CSI500/400 Starts: C Language Programming for OS Students

Prof. Seth Chaiken

Aug. 27, 2012 to August 31, 2012
Administration

About This Course

Lecture 02

Lecture 03

Lecture 04

Lecture 06
  Some C Elements
  Data Structure Example
  Virtual Memory Contents: Linux Specific
  C Storage Classes

One C program Example

C and Assembly Relationships
  Lifetime versus Scope
  Automatic
  Static
  Dynamic
Web site: http://www.cs.albany.edu/~sdc/CSI500

**CS UNIX Account**
Write your name on the attendance sheet.

▶ If you do not have a CS Department UNIX account, write that and fill out the application form. (Give a GOOD PASSWORD).

▶ If you HAVE a CS Department UNIX account, write the Login ID on the attendance sheet.

**First Week**

1. Lab Thu and Tues: Copy, run and modify x386 assembly language and C under Linux. Assignment now on the web. Followup: Learn the basic addressing modes of x386 assembly language.

2. Homework: Read and address questions from the first half of the MOS text intro. chapter graduate student style.
What is an OS?

1. Operating System
2. Is Software
3. The OS interfaces application software with people, storage and communication media, and exotic devices like robots (through the hardware).

We’ll study HOW OPERATING SYSTEMS WORK

1. What they do.
2. How to use them.
Why study OS?

1. Create better applications, deploy and manage systems better, and fix problems more efficiently.
2. Gain (maybe you’re first) experience with a highly complex computing system. Become able to work with complexity.
3. Learn to think system-wide.
4. Know some technology good for a few years (like 5?).
5. Maybe build device drivers or perhaps even lower-level OS components, perhaps in embedded applications like cell phones and robots.
C is a big part of this course.

Reason ONE

- The interface between popular OS/hardware combinations and application software is the C language and execution model. POSIX says so! (Prepare for quiz question: Find out what POSIX is.)
- Unlike Java, C exposes the addressable memory.

Reason TWO

- Most currently active OS are coded mostly in C.
- C with GNU C’s `asm` extension provides the kind of hardware access that OS code needs. There are very few assembly language files in the Linux kernel.
The Big Picture

- (When system is not idle,) most of the time, the computer hardware executes **machine instructions**.
- The CPU keeps track of the memory address of the current machine instruction in the Program Counter register (PC).
- The CPU keeps track of a small amount of data in a few General Purpose and/or Index Registers.
- The CPU activates RAM to fetch machine instructions.
- Each machine instruction tells the CPU to sometimes activate RAM to fetch data, sometimes fetch and/or change data in (CPU) registers, sometimes activate RAM to store data there. Also the new value for the PC is determined.

This part of **beautiful interface** operates most often.
The address space ranges from 0 to $2^{32} - 1$, which is 4 Gigabinary -1, approximately 4 billion!
Virtualization

OS + hardware system features implements ONE, SEPARATE, mostly ISOLATED, INDEPENDENT Virtual Computer CPU (1 or more with REGISTERS) + Addressable Memory one FOR EACH PROCESS currently running a program

Good Idea!
Imagine that ASM program from lab, by itself, in a bare hardware computer!
Lab 01 Purpose

1. A machine language program, coded in ASM for convenience.
2. It use regular instructions PLUS system call instructions.
3. The system call instructions do complex stuff.
4. The instructions plus the data reside in addressable memory.
5. A C program is compiled into an ASM program and assembled into a machine program.
6. A shell is an ordinary program that reads your typing and does things in response to it. IT CAN ALSO READ from a shell script file. Some of those things is call systems calls, like exec to run other programs!
7. make is a kind of shell.
   - Scripted by your Makefile
   - make’s algorithm figures out if files built from dependency input files must be rebuilt.
   - make orchestrates building the Linux kernel.
8. (Learning to use OS Lab technology EFFICIENTLY.)
Abstraction used by C Programmers—Behold Beauty

The beautiful interface is

- CPU(s) (Instruction Set Architecture from real hardware) limited to user-mode instructions (including traps for system calls)
- plus a Virtual Memory, containing program instructions and data.

It’s most beautiful seen in C.
Occasionally

- When the machine instruction is (for us) int 0x80 (or sysenter), the CPU starts to run machine instructions from the OS kernel.
- When a (RAM) memory access exception occurs, (or an illegal instruction is run), the CPU starts to run machine instructions from the OS kernel.
- When a periodic timer ticks, or an I/O device has more data or something, the CPU starts to run machine instructions from the OS kernel because an interrupt occurs.
- The OS kernel code might copy data to/from the part of memory used by an application.
- The OS kernel code might decide to make the CPU execute machine instructions from a different user process or thread than before.

CSI333 (Machine/Assembly language and C) are serious prerequisites for this course.
Once upon a time, there was a number in memory (RAM, Randomly Addressed Memory).

