601.220 Intermediate Programming

Pointers

Outline

- Motivation
- Pointer basics
- Pointers as function parameters
- Arrays and functions (parameters and returning)

Writing a swap method in C

```
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

- Suppose the above function is called within main as `swap(a,b)`, where `a` and `b` are ints.
  - What's wrong here?

Writing a swap method in C

```
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

- Suppose the above function is called as `swap(a,b)`, where `a` and `b` are ints.
- Pass-by-value semantics mean that changes made to `x` and `y` within `swap` are never reflected in caller's argument values!
    - That is, `a` and `b` will remain unchanged in `main`

Local variables

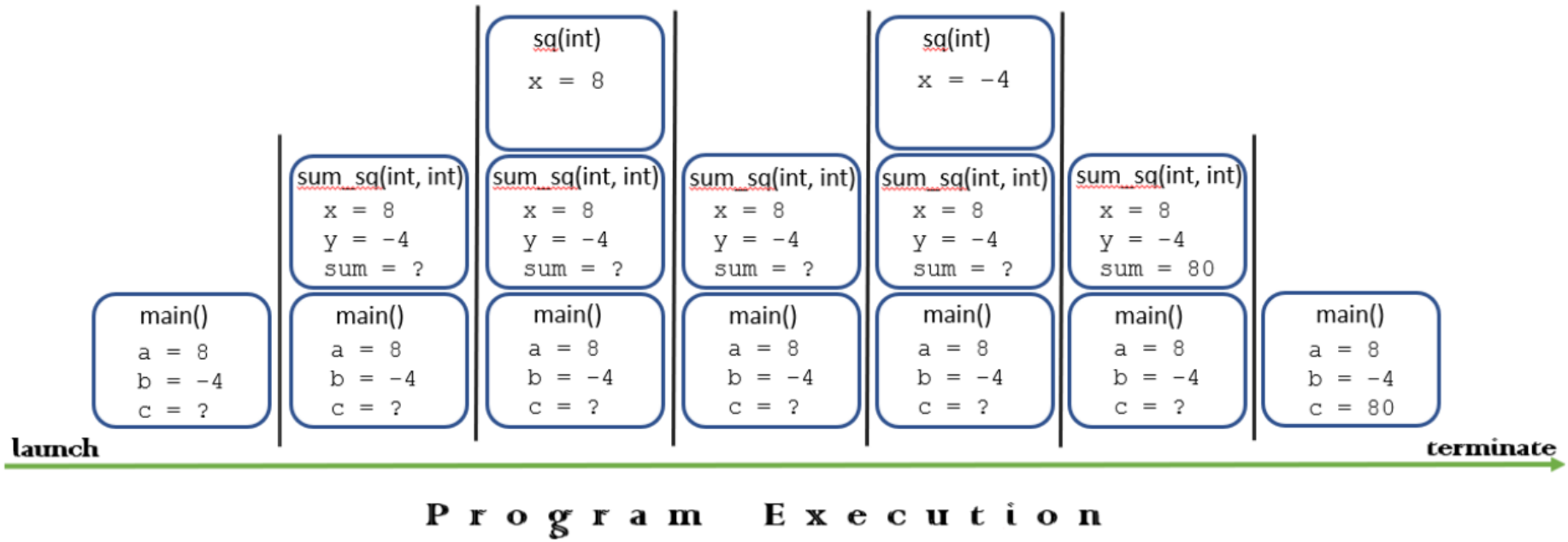
- Variables declared within a function get situated in memory within a structure called the program execution *stack* or *call stack*
  - Each function call gets a new *stack frame* (*activation record*) put on top of ("pushed onto") the stack in mememory; each frame has space for local variable values
  - Space is allocated within the stack frame for parameters as well, and values of arguments are copied into that frame
    - Consider how recursive functions keep track of the argument values for the current call...

Local variables

- Once the current function call returns, the stack frame is removed from ("popped off of") the stack, and execution returns to the calling function (whose frame happens to be the topmost remaining one on the stack)
  - This means that a function's local variables don't live beyond that specific function's call

Program execution stack - example

```
int sq(int x) { return x * x; }
int sum_sq(int x, int y) { int sum = sq(x) + sq(y); return sum; }
int main() {
    int a = 8, b = -4;
    int c = sum_sq(a, b);
    return 0;
}
```



Program Execution

Writing a swap function in C

- In the swap function, adjusting values in swap function stack frame didn't make any impact in calling function's stack frame
- What we really needed was access to caller's stack frame instead. . . this is a job for *pointers*!

