Software Systems

Day 3 – Stack/State Diagrams, Compilation

Agenda

- Announcements
- Switch Exercise
- Stack and State Diagrams
- Compilation

Announcements

- Assignment 0 released - see Discord announcement or GitHub.
  - If you’re stuck, ask on Discord or at CA hours.
- In general, assignment deadlines will be at the same time each week.

Exercise: Switch

- Use scanf to read in a string of the form "x + y" or "x * y", where x and y are integers.
  - So read in something like 6 * 7.
- Then use a switch statement to evaluate the result.
  - If the operator is not + or *, then return –1.
- Finally, print the result using printf.

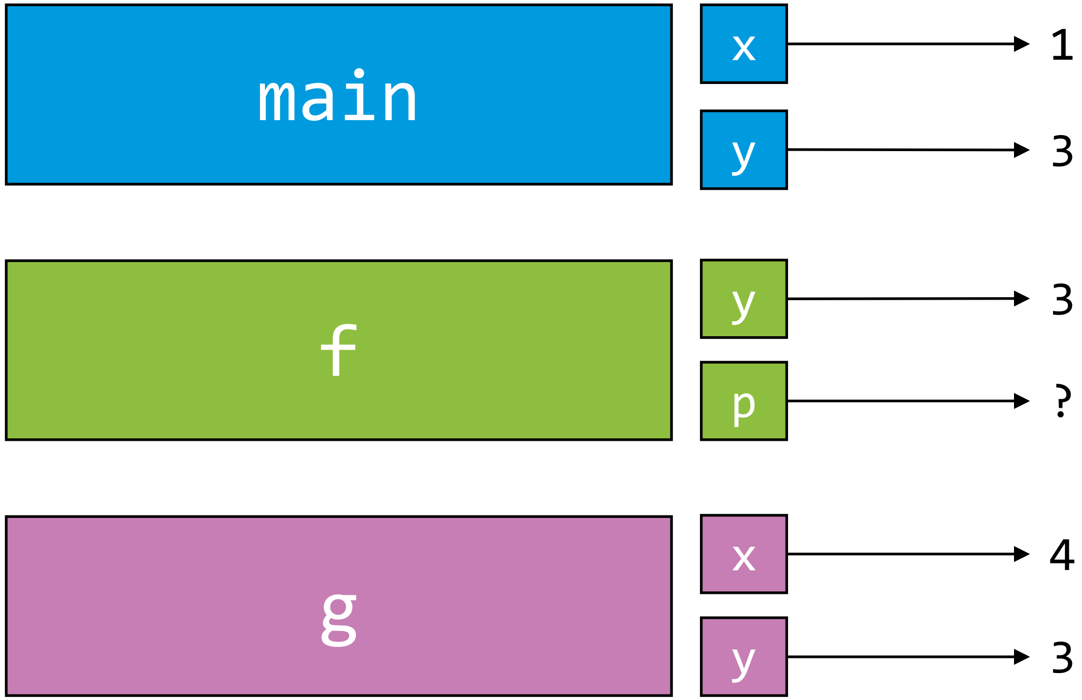
Exercise: Switch

```
int main(void) {
  int x, y, z;
  char op;
  scanf("%d %c %d", &x, &op, &y);
  switch(op) {
    case '+':
      z = x + y;
      break;
    case '*':
      z = x * y;
      break;
    default:
      z = -1;
  }
  printf("%d %c %d is %d\n", x, op, y, z);
  return 0;
}
```

