



# csci 112

## Programming with C

### Test 3 Review



# Intro

- 25 questions
  - Just T/F and multiple choice
    - no multiple select or matching
  - 75 points total
    - Extra credit for evaluations will show as additional 2.25 points
      - which is equivalent to 3 points out of 100
- Test opens at 8am on 11/18 and closes at 11pm on 11/19
  - Once you start it, you have 50 minutes
- There is the pledge at the top of each question that you did not use any extra help (just like in the other tests)



## Chapter 7, section 7.7

# Enumerated Types and Defined Types

# User Defined Types

- Up until now, we have only looked at four basic types in C (along with pointers and arrays)
- We can actually define our own data types using the typedef keyword
- An example of a type definition is:

```
typedef int counter_t ;
```

- This code defines a new type that is the same as an int
- The following declarations are now both equivalent (and legal):

```
int flag = 0;  
counter_t flag = 0;
```

Counter  
Counter\_t

# Enumerated Types

- It is common practice to assign numerically increasing integer values to a set of variables belonging to a typedef, so C gives us the option to use enum

```
enum DAYS{  
    sunday ,  
    monday ,  
    tuesday ,  
    wednesday ,  
    thursday ,  
    friday ,  
    saturday  
};
```

Chapter 7, pgs 409-414

- Now sunday through saturday have the values 0 through 6

# Combining Enum and Typedef

```
typedef enum {  
    sunday ,  
    monday ,  
    tuesday ,  
    wednesday ,  
    thursday ,  
    friday ,  
    saturday  
} day_t;
```

# Chapter 10

## User-defined Structure Types

# Struct

## Structured Collection of Data



- We want to categorize all the planets
- We want to know the name, diameter, how many moons, time to orbit and how long it takes to rotate
- We can do this by creating our own data type using the keyword “struct”




# Struct

- **2 ways to do it**

```
struct Planet {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
};  
// in a function  
struct Planet p1, p2;
```

## MY PREFERRED METHOD



```
typedef struct {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
} Planet;  
// in a function  
Planet p1, p2;
```

# Struct

- Syntax

```
typedef struct {  
    <type> <id>;  
    <type> <id>;  
    ...  
} <name of structure data type>;
```

# Struct

## Setting the element values



```
typedef struct {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
} Planet;  
  
// in a function  
Planet p1, p2;  
p1.moons = 16;  
strcpy(p1.name, "Jupiter");
```

# Struct Operations



```
typedef struct {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
} Planet;  
  
// in a function  
Planet p1, p2;  
p1.moons = 16;  
strcpy(p1.name, "Jupiter");  
p2 = p1;  
printf("%s %lf %d", p1.name, p1.diameter, p1.moons);
```

# Struct

## A Structure As A Pointer



```
typedef struct {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
} Planet;  
// in main  
Planet p1;  
setPlanet(&p1);  
..  
void setPlanet(Planet *p) {  
    (*p).moons = 0;  
    strcpy((*p).name, "none");  
    ...  
}
```

DON'T USE THIS METHOD

# Struct

## A Structure As A Pointer



```
typedef struct {  
    char name[70];  
    double diameter;  
    int moons;  
    double orbit_time,  
           rotation_time;  
} Planet;  
// in main  
    Planet p1;  
    setPlanet(&p1);  
    ..  
}  
void setPlanet(Planet *p) {  
    p->moons = 0;  
    strcpy(p->name, "none");  
    ...  
}
```

# Struct

## A Structure Passed by Value



```
typedef struct {
    char name[70];
    double diameter;
    int moons;
    double orbit_time,
           rotation_time;
} Planet;
// in main
    Planet p1;

    ...
    printPlanet(p1);
    ...
}
void printPlanet(Planet p) {
    printf("name is %s\n", p.name);
    ....
}
```