Pointers

- A *pointer* is a variable that contains the address of a variable.
  - Every pointer points to a specific data type (except a *pointer to void*, but more about that later)
  - Declare a pointer using type of variable it will point to, and a `*`
  - `int *p;` says `p` is the name of a variable that holds the address of an `int`
- Operations related to pointers
  - address-of operator `&`: returns address of whatever follows it
  - dereferencing operator `*`: returns value being pointed to

Examples using & and *

```
int x = 1;
int y = 2;

int *ip;          /* declare ip, a pointer to int */

ip = &x;          /* ip now "points to" x */

y = *ip;          /* y is now 1 */

*ip = 0;          /* x is now 0 */
```

Now, let's return to writing a swap function

```
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

- Suppose the above function is called as `swap(a,b)`, where `a` and `b` are ints.
- Pass-by-value semantics mean that changes made to `x` and `y` within `swap` are never reflected in caller's argument values!
  - That is, `a` and `b` will remain unchanged in `main`

An improved swap function

```
void swap(int *px, int *py) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

- The call in main will now be `swap(&a, &b)`, since we need to send in the addresses of `a` and `b`
- Pointer arguments enable a function to access and modify values in the calling function

More examples using & and *

- Suppose `ip` points to integer `x`. Then `*ip` can be used anywhere that `x` makes sense:
  - `*ip = *ip + 10`
- Unary operators & and * bind more tightly than binary arithmetic ops
  - So, `y = *ip + 1` means take whatever `ip` points to, add one to it, then assign it to `y`
  - And we can write: `*ip += 1` or `++*ip` to increment what `ip` points to.
  - But note that parentheses are needed for `(*ip)++` since unary operators associate from right to left.

Arrays within a stack frame

- Recall that arrays, when passed to a function, aren't copied over as a single int variable would be
  - Instead, the *address* of the array is copied over, as this is more efficient
  - But since calling function remains on stack while called function is executing, the address is still valid, so called function can access it

Arrays as function arguments

```c
// passArray.c:
#include <stdio.h>
void changeFirst(int a[]) {
     a[0] = 99;   //change first item in array
}
int main() {
    int list[300];
    for (int i = 0; i < 300; i++) {
        list[i] = i;
    }
    changeFirst(list);
    for (int i = 0; i < 3; i++) {
        printf("%d ", list[i]);  //output first 3
    }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic passArray.c
$ ./a.out
99 1 2
$
```

Returning an array from a function

```c
// returnArray.c:
#include <stdio.h>
int * createArray(int size) {
    int a[size];  //declare an int array of specified size
    //...some initialization could happen here...
    return a;     //return the locally-allocated array
}
int main() {
    int *list = createArray(10);
    for (int i = 0; i < 10; i++) {
        printf("%d ", list[i]);
    }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic returnArray.c
returnArray.c: In function ‘createArray’:
returnArray.c:5:12: warning: function returns address of local variable
    5 |     return a;     //return the locally-allocated array
      |            ^
```

Arrays and functions

- Arrays returned from functions are passed by address also; only a copy of the address is sent back to caller
  - But if the address is of an array that lives in the function's stack frame, the array won't survive after the function returns (the frame will be popped!)
- As a result, we can't expect to create an array that lives in a function's stack frame and then return it to the calling function
  - But we'll soon see a way to send back a new array from a function...

Allocating a very large array

```c
// largeArray.c:
#include <stdio.h>

int main() {
    int list[10000000];
    for (long i = 0; i < 10000000; i++) {
        printf("%d ", list[i]);
    }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic largeArray.c
$ ./a.out
Segmentation fault (core dumped)
```

Allocating a very large array

- Stack frames have a limited size
- On the last slide, we attempted to allocate an array within a function's stack frame, but the array was too large for the frame
    - A segmentation fault resulted

Limitations of arrays allocated within a stack frame

- We've just seen that arrays allocated within a stack frame ("static allocation") have several limitations
  - Size of array is limited by size of stack frame
  - Arrays created within a called functions stack frame can't be accessed by calling function (since lifetime of array ends when called function returns)
  - Prior to C99, another limitation existed:
    - Needed to know size of array prior to run-time - couldn't ask for array of size `n` when `n` was a value input by user!
- To get around these limitations, we'll soon learn about *dynamic allocation*