Functions

CS449 Fall 2017
Functions in C

• What makes it a procedural language
• A function is a name for a self-contained group of statements that performs a task.
• The statements inside a function can be executed by calling it.
• What are functions good for?
  – Code modularization (better readability)
  – Reusability (e.g. the C Standard Library)
  – Implementing recursive algorithms
```c
#include <stdio.h>

int add(int a, int b);

int main()
{
    int x = 3, y = 4, sum = 0;
    sum = add(x, y);
    printf("Sum: %d\n", sum);
    return 0;
}

int add(int a, int b)
{
    return a+b;
}
```

>> ./a.out
Sum: 7


#include <stdio.h>

int add(int a, int b);

int main()
{
    int x = 3, y = 4, sum = 0;
    sum = add(x, y);
    printf("Sum: %d\n", sum);
    return 0;
}

int add(int a, int b)
{
    return a+b;
}

• Syntax: <return type> <name> ( <parameter list> );
  – E.g. “int add(int a, int b);”
  – E.g. “int printf(const char* format, ...);”
• Declares the function prototype
• Function prototype
  – Type of the function
  – Consists of function name + return type + parameter types
  – Crucial for type checking and generating correct type conversions during function call
• Must come before call (if function definition doesn’t)
• Can be outside functions (global scope of entire file) or inside another function (local scope of function)
• Header file (e.g. stdio.h) contains function declarations
  – #include copies and pastes contents of header file
#include <stdio.h>

```c
int add(int a, int b);

int main()
{
    int x = 3, y = 4, sum = 0;
    sum = add(x, y);
    printf("Sum: %d\n", sum);
    return 0;
}

int add(int a, int b)
{
    return a+b;
}
```

• Syntax: `<return type> <name> ( <parameter list> )
  { declarations and statements }
  – E.g. “int add(int a, int b) { return a+b; }”

• Consists of:
  – Function prototype
  – Local variable declarations
  – Statements

• main() is also a function, one that is called at the beginning of the program

• Must match exactly function prototype in declaration
  – Must return a value of the return type
  – “void” return type requires no return value (just do “return;” to exit function or nothing at the end)

• A function cannot be defined inside another function
Function Call

```c
#include <stdio.h>
int add(int a, int b);

int main()
{
    int x = 3, y = 4, sum = 0;
    sum = add(x, y);
    printf("Sum: %d\n", sum);
    return 0;
}

int add(int a, int b)
{
    return a+b;
}
```

- Syntax: `<name> ( <argument list> );`
  - E.g. “add(x, y);”
- Consists of:
  - Function name
  - Arguments (expressions that evaluate to each respective type in parameter list)
- If number of arguments differ from number of parameters, it results in a compile error
- If argument types differ from parameters, arguments are coerced into parameter types when possible
- All arguments are passed by value
Passing Arguments by Value

• All arguments are **passed by value** in C
• Meaning: arguments are **copied** to parameters
  – Arguments and parameters are names for different storage locations
• Compare: Java
  – The same: all arguments passed by value in Java
  – Slight difference in what those “values” are
    • Values in Java: primitive values, object references
    • Values in C: primitive values, pointers, structs
Goals of Argument Passing in C

• Provide input values to a function
  – E.g. “int add(int x, int y)”
  – No need to belabor; did this in Java all the time

• Modify input values to a function
  – E.g. “swap” function swaps the values of two variables
  – How can this be done when arguments are copied?

• “Return multiple values” from a function
  – E.g. “divide” function needs to return a quotient and a remainder
  – But allowed to return only one value from function?
Goals of Argument Passing in C

• Provide input values to a function
  – E.g. “int add(int x, int y)”
  – No need to belabor; did this in Java all the time

• Modify input values to a function
  – E.g. “swap” function swaps the values of two variables
  – How can this be done when arguments are copied?

• Return multiple values* from a function
  – E.g. “divide” function needs to return a quotient and a remainder
  – But allowed to return only one value from function?

