

# CS 137 - Programming Principles

Kevin Carruthers

Fall 2012

---

## Outline

We will be covering

1. Basic C programming concepts
  - Variables
  - Integers, chars, expression evaluation
  - Conditionals
  - Loops (do, do while, for)
2. Functions, parameters, recursion
3. Arrays and pointers
4. Structures
5. Sorting, searching, time and space complexity

## Absolute Basics

```
// import standard input/output library
#include <stdio.h>

// main function returns an integer and takes no (void) parameters
int main(void) {
    // print formatted "Hello, World!"
    printf("Hello, World!");

    // returns 0, ie successful completion
```

```

    return 0;
}

```

## Inputs

```

#include <stdio.h>

int main(void) {
    // declare variables
    int a,b,r;

    // take two integers as input, assign them to a and b. %d is for integers
    scanf("%d,%d",&a,&b);

    // so long as b is non-zero...
    while(b) {
        // r is the remainder of a divided by b
        r = a % b;
        a = b;
        b = r;
    }
    printf("%d\n",a);

    return 0;
}

```

## Integer Data Types

Table 1: Types of integers and their sizes and ranges.

Type	Size	Range
unsigned char	1 byte	$[0, 2^8 - 1]$
char	1 byte	$[-2^{8-1}, 2^{8-1}]$
unsigned short	2 bytes	$[0, 2^{16} - 1]$
short	2 bytes	$[-2^{16-1}, 2^{16}]$
unsigned int	4 bytes	$[0, 2^{32} - 1]$
int	4 bytes	$[-2^{32-1}, 2^{32-1}]$
long	4/8 bytes	$[-2^{64-1}, 2^{64-1}]$

# Logic

False is denoted by zero, true is denoted by non-zero (traditionally one). The logical operators are

- NOT (!)
- OR (||)
- AND (&&)

## DeMorgan's Identities

- $!(P \ \&\& \ Q) == !P \ || \ !Q$
- $!(P \ || \ Q) == !P \ \&\& \ !Q$

# Loops

- `while(expression)`  
    statement // executes at least 0 times
- `do`  
    statement  
    while(expression); // executes at least 1 time

# Functions

```
// create function gcd which can be called with gcd(x,y) and returns an integer answer
int gcd(int a, int b) {
    int r = a % b;
    while(r) {
        a = b;
        b = r;
        r = a % b;
    }

    return b;
}

int main() {
    // print the result of a function call with arguments 806 and 338
    printf("%d\n",gcd(806,338));
}
```

```

    return 0;
}

// include boolean library (defines boolean type with true and false)
#include<stdbool.h>
#include<stdio.h>

bool isLeap(int year) {
    if(year%400 == 0) return true;
    else if(year%100 == 0) return false;
    else if(year%4 == 0) return true;
    else return false;
}

bool isPrime(int num) {
    int divisor = 2;
    if(num <= 1) return false;
    // expressions can be evaluated within loop tests
    while(num/divisor >= divisor) {
        if(num % divisor == 0) {
            return false;
            divisor++
        }

        return true;
    }
}

```

## Asserts

```

// include assert library
#include<assert.h>

bool leap(int year) {
    // if year <= 1582, terminates with error
    assert(year > 1582);

    ...
}

```

## Seperate Compilation

If you have multiple files (ie functions in one file, main program in another), declare the function in the main file as

```
void func(int number);
```

and compile as

```
% gcc -o output functions.c main.c
```

## Header Files

You can also declare the functions in a header file as

```
#ifndef HEADER_H
#define HEADER_H
void func(int number);
#endif
```

and in your main file

```
#include <header.h>
```

## Recursion

```
int gcd(int a, int b) {
    if(!b) return a;
    // call itself with updated/new arguments
    else return gcd(b,a%b);
}
```

## Locality

```
void func(int a) {
    // changes the value of a... within func
    a = 42;
}
```

```
int main() {
    int a = 3;
    func(a);
    // a has not been changed in this scope
}
```

```

    printf("%d\n",a);

    return 0;
}

```

## Arrays

```
int a[3] = {10,30,50};
```

creates an array, we can access the elements by

```
a[2];
```

which returns 50. Very useful is

```

// beginning with i = 0, iterate n times
// n is the number of elements in a
// i is a count of how many times we've gone through the loop
for(int i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    // add the current element to the sum
    sum += a[i];
}

```

Note that a for loop is defined as

```

for(initialization; condition; increment) {
    statement
}

```

where any one or more parameters may be removed.

