C Functions

CS 2060 Week 4

Prof. Jonathan Ventura

Modularizing programs

- In C, **functions** are used to *modularize* programs.
  - The C standard library provides *pre-packaged* functions available to all C programs, such as:
    - Mathematical calculations (math.h)
    - String manipulations (string.h)
    - Character manipulations (ctype.h)
    - Input/output (stdio.h)
  - `printf` and `scanf` are examples of functions from the C standard library.
- You can write your own **programmer-defined functions** as a way to modularize and share code.

Structure of functions in C

- C functions are **invoked** by a **function call**.
- The **calling function** does not know how the function operates; only its **arguments** and **return value** type.
- The function can in turn call other functions (or call itself).

- Functions provide a basic way to manage code by:
  - Hiding implementation details, or **abstraction**
  - Supporting code reusability
  - Avoiding repeated code

Examples of function calls

- Examples of function calls:

```
double result = sqrt(900.0);
printf(''%.2f'',result);
```

- Nested function calls:

```
printf(''%.2f'', sqrt(900.0));
```

Standard C math library functions in math.h

Function	Description
`sqrt(x)`	$\sqrt{x}$
`cbrt(x)`	$x^{(1/3)}$ (C99 and C11 only)
`exp(x)`	e^x
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`fabs(x)`	$\lvert x \rvert$
`ceil(x)`	$\lceil x \rceil$
`floor(x)`	$\lfloor x \rfloor$
`pow(x,y)`	x^y
`fmod(x,y)`	remainder of $\frac{x}{y}$
`sin(x),cos(x),tan(x)`	$\sin(x), \cos(x), \tan(x)$

Table: Commonly used math library functions

Writing custom functions

- A function definition must be preceded by a **function prototype**, e.g.:

```
int myfn( int x );
```

  - The function prototype gives the name of the function `myfn`, the arguments list `(int x)` and the return value type `int`.
  - Note the semicolon at the end of the line.

- Then later in the source code, the **function definition** gives the body of the function:

```
int myfn( int x ) { return x+1; }
```

  - The function definition repeats the function prototype.
  - The function body must be wrapped in curly braces and no semicolon is needed.

Examples of function definitions

```c
#include <stdio.h>

int square( int y );  // function prototype

int main()
{
    for ( int x = 1; x <= 10; ++x )
    {
        printf("%d ",square(x));
    }

    puts(""); // prints a string followed by \n
}

// function definition
int square( int y )
{
    return y * y;
}
```

Examples of function definitions

```c
#include <stdio.h>

int maximum( int x, int y, int z );

int main()
{
    int a, b, c;
    scanf( "%d %d %d", &a, &b, &c );

    printf("max of a,b,c is: %d\n", maximum(a,b,c) );
}

int maximum( int x, int y, int z )
{
    int max = x;
    if ( y > max ) max = y;
    if ( z > max ) max = z;
    return max;
}
```

Function calls must have the right number of arguments

```c
#include <stdio.h>

int maximum( int x, int y, int z );

int main()
{
    int a, b, c;
    scanf( "%d %d %d", &a, &b, &c );

    printf("max of a,b is: %d\n", maximum(a,b) ); // error
}

int maximum( int x, int y, int z )
{
    int max = x;
    if ( y > max ) max = y;
    if ( z > max ) max = z;
    return max;
}
```

Error checking function calls

- The compiler uses the function prototype to check whether a function call is correctly formed.
  - Correct number of arguments
  - Argument types match
  - Has return value
- We are also allowed to define a function without the prototype, as long as the definition precedes the function call.
- In this case, the compiler infers the function prototype from the function call.
  - Thus we cannot rely on the compiler for function call error checking.

