Pointers in C

*Adapted from Dr. Sally's slides

Pointers in C

```
char c;
c = 'B';

char * pC;
pC = &c;

printf("%c", *pC);    \\ Will print 'B'

*pC = 'C';
```

"Address"

Index	Memory	
1000		
1001		
1002		
1003	'B'	c
1004		
1005		
1006	1003	pC
1007		
1008		
1009		
1010		
1011		
1012		
1013		
1014		

Pointers in C

- To create a variable that will hold an address:
```
data-type-name * varname;
int * pCount;
CAT_T * catArray;
```
- All pointers are addresses. This means that all pointer values are positive integers.
- To find out the address of a variable: `&variableName`
  - E.g., `pCount = &count;   /* get address of count */`
- To get or set the data stored at an address: `*pointerVariableName`
- E.g.,
```
int result = *pCount;    /* get data stored at address pCount */
*pCount = 10;  /* set new value into address referenced by pCount */
```

Common Mistake:

- `char a, b;` ← Both `a, b` are char values
- `char * a, b;` ← Only `a` is pointer, `b` is char value!

If you want both to be pointers:

- `char * a, * b;`

Pointers & Arrays in C

The following are equivalent in C:

```
char myCharArray[]      ⇔   char * myCharArray      ⇔   &myCharArray[0]
int myIntArray[]        ⇔   int * myIntArray        ⇔   &myIntArray[0]
char * charPtrArray[]   ⇔   char ** charPtrArray    ⇔   &charPtrArray[0]
DATE_T dateArray[]      ⇔   DATE_T * dateArray ⇔ &dateArray[0]
```

So we can do, e.g.,

```
char* newString = calloc(strlen(oldString)+1,sizeof(char));
for (i = 0; i < strlen(oldString); i++)
     newString[i] = oldString[i];
```

Dynamic Allocation in C

*Adapted from Dr. Sally's slides

Dynamically Allocation in C: Calloc

Use `void * calloc(int elementCount,int elementSize)`

- `calloc` allocates `elementCount * elementSize` consecutive bytes and sets their values to 0
- `calloc` returns the address of the first allocated byte, or NULL for error

```
#define ARRAYSIZE 300
int* myValues = calloc(ARRAYSIZE,sizeof(int));
if (myValues == NULL)
{
    /* handle error condition */
}
...
```

Dynamically Allocation in C: Other functions

```
void * malloc(int totalBytes)
```

- *Almost* same as `calloc(elementCount, elementSize)`
- Except: Does not initialize value to zeros

```
char * strdup(char* stringToCopy)
```

- For allocating memory and copying into `stringToCopy` its value
- In other words, almost the same as

```
char* newString = calloc(strlen(oldString)+1,sizeof(char));
for (i = 0; i < strlen(oldString); i++)
    newString[i] = oldString[i];
```

- Except: Arrays created with `strdup()` should be treated as read-only

Releasing Memory with free()

- Memory is a limited resource
- When no longer need, dynamically-allocated memory should be released
- To do this, use the `free()` function.

```
void free(void* pointerToDynamicMemory);
```

- The argument is the pointer returned from a call to `calloc()`
- Good practice: Free memory at the end of functions, in a single block of code

Common Mistakes

- **Not checking the allocated pointer**
  - Always check after a call to `calloc` (or `malloc` or `strdup`) to make sure that the returned value is not NULL.
  - If the pointer is NULL, the memory allocation failed.
    - You should handle this as an error. Do not continue processing.
- **Not freeing allocated memory when you are done with it**
  - This is called a ***memory leak***.
  - Memory is "leaking" out of the available pool, taken out but never returned.
  - In a small program, this usually won't be noticed (but it is still a bug)
  - In a large (real-world) program, memory leaks can have many serious consequences, including unexpected failures and reduced performance.