Stack Diagrams

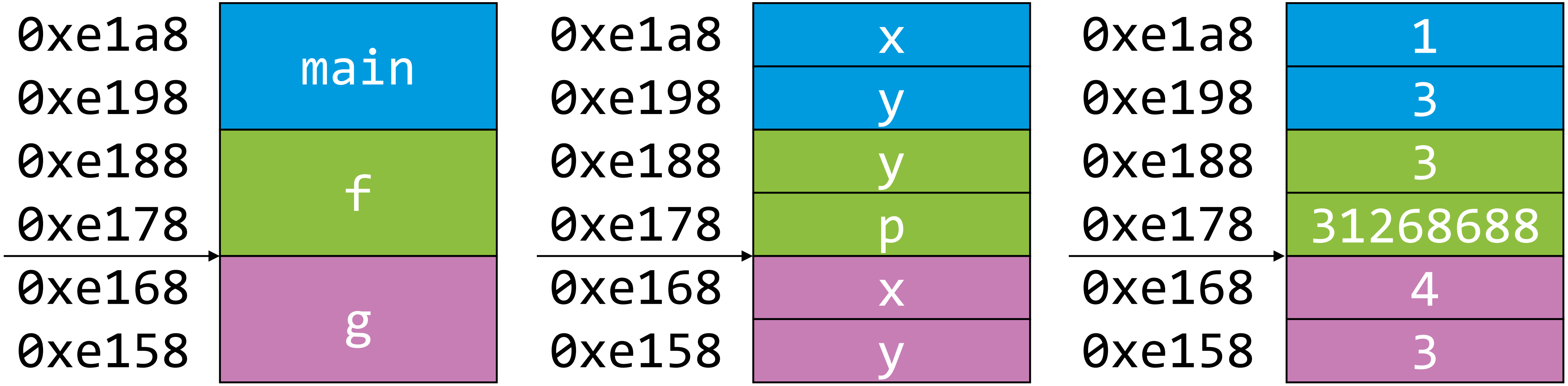
- You may remember stack diagrams in Python.
- They show you what functions are running and what each variable is.

```
def f(y):
    p = g(y, y)
    return p

def g(x, y):
    x += 1
    return x * y

def main():
    x = 1
    y = x + 2
    f(y)
```



Stack Diagrams

- In C, things look similar, but differ from Python in a few ways.



Stack Diagrams: Exercise

- Draw a stack diagram for the following code at the point indicated.

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

void test_add() {
  int sum = add(3, 4);
  printf("%d\n", sum);
}
```

```
char *check_parity(int n) {
  switch (n % 2) {
    case 0:
      return "n is even";
    case 1:
      return "n is odd";
  }
}

int main() {
  test_add();
  char *s = check_parity(3);
  printf("%s\n", s);
}
```

Stack Diagrams: Exercise

- Draw a stack diagram for the following code at the point indicated.

```
int add(int x, int y) {
  int z = x + y;
  return z;
}
```

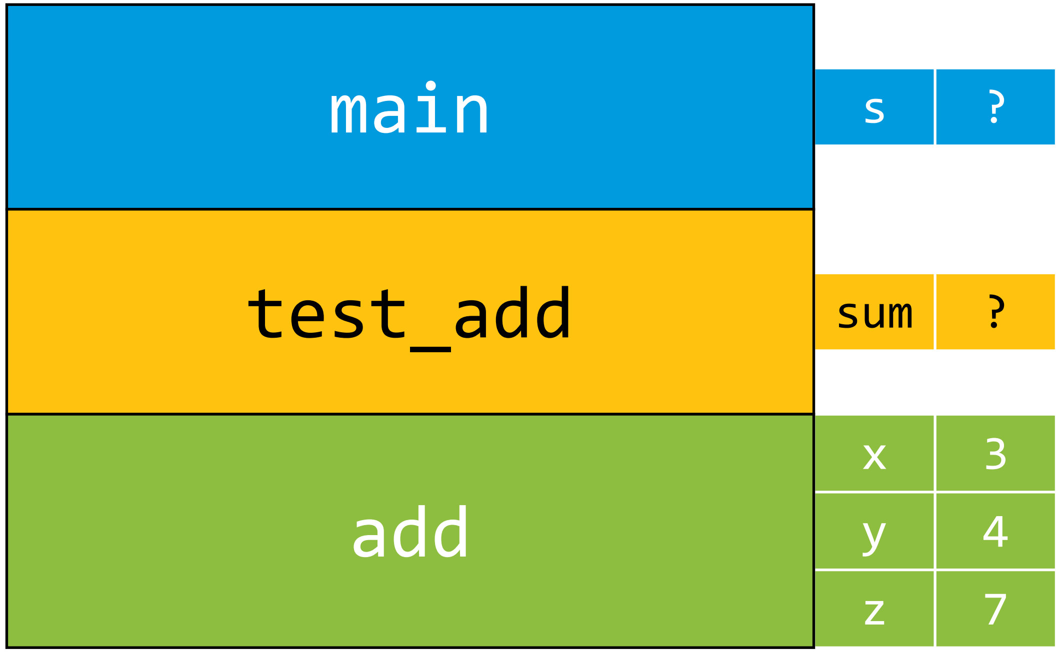
```
void test_add() {
  int sum = add(3, 4);
  printf("%d\n", sum);
}
```

