CSC215 Lecture

Advanced Pointers

Outline

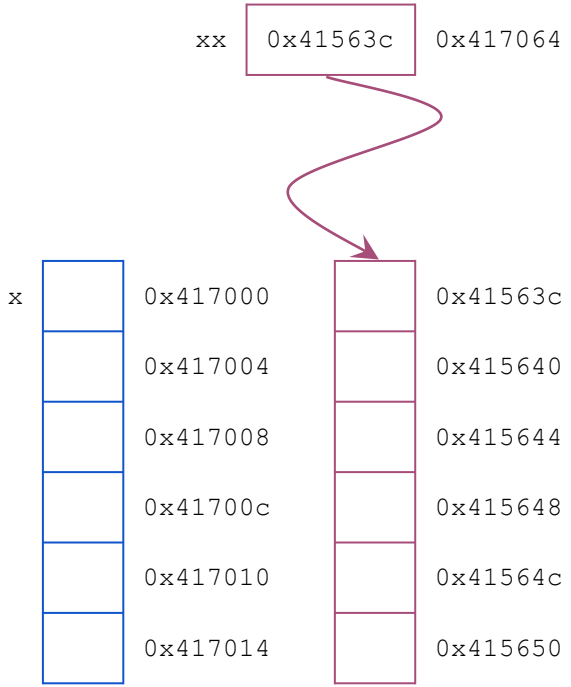
- Pointer to Pointer
  - Pointer Array
  - Strings Array
  - Multidimensional Array
- void Pointers
- Incomplete Types
- Pointer to Function

Array vs. Pointer

```
#include <stdio.h>
int main() {
  int x[5];
  int* xx = (int*)malloc(5*sizeof(int));

  printf("%p\n", x);
  printf("%p\n", x+1);
  printf("%p\n", &x);
  printf("%p\n", &x+1);
  printf("%d\n", (int)sizeof(x));
  printf("==========\n");
  printf("%p\n", xx);
  printf("%p\n", xx+1);
  printf("%p\n", &xx);
  printf("%p\n", &xx+1);
  printf("%d\n", (int)sizeof(xx));
  return 0;
}
```

```
0x417000
0x417004
0x417000
0x417014
20
==========
0x41563c
0x415640
0x417064
0x417068
4
```



Pointer to Pointer

- Pointer represents address to variable in memory
- Address stores pointer to a variable is also a data in memory and has an address
- The address of the pointer can be stored in another pointer
- Example:

```
int n = 3;
int *pn = &n; /* pointer to n */
int **ppn = &pn; /* pointer to address of n */
```

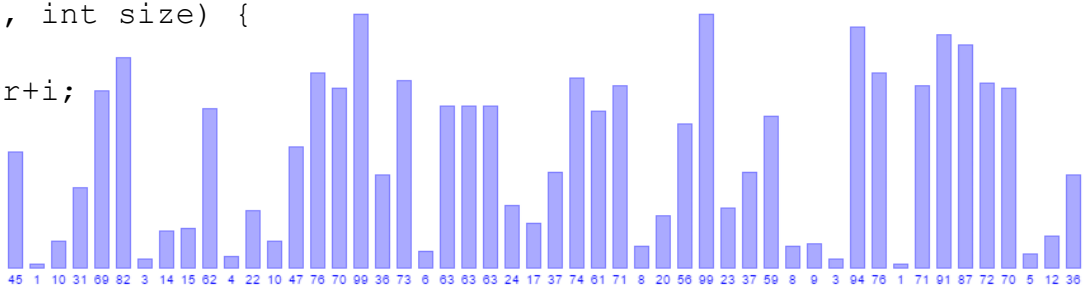
- Many uses in C: pointer arrays, string arrays, multidimensional arrays

Pointer Arrays Example

- Assume we have an array `int arr [20]` that contains some numbers
  `int arr[20]= {73,59,8,82,48,82,84,94,54,5,28,90,83,55,2,67,16,79,6,52};`
- Want to have a sorted version of the array, but not modify arr
- Declare a pointer array: `int* sarr[20]` containing pointers to elements of arr and sort the pointers instead of the numbers themselves
- Good approach for sorting arrays whose elements are very large (like strings)
- Example: insert sort
  - `void shift _element(int* sarr[], int i)`
  - `void insert_sort(int arr[], int* sarr[], int size)`

Pointer Arrays Example

```
#include <stdio.h>
void shift_element (int* sarr[], int i) {
  int* p2i;
  for (p2i = sarr[i]; i > 0 && *sarr[i-1] > *p2i; i--)
    sarr[i] = sarr[i-1];
  sarr[i] = p2i;
}
void insert_sort(int arr[], int* sarr[], int size) {
  int i;
  for (i=0; i < size; i++) sarr[i] = arr+i;
  for (i=1; i < size; i++)
    if (*sarr[i] < *sarr[i-1])
      shift_element(sarr, i);
}
int main(){
  int i, arr[20]={73,59,8,82,48,82,84,94,54,5,28,90,83,55,2,67,16,79,6,52}, *sarr[20];
  insert_sort(arr, sarr, 20);
  for (i = 0; i < 20; i++) printf("%d\t", *(sarr[i]));
  return 0;
}
```



String Array Example

- An array of strings, each stored as a pointer to an array of chars
  - each string may be of different length

```
char word1[] = "hello";    /* length = 6 */
char word2[] = "goodbye";  /* length = 8 */
char word3[] = "welcome!"; /* length = 9 */
char* str_arr[] = {word1, word2, word3} ;
```

- Note that str_arr contains only pointers, not the characters themselves!

