C Tutorial
Session #4

- More on Arrays and Pointers
- Memory allocation
- Structures, Unions & typedef

Review

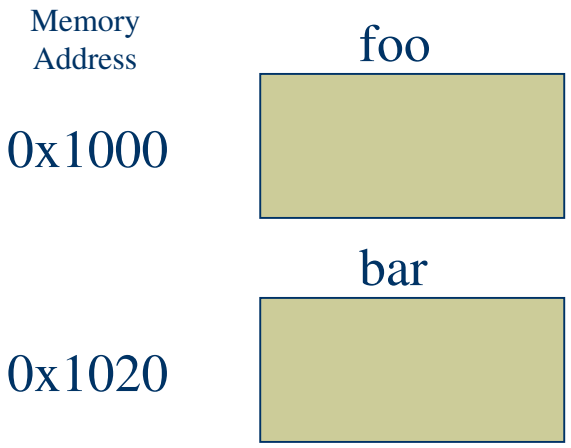
- ```
  int iArr[30];
  &iArr[0] == &iArr; // equivalent
  ```

- ```
  int (*p)[2]; // pointer to array of 2 integers
  int *p[2];  // array of 2 pointers to integers
  ```

- ```
  int foo = 100;
  int *bar;
  bar = &foo;
  *bar = 200;
  ```

  What is:

  1. foo;
  2. &foo;
  3. bar;
  4. *bar;
  5. &bar;


  ```

Pointers to multidimensional arrays

```
int Zippo[2][4];
```

- Zippo is the starting address of this 2-d array. It is same as &Zippo[0].
- Zippo[0] itself is an array of 4 integers.
- Zippo[0] is same as &Zippo[0][0].

Difference between arrays and pointers

- Array name is a constant.
- Pointer is a variable.
- Example:

```
char hello[] =“hello”;
char *helle = “helle”;
hello++;  //illegal
helle++; //legal
```

Dynamic arrays

- You don't always know your array size in advance.
- Static arrays waste a lot memory.
- `malloc()` and `free()`

malloc ( )

- Example of the usage of malloc ( ):

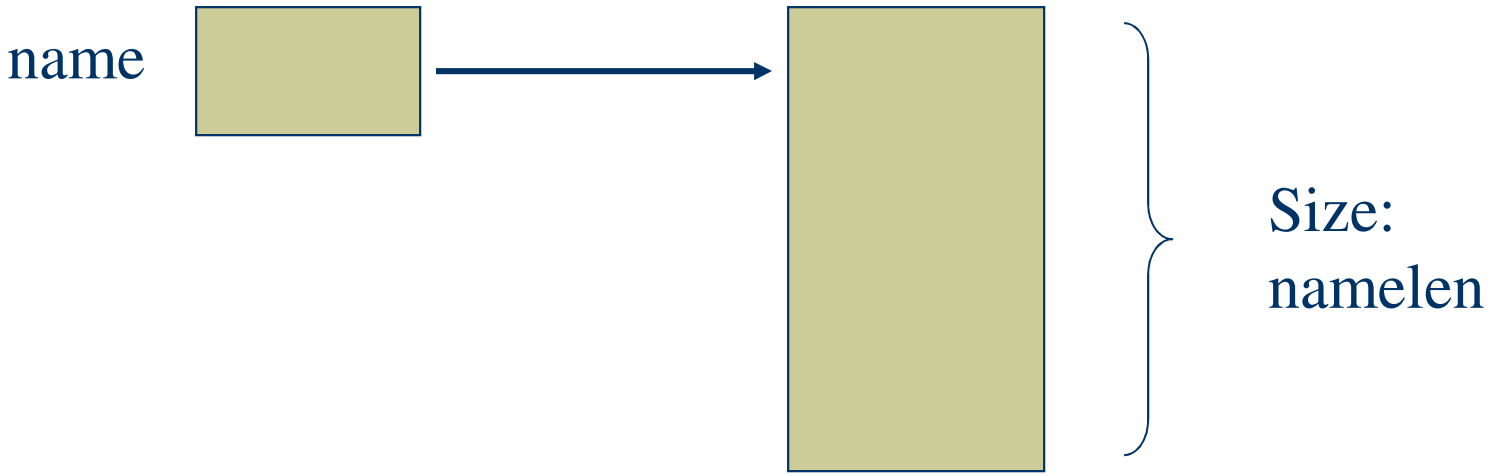
```
char *name;
name = (char *) malloc(length * sizeof(name));
```

Dynamically allocate memory

- *malloc* allocates a chunk of memory and returns the starting address of the memory.
- *free* frees memory
- Note: you are responsible for releasing the memory allocated by malloc function.

An example of malloc

