CSC215 Lecture

Pointers and Arrays

Outline

- Physical and virtual memory
- Pointers
  - Declaration, operators, casting
  - Passing as arguments and returning from functions
- Arrays
  - Declaration, initialization, accessing individual elements
  - Arrays as constant pointers
  - Multidimensional arrays
- Pointer Arithmetic
  - Assignment, addition and subtraction, increment and decrement, comparative operators
  - Unary operators precedency
- Cryptic C code

Pointers and Memory Addresses

- Physical memory: physical resources where data can be stored and accessed by your computer
  - Cache
  - RAM
  - hard disk
  - removable storage
- Physical memory considerations:
  - Different sizes and access speeds
  - Memory management – major function of OS
  - Optimization – to ensure your code makes the best use of physical memory available
  - OS moves around data in physical memory during execution
  - Embedded processors – may be very limited

Pointers and Memory Addresses

- Virtual memory:
  - abstraction by OS
  - addressable space accessible by your code
- How much physical memory do I have?
  Answer: 2 MB (cache) + 2 GB (RAM) + 100 GB (hard drive) + . . .
- How much virtual memory do I have?
  Answer: <4 GB (32-bit OS)
- Virtual memory maps to different parts of physical memory
- Usable parts of virtual memory: stack and heap
  - stack: where declared variables go
  - heap: where dynamic memory goes

Pointers and variables

- Every variable residing in memory has an address!
  - What doesn't have an address?
    - register variables
    - literals/preprocessor defines
    - expressions (unless result is a variable)

- C provides two unary operators, `&` and `*`, for manipulating data using pointers
  - address operator `&`: when applied to a variable `x`, results in the address of `x`
  - dereferencing (indirection) operator `*`:

    when applied to a pointer, returns the value stored at the address specified by the pointer.

- All pointers are of the same size:
  - they hold the address (generally 4 bytes)
  - pointer to a variable of type T has type T*
  - a pointer of one type can be converted to a pointer of another type by using an explicit cast:

    `int *ip;`   `double *dp;`   `dp= (double *)ip;` OR `ip = (int*)dp;`

Examples

```
char a;              /* Allocates 1 memory byte */
char *ptr;           /* Allocates memory space to store memory address */
ptr = &a;            /* store the address of a in ptr. so, ptr points to a */
int x = 1, y = 2, z[10]={0, 1, 2, 3, 4, 5, 4, 3, 2, 1};
int *ip;             /* ip is a pointer to int */
ip = &x;             /* ip now points to x */
y = *ip;             /* y is now 1 */
*ip = 0;             /* x is now 0 */
ip = &z[0];          /* ip now points to z[0] */
printf("%d %d %d", x, y, *ip);
y = *ip + 1;
printf("%d %d %d", x, y, *ip);
*ip += 1;
printf("%d %d %d", x, y, *ip);
```

```
0 1 00 1 00 1 1
```

Dereferencing & Casting Pointers

- You can treat dereferenced pointer same as any other variable:
  - get value, assign, increment/decrement
- Dereferenced pointer has new type, regardless of real type of data
- null pointer, i.e. 0 (NULL): pointer that does not reference anything
- Can explicitly cast any pointer type to any other pointer type
  `int* pn; ppi = (double *)pn;`
- Implicit cast to/from `void *` also possible
- Possible to cause segmentation faults, other difficult-to-identify errors
  - What happens if we dereference ppi now?

Passing Pointers by Value

```c
/* Does not work as expected*/
void swap(int a, int b){
  int temp = a;
  a = b;
  b = temp;
}

int main(){
  int a[] = {3, 5, 7, 9};
  swap(a[1], a[2]);
  printf("a[1]=%d, a[2]=%d\n", a[1], a[2]);
  return 0;
}
```

```c
/* Works as expected*/
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(){
  int a = {3, 5, 7, 9};
  swap(&a[1], &b[2]);
  printf("a[1]=%d, a[2]=%d\n",a[1], a[2]);
  return 0;
}
```

Function Returning a Pointer

- ❑ Functions can return a pointer
  Example: `int * myFunction() { . . . }`

- ❑ But: never return a pointer to a local variable

```
#include <stdio.h>
char * get_message ( ) {
  char msg[] = "Hello";
  return msg;
}
int main ( void ){
  char * str = get_message() ;
  puts(str);
  return 0;
}
```

```
#include <stdio.h>
char * get_message ( ) {
  static char msg[] = "Hello";
  return msg;
}
int main ( void ){
  char * str = get_message() ;
  puts(str);
  return 0;
}
```

- ❑ unless it is defined as static

- ❑ Multiple returns? Use extra parameters and pass addresses as arguments.

