Midterm 1 Announcement

- Sixth week: Feb. 14 (Thursday) during the class.
  - Close book, close notes, no electronic devices
  - 50 minutes sharp
  - Multiple choices
- Bring #2 pencils for blackening the bubbles for answers
- Bring student IDs in case TAs need to check
- No makeup exam
  - (as specified in policy) unless doctor’s note says you are sick

We have seen there are many ways to address array elements

- For a char array buf[], we cannot write buf = ‘a’; (let’s try to compile it)
- But we can write *buf = ‘a’; (This is same as buf[0] = ‘a’;)
- And write *(buf + 1) = ‘b’; (Same as buf[1] = ‘b’;)
- For array x[][], we can write *(x[0] + 1) = ‘c’;
- For pointer p, we can write p++[1] = ‘x’;
  - This offers the flexibility to access a section of the array starting from the middle of the array
- Since these involve other operations such as ++ and multiple dimensions, we will come back to analyze this example later.

Function calls: another postfix expression

- Fixed number of arguments: f(arg1, arg2, arg3)
- Variable number of arguments: f(arg1, arg2, arg3, …)
- The arguments and the function designator are completely evaluated, including all side effects, before the function is entered.

- *However, the order of evaluation of arguments is unspecified;*
- *!!! Not necessarily from left to right !!!*
  - Therefore, it is not a good idea to let an argument modify a variable
    - E.g. f(i++, i--);
  - It is also not a good idea to let more than one argument contain functions calls that have side effects, e.g. having printf() inside
    - E.g. f(g(), h()+1); /* in case g() and h() have side effects */

An Example (*argumentOrder.c*) Look at the order of argument evluations

```
#include <stdio.h>
int f(int, int);
int g();
int h();

main() {
	int i=0;

	f(i,i--);
	f(g(), h());
}

int f(int a, int b){
	printf("a = \t%d\n", a);
	printf("b = \t%d\n", b);
	return 0;
}
```

```
int g( ) {
	printf("g() is called \n");
	return 0;
}

int h( ) {
	printf("h() is called \n");
	return 0;
}
```

Argument/Parameter passing in function call

- Here we take the opportunity to explain how, at run time, parameters are passed during a function call in a C program
  - In function call *f(arg1, arg2, …);* arg1 and arg2 etc are called *(actual) arguments*
  - In function declaration or definition *int f(p1, p2, …) {}* *p1* and *p2* etc are called *(formal) parameters*.
- The hardware/OS vendor specifies an *application binary interface (ABI)* for software writers to follow
  - Compiler writers follow it to generate machine code
  - Third-party library writers follow it to allow other programmers to invoke the library routines correctly

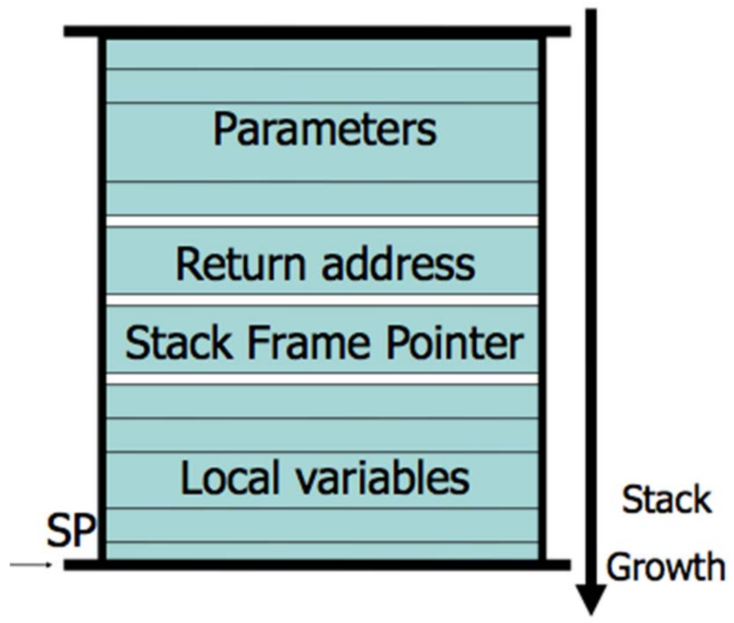
Calling Convention

- An important part of the ABI is the *calling convention*, which
  - *controls how functions' arguments are passed and return values retrieved;*
    - *for example, whether all parameters are passed on the stack or some are passed in registers,*
    - *which registers are used for which function parameters, and*
    - *whether the first function parameter passed on the stack is pushed first or last onto the stack*
      - *(quote from wikipaedia)*
    - *Note that the order of argument/parameter push/pop and the order of argument evaluation are two separate issues*

“Call by value” in C

- The most important thing to understand about parameter passing in C is the “call by value” concept
- Let us look at what happens at the time of a function call

Illustration of a stack frame



A possible calling sequence

1. The caller evaluates the argument (in some order)
   - This may cause another function to be called (if an argument contains function calls
2. After all arguments are evaluated (and saved to compiler-generated temporary variables), they are ready to be “pushed” to the stack (for the callee to access)
3. Every function reserves a “*frame*” on the program stack
4. When calling a function, a new frame is pushed to the stack
   - Either by the instructions in the caller or by those in the callee (depending on the calling convention)
   - Pushing is done by making the new frame pointer pointing to the old stack pointer (i.e. the old top of stack)