```
char *name = (char *)malloc(namelen * sizeof(char));
```



Sample code

```
int main(){
   char name[] = "I am a string.";
   char *temp;

   temp = (char *)malloc( (strlen(name)+1) *
             sizeof(char));
   strcpy(temp, name);
   …
   free(temp);
}
```

Caution: memory leak

```
char str1[] = "string one";
char str2[] = "string two";
char *temp;

temp = (char *) malloc ((strlen(str1) + 1) *
                        sizeof(char));
strcpy(temp, str1);
temp = (char *) malloc ((strlen(str2) + 1) *
                        sizeof(char));
strcpy(temp, str2);
free(temp);
```

Two dimensional dynamic array

- Use pointer to pointers.
- Syntax:

  *<type> **name;*
- Example:

  *int **arr;*
- *arr* is a pointer to an integer pointer. It can be used as a 2-d array.

Sample usage

- Suppose we need a 2-d array to store a matrix. However, neither the column size nor the row size of the matrix is known. Thus, we need to use a 2-dim dynamic array.
- Code:

```
int **matrix;
```

- Question: how do we allocate 2-dim memory.

Matrix example

```
int **matrix;
matrix = (int **)malloc(row * sizeof(int *));
for (i = 0; i< col; i++) {
  matrix[i] = (int *)malloc(col *sizeof(int));
}
```

Caution: we need a for loop to free that memory

Dynamically change array size

- Sometimes we need to change array size in the middle of programs.
- The *realloc* function.
- Syntax:

  realloc (baseadd, newsize);

Example of using realloc

```
char f_name[] = "John";
char l_name[]= "Smith";
char *name = (char *)malloc((strlen(f_name)+1) *
                    sizeof(char));
strcpy(name, f_name);
name = (char *)realloc(name,
   (strlen(f_name)+strlen(l_name)+1) *
   sizeof(char));
strcat(name, l_name);

// memory leak…
sprintf (name, "%s, %s", l_name, f_name);
```

Introduction to structures

- Choosing a good way to represent data is very important
- “A structure is a collection of one or variables, possibly of different types, grouped together under a single name for convenient handling.”

```
struct point {
    int x;
    int y;
};
```

Example: book inventory

```
struct book {
    char title[MAX];
    char author[MAX];
    float value;
    }
struct book library; //declare a book variable
struct book library1;

gets(library.title);
scanf("%f", &library.value);

struct book library2 = {"The Host",
   "Meyers, Stephanie", 19.95");
```

Structures (cont.)

- Operations on a structure
  - copy it
  - assign to it as a unit
  - take its address with &
  - accessing its members
- Pointers to structures

```
struct book library, *bk;
bk = &library;
strcpy (bk->title, "The Hunger Games”);
```

- Arrays of Structures

```
struct book library[MAX];
```

Unions

- Unions allow a variable to hold different types (at different times)

```
union u_tag {
    int ival;
    float fval;
    char &sval;
} u;
```

- Allocated to hold the largest possible value
- Accessed in same manner as structures:

```
printf ("string val %s", u.sval);
printf ("int val %d", u.ival);
```

typedef

- Create new data type names

```
typedef int Length;
Length len, maxlen, *lengths[];
typedef char *String;
typedef struct tnode *Treeptr;
typedef struct tnode {     // tree node
   char *word;             // points to the text
   int  count;             // number of occurences
   Treeptr left;           // left child
   Treeptr right;          // right child
} Treenode;
Treeptr talloc(void) {
   return (Treeptr) malloc (sizeof (Treenode));
}
```

- typedef does not create a new type, only adds a new name for some existing type