Functions in C

• A group of statements performing a task
  – Statements inside function is executed by calling it using the () operator

• Uses:
  – Readability (through code modularization)
  – Reusability (e.g. the C Standard Library)
  – Implementing recursive algorithms through recursion (we will learn what this means soon)

• What makes C a procedural language
  – Cf. Java is also a procedural language, and it is also an object-oriented language (something C is not)
#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;
}
#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;
}
```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;
}
```

**Function Definition**

- **Syntax:** `<return type> <name> ( <parameter list> )
{ declarations and statements }
– E.g. “int add(int a, int b) { return a+b; }”
- **Consists of:**
  - Function prototype (must match declaration)
  - Local variable declarations
  - Statements
- **main()** is a special function that is called at the start of the program
- **“return”** statement exits from function
  - Must return a value of the return type
  - If return type is “void”, no return value needed
    - Just do “return;” when exiting
    - “return;” at the end can be omitted
# 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 converted into parameter types when possible

- All arguments are passed by value
Arguments are Passed by Value

• All arguments are **passed by value** in C
• Meaning: arguments are **copied** to parameters
  – Argument variables and parameter variables reside in different memory locations
  – Modifying parameters will not modify arguments
• Cf. Java
  – All arguments are also 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
When Passing by Value is Insufficient

• Works just fine for providing input values to a function
  – E.g. “int add(int x, int y)”
  – add() does not need (nor want) to change x or y

• But what if you want to modify the values of the input variables to a function?
  – E.g. “void swap(int x, int y)”
  – swap() needs to swap the values of x and y
  – How can this be done when just the values are passed? (Any modifications inside swap will not affect x and y)

• Answer: pass **pointers** to those variables
  – Can’t modify pointer but can modify what it points to
(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;
}
```

>> ./a.out
x: 3, y: 4
x: 3, y: 4
(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;
}
```

- **Problem:**
  - Parameters “a” and “b” occupy memory 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

```
#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
#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;
}

• “remainder” modified through pointer
• Note:
  – Value in “remainder” variable is not even used inside “divide()”.
  – “remainder” variable is a receptacle to store a “return value” of the function
• Function can “return” multiple values by...
  – Returning one value normally
  – Returning the rest by passing pointers to variables to receive those values
• Exactly the reason for & in scanf("%d", &x)
  – Scanf is not interested in value of x
  – x is a receptacle to store the scanned value
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
    • $\text{Visit(node)} = \text{Visit(left node)} + \text{Visit(right node)}$
  – Binary Search
    • $\text{Search(array)} = \text{Search(left half)} + \text{Search(right half)}$
Example of Fibonacci Numbers

```c
#include <stdio.h>
int fibonacci(int); 
int main()
{
    int i;
    for(i = 0; i < 10; ++i) {
        printf("%d\n", 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 point to code (not only data)
  – Code is also stored in memory and have addresses
• Value of a function is a pointer to its code (similar to arrays)
  – () operator takes function as operand and jumps to the code pointed to by that function
  – Value of function cannot be overwritten
  – But value can be assigned to function pointers
  – Can use function pointers to call functions
• Useful when you want a single function call to jump to different functions in different situations
  – E.g. Depending on day of week, you might either go jogging, sleep in, or have brunch
  – One call to do the appropriate thing at the appropriate time?
Example of Function Pointers

```c
#include <stdio.h>
void jog() { printf("Jog\n"); }
void sleep() { printf("Back to sleep\n"); }
void brunch() { printf("Brunch\n"); }
void (*f[7])() = {jog, jog, jog, jog, jog, sleep, brunch};

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

>> ./a.out
Jog
Jog
Jog
Jog
Back to sleep
Brunch
Function Pointer Declaration

```c
#include <stdio.h>

void jog() { printf("Jog\n"); }
void sleep() { printf("Back to sleep\n"); }
void brunch() { printf("Brunch\n"); }

void (*f[7])() = {jog, jog, jog, jog, jog, sleep, brunch};

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

- **Syntax:** `<return type> (*<name>)
  (<parameter list>)`
- e.g. “void (*todo)()“
  - Meaning: “todo” is a pointer to a function with a return type void and a parameter list of ()
- e.g. “void (*f[7])()“
  - Meaning: “f” is an array of 7 pointers to functions with a return type void and a parameter list of ()
- Any function assigned to the function pointer should match its type
#include <stdio.h>

void jog() { printf("Jog\n"); }
void sleep() { printf("Back to sleep\n"); }
void brunch() { printf("Brunch\n"); }

void (*f[7])(() = {jog, jog, jog, jog, jog, sleep, brunch};

int main()
{
    int i;
    void (*todo)();
    for(i = 0; i < 7; ++i) {
        todo = f[i];
        todo();
    }
    return 0;
}
# Why Function Pointers?

```c
#include <stdio.h>
void todo(int date) {
    if(date < 5) {
        printf("Jog\n");
    } else if(i == 5) {
        printf("Back to sleep\n");
    } else {
        printf("Brunch\n");
    }
}

int main()
{
    int i;
    for(i = 0; i < 7; ++i) {
        todo(i);
    }
    return 0;
}
```

- Alternative implementation without function pointers
- Without function pointers code becomes...
  - Less modular
    - Now a single function `todo` is forced to perform disparate tasks
  - Less flexible
    - Now dates are hard-coded into `todo` and can’t be changed
- With array of function pointers, array can be updated at runtime to change your weekly schedule
Pitfall 1: Pass by value

• What do you think the following will print?
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;
}
```