Multidimensional Arrays

- C permits multidimensional arrays specified using [ ] brackets notation:

```
int world[20][30]; /* a 20x30 2-D array of integers */
```

- Higher dimensions are also possible:

```
char big_matrix[15][7][35][4]; /* what are the dimensions of this?
                               /* what is the size of big_matrix? */
```

- Multidimensional arrays are rectangular, while pointer arrays can be of any shape
- See: Lecture 05, Lab 05, Lecture 07

void Pointers

- C does not allow declaring or using void variables.
- void can be used only as return type or parameter of a function
- C allows void pointers
  - What are some scenarios where you want to pass void pointers?
- void pointers can be used to point to any data type

```
int x; void* px = &x; /* points to int */
float f; void* pf = &f; /* points to float */
```

- void pointers cannot be dereferenced
  - The pointers should always be cast before dereferencing

```
int x=5; void* px=&x;
printf("%d",*px); /* warning: dereferencing 'void *' pointer
                            error: invalid use of void expression */
printf("%d",*(int*)px); /* valid */
```

Incomplete types

- Types are partitioned into:
  - object types (types that fully describe objects)
    Example:
    - `float x;`
    - `char word[21];`
    - `struct Point (int x, int y};`
  - function types (types that describe functions)
    - characterized by the function's return type and the number and types of its parameters
  - incomplete types (types that describe objects but lack information needed to determine their sizes)
    - A struct with unspecified members: Ex. `struct Pixel;`
    - A union with unspecified members: Ex. `union Identifier;`
    - An array with unspecified length: Ex. `float[]`

- A pointer type may be derived from:
  - an object type
  - a function type, or
  - an incomplete type

Pointer to Incomplete Types

- Members of a struct must be of a complete type
- What if struct member is needed to be of the same struct type?

```
struct Person{
  char* name;
  int age;
  struct Person parent; /* error, struct Person is not complete yet */
};
```

- Pointers may point to incomplete types

```
struct Person{
  char* name;
  int age;
  struct Person* parent; /* valid */
}
```

- Good news for linked lists!

Function Pointers

- Functions of running program are stored in a certain space in the main-memory
- In some programming languages, functions are first class variables (can be passed to functions, returned from functions etc.).
- In C, function itself is not a variable
  - but it is possible to declare pointer to functions.
- Function pointer is a pointer which stores the address of a function
  - What are some scenarios where you want to pass pointers to functions?
- Declaration examples:

```
int (*fp1)(int)
int (*fp2)(void* ,void*)
int (*fp3)(float, char, char) = NULL;
```

- Function pointers can be assigned, passed to/from functions, placed in arrays etc.

Function Pointers

- Typedef Syntax:

```
typedef <func_return_type> (*<type_name>)(<list_of_param_types>);
```

- Declaration Syntax:

```
<func_return_type> (*<func_ptr_name>)(<list_of_param_types>); /* or */
<type_name> <func_ptr_name>;
```

- Assignment Syntax:

```
<func_ptr_name> = &<func_name>; /* or */
<func_ptr_name> = <func_name>;  /* allowed as well */
```

- Calling Syntax:

```
(*<func_ptr_name>)(<list_of_arguments>); /* or */
<func_ptr_name>(<list_of_arguments>);    /* allowed as well */
```

- Example:

```
void print_sqrt(int x){
      printf("%.2f\", sqrt(x));
    }
```

```
/* use */ void (*func)(int);
func = &print_sqrt;

(*func)(25);
```

Function Pointers Examples

```c
#include <stdio.h>
#include <math.h>

int f1(float a){
  return (int)ceil(a);
}

int f2(float a){
  return (int)a;
}

int main(){
  int (*func)(float);
  float  f;
  scanf("%f", &f);
  func = (f - (int)f >= 0.5)? &f1:&f2; /* or f1:f2 */
  printf("Rounding of %f is %d\n", f, *func(f) /* or func(f) */);
  return 0;
}
```

```
> test
3.7
Rounding of 3.70 is 4
> test
3.3
Rounding of 3.30 is 3
```

```c
typedef int(*Fun)(float);
Fun func;
```

Function Pointers: Callbacks

- Definition: Callback is a piece of executable code passed to functions.
- In C, callbacks are implemented by passing function pointers.
- Example:

```
void qsort(void* arr, int num,int size,int (*fp)(void* pa, void* pb))
```

  - `qsort()` function from the standard library can be used to sort an array of any datatype.
  - How does it do that? Callbacks.
    - `qsort()` calls a function whenever a comparison needs to be done.
    - the function takes two arguments and returns (<0 , 0 , >0) depending on the relative order of the two items.

```
int a rr [ ] ={ 1 0 , 9 , 8 , 1 , 2 , 3 , 5 };
int asc ( void* pa , void* pb ){
  return ( *(int*)pa - *(int*)pb ) ;
}
int desc ( void* pa , void* pb ){
  return ( *(int*)pb - *(int*)pa ) ;
}
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), asc);
     qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), desc);
```