Common Mistakes II

- **Using a dynamically allocated pointer after it has been freed**
  - Once you call `free()` on a pointer, you must not use that pointer again
  - Set your pointers to NULL after you call free

```
free(names1);
names1 = NULL;
```

  - A related problem is freeing the same pointer twice
- **Calling free on a pointer that was not dynamically allocated**

```
int x = 100;          // declares an integer variable
int* pX = NULL;       // declares an int pointer variable
pX = &x;              // sets the value of the pointer to the address of x
…
free(pX);             // Will crash!! pX does not point to dynamic memory!
```

For Self-Study: Memory in C

C has three separate pools of memory

- **Static memory**
  - Holds data items declared outside any function
  - Can access anywhere in program
  - Allocated as part of the compile/link process
  - Lifetime: entire execution of the program
- **Stack memory**
  - Holds data items declared within a function or block
  - Can access only within that function or block (or within nested blocks)
  - Allocated when execution enters the block & Released when execution leaves the block
  - Lifetime: only during the time code in the function/block is executing
- **Heap memory**
  - Significantly larger capacity than stack memory
  - Holds data items created by calls to calloc, malloc, etc.
  - Allocated when program calls calloc, released when program calls free
  - Lifetime: determined by the programmer

Example

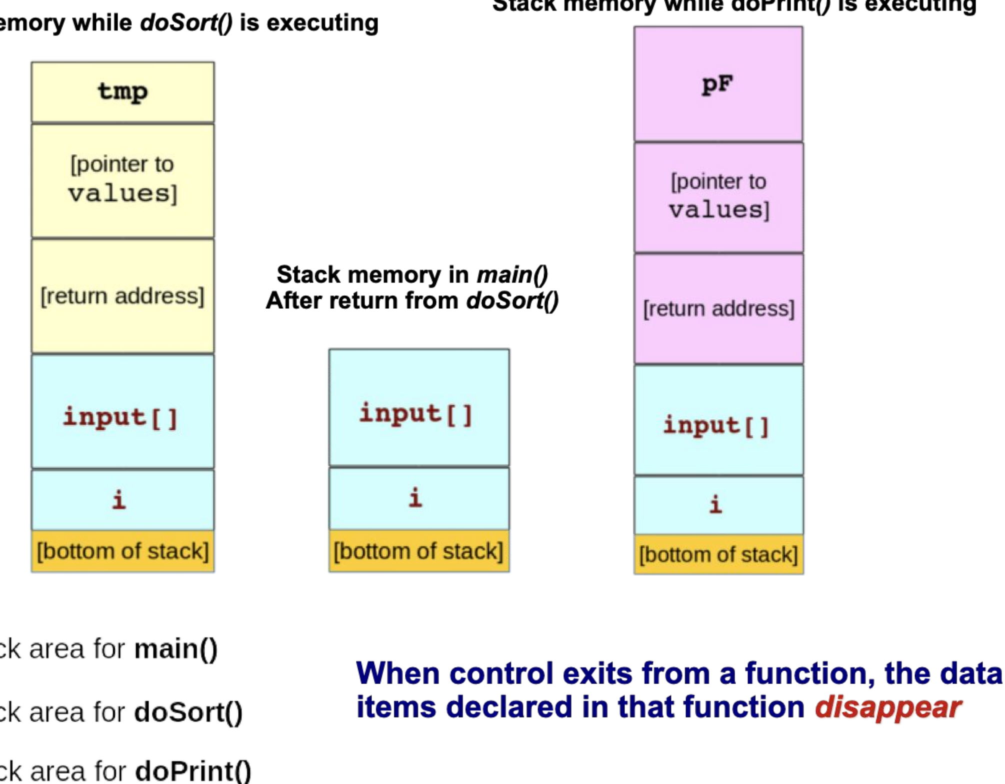
```
#define MAX_ELTS 20
int values[MAX_ELTS];

void doSort(int numbers[])
{
    int tmp=0;
    ...
}

void doPrint(int numbers[])
{
    FILE* pF = NULL;
    …
}

int main()
{
    int i;
    char input[64];
    for (i = 0; i < MAX_ELTS; i++)
        {
              fgets(input,sizeof(input),stdin);
              sscanf(input,"%d",&values[i]);
        }
    doSort(values);
    doPrint(values);
}
```

Static data items in RED

Stack data items in GREEN

This example does not use the heap

Stack memory during execution



When control exits from a function, the data items declared in that function *disappear*