All numbers and other data in memory are in binary.

The machine instructions are in memory and in binary too.

Our number has 32 bits.

1101 1110 1010 1101 1011 1110 0000 1111

Written in hexadecimal: DEADBE0F (D = 1101 = 13, etc.)

A 4-bit number, in the 0-15 range, is called a nibble.
We want the computer to print this number.

To print the string like DEADBEB0F from 110111010101101111000001111,

- We have to store in memory the ASCII codes for those letters.
- The code for each letter is an 8 bit number from 0 to 255.
- We have to tell the Operating System to fetch 8 of those codes and print the corresponding letters or digits DEADBEB0F.

1. Nibble 1010 = Ten is printed A, 1011 = Eleven is printed B, ...
\[ 1101 = Thirteen \) is printed D.

2. The code for A is 65 (look up with man ascii, code for B is 65+1 = 66, etc.)

3. So, our program must take the 1101 (equals 13),
4. calculate how much away it is from it is from Ten (Thirteen - Ten = Three),
5. add this (Three) to the code for A (65 + 3 = 68).
6. store 68 (in 8-bit binary) in memory so the OS can fetch it.
Lecture 02

OS (Operating System) software. The OS Kernel is either part of the OS or is the whole OS, depending on the speakers' opinion, politics, business strategy.

What the OS Kernel does

OS Kernel plus the Hardware execute application programs in Machine Language. Assembly language expresses machine language in a way that’s much easier for people.

Machine instructions:

▶ Ordinary instructions (pushl, movl, movb, je, jle, call, ret, addl, addb, popl, jmp, subl) make hardware do simple things with registers and RAM memory.

▶ System call (int for “interrupt”) instructions seem to make the computer do something complicated (like copy data to or from a network connection or disk)

We say “seem to” because the OS kernel supplies the instructions for all that complicated stuff.
What the OS kernel plus hardware implement is sometimes called an ABSTRACT or VIRTUAL MACHINE

What is multiprogramming?
Modern systems have MANY SUCH VIRTUAL MACHINES running AT THE SAME TIME (or seeingly) in IN THE SAME COMPUTER HARDWARE

What is a process?
MOST IMPORTANT SUBJECT CONCEPT!
A PROCESS is one single RUN of one program within one “virtual machine” that runs during a particular period of time.

What is multiprocessing?
Several processes run ACTUALLY AT THE SAME TIME, in PARALLEL; each one is run by a DIFFERENT CPU CORE.
Yes, the word choice is misleading! Due to human history.
Dramatization

Multiprocessing: move one hand in a circle and the other hand up and down AT THE SAME TIME.
Multiprogramming: move one hand in a circle. When interrupted by the clock, the hand goes into the kernel and decides whether to continue circling or to resume going up and down. If it decides to go up and down, it does that until interrupted again.
Do you want to be a TA? and the NON-SUPRISE QUIZ

QUIZ
What did you find out about POSIX?

UG Teaching Assistant opportunities
Ugrads: Check your email from CSIB-L
See teaching assistant topic on
http://ccistudentcenter.posterous.com/

Finish the class
On your quiz sheet, write a catalog of (1) the names of assembly language instructions used and (2) the addressing or data access modes used.
Some of this was written on the scribbled notes now linked on the web site.
What is threading when a thread runs:

```asm
.section .text
.globl _start
_start: nop
    pushl $printbuf
    pushl $0xDEADBE0F
    call ltohex
    movl $4, %eax
    movl $1, %ebx
    movl $printbuf, %ecx
    movl $9, %edx
    int $0x80
    movl $1, %eax
    movl $-1, %ebx
    int $0x80

ltohex: pushl %ebp
    movl $8, %ecx
    movl %esp, %ebp
    movl 8(%ebp), %edx
    pushl %ebx
    movl 12(%ebp), %ebx
    jmp .L2
.L10: addl $48, %eax
    movb %al, -1(%ecx,%ebx)
    subl $1, %ecx
    je .L8
.L11: sarl $4, %edx
.L2: movl %edx, %eax
    andl $15, %eax
    cmpb $9, %al
    jle .L10
    addl $55, %eax
    movb %al, -1(%ecx,%ebx)
    subl $1, %ecx
    jne .L11
.L8: popl %ebx
    popl %ebp
    ret

.section .data
printbuf: .ascii "12345678\n"
```
Lecture 03

Finish the catalog of (1) instruction names and (2) addressing and other data access modes used in our demo of a thread running an assembly language program.

Learning objectives

- Demonstrate “Reading knowledge” of x386 assembly language by explaining what each instruction is for in short code sequences like that in the *Understanding the Linux Kernel* book.

- Explain what each operation in C program does in terms of what memory accesses are made, what storage class of memory (stack, free space, static, program memory), what decisions are made (including loop control) and how function calls and system calls are made.
We want the computer to print this number.

