptr - i → points to the ith element before ptr
  - These are equivalent expressions

    ptr = ptr + 1;

    ptr += 1;

    ptr++;

Pointers

- Pointer Arithmetic

  Given ptr = arr;

```
ptr → points to arr[0]
*ptr → refers to the contents of arr[0]
(ptr + 1)→ points to arr[1]
*(ptr + 1)→ refers to the contents of arr[1]
(ptr + i)→ points to arr[i]
*(ptr + i)→ refers to the contents of arr[i]
(ptr - i)→ points to arr[-i]
*(ptr - i)→ refers to the contents of arr[-i]
```

Testing

- Levels of testing
  - *Unit* -> workbench testing
  - *Integration* -> module-to-module, interfaces
  - *System* -> production simulation, performance
- Approaches
  - *White Box* -> code paths
  - *Black Box* -> requirements
  - *Model Office (or Beta)* -> “Let the customer test it”

Testing

- Test Plan vs Test Cases
  - *Test Plan* -> who, what, when
  - *Test Cases* -> how
- Developer Tools
  - *Products* -> Clarify/Purify, Test Case Mgmt Systems
  - Assert Macro
  - Conditional Preprocessing

Testing

- Conditional Preprocessing
  - #ifdef, #ifndef, #else, #endif
    - ```
      #ifdef SPECIAL_DEBUG
          printf( "fillet radius  =  %d \n", radius );
      #endif
      ```
    - ```
      #ifndef  NDEBUG
          void assert( int expression);
      #else
          #define assert(x)  ((void) 0)
      #endif
      ```

Testing

- Review #define
  - Preprocessor Statement
    - ```
      #ifndef TRUE
      #define TRUE 1
      #endif
      ```
  - As Compiler Directive (see reference for details)
    - cc –DTRUE=1 –DSPECIAL_DEBUG
      ```

Testing

- assert() - verify program assertion
  - #include <assert.h>
  - void assert(int expression);
  - Example

```
static int parse_sscanf( char * in_str, struct parse_s * parse_struct )
{
    ...
    assert( in_str );
    assert( parse_struct );
    ...
}
```

  - If assertion expression is NOT TRUE and if NDEBUG is NOT defined
    - Aborts program
    - Prints the expression, the file and line number

Strings

- In C, strings are not intrinsic
- A string is a character array ending with a *null character* ('\0')
- Or said another way, a string is a *null-terminated* character array
- The *null character* has the ASCII value 0
- The *null* character is logically different then NULL

String

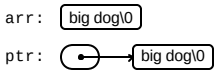
- String Constant
  - A string constant is an array of characters and is accessed by a pointer to its first element
  - Example → "big dog"
  - Usage
    - char arr[ ] = "big dog";
    - char * ptr = "big dog";
    - puts(arr);
    - puts(ptr);
    - puts("big dog");

String

- String Constant
  - The result of modifying the contents of a string constants is undefined (non-deterministic)
  - Initializing an array with a string constant
    - char arr[ ] = "big dog";
      is equivalent to:
      char arr[ ] = {'b','i','g',' ','d','o','g','\0' };
    - In both cases, the compiler allocates space to hold the characters and initializes the array elements.

String

- String Constant
  - Declaring a pointer to a string constant in not equivalent
    - char *ptr = "big dog";
  - Consider memory ...



Strings

- Compare and Assign
  - In C, strings cannot be assigned to other strings. Instead, the contents of one string are copied to other strings.
  - In C, strings cannot be compared. Instead, a character by character evaluation is performed.

Strings

- Compare and Assign

```
#include <stdio.h>
int main ( void )
{
    /* str1 is an 80 character strings */
    char str1[80+1];
    strcpy( str1, "big dog" );
    if ( strcmp( str1, "big dog" ) == 0 )
    {
        puts( "strings are equal" );
    }
}
```

```
void strcpy( char str1[], char str2[] )
{
    int inx = 0;
    while ( str1[inx] = str2[inx] )
        inx++;
}
```

Strings

- Safe String Copy
  - When source is large then target can accommodate ...

    char source[] = “the quick brown fox”;
    char target[10];
    strcpy(target, source);
  - ... strcpy results in non-deterministic memory error

Strings

- Safe String Copy
  - method using use *strncpy*

```
1.#define TARGET_LEN  10
  char source[] = “the quick brown fox”;
2. /* + 1 for ‘\0’ */
  char target[TARGET_LEN + 1];
3.strncpy( target, source, TARGET_LEN );
4.target[TARGET_LEN] = ‘\0’;
```

Strings

- Safe String Copy
  - When source is smaller then target can accommodate ...

```
1.#define TARGET_LEN  10
  char source[] = “hello”;
2./* + 1 for ‘\0’ */
  char target[TARGET_LEN + 1];
3.strncpy( target, source, TARGET_LEN )
4.target[TARGET_LEN] = ‘\0’;
```

String Parsing

- **int sscanf(char src[], char format[], ...);**
  - Reads from src string.
  - Converts according to format string (see reference for details),
  - Addition parameters (variable arguments) are ***pointers***.
  - Converted contents are stored in objects pointed by addition parameters.
  - Returns number of successful conversion.

String Parsing

- **int sscanf(char src[], char format[], ...);**
  - Defined in stdio.h.
  - Example

```
#define NUM_STR_LEN 10
char src[] = "number  3.2";
char numstr[NUM_STR_LEN + 1];
double number;
int retnum = sscanf(  str, "%s  %lf", numstr, &number );
```

  - Problems: objects must be big enough to hold converted contents.

String Parsing

- char * strtok(char src[], char set[]);
  - separates src in to *tokens* according to delimiters.
  - call strtok for each token, possible with different delimiters.
  - first call includes src, subsequent calls pass a NULL pointer.
  - returns pointer to null-terminated token
  - returns NULL if no token found.

String Parsing

- **char * strtok(char src[], char set[]);**

```
#define NUM_STR_LEN 10
char *  ptr = NULL;
char src[] = "number  3.2";
char numstr[NUM_STR_LEN + 1];
double number;
int  retnum;
ptr = strtok( src, " " );
if ( ptr == NULL )
{
     puts( "numstr not found" );
     return;
}
else
{
     strncpy( numstr, ptr, NUM_STR_LEN );
     numstr[NUM_STR_LEN]  = '\0';
}
ptr = strtok( NULL, " " );
if ( ptr == NULL )
{
     puts( "number not found" );
     return;
}
else
{
     number = strtod( ptr, NULL );
}
```

String to Number Conversion

- String to double ➔
  double strtod(const char * str, char ** ptr);
- String to long ➔
  long strtol(const char * str, char ** ptr, int base);
- String to unsigned long ➔
  unsigned long strtoul(const char * str,
  char ** ptr, int base);

String to Number Conversion

- String to Number Conversion
  - Usage

```
#include <stdlib.h>
{
        double value = 0.0;
        char * errStr = NULL;
        char * inputStr = "3.14";
        /* on return, errStr is a pointer to the last
         * character parse in inputString */
        value = strtod( inputString, &errStr );
        /* OR, pass NULL to ignore error string parameter */
        value = strtod( inputString, NULL );
}
```

- Problems with atio, atof, atol
  - H&S, p336, Sec. 13.9...
    "If atio, atof or atol are unable to convert the input string, their behavior is <u>undefined</u>."
  - These methods are non-deterministic !!!