## Floating-Point Numbers

```

double x = 4/5;           // 0.0
double x = 4.0/5;         // 0.8
double x = (double)4/5;   // 0.8

```

## Polynomials

Polynomials are often represented as arrays, ie  $3 + 4x - x^2$  is represented as

```
double f[] = {3,4,-1};
```

and must be evaluated with

```
double eval(double f, int n, double x);
```

## Example (Horner's Method)

```
#include<stdio.h>

double horner(double f[], int n, double x) {
    double h = f[n-1];

    // declaring a variable within a loop statement requires compilation with c99 standard
    for(int i = n-2; i >= 0; i--) {
        h = h * x + f[i];
    }
    return h;
}

int main() {
    double f[] = {2,9,4,3};

    // we could replace the 4 by the sizeof() stuff we did earlier
    printf("%g\n", horner (f,4,0));
    printf("%g\n", horner (f,4,1));
    printf("%g\n", horner (f,4,2));

    return 0;
}
```

## Math

```
#include <math.h>

has stuff like sin, cos, tan, exp, log, M_PI (ie pi as a constant)...
```

```
#include<stdio.h>
#include<math.h>

// let's find the root of this function (if it's continuous on [a,b])
double f(double x) {
    return x-cos(x);
}

double bisect(double a, double b, double epsilon, int maxIters) {
    double m;

    for(int i = 0; i < maxIters; i++) {
        // find the mindpoint
```

```

    m = (a+b)/2;
    // fabs is from the math library, it return the absolute value of a variable
    if(fabs(f(m)) <= epsilon) return m;
    // figure out which half has the answer within it
    if(f(m) > 0) {
        b = m;
    }
    else {
        a = m;
    }
}
return m;
}

int main() {
    printf("%g\n", bisect(-10,10,0.001,1000000));

    return 0;
}

```

## Structs

```

// defines a variable containing two other variables
struct tod {
    int hours;
    int minutes;
};

int main() {
    // declare a struct like this
    struct tod now = {13,46};
    struct tod later;
    // access member variables
    later.hours = now.hours + 3;
    later.minutes = now.minutes;

    return 0;
}

```

Structs can be passed as parameters or returned, just like pre-defined variables.



## Typedefs

```
// define a new type (not a struct anymore...)
typedef struct {
    int hours;
    int minutes;
} tod;

int main() {
    // notice the lack of struct in the definition?
    tod now = {13,53};

    return 0;
}
```

## Designated Initializers

```
int a[4] = {[2]=3, [0]=99};
gives us {99,0,3,0}
```

## Pointers

```
int main() {
    // create a new integer in memory, assign it a value of 6
    int i = 6;
    // create a pointer to a memory address, have it point to the address of i
    int *p = &i;
    // set the contents pointed to by p to 10
    *p = 10;

    // prints 10
    printf("%d\n",i);

    return 0;
}
```

## Coding Format

```
int* p, i;
is the same thing as
```

```
int *p, i;
```

so

```
int *p;
```

is preferred.

## Assignment

```
int main() {
    int i = 6;
    int *p = &i;
    int *q;
    // no stars here because we're assigning the memory address
    q = p;
    *q = 17;

    // prints 17
    printf("%d\n",i);

    return 0;
}
```

## Arguments

```
void swap(int *p, int *q) {
    int temp = *p;
    // stars here so we assign the data
    *p = *q;
    *q = temp;
}
```

```
int main() {
    int i=1, j=2;
    swap (&i, &j);

    // prints 2, 1
    printf("%d, %d\n",i,j);
}
```

## Return Values

```
int *largest(int a[], int n) {
    int m = 0;
    for(int i = 1; i < n; i++) {
        if(a[i] > a[m]) m = i;
    }
    // return the address as a pointer
    return &(a[m]);
}

int main() {
    int test[] = {0,1,2,3,2,1,0};
    // p will point to the largest element
    int *p = largest(test, sizeof(test)/sizeof(test[0]));
    *p = 100;

    // the third element is the largest... note how it has been changed to 100
    printf("%d\n",test[3]);

    return 0;
}
```

## Arithmetic

```
int a[8], *p, *q. i;
p = &a[2];
q = p+3;    // q = &a[5]
p += 3;     // p = q = &a[5]
p = q - 3;  // p = &a[2]
i = q - p;  // i = 3
```

## Memory

```
int *p = NULL;
```

points to nothing (null pointer) and takes no memory. We can create dynamic storage and memory allocation with