**DON'T USE THIS METHOD**

# Struct12.c Explained

- In main:
  - `Student_t st_read; // Address is 59c0`
  - `st_read = StudentScan(); // st_read keeps 59c0`
- In `StudentScan()`:
  - `Student_t studnt; // Address is 5860`
  - `// name: 5860 (first element in struct) – same address as studnt`
  - `// id: 5888 (5860+28Hex(40)) year, 588c (5888+4), year: 5890 (588c+4)`
- Back in main:
  - `Student_t st_upd; // Address is 5980`
  - `st_upd = student_update_credits(st_read);`
- In `student_update_credits(Student_t st) { // st (input param) has address: 5900)`
  - `- st.credits += 15;`
  - `return st; // st is local and output (as st_read) – not the same as st_read which was input to this function`
- Back in main:
  - `// &st_read.name: 59c0 (first element in struct) – same address as st_read, &st_upd.name: 5980 (same as st_upd)`
  - `// st_read.name: 59c0 – same address as st_read, st_upd.name: 5980 (same as st_upd)`
  - `st_read.credits – 12 (as read in)`
  - `st_upd.credits – 27 (12 + 15)`

local now to function

```
typedef struct {  
    char name[40];  
    int id, credits;  
    char year;  
} Student_t;
```





# Struct13.c Explained

- In main:
  - Student st\_read; // Address is 59c0
  - st\_read = StudentScan(); // st1 keeps 59c0
- In StudentScan():
  - Student\_t studnt; // Address is 5860
  - // name: 5860 (first element in struct) – same address as studnt
  - // id: 5868 (5860+8) year, 586c (5888+4), year: 5870 (586c+4)
- Back in main:
  - Student\_t st\_upd; // Address is 5980
  - st\_upd = student\_update\_credits(str\_read);
- In student\_update\_credits(Student\_t st) { // st (input param) has address: 5900)
  - - st.credits += 15;
  - return st; // st is local and output (as str) – not the same as st\_read which was input to this function
- Back in main:
  - // &st\_read.name: 59c0 (first element in struct) – same address as st\_read, &st\_upd.name: 5980 (same as st\_upd)
  - // st\_read.name: 76b0, st\_upd.name: 76b0
  - st\_read.credits – 12 (as read in)
  - st\_upd.credits – 27 (12 + 15)

local now to function

```
typedef struct {  
    char *name;  
    int id, credits;  
    char year;  
} Student_t;
```

# Chapter 11

## File Input/Output

## Standard Input/Output

- `printf()` sends characters to standard output (`stdout`).
- `scanf()` reads characters from standard input (`stdin`).

By default, `stdout` and `stdin` both use the console.

## Redirection

Within linux, we can redirect `stdout` and `stdin` to point elsewhere.

We specify this when typing the command to start the program.

- If we redirect `stdout` to use a file, `printf()` sends characters to the file instead of the console.
- If we redirect `stdin` to use a file, `scanf()` reads characters from a file rather than the console.



# Redirection - Usage

## Redirecting stdout

Redirect stdout by adding the ' $>$ ' character, followed by the name of the file to redirect to.

```
$ ./ program > output_file . txt
```

## Redirecting stdin

Redirect stdin by adding the ' $<$ ' character, followed by the name of the file to redirect from.

```
$ ./ program < input_file . txt
```

We can also do both:

```
$ ./ program < input_file . txt > output_file . txt
```



## Redirecting stderr

Redirect stderr by adding the '2>' character, followed by the name of the file to redirect to.

```
$ ./ program > output_file.txt 2> error.txt
```



# Pointers to Files - Declaration

To point a file pointer at a particular file, we must initialize it.

Use `fopen()` to point to get the address of a FILE object.

The `fopen()` function is found in the header file “`stdio.h`”.

## Initialization

```
inp = fopen (" input_file . txt", " r");  
outp = fopen (" output_file . txt", " w");
```

- The first argument is the file we wish to access.
- The second argument is *how* we want to access the file.
  - "r" - read from the file
  - "w" - write to the file (erase any previous content)
  - "a" - append to the file (do not erase)

file1.c  
file2.c



# Pointers to Files - Moving the Pointer

The FILE pointer moves forward automatically as you read.  
What if we want to move the pointer ourselves?

We can use `fgetpos()` and `fsetpos()` to move the pointer manually.

## Usage

- The position in a file is represented in C by the type 'fpos\_t'.
- `fgetpos(inp, &pos)` stores the position for `inp` in `pos`.
  - The first argument is the file pointer.
  - The second argument is the `fpos_t` variable.
- `fsetpos(inp, &pos)` sets the position for `inp` to `pos`.
  - The first argument is the file pointer.
  - The second argument is the `fpos_t` variable.

file3.c



# Pointers to Files - Moving the Pointer

## Example

```
FILE *inp = fopen("in.txt", "r");  
fpos_t start_pos;  
fgetpos(inp, &start_pos);  
// do something ...  
fsetpos(inp, &start_pos);
```

Note that we can also move the pointer back to the beginning of the file at any time using the `rewind()` function.