Argument coercion

```c
#include <stdio.h>

int maximum( int x, int y, int z )
{
    int max = x;
    if ( y > max ) max = y;
    if ( z > max ) max = z;
    return max;
}

int main()
{
    float a, b, c;
    scanf( "%f %f %f", &a, &b, &c );

    printf("max of a,b,c is: %f\n", maximum(a,b,c) );
}
```

C's usual arithmetic conversion rules

- As with arithmetic expressions, C will automatically convert numeric types in function calls when necessary.
- C's **usual arithmetic conversion rules** determine how numeric types can be safely converted.
- The following (incomplete) list gives the **promotion** hierarchy:
  - `double`
  - `float`
  - `unsigned int`
  - `int`
  - `short`
  - `char`

Header files

- Header files (.h files) typically contain a long list of function prototypes.
- The function definitions are given in a separate source code (.c) file.

- Code which wants to use the functions only needs to include the header file which contains the function prototypes.

Header files

- Header files (.h files) typically contain a long list of function prototypes.
- The function definitions are given in a separate source code (.c) file.

- Code which wants to use the functions only needs to include the header file which contains the function prototypes.
- The compiler checks that the function calls match the prototypes.

Header files

- Header files (.h files) typically contain a long list of function prototypes.
- The function definitions are given in a separate source code (.c) file.

- Code which wants to use the functions only needs to include the header file which contains the function prototypes.
- The compiler checks that the function calls match the prototypes.
- Then the linker links the function calls to the function definitions.

The function call stack

- C manages function calls using a **function call stack**.
- The **stack** is one of the basic data structures in computer science.
  - Imagine a stack of pancakes.
  - We add a new pancake by placing it on the top of the stack.
    - This is called **pushing** onto the stack.
  - We remove a pancake by taking it off of the top of the stack.
    - This is called **popping** off of the stack.

The function call stack

- The stack is a **last-in, first out (LIFO)** data structure.
  - The *last item* added is the *first item* to be removed.
- When we call a function, we jump to a different location in the code.
  - We need to remember where in the code to return to when the function call returns.
  - The stack is perfect for this,
    - We **push** the return location onto the stack when calling the function.
    - We **pop** the return location off when the function call is done.
  - If the function calls another function, it will again push onto the stack and then pop it off later.

Call stack example

```c
int bar( int c )
{
    return c + 10;
}

int foo( int b )
{
    return bar(b) + 10;
}

int main()
{
    int a = foo(10);
}
```

- Function call stack:
  - main

Call stack example

```
int bar( int c )
{
    return c + 10;
}

int foo( int b )
{
    return bar(b) + 10;
}

int main()
{
    int a = foo(10);
}
```

- Function call stack:
  - main
  - foo

Call stack example

```c
int bar( int c )
{
    return c + 10;
}

int foo( int b )
{
    return bar(b) + 10;
}

int main()
{
    int a = foo(10);
}
```

- Function call stack:
  - main
  - foo
  - bar

Call stack example

```
int bar( int c )
{
    return c + 10;
}

int foo( int b )
{
    return bar(b) + 10;
}

int main()
{
    int a = foo(10);
}
```

- Function call stack:
  - main
  - foo

Call stack example

```c
int bar( int c )
{
    return c + 10;
}

int foo( int b )
{
    return bar(b) + 10;
}

int main()
{
    int a = foo(10);
}
```

