October 24, 2009: 350.org @ Space Needle

David Notkin ● Autumn 2009 ● CSE303 Lecture 12

## Upcoming schedule

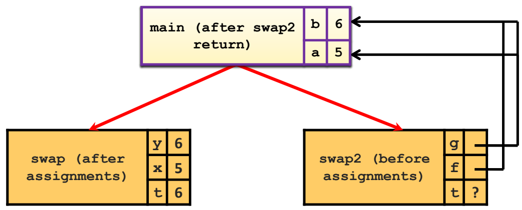
| Today 10/23 | *Monday 10/26* | Wednesday 10/28 | Friday 10/30 | Monday 11/2 |
|---|---|---|---|---|
| Finish-up Wednesday<br>Some specifics for HW3<br>Social implications Friday | Memory management | | Midterm review | Midterm |

- Swap
- Arrays as parameters and returns
  - Arrays vs. pointers
- The heap
  - Dynamic memory allocation (malloc, calloc, free)
  - Memory leaks and corruption



```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int a,b;
  scanf("%d %d",&a,&b); printf("Before swap: a=%d b=%d\n",a,b);
  swap(a,b);      printf("After return from swap: a=%d b=%d\n",a,b);
  swap2(&a,&b); printf("After return from swap2: a=%d b=%d\n",a,b);
}
swap(int x,int y) {
  int t;
  t = x; x = y; y = t;
  printf("Before return from swap: x=%d y=%d\n",x,y);
}
swap2(int *f,int *g) {
  int t;
  t = *f; *f = *g; *g = t;
  printf("Before return from swap2: f=%d g=%d\n",*f,*g);
}
```






## Arrays and pointers

- A pointer can point to an array element
- An array's name can be used as a pointer to its first element
- The `[]` notation treats a pointer like an array
  - `pointer[i]` is `i` elements' worth of bytes forward from pointer

```
int a[5] = {10, 20, 30, 40, 50};

int* p1 = &a[3];   // a's 4th element
int* p2 = &a[0];   // a's 1st element
int* p3 = a;       // a's 1st element

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;
```

**Final array contents:**
**{200, 400, 500, 100, 300}**

## "pointer[i] is i elements' worth of bytes" – what is an "elements' worth"?

```
int x;
int a[5];

printf("int=%d,double=%d\n",
        sizeof(int),
        sizeof(double));
printf("x uses %d bytes\n",
        sizeof(x));
printf("a uses %d bytes\n",
        sizeof(a));
printf("a[0] uses %d bytes\n",
        sizeof(a[0]));
```

int=4, double=8
x uses 4 bytes
a uses 20 bytes
a[0] uses 4 bytes

- `sizeof(type)` or `sizeof(variable)` returns memory size in bytes
- Arrays passed as parameters do not remember their size

```
int a[5];
printf("a uses %d bytes\n",
        sizeof(a));
f(a);

void f(int a[]) {
    printf("a uses %2d
       bytes in f\n",
       sizeof(a));
}
```

a uses 20 bytes
a uses 4 bytes in f

## Arrays as parameters

- Array parameters are passed as pointers to the first element; the `[]` syntax on parameters is only a convenience – the two programs below are equivalent

```
void f(int a[]);
int main(void) {
    int a[5];
    ...
    f(a);
    return 0;
}
void f(int a[]) {
    ...
}
```

```
void f(int* a);
int main(void) {
    int a[5];
    ...
    f(&a[0]);
    return 0;
}

void f(int* a) {
    ...
}
```

## Returning an array

- Stack-allocated variables disappear at the end of the function: this means an array cannot generally be safely returned from a method

```
int main(void) {
    int nums[4] = {7, 4, 3, 5};
    int nums2[4] = copy(nums, 4);   // no
    return 0;
}
int[] copy(int a[], int size) {
    int i;
    int a2[size];
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
    return a2;   // no
}
```

## Pointers (alone) don't help

- A *dangling pointer* points to an invalid memory location

```
int main(void) {
    int nums[4] = {7, 4, 3, 5};
    int* nums2 = copy(nums, 4);
    // nums2 dangling here
    ...
}
int* copy(int a[], int size) {
    int i;
    int a2[size];
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
    return a2;
}
```