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

```
void *malloc(size_t size);           // allocates size amount of memory
void *realloc(void *p, size_t size); // reallocates previously allocated memory block
void free(void *p);                  // released memory allocated by malloc
```

```
// creates an array of size n filled with natural numbers one through n
int *numbers(int n) {
    int *p = (int *) malloc(n*sizeof(int)); // (int *) is technically optional, but good p
    for (int i = 0; i < n; i++) p[i] = i+1;
    return p;
}
```

## Safety

```
// adds an assert for error handling
void *safeMalloc(size_t size) {
    void *p = malloc(size);
    assert(p);
    return p;
}
```

## Structs

```
struct tod{int hour, minute;};
struct tod *t = (struct tod *) malloc(sizeof(struct tod));
```

## Strings

A string is an array of chars terminated by

"\0"

Examples:

```
char s[ ] = "hello";
char s[ ] = {'h', 'e', 'l', 'l', 'o', '\0'};
char *s = "hello";
```

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

```
// first counter
int count(char s[ ], char c) {
    int n = 0;
    for(int i = 0; s[i] != '\0'; i++) {
        if(s[i] == c) n++;
    }
    return n;
}
```

```
// second alternative
int count (char *s, char c) {
    int n = 0;
    for (; *s; s++) {
        if(*s == c) n++;
    }
    return n;
}

int main() {
    char *hi = "Hello World!";
    printf("%d\n", count(hi,'l'));

    return 0;
}
```

## String Library

```
// string library
#include <string.h>

size_t strlen(const char *s);           // returns the length of the string
char *strncpy(char *s0, const char *s1); // copies s1 onto s0 (s0 must be big enough
                                           // strncpy copies first n characters
char *strcat(char *s0, const char *s1); // concatenation
int strcmp(const char *s0, const char *s1) // alpha betical compare gives <0, 0, or >0
```

## Vectors

Vectors are better arrays with built in safety.

```
struct vector {
    int *a;           // actual array of data
    int size, length; // allocated and used storage
}

// proper initialization, for any new vector
struct vector *vectorCreate() {
    struct vector *v = (struct vector *) safeMalloc(sizeof(struct vector));
    v->size = 1;
    v->length = 0;
    v->a = (int *)safeMalloc(sizeof(int));
}
```

```

    return v;
}

// garbage handling and cleanup
struct vector *vectorDelete(struct vector *v) {
    // unnecessary, but recommended
    if(v) {
        free(v->a);
        free(v);
    }
    return (struct vector *) 0;
}

// behind-the-scenes assignment
void vectorSet(struct vector *, int index, int value) {
    assert(v && index >= 0);
    if (index >= v->size) {
        do
            v->size *= 2;
        while(index >= v->size);
        v->a = (int *) safeRealloc((void *) v->a, v->size*sizeof(int));
    }
    while(index >= v->length) {
        v->a[v->length] = 0;
        v->length++;
    }
    v->a[index] = value;
}

// behind-the-scenes return
int VectorGet(struct vector *v, int index) {
    assert(v && index >= 0 && index < v->length);
    return v->a[index];
}

// safe size function
int vectorLength(struct vector *v) {
    assert(v);
    return v->length;
}

```

# Big O Notation

$O(n)$  is a measure of complexity. Examples:  $3x^2 + 2 = O(x^2)$ ,  $6 \sin(x) = O(1)$ ,  $13 \log x = O(\log x)$ . We can also have best and worst case complexity, ie best case:  $O(1)$  (constant time), worst case:  $O(n)$  (linear time). We also have logarithmic, quadratic, and polynomial time.

## Sorting

Different types of sorting, all have different time and memory complexities.

### Selection

Find the smallest element, swap it with the first element, repeat. Best case:  $O(n^2)$ , average case:  $O(n^2)$ , worst case:  $O(n^2)$ .

### Insertion

Find where an element should go, shift up elements above that, insert it, repeat  $n-1$  times. Best case:  $O(n)$ , average case:  $O(n^2)$ , worst case:  $O(n^2)$ .

### Merge

Divide array in half, sort each half, merge the results. Sort results while merging. Sort "left side" of each array. Best case:  $O(n \log n)$ , average case:  $O(n \log n)$ , worst case:  $O(n \log n)$ .

### Quick

Pick a random pivot element, recursively sort both sides. Note this does not require a temporary array. It can also be improved by better choice of pivots, quicker partitioning methods, or optimized compiler functions. Best case:  $O(n \log n)$ , average case:  $O(n \log n)$ , worst case:  $O(n^2)$ .

## Binary Search

To search a sorted array, check the middle element, check the middle in whatever the direction your answer is, repeat. Time complexity is  $O(\log n)$ , which beats linear search ( $O(n)$ ).