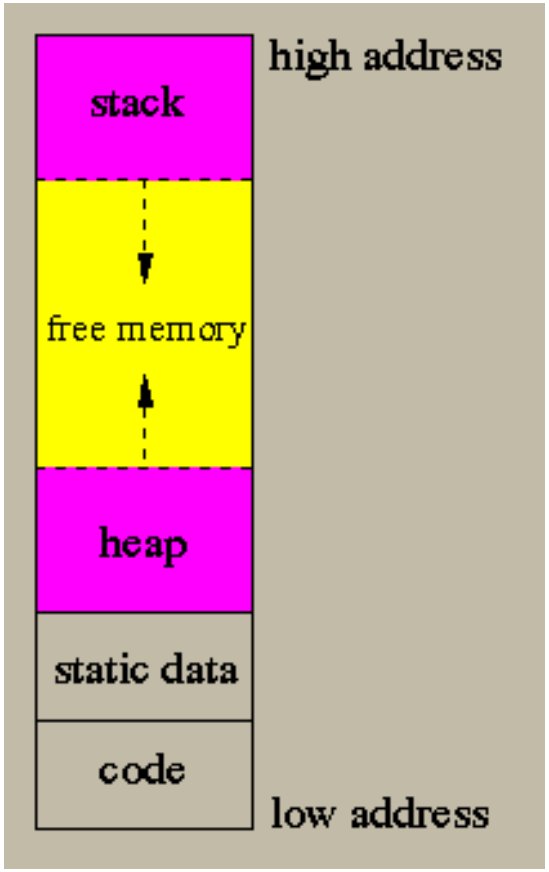
- Function call stack:
  - main

Stack frames

- The function call stack contains not just the return address but the **stack frame**.
- The stack frame contains the return address as well as the function's **local variables**.
  - Local or **automatic** variables are destroyed when the function ends,
  - as opposed to **global variables** which exist for the duration of the program's execution.
- The currently executing function's stack frame is always at the **top** of the stack.

Stack and heap in the MIPS architecture

- When we push a stack frame, we must allocate memory for it.
  - The stack actually grows downward in this illustration.
- Dynamic memory allocations take place in the **heap**, which grows upward.
  - Program code and data also take up some fixed portion of memory at the bottom.
- Running out of stack space is called **stack overflow** – this is a *fatal* error.



Stack frame example

```
int myfn( int a, int b, int c )
{
    int d = a * b;
    return d + c;
}

int main()
{
    int value = 10;
    int result = myfn( value, value, value );
}
```

- The compiler determines at compile-time the size of the stack frame by analyzing the function.
- What is the size of `myfn`'s stack frame?
  - Assume we are on a 32-bit system...

Stack frame example

```
int myfn( int a, int b, int c )
{
    int d = a * b;
    return d + c;
}

int main()
{
    int value = 10;
    int result = myfn( value, value, value );
}
```

- myfn's stack frame:
  - Return address:
    - 32-bits: 4 bytes
  - Arguments:
    - int a: 4 bytes
    - int b: 4 bytes
    - int c: 4 bytes
  - Local variables:
    - int d: 4 bytes
  - Return value:
    - int: 4 bytes
- Total: 24 bytes

Passing arguments by value and by reference

- In C, the function arguments are **passed by value**.
  - This means that a *copy* of the argument is made when the function is called.
  - The function cannot change the value of the variable in the calling function's stack frame.
- The alternative is **pass-by-reference**, where the called function operates directly on the calling function's variable.
- Pass-by-value is "safer" since it avoids accidental **side effects** when the calling function's variable is changed mistakenly.

Pass-by-value example

```c
#include <stdio.h>

int myfn( int a, int b )
{
    a += b;
    return a;
}

int main()
{
    int a = 10;
    myfn( a, 10 );
    printf( "%d\n", a );
}
```

- What does this program output?

Pass-by-value example

```c
#include <stdio.h>

int myfn( int a, int b )
{
    a += b;
    return a;
}

int main()
{
    int a = 10;
    myfn( a, 10 );
    printf( "%d\n", a );
}
```

- What does this program output?

  10

Pass-by-value example

```c
#include <stdio.h>

int myfn( int a, int b )
{
    a += b;
    return a;
}

int main()
{
    int a = 10;
    myfn( a, 10 );
    printf( "%d\n", a );
}
```

- What does this program output?

  10

- a is passed by value, so `myfn()` cannot change the value of `a` in `main()`.

Pass-by-value example

```c
#include <stdio.h>

int myfn( int a, int b )
{
    a += b;
    return a;
}

int main()
{
    int a = 10;
    myfn( a, 10 );
    printf( "%d\n", a );
}
```

- What does this program output?