- This has one argument: the FILE pointer.

```
rewind(inp);
```





# Binary Files

- To read large files in and print them, your program has to expend a lot of effort to convert it to binary then convert it back to ints, etc to print them.
- If there is no need for us to have to read these files, then we can make them binary files.
  - then we don't have to do this translation.
- A binary file has the computer's internal representation of each data element in the file.
- Using binary files has the disadvantage of increasing the file size.
  - You can do a compression on the file for storage, which you can't do for text files.
- Other disadvantages - can't be read by a human, is not portable.

# Binary File Function Calls

fopen – modes: use wb for write binary, rb for read binary

fclose – no change from a text file

fread - /public/examples/chap11/bfile3.c

fwrite - /public/examples/chap11/bfile1.c

# Advantages

Can read/write whole arrays/data structures with one call

Make sure you understand `/public/examples/chap11/bfile3.c`



# Chapter 9

## Recursion

# Recursion

- Definition
  - Recursion describes the process when a function calls itself.
  - A function,  $f$ , is also said to be recursive if it calls another function,  $g$ , which in turn calls  $f$ .
- Recursion is sometimes less efficient than iterative (loop based) approaches, but often provides a more natural and simple solution.
- Recursion is a powerful programming tool to solve certain problems.

# Recursion

- Recursion uses a strategy known as divide and conquer. This approach continually reduces the problem size until it reduces to a simple case with an obvious solution.
- The simple cases are known as base cases.
- The other cases, known as recursive cases, redefine the problem in such a way as to move closer to the base case.

# Basics

- The function needs to have these basics:  
if this is a simple or base case  
    solve it  
else  
    redefine the problem using recursion

# Bad Recursion

```
void bad_recursion(int n) {  
    printf("%d\n", n);           // BAD: missing base case  
    if (n%2 == 0) {              // GOOD: reduction case  
        bad_recursion(n/2); }    // BAD: only if n is even  
    else {  
        bad_recursion(3*n + 1); // BAD: step does not  
    }                           //      reduce problem  
}
```

- Recursion will not terminate for all possible inputs of positive  $n$ .
- A missing base case means the recursion will not "bottom out".
- This is analogous to an infinite loop.



# Multiprocessing using processes and threads

# Terminology (1 of 4)

- multitasking
  - dividing a program into tasks that operate independently of one another
- linear programming
  - writing a sequence of program instructions in which each instruction depends on the completion of the previous instruction
- parallel programming
  - execution of multiple programs at the same time

# Processes (1 of 2)

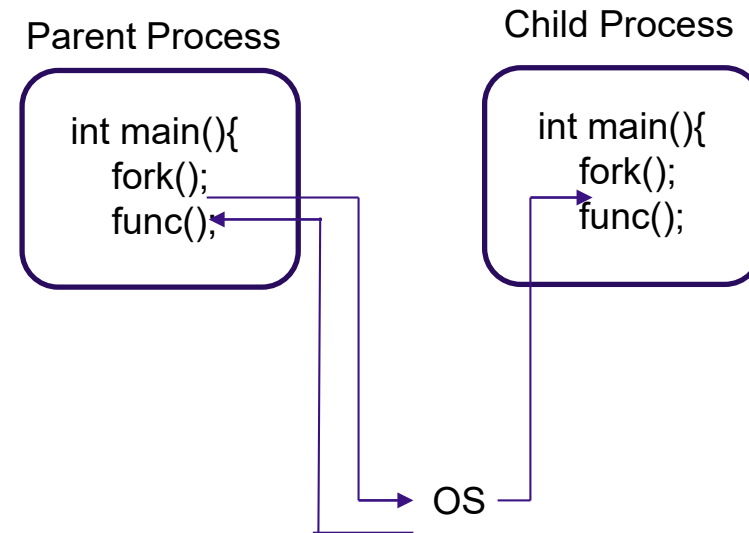
- process ID
  - a unique identifier given to a process by the operating system
- child process
  - a new process that is created by a currently executing process (the parent process)
- parent process
  - the currently executing process that has created one or more new child processes

