Functions

CS449 Spring 2016

Procedural Languages

Procedural programming uses a **list of instructions** to tell the computer what to do step-by-step.

Procedural programming relies on **procedures**, also known as **routines** or **subroutines**.

A procedure contains a series of computational **steps to be carried out**.

Procedural programming is also referred to as **imperative programming**.

Procedural programming languages are also known as **top-down** languages.

Procedural programming is **intuitive** in the sense that it is very similar to how you would expect a program to work. If you want a computer to do something, you should provide step-by-step instructions on how to do it.

It is, therefore, no surprise that most of the **early programming languages** are all procedural. Examples of procedural languages include **Fortran, COBOL and C**, which have been around since the 1960s and 70s.

Functions in C

- What makes it a procedural language
- Loosely defined, 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 invoking or calling it.
- What are functions good for?
  - Code modularization (better readability)
  - Reusability (e.g. the C Standard Library)
  - Implementing recursive algorithms

Running Example

```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
```

Function Declaration

```
#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);”, “int printf(const char*);”
- 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 memory allocations 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)
- Parameter names are optional – ignored by compiler

Function Definition

```
#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>) { 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

```
#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 for number of parameters, it results in a compile error
- If argument types differ from parameters, arguments are coerced into parameter types
- All arguments are passed by value

Passing Arguments by Value

- The function defines a *parameter*, and the calling code passes an *argument*. Mnemonic:

Parameter = **p**arking space, **A**rgument = **a**utomobile.

- All arguments are passed by value in C
- Meaning: arguments are **copied** to parameters
  - Argument and parameter refer to different locations
- Compare: call by reference (e.g. C++)
  - Argument and parameter refer to the same location
- Compare: Java
  - **The same**: all arguments are passed by value in Java
  - Forget about anyone who told you java arguments are passed by reference (they mean “references to objects are passed by value”).

Why is argument passing needed?

- A function has access to the following locations:
  - Global variables (variables declared outside functions)
  - Local variables (variables declared inside the function)
  - Parameters (variables declared in parameter list)
  - BUT not the local variables or parameters of caller function
- Arguments allow local variable and parameter values to be passed from **caller** function to **callee** function
- But what if callee wants to modify the value of a caller local variable or parameter?

(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;
}
```

```
>> ./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?

(Correct) 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?
  - Use pointers as arguments
  - Parameters a and b still refer to storage locations that are different from x and y
  - But since the value of a and &x are identical (copied), the storage locations *a and *(&x) (or just x) are identical
- Impossible to do in Java (since it has no pointers)
  - Can modify content of objects passed as arguments
  - Cannot modify primitives or references to objects

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: Traverse(node) = Traverse(left node) + Traverse(right node)
  - Binary Search: Search(sorted array) = Search(left half) + Search(right half)
- C allows all types of recursion
  - Linear, binary, tail, mutual, nested

Example of Fibonacci Numbers

```
#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 even point to **functions** (not only data)
- Useful when you want one function call to perform different tasks (i.e. call a different function) in different situations.
  - E.g. Depending on day of week, when your 7:00 AM alarm rings, you might either go jogging, make breakfast, or just go back to sleep.
- Value of function name is the address of the function or the function pointer (just like an array name)
- Function name is not an l-value (cannot be assigned to, just like an array name)

Example of Function Pointers

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

```
>> ./a.out
a: 3, b: 4, ret: 7
a: 4, b: 3, ret: 0
```

Function Pointer Declaration

```
#include <stdio.h>
int add(int *a, int *b) {
  return *a+*b;
}
int swap(int *a, int *b) {
  int temp=*a; *a=*b; *b=temp; return 0;
}
void doIt(int *a, int *b, int (*f)(int*, int*)) {
  int ret = (*f)(a, b);
  printf("a: %d, b: %d, ret: %d\n", *a, *b, ret);
}
int main() {
  int x = 3, y = 4;
  int (*g)(int*, int*) = add;
  doIt(&x, &y, g);
  g = swap;
  doIt(&x, &y, g);
  return 0;
}
```

- Syntax: <return type> (*<name>) (parameter list)
  - e.g. “int (*g)(int*, int*)“
- Interpretation:
  - “g is a pointer with a return type int and a parameter list of (int, int)”
- Any function assigned to the function pointer must match its prototype exactly, or it will result in a type mismatch error

Function Pointer Call

```
#include <stdio.h>
int add(int *a, int *b) {
  return *a+*b;
}
int swap(int *a, int *b) {
  int temp=*a; *a=*b; *b=temp; return 0;
}
void doIt(int *a, int *b, int (*f)(int*, int*)) {
  int ret = (*f)(a, b);
  printf("a: %d, b: %d, ret: %d\n", *a, *b, ret);
}
int main() {
  int x = 3, y = 4;
  int (*g)(int*, int*) = add;
  doIt(&x, &y, g);
  g = swap;
  doIt(&x, &y, g);
  return 0;
}
```

- Syntax: (*<name>)(argument list)
  - e.g. “(*f)(a, b)“
- Interpretation:
  - “call function pointed to by p with argument list (a, b)”

Pitfall 1: String update

- 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;
}
```

- It will print Hello, because “str” and “s” refer to different locations

Pitfall 1: String update

- Solution:

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

Pitfall 2: String update

- What will happen with the following code?

```
void foo(char *s) { s[0] = ‘h’; }
int main()
{
  char *str = "Hello";
  foo(str);
  printf("%s\n", str);
  return 0;
}
```

- It will result in a segmentation fault (attempt to write to code section)
- Problem: “Hello” is a string constant so is not part of the modifiable data section of the program

Pitfall 2: String update

- Solution:

```
void foo(char *s) { s[0] = ‘h’; }
int main()
{
  char str[100];
  strcpy(str, “Hello”);
  foo(str);
  printf("%s\n", str);
  return 0;
}
```

Pitfall 3: Undeclared Functions

```
int main()
{
  menu();
}
void menu()
{
  //...
}
```

- Always remember to put either a prototype for the function or the entire definition of the function above the first time you use the function.

Pitfall 3: Undeclared Functions

- Solution:

```
void menu();
int main()
{
  menu();
}
void menu()
{
  …
}
```

Pitfall 4: Phantom returned values

```
int foo (a)
{
    if (a)
        return(1);
}
```

- Buggy, because sometimes no value is returned
- Make sure your functions always return some value (if not void functions)

Pitfall 5: Unsafe returned values

```
char *f() {
   char result[80];
   sprintf(result,"any string here");
   return(result);
 }


int g()
{
   char *p;
   p = f();
   printf("f() returns: %s\n",p);
}
```

- Problem: result is allocated on the stack rather than in data segment.
- Program might execute correctly as long as nothing has reused the particular piece of stack occupied by result.

Pitfall 5: Unsafe returned values

- Solution:

```
char *f() {
   char * result = "any string here";
   return(result);
 }

int g()
{
   char *p;
   p = f();
   printf("f() returns: %s\n",p);
}
```

Practice Problem 1

Write an iterative function in C, which gets an integer ***N*** as input and calculates/returns the factorial, denoted ***N!***, of that number:

*N!=1*2*3*4*5*….*(N-2)*(N-1)*N*

Practice Problem 2

Write a recursive function in C, which gets an integer ***N*** as input and calculates/returns the factorial, denoted ***N!***, of that number:

Practice Problem 3

Write an iterative function, which gets an integer ***N*** as input and returns the N-th Fibonacci number. Compare the performance of the recursive and the iterative functions for calculation of the Fibonacci numbers.