  10

- a is passed by value, so `myfn()` cannot change the value of `a` in `main()`.
- (It is not an error to ignore a return value.)

Random number generation

- The `rand()` function (defined in `stdlib.h`) generates random integers between `0` and `RAND_MAX`.

```
int i = rand();
```

- The numbers are intended to *uniformly distributed*, meaning every number between `0` and `RAND_MAX` has equal chance of being generated.
- `RAND_MAX` is platform- and compiler- specific, but must be at least 32767.

Random numbers on intervals

- How could we generate a random number between 0 and 99 ?

Random numbers on intervals

- How could we generate a random number between 0 and 99 ?

```
int i = rand() % 100;
```

Random numbers on intervals

- How could we generate a random number between `0` and `99` ?

```
int i = rand() % 100;
```

- How could we generate a random number between `1` and `100` ?

Random numbers on intervals

- How could we generate a random number between 0 and 99 ?

```
int i = rand() % 100;
```

- How could we generate a random number between 1 and 100 ?

```
int i = rand() % 100 + 1;
```

Random numbers on intervals

- How could we generate a random number between 0 and 99 ?

```
int i = rand() % 100;
```

- How could we generate a random number between 1 and 100 ?

```
int i = rand() % 100 + 1;
```

- How could we generate a random number between -50 and 49 ?

Random numbers on intervals

- How could we generate a random number between 0 and 99 ?

```
int i = rand() % 100;
```

- How could we generate a random number between 1 and 100 ?

```
int i = rand() % 100 + 1;
```

- How could we generate a random number between -50 and 49 ?

```
int i = rand() % 100 - 50;
```

Rolling a six-sided die

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    for ( unsigned int i = 1; i <= 20; i++ )
    {
        printf( "%10d", 1 + rand() % 6 );

        if ( i % 5 == 0 ) puts("");
    }
}
```

Counting number frequencies

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned int num_heads = 0, num_tails = 0;

    for ( unsigned int trial = 1; trial <= 20000000; trial++ ) {
        int flip = rand() % 2;
        switch ( flip ) {
            case 0:
                ++num_heads;
                break;
            case 1:
                ++num_tails;
                break;
        }
    }

    printf("%s%13s\n", " Side", "Frequency");
    printf("Heads%13u\n", num_heads);
    printf("Tails%13u\n", num_tails);
}
```

Seeding the random number generator

- Running the program twice will produce the exact same output.
- This is because the `rand()` actually produces **psudorandom numbers**.

- By **seeding** the random number generator, we can produce different results.

Seeding the random number generator

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned int seed;
    printf("%s", "Enter seed: ");
    scanf("%u", &seed );

    srand(seed);

    for ( unsigned int i = 1; i <= 10; ++i ) {
        printf("%d ", rand()%10 );
    }

    puts("");
}
```

Seeding the random number generator

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // ??
    srand(rand());

    for ( unsigned int i = 1; i <= 10; ++i ) {
        printf("%d ", rand()%10 );
    }

    puts("");
}
```

Seeding the random number generator

- Typically we use the current clock time to seed the random number generator:

```
srand(time(NULL));
```

- `time(NULL)` (defined in `time.h`) returns the number of seconds since midnight on January 1, 1970 ("the epoch").

Seeding the random number generator

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand(time(NULL));

    for ( unsigned int i = 1; i <= 10; ++i ) {
        printf("%d ", rand()%10 );
    }

    puts("");
}
```

Creating enumerations with enum

- The enum keyword is used to create enumeration constants – keywords with custom names that can be used as markers or flags in your code.

```
enum Status { WHITESPACE, LETTER };
```

- Here we define two constants WHITESPACE and LETTER.
- We have also defined a variable type enum Status which can only be set to those values.