```
char *check_parity(int n) {
  switch (n % 2) {
    case 0:
      return "n is even";
    case 1:
      return "n is odd";
  }
}
```

```
int main() {
  test_add();
  char *s = check_parity(3);
  printf("%s\n", s);
}
```

Stack Diagrams: Exercise

- Draw a stack diagram for the following code at the point indicated.



Think OS Chapter 1

- Stages of compilation
  - Preprocessing: use macros like #include to rewrite the source code.
  - Parsing: turning the source code into a "sentence diagram" of sorts.
  - Static checking: making sure that types, etc. line up throughout the code.
  - Code generation: turning the source code into machine language.
  - Linking: putting compiled code and libraries together into an executable.
  - Optimization: making it faster.
- We'll explore this more today through a multi-part exercise.

Exercise: Compilation

- Step 0: Acquire and Check Files
  - Run git pull upstream main to get the source file for today.
  - cd into that directory and make sure you can compile and run it.
  - gcc hello.c, then run a.out.
  - Does everything work?

Exercise: Compilation

- Step 1: Executable Control
  - Now compile it so that the executable is called hello instead of a.out.
  - What command do you use?
  - Does the order of command-line arguments matter?

Exercise: Compilation

- Step 2: Explore the Object File
  - Compile hello.c with the -c flag. What does this do?
  - Use the nm command to see what functions are defined in hello.c, and which ones are used but not defined.
  - Define a new function in hello.c, and add a reference to another function.
  - Run gcc -c and nm again. What effect did the previous step have on the output?

Exercise: Compilation

- Step 3: Explore a library file
  - Use gcc --print-file-name=libc.a to find out where libc.a lives.
  - Then use nm on that path. What is in libc.a?

Exercise: Compilation

- Step 4: Explore assembly
  - Compile hello.c using the -S flag, which generates assembly.
  - What happens if you then try to compile hello.s with gcc?
  - Open hello.s in a text editor and try to figure out what the various sections represent. Particularly, how do you think the main function works?

Exercise: Compilation

- Step 5: Add some assembly
  - Add the following lines to hello.c after the puts call:
    ```
    int a = 3;
    int b = 4;
    int c = a + b;
    printf("c is %d\n", c);
    ```
  - Compile hello.c into assembly again. How does this generated code (hello.s) compare to what you saw previously?

Exercise: Compilation

- Step 6: Play with assembly
  - Open hello.s from the previous step.
  - You should see an addl instruction. Replace it with subl.
  - If you compile this program and run it, what do you expect to get? Why?
  - Compile hello.s and run it. Does the output match what you expect?

Exercise: Compilation

- Step 7: Optimize assembly
  - Compile hello.c with the -O2 and -S flags.
  - Open hello.s. How is it different from previous versions?

Exercise: Compilation

- Step 8: Explore compilation
  - Add the following lines to hello.c, after where you added the last set of lines:
    ```
    if (c % 2 == 0) {
        printf("c is even\n");
    } else {
        printf("c is odd\n");
    }
    ```
  - Compile the program to assembly again. How is the modulus operator (%) compiled?
  - What happens if you instead write c % 3?

Exercise: Compilation

- Step 9: Switch it up
  - Rewrite the if statement from the previous step using a switch statement.
  - Compile the program to assembly again. How does it compare to the previous results?
  - What happens if you leave off the break statements?
  - Why might this tell you why the switch statement "falls through" each case?

Exercise: Compilation

- Step 10: Explore the preprocessor
  - Run gcc with the -E flag to only run the preprocessor.
  - Use the wc command to count the lines in the resulting file. How does it compare to the length of the original file?
  - Now add #include <stdlib.h> to hello.c, and run the above steps again. How did that change the length of the preprocessed file?