Arrays

- Fixed-size sequential collection of elements of the same type
- Primitive arrays implemented as a pointer to block of contiguous memory locations
  - lowest address corresponds to the first element and highest address to the last element
- **Declaration:** <element_type> <array_name>**[**<positive_int_array_size>**]**;
  Example: int balance[8]; /* allocate 8 int elements*/
- **Accessing individual elements:** <array_name>[<element_index>]
  Example int a = balance[3]; /* gets the 4th element's value*/
- **Array Initializer:** <type> <name>**[**<optional_size>**] = {**<comma-sep elements>**}**;
  <optional_size> must be >= # of elements

Arrays

- Under the hood: the array is constant pointer to the first element

  ```
  int *pa = arr; ⇔ int *pa = &arr[0];
  ```

- Array variable is not modifiable/reassignable like a pointer

  ```
  int a[5];
  int b[] = {-1, 3, -5, 7, -9};
  a = b;
  error: assignment to expression with array type
  ```

- `arr[3]` is the same as `*(arr+3)`: to be explained in few minutes

- Iterating over an array:

  ```
  int i;                                    int *pi;
  for(i = 0; i < n; i++)        ⇔        for(pi = a; pi < a + n; pi++)
    arr[i]++;                                         (*pi)++;
  ```

Strings

- There is no string type, we implement strings as arrays of chars

```
char str[10]; /* is an array of 10 chars or a string */
char *str; /* points to 1st char of a string of unspecified length
                       BUT no memory is allocated here! */
```

- Header file `string.h` in the standard library has numerous string functions
  - they all operate on arrays of chars and include:

    `strcpy(s1, s2)` : copies s2 into s1 (including '\0' as last char)
    `strncpy(s1, s2, n)` : same but only copies up to n chars of s2
    `strcmp(s1, s2)` : returns a negative int if s1 < s2, 0 if s1 = = s2 and a positive int if s1 > s2
    `strncmp(s1, s2, n)` : same but only compares up to n chars
    `strcat(s1, s2)` : concatenates s2 onto s1 (this changes s1, but not s2)
    `strncat(s1, s2, n)` : same but only concatenates up to n chars
    `strlen(s1)` : returns the integer length of s1
    `strchr(s1, ch)` : returns a pointer to the 1st occurrence of ch in s1 (or NULL if not found)
    `strrchr(s1, ch)` : same but the pointer points to the last occurrence of ch
    `strstr(s1, s2)` : substring, return a pointer to the char in s1 that starts a substring that matches s2, or NULL if the substring is not present

Arrays

- Array length? no native function

```
#include <stdio.h>
int main() {
  char* pstr = "CSC215";
  printf("%s\t%d\n", pstr, sizeof(pstr));
  char astr[7] = "CSC215";
  printf("%s\t%d\n", astr, sizeof(astr));
  int aint[10];
  printf("%d\t%d\n", sizeof(aint[0]), sizeof(aint));
  int* pint = aint;
  printf("%d\t%d\n", sizeof(pint[0]), sizeof(pint));
  return 0;
}
```

```
CSC215    4
CSC215    7
4         40
4         4
```

- How about: `sizeof(arr)==0?0 : sizeof(arr)/sizeof(arr[0]);`
  can be defined as a macro:

```
#define arr_length(arr)(sizeof(arr)==0?0 : sizeof(arr)/sizeof((arr)[0]))
```

Multidimensional Arrays

- **Syntax:** <type> <name>**[**<dim1size>**][**<dim2size>**]...[**<dimNsize>**]**;
  Example: `int threedim[5][10][4];`

- **Initializer:** `= { { {..},{..},{..}}, {...}, {...}}`
  Example: `int twodim[2][4]={{1,2,3,4},{-1,-2,-3,-4}}; /* or simply: */`
  `int twodim[2][4]={1, 2, 3, 4, -1, -2, -3, -4};`

  - You cannot omit any dimension size if no initializer exists

- **Accessing individual elements**:
  <name>**[**<dim1index>**][**<dim2index>**]...[**<dimNindex>**]**
  Example: **`twodim[1][2]`**`=5; printf("%d\n", `**`twodim[0][3]`**`);`

- **Allocation**:

twodim 3bf71a0c	3bf71a10	3bf71a14	3bf71a18	3bf71a1c	3bf71a20	3bf71a24	3bf71a28
1	2	3	4	-1	-2	5	-4