# fork()

- Fork system call is used for creating a new process, which is called **child process**, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which are in use in the parent process.
- The **CPU executes a program** that is stored as a sequence of machine language instructions in main memory. It **does** this by repeatedly reading, or fetching, an instruction from memory and then carrying out, or **executing**, that instruction.
- It takes no parameters and returns an integer value. Below are different values returned by fork().  

forkfile.c
- **Negative Value**: creation of a child process was unsuccessful.  
**Zero**: Returned to the newly created child process.  
**Positive value**: Returned to parent or caller. The value contains process ID of newly created child process. from <https://www.geeksforgeeks.org/fork-system-call/>  
<https://www.microfocus.com/documentation/visual-cobol/VC23/VS2015/BKMTMTINTRS002.html>

# fork()



<https://www.geeksforgeeks.org/fork-system-call/>

# How many child processes will fork0a produce?

fork0a.c

```
fork (); // Line 1
fork (); // Line 2
fork (); // Line 3

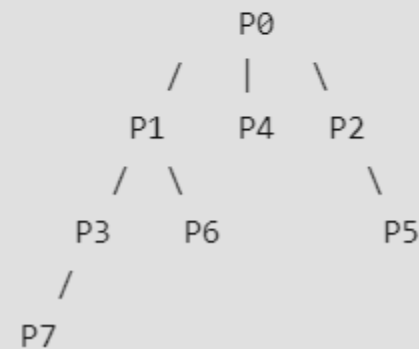
      L1      // There will be 1 child process
    /      \  // created by line 1.
  L2      L2  // There will be 2 child processes
 /  \    /  \ // created by line 2
L3  L3  L3  L3 // There will be 4 child processes
                // created by line 3
```

The main process: P0

Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7



## fork()

A process executes the following code

```
for (i = 0; i < n; i++) fork();
```

The total number of child processes created is

- (A)  $n$
- (B)  $2^n - 1$
- (C)  $2^n$
- (D)  $2^{(n+1)} - 1$

<https://www.geeksforgeeks.org/fork-system-call/>

**ANSWER IS B**

# Table 14.1 Some Process Functions from `unistd.h` and `wait.h`

Function	Purpose: Example	Parameters	Result Type
<code>fork</code>	If successful, creates a new process and returns the process ID of the new process (to the parent) and 0 (to the new process). If not successful, returns -1 to the parent. <code>pid = fork();</code>	None	<code>pid_t</code>
<code>wait</code>	Returns the process id of the next child process to exit. The exit status is written into the memory location pointed to by <code>status_ptr</code> .  <code>pid = wait( &amp;status_ptr );</code>	<code>int* status_ptr</code>	<code>pid_t</code>
<code>execl</code>	Replaces the instructions in the process that is executing with the instructions in the executable file specified by the <code>path</code> and <code>file</code> arguments. The argument <code>path</code> specifies the full path name including the executable file name; the argument <code>file</code> specifies just the executable file name; the ellipses indicates that there may be more arguments, but the last argument is always <code>NULL</code> .  <code>execl ("prog.exe", "prog.exe", NULL);</code>	<code>const char *path</code> <code>const char *file</code> <code>. . .</code> <code>NULL</code>	<code>int</code>

A call to `wait()` blocks the calling process until one of its child processes exits or a signal is received.

`execl1.c`, `execl2.c`, `execl3.c`



# Interprocess Communications and Pipes (1 of 2)

- interprocess communications
  - the exchange of information between processes that are running on the same CPU and that have a common ancestor
- pipe
  - a form of interprocess communications that consists of two file descriptors, one opened for reading and the other opened for writing

# Table 14.2 Some Interprocess Communications Functions from `unistd.h`

Function	Purpose: Example	Parameters	Result Type
<code>pipe</code>	If successful, creates a new pipe and returns a value of 0. The read and write file descriptors are written into the array argument. If not successful, returns -1.  <code>pipe (filedes);</code>	<code>int filedes[2]</code>	<code>int</code>
<code>dup2</code>	If successful, duplicates the file descriptor <code>oldfiledes</code> into the file descriptor <code>newfiledes</code> and returns <code>newfiledes</code> . If not successful, returns -1.  <code>dup2 (oldfiledes, newfiledes);</code>	<code>int oldfiledes</code> <code>int newfiledes</code>	<code>int</code>

# Cybersecurity in C

- <https://cybersecurityguide.org/resources/coding-for-cybersecurity/>
- Why is C popular for cybersecurity?