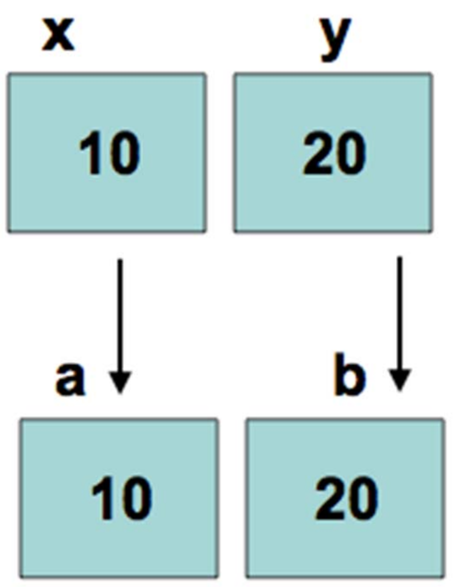
- Normally the hardware has only one set of registers used by all functions,
  - Therefore, registers must be saved before a new function is called
  - They are saved to either the caller's frame or callee's frame or split between them
  - It may be caller's job or callee's job to save, or they may split the saving operations (all depending on the ABI)
- Registers that must be saved
    - Program counter (PC) of the instruction to be executed when call returns to the caller (some hardware allows PC to be saved to a register, but eventually that register may need to be saved)
    - The stack pointer (top of the stack frames) right before the new function call
    - Other registers whose values are still needed by the caller (its compiler's job to find out which ones)
  - The "frame pointer" (i.e. the base address of the stack frame of the caller in the stack) may also be saved to the stack

- The *actual arguments* are now pushed to the argument registers or stack locations known to the callee (depending on the ABI)
- In C, one does not pass the entire array or structure through the stack
  - Recall in Java, caller just passes the *reference* to an object (i.e. the base address) to a called method
  - In C, call passes the pointer to an array, a function, or structure to the called function
  - In C, caller may also passes arithmetic values to the called function, as in Java
  - The callee accesses the passed pointers and arithmetic values by the name of the *formal parameters*
  - Upon return, those passed values and pointers are "gone" from the stack, not to be accessed by the caller

- Therefore, when caller makes a variable, x, one of the actual arguments, the variable’s value will not be altered by the callee even if the corresponding formal parameter is altered by callee
- This is because x’s original location is not in the new stack frame, *the new stack frame has only a copy*
- If the caller wants the callee to modify its own variables, then the caller must *pass the addresses* of the variables to the callee

Example to see effect of call by value

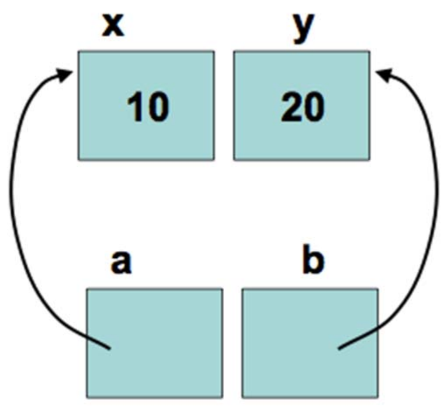
```
• int y = 20, x = 10;
• swap(x,y);

• void swap(int a, int b) {
•     int t = a;
•     a = b;
•     b = t;
• }
```



To swap!

- `int y = 20, x = 10;`
- `swap(&x,&y);`

```
• void swap(int* a, int* b) {
•     int t = *a;
•     *a = *b;
•     *b = t;
• }
```



Return statement

Terminates the execution of a function and returns control to caller

- The return value is first converted to declared return type
- The return value is placed in the "return-value" register or some stack location known to the caller
- The rest of the *return sequence* reverses the calling sequence

Unix Command Line

- On Unix, we use the main function’s array of pointers arguments to capture the command line arguments
  - The Unix shell interpreter extracts the command line arguments, which are (eventually) passed to main()

```
#include <stdio.h>
int main(int argc, const char* argv[])  {
        int i;
        printf("Number of arguments \t %d\n", argc);

          for (i = 0; i < argc; ++i)
          printf("next argument \t %s\n", argv[i]);
return 0;
}
```

Variable length of parameter list

- So far we have seen two examples of parameter lists that have variable lengths
  - printf
  - main
- For general cases of variable-length argument list, both its use in a program and the implementation by the compiler are quite complex
  - We study this much later in the semester

Quiz 2.1

- The value of ‘b’ – ‘a’ is
- (a) 0
- (b) 1
- (c) neither

- Answer is (b)
  - The integer code of ‘a’ to ‘z’ is consecutively incremented in the ASCII table

Quiz #2 - 2

- If both file1.c and file2.c contains the definition

```
static int f() { }
```

Then the command `gcc  file1.c file2.c` will cause the compiler to warn that function f() has multiple definitions

(a) True

(b) False

- The answer is (b)
  - The “static” qualifier limits the scope of int f() definition to the file only. The static definition in one file does not conflict with any definition of the same identifier in other files.

Quiz #2 - 3

- When the following program runs, “x = 0” will be printed

```
#include <stdio.h>
void f();

main() {
    int i;
    for (i=0; i<10; i++)
        f();
}

void f() {
    static int x = 0;
    printf(“x = %d\n", x);
    x++;
    return;
}
```

(a) Once
(b) Twice
(c) 10 times
(d) None of the above

- The answer is (a)
  - The “static” qualifier applied to the local variable “i” makes it carries the value updated in one call to f to the next call. The initial value 0 is assigned to “i” only at the first time f is called. This is actually the only way to make such static local variable useful. Otherwise there is no way to initialize the value of “i” while allowing its updated value to be carried from call to call.