*Use Pointers!
(Wrong) Example of Swap Function

```c
#include <stdio.h>
void swap(int a, int b);
int main()
{
    int x = 3, y = 4;
    printf("x: %d, y: %d\n", x, y);
    swap(x, y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```bash
>> ./a.out
x: 3, y: 4
x: 3, y: 4
```
(Wrong) Example of Swap Function

#include <stdio.h>
void swap(int a, int b);
int main()
{
    int x = 3, y = 4;
    printf("x: %d, y: %d\n", x, y);
    swap(x, y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

• Problem:
  – Parameters “a” and “b” refer to storage locations that are different from “x” and “y”
• What is the solution?
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
    int x = 3, y = 4;
    printf("x: %d, y: %d\n", x, y);
    swap(&x, &y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
# Example of Division Function

```c
#include <stdio.h>

int divide(int a, int b, int *rem);

int main()
{
    int x = 7, y = 3, quotient, remainder;
    quotient = divide(x, y, &remainder);
    printf("quotient: %d, remainder: %d\n", quotient, remainder);
    return 0;
}

int divide(int a, int b, int *rem)
{
    *rem = a % b;
    return a / b;
}

>> .a.out
quotient: 2, remainder: 1
```
Recursion

• A function calling itself, or a group of functions calling each other in a cyclic pattern

• Useful in expressing many algorithms. E.g.:
  – Fibonacci series: $F(n) = F(n-1) + F(n-2)$
  – Tree traversal: $\text{Traverse}(\text{node}) = \text{Traverse}(\text{left node}) + \text{Traverse}(\text{right node})$
  – Binary Search: $\text{Search}(\text{sorted array}) = \text{Search}(\text{left half}) + \text{Search}(\text{right half})$
Example of Fibonacci Numbers

```c
#include <stdio.h>
int fibonacci(int);  
int main()
{
    int i;
    for(i = 0; i < 10; ++i) {
        printf("%d 
", fibonacci(i));
    }
    return 0;
}

int fibonacci(int n)
{
    if(n == 0 || n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

>> ./a.out
Num: 1 1 2 3 5 8 13 21 34 55
Function Pointers

• Pointers can even point to functions (not only data)
• Useful when you want a function call to perform a different task (i.e. call a different function) in different situations.
  – E.g. When your 7:00 AM alarm rings, you might either go jogging, make breakfast, or just go back to sleep, depending on day of week
• Value of function name is the address of the function or the function pointer (just like an array name)
• Function name is a constant (cannot be assigned to, just like an array name)
#include <stdio.h>
void jog() { printf("Jog\n"); }
void sleep() { printf("Back to sleep\n"); }
void breakfast() { printf("Breakfast\n"); }
void (*f[7])(()) = {jog, jog, jog, jog, jog, sleep, breakfast};

int main()
{
    int i;
    void (*todo)();
    for(i = 0; i < 7; ++i) {
        todo = f[i];
        (*todo)();
    }
}

> ./a.out
Jog
Jog
Jog
Jog
Jog
Back to sleep
Breakfast
#include <stdio.h>
void jog() { printf("Jog\n"); }
void breakfast() { printf("Breakfast\n"); }
void sleep() { printf("Back to sleep\n"); }
void (*f[7])() = {jog, jog, jog, jog, jog, sleep, breakfast};

int main()
{
    int i;
    void (*todo)() = NULL;
    for(i = 0; i < 7; ++i) {
        todo = f[i];
        (*todo)();
    }
}
#include <stdio.h>

void jog() { printf("Jog\n"); }  
void breakfast() { printf("Breakfast\n"); }  
void sleep() { printf("Back to sleep\n"); }  
void (*f[7])() = {jog, jog, jog, jog, jog, sleep, breakfast};

int main()
{
    int i;
    void (*todo)() = NULL;
    for(i = 0; i < 7; ++i) {
        todo = f[i];
        (*todo)();
    }
}

• Syntax: (*<name>)(argument list)
• e.g. "(*todo)()"
  – Meaning: call function pointed to by "todo" with argument list ()
Why Function Pointers?

#include <stdio.h>

void jog() { printf("Jog\n"); }
void breakfast() { printf("Breakfast\n"); }
void sleep() { printf("Back to sleep\n"); }

int main()
{
    int i;
    for(i = 0; i < 7; ++i) {
        if(i < 5) {
            jog();
        } else if(i == 5) {
            breakfast();
        } else {
            sleep();
        }
    }
}

• See alternative implementation without function pointers on left
• Without function pointers code is...
  – Harder to read / messier
  – Less efficient
    (Potentially must evaluate multiple if conditions to get to correct call)
  – No room for flexibility
    (With function pointers, you could change behavior for each day by simply updating the pointer array)
Pitfall 1: Pass by value

- What do you think the following will print?

```c
void foo(char *s) { s = "World"; }
int main()
{
    char *str = "Hello";
    foo(str);
    printf("%s\n", str);
    return 0;
}
```

- Problem: “str” and “s” refer to different locations
Pitfall 1: Pass by value

• Solution:

```c
void foo(char **s) { *s = "World"; }
int main()
{
    char *str = "Hello";
    foo(&str);
    printf("%s\n", str);
    return 0;
}
```