**Federico Mengozzi**, studied Computer Science at University of California, Davis



Answered June 29, 2017

The first reason that comes to my mind is that there many devices running some Unix distributions, I mean **so many**. Since the entire kernel of Unix is (or was ,I don't know if other languages are being used today) written in C, a good knowledge in C is mandatory to deal with any sort of computer security.

C is such a powerful language, it allows to explore every spot of an entire operating system. It's a widely used language when it comes to communicate with the operating system, a tons of applications have a C core.

In order to provide security a system is usually secured bottom-up, so one of the main concern of security is to have a solid, and most important **secure**, core to build about everything on top of it. And guess what? Very often this core is coded in C.

# Security vulnerabilities in C

# Security Issues with C

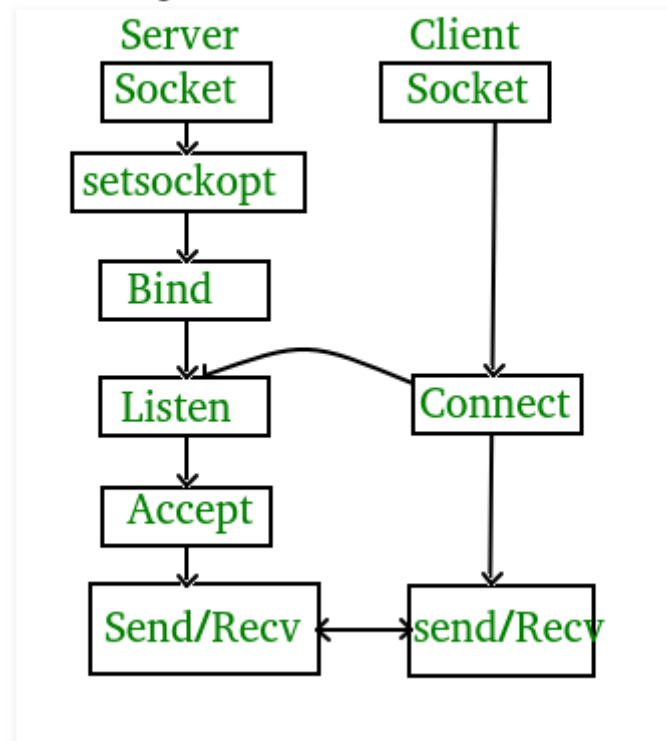
- Formatting string attacks
  - Using printf and not checking user inputs
  - Using %s and %x to print data from locations in memory
  - You can also use %n to write data to arbitrary locations
- Integer overflow -  
<http://projects.webappsec.org/w/page/13246946/Integer%20Overflows#:~:text=An%20integer%20overflow%20during%20a,when%20the%20data%20is%20copied>.
  - **Security Impact of Integer Operations**
  - Attackers can use these conditions to influence the value of variables in ways that the programmer did not intend. The security impact depends on the actions taken based on those variables. Examples include, but are certainly not limited, to the following:
    - An integer overflow during a buffer length calculation can result in allocating a buffer that is too small to hold the data to be copied into it. A buffer overflow can result when the data is copied.
    - When calculating a purchase order total, an integer overflow could allow the total to shift from a positive value to a negative one. This would, in effect, give money to the customer in addition to their purchases, when the transaction is completed.
    - Withdrawing 1 dollar from an account with a balance of 0 could cause an integer underflow and yield a new balance of 4,294,967,295.
    - A very large positive number in a bank transfer could be cast as a signed integer by a back-end system. In such case, the interpreted value could become a negative number and reverse the flow of money - from a victim's account into the attacker's.

# Communication between 2 programs

- sockets or pipes

At least one program has to run in the background  
To run a program in the background: &

State diagram for server and client model



write.c/read.c

# Pipes vs Sockets

- Use pipes:
  - when you want to read / write data as a file within a specific server. If you're using C, you read() and write() to a pipe.
  - when you want to connect the output of one process to the input of another process... see [popen\(\)](#)
- Use sockets to send data between different IPv4 / IPv6 endpoints. Very often, this happens between different hosts, but sockets could be used within the same host