Multidimensional Arrays

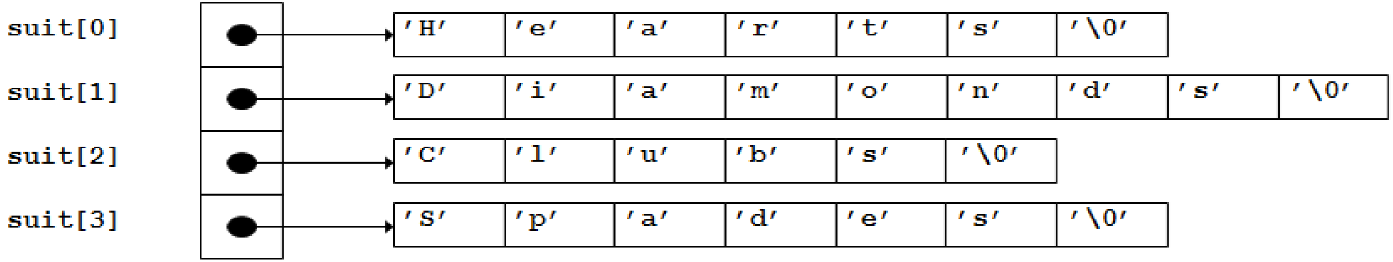
- **Pointer style:** <type> ** <name>; /* add * for every extra dimension */
  a pointer to the 1st element of an array, each element of which is a pointer to the 1st element in an array
- More flexibility:
  Example: char b[4][7] = {"CSC111", "CSC113", "CSC212", "CSC215"};
  char *bb[] = {"CSC215", "This is a beautiful morning","M","I guess so"};
- Still have [ ]?
  - To define pure pointer 2D array:
    - Declare <type>** x variable
    - Allocate memory for N elements of type <type>* (1st dimension)
    - For each of these elements, allocate memory for elements of type <type> (2nd dimension)
  - Ignore it for now, you need to learn about memory managements in C first.
- Arguments to main: int main(int argc, char** argv){ … }
  - Name of the executable is always the element at index 0
    for (i=0; i<argc; i++) printf("%s\n", argv[i]);

Arrays of Pointers

- ❑ Example is an array of strings:

```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

  - strings are pointers to the first character
  - `char *` each element of suit is a pointer to a char
  - strings are not actually stored in the array `suit`, only pointers to the strings are stored
  - `suit` array has a fixed size, but strings can be of any size



Pointer Arithmetic

- ❑ Assignment operator = : initialize or assign a value to a pointer
  - value such as 0 (`NULL`), or
  - expression involving the address of previously defined data of appropriate type, or
  - value of a pointer of the same type, or different type casted to the correct type
- ❑ Arithmetic operators + , −: scaling is applied
  - adds a pointer and an integer to get a pointer to an element of the same array
  - subtract an integer from a pointer to get a pointer to an element of the same array
  - Subtract a pointer from a pointer to get number of elements of the same array between them
- ❑ Increment/Decrement ++ , --: scaling is applied
  - result is undefined if the resulting pointer does not point to element within the same array
- ❑ Comparative operators:
  - == , != : can be used to compare a pointer to 0 (NULL)
  - == , != , > , >= , < , <= : can be used between two pointers to elements in the same array
- ❑ All other pointer arithmetics are illegal

Example: Increment/Decrement Operators

```c
#include <stdio.h>
int main (){
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < 3; i++){
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Var bf882b30	10
bf882b34	100
bf882b38	200
i	2
ptr	bf882b38

Example: Comparative operators

```
#include <stdio.h>
const int MAX = 3;
int main (){
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;
  while ( ptr <= &var[MAX - 1] ){
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* point to the next location */
    ptr++;
    i++;
  }
  return 0;
}
```

Precedence of Pointer Operators

- Unary operators `&` and `*` have same precedence as any other unary operator
  - with associativity from right to left.
- Examples:

```
c=*++cp          c=*(++cp)
c=*cp++          c=*(cp++)
c=++*cp          c=++(*cp)
       ???                      c=(*cp)++
```

Segmentation Fault & Pointer Problems

- OS assigns a portion of the memory to your program
  - Any access attempt to a memory outside this portion is impermissible
- Segmentation fault <u>may</u> be a result of any of the following main causes:
  - Dereferencing NULL
  - Dereferencing an uninitialized pointer
  - Dereferencing a freed or out-of-scope pointer
  - Writing off the end of an array
- Usually the state of the program when the fault occurs is dumped to the disk
  - You might have seen the message: "Segmentation Fault (core dumped)"
  - The core helps in debugging the program and finding out what part of it causes the fault
  - You need to use a debugger: e.g. GDB

Cryptic vs. Short C Code

- Consider the following function that copies a string into another:

```
void strcpy(char *s, char *t){
            int i;
            i = 0;
            while ((*s = *t) != '\0') {
              S++;
              T++;
            }
          }
```

  - Now, consider this

```
void strcpy(char *s, char *t){
  while ((*s++ = *t++) != '\0');
}
```

  - and this

```
void strcpy(char *s, char *t){
  while (*s++ = *t++);
}
```

- Obfuscation (software)
- The International Obfuscated C Code Contest *http://www.ioccc.org/*