## Our conundrum

- We'd like to have C programs with data that are
  - Dynamic (size of array changes based on user input, etc.)
  - Long-lived (doesn't disappear after the function is over)
  - Bigger (the stack can't hold all that much data)
- Currently, our solutions include:
  - Declaring variables in main and passing as "output parameters"
  - Declaring global variables (do not want)

## The heap

- The *heap* (or "free store") is a large pool of unused memory that you can use for dynamically allocating data
- It is allocated/deallocated explicitly, not (like the stack) on function calls/returns
- Many languages (e.g. Java) place all arrays/ objects on the heap

```
// Java
int[] a = new int[5];
Point p = new Point(8, 2);
```

## `malloc`: allocating heap memory

- `variable = (type*) malloc(size);`
- `malloc` function allocates a heap memory block of a given size
  - returns a pointer to the first byte of that memory
  - can/should cast the returned pointer to the appropriate type
  - initially the memory contains garbage data
  - often used with `sizeof` to allocate memory for a given data type

```
int* a = (int*) malloc(8 * sizeof(int));
a[0] = 10;
a[1] = 20;
...
```

## `calloc`: allocate and zero

- **`variable = (type*) calloc(count, size);`**
- **`calloc`** function is like **`malloc`**, but it zeros out the memory
  - also takes two parameters, number of elements and size of each
  - preferred over **`malloc`** for avoiding bugs (but slightly slower)

```
#include <stdlib.h>
// int a[8] = {0};    <-- stack equivalent
int* a = (int*) calloc(8, sizeof(int));
```

## Returning a heap array

- To return an array, **`malloc`** it and return a pointer
  - Array will live on after the function returns

```
int main(void) {
int nums[4] = {7, 4, 3, 5};
int* nums2 = copy(nums, 4); ...

int* copy(int a[], int size) {
    int i;
    int* a2 = malloc(size * sizeof(int));
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
    return a2;
}
```

## `NULL`: an invalid memory location

- In C, **`NULL`** is a global constant whose value is **`0`**
- If you **`malloc/calloc`** but have no memory free, it returns **`NULL`**
- You can initialize a pointer to **`NULL`** if it has no meaningful value
- Dereferencing a null pointer will crash your program

```
int* p = NULL;
*p = 42;            // segfault
```

- Exercise : Write a program that figures out how large the stack and heap are for a default C program.

## Deallocating memory

- Heap memory stays allocated until the end of your program
- A *garbage collector* is a process that automatically reclaims memory no longer in use
  - Keeps track of which variables point to which memory, etc.
  - Used in Java and many other modern languages; ***not in C***

```
// Java
public static int[] f() {
    int[] a = new int[1000];
    int[] a2 = new int[1000];
    return a2;
}   // no variables refer to a here; can be freed
```

## Memory leaks

- A *memory leak* is a failure to release memory when no longer needed.
  - easy to do in C
  - can be a problem if your program will run for a long time
  - when your program exits, all of its memory is returned to the OS

```
void f(void) {
    int* a = (int*) calloc(1000, sizeof(int));
    ...
}   // oops; the memory for a is now lost
```

## `free`: releases memory

- **`free(pointer);`**
- Releases the memory pointed to by the given pointer
  - precondition: pointer must refer to a heap-allocated memory block that has not already been freed
  - it is considered good practice to set a pointer to **`NULL`** after freeing

```
int* a = (int*) calloc(8, sizeof(int));
...
free(a);
a = NULL;
```

## Memory corruption

- If the pointer passed to free doesn't point to a heap-allocated block, or if that block has already been freed, bad things happen
  - you're lucky if it crashes, rather than silently corrupting something

```
int* a1 = (int*) calloc(1000, sizeof(int));
int a2[1000];
int* a3;
int* a4 = NULL;

free(a1);     // ok
free(a1);     // bad (already freed)
free(a2);     // bad (not heap allocated)
free(a3);     // bad (not heap allocated)
free(a4);     // bad (not heap allocated)
```

## Questions?