```
enum Status status = WHITESPACE;
```

  - Google Style Guide: constant names in all capitals scare small children.

Enumeration example

```c
#include <stdio.h>
#include <stdbool.h>

enum Status { WHITESPACE, LETTER };

int main()
{
    char input = ' ';
    enum Status status = WHITESPACE;
    int num_words = 0;

    while ( ( input = getchar() ) != EOF ) {
        switch ( input ) {
            case ' ':
            case '\n':
            case '\t':
                status = WHITESPACE;
                break;

            default:
                if ( status == WHITESPACE ) num_words++;
                status = LETTER;
        }
    }

    printf("%d\n",num_words);
}
```

Storage classes

- C provides **storage class specifiers** which change how variables are stored and their duration in the program.
- For now we will discuss `static`.

```
static int counter = 0;
```

- A variable declared `static` inside a function persists beyond the life of the function call – it is not destroyed at the end of the function call's execution.

static example

```c
#include <stdio.h>

int square( int x )
{
    static int counter = 0;

    counter++;
    printf("This function has been called %d times.\n", counter);

    return x*x;
}

int main()
{
    int sum = 0;
    for ( int i = 1; i <= 10; i++ ) sum += square(i);
    printf("sum: %d\n", sum );
}
```

Scope rules

- **Identifiers** (such as variables, labels and functions) have a defined **scope**.
- The scope of an identifer is the portion of the program in which the identifier can be referenced.

```
int myfn( int a ) { int x = a + 5; return x; }
```

```
int main() { x +=5; }
```
(error)

- For example, here variable `x` can only be referenced from inside the function `myfn`.
- We can't reference variable `x` in `myfn` from `main`.

Scope rules

- **Identifiers** (such as variables, labels and functions) have a defined **scope**.
- The scope of an identifer is the portion of the program in which the identifier can be referenced.

```
int myfn( int a ) { int x = a + 5; return x; }
```

```
int main() { x +=5; }
```
(error)

- For example, here variable `x` can only be referenced from inside the function `myfn`.
- We can't reference variable `x` in `myfn` from `main`.

Block scope

- Curly braces define a **block**.

  `int myfn( int a )` { (start of block)

  `int x = a + 5; return x;`

  } (end of block)

- Local variables defined inside a function have **block scope**, meaning that they can only be referenced from inside the function's block.

Block scope

- A block is also defined other constructs such as `if`, `for` and `while`:

```
if ( a == 5 ) { int b = a+5; printf("%d", b ); }
while ( true ) { char input = getchar(); }
```

- We also can define a block inside a function explicitly.
- With nested blocks, inner scopes might **hide** identifiers in outer scopes.

Block scope

```c
#include <stdio.h>

int main()
{
    int x = 5; // local to main

    { // start new scope
        int x = 7;
        printf("x in inner scope is: %d\n", x );
    } // end new scope

    printf("x in outer scope is: %d\n", x );
}
```

File scope

- Variables defined *outside* any function are **global variables** with **file scope**.
- File scope means the identifier can be referenced anywhere in the source code file.
- Function prototypes and definitions also have file scope.

File scope

```c
#include <stdio.h>

int x = 10; // global variable (file scope)

void changeX()
{
    x += 10;
}

int main()
{
    printf("x is %d\n", x );

    changeX();

    printf("x is %d\n", x );
}
```

Recursion

- **Recursion** is when a function calls itself.

```
int myfn( int a ) { return myfn(a); }
```

- When not done carefully, recursion causes an infinite loop (stack overflow will crash the program).
- Some stop condition is needed to exit the recursion.

Recursion example

```c
#include <stdio.h>

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

int main()
{
    int n;
    scanf( "%d", &n );

    for ( int i = 0; i < n; i++ )
    {
        printf( "%d ", fibonacci( i ) );
    }
    puts("");
}
```

Next Time

- Homework 5 due next Monday 11:55 PM.
- Chapter 6: C Arrays