To print the string like DEADBE0F from 110111101011011011111000001111,

- We have to store in memory the ASCII codes for those letters.
- The code for each letter is an 8 bit number from 0 to 255.
- We have to tell the Operating System to fetch 8 of those codes and print the corresponding letters or digits DEADBE0F.

1. Nibble 1010 = Ten is printed A, 1011 = Eleven is printed B, ...
   1101 = Thirteen is printed D.
2. The code for A is 65 (look up with man ascii, code for B is 65+1 = 66, etc.)
3. So, our program must take the 1101 (equals 13),
4. calculate how much away it is from it is from Ten (Thirteen - Ten = Three),
5. add this (Three) to the code for A (65 + 3 = 68).
6. store 68 (in 8-bit binary) in memory so the OS can fetch it.
But, if the nibble is between 0 and 9, we add the code for the ZERO character 48 to the nibble to get the ASCII code. Now sketch on paper how to do this for all 8 nibbles. Strategy: Maintain in a 32 bit variable the nibbles NOT YET CONVERTED, and SHIFTED so the next one to convert is at the end. Loop 8 times:

1. Copy the last nibble into another variable.
2. Decide to add ‘0’ = 48 versus ‘A’ = 10, depending on whether the nibble is < 10 or not.
3. Copy the sum into the right spot in memory.
4. Decide whether to continue the loop.
5. If continuing, shift the nibbles not yet converted, update the “right destination spot” variable, and update the loop counter.
First, write a program to TEST your project.

```c
#include <unistd.h>
int main(int argc, char *argv[])
{
    char buffer[] = "12345678\n";
    ltohex(0xDEADBE0F,buffer);
    write(1,buffer,9);
    return 0;
}
```

And a Makefile to automate the build

```make
main : main.o ltohex.o
<tab here>gcc -o main main.o ltohex.o
main.o : main.c
<tab here>gcc -c main.c
ltohex.o : ltohex.c
<tab here>gcc -c ltohex.c
```
void ltohex(long number, char *dest)
{
    int nibbles = 8;
    dest = dest + 7;
    while(nibbles)
    {
        char digit = number & 0xF;
        if(digit < 10)
            digit = digit + '0';
        else
            digit = digit - 10 + 'A';
        *dest = digit;
        dest--;
        dest--;
        number = number >> 4;
        nibbles = nibbles - 1;
    }
}
How I got an equivalent assembly language program

1. Developed your LAB 01 material including demonstration of making Linux system calls in assembly language to print 9 characters from a memory buffer.
2. Wrote and debugged ltohex.c
3. Compiled ltohex.c to ASSEMBLY LANGUAGE ONLY with the command
   gcc -S -O2 ltohex.c
4. Coded in assembly language setting up the parameters and calling ltohex.
5. Copied the assembly language in ltohex.s and removed space and other stuff not really needed for a bare runnable program.
What is threading when a thread runs:

```assembly
.section .text
.globl _start
_start: nop
        pushl $printbuf
        pushl $0xDEADBE0F
        call ltohex
        movl $4,%eax
        movl $1,%ebx
        movl $printbuf,%ecx
        movl $9,%edx
        int $0x80
        movl $1,%eax
        movl $−1,%ebx
        int $0x80
ltohex: pushl %ebp
        movl $8, %ecx
        movl %esp, %ebp
        movl 8(%ebp), %edx
        pushl %ebx
        movl 12(%ebp), %ebx
        jmp .L2
.L11: addl $48, %eax
        movb %al, −1(%ecx,%ebx)
        subl $1, %ecx
        je .L8
.L10: addl $55, %eax
        movb %al, −1(%ecx,%ebx)
        subl $1, %ecx
        jne .L11
.L8:    popl %ebx
        popl %ebp
        ret
.section .data
printbuf: .ascii  "12345678\n"
```
Summary

1. When single thread runs, it “threads” through RAM memory executing one machine language instruction after another.
2. The thread CARRIES ALONG a handful of REGISTERS and data in 1, 2, 3 or so of them changes when each instruction executes.
3. The thread also sometimes copies data from or to RAM Memory.
4. System call machine instructions like int 0x80 seem to perform complicated operations, including printing, in a single step.
5. The thread doesn’t see the enormous number of machine instructions from the OS kernel executed between the system call and its return.

Abstraction: System call number 4 prints stuff.
Reality: Between int 0x80 and the movl ... after it, the CPU runs many many instructions from the OS kernel, and they activate the graphics chip to change what’s displayed on the screen.
Lecture 04

Homework 2 Assigned

Finish Chapter 2 and answer 8 questions FROM THE WEBSITE.

General 2 week assignment

Become able to explain the purpose of every line of our assembly language program and correlate it with the operations in the C function ltohex().
Consider a system that has two CPUs and each CPU has two threads (hyperthreading).
Suppose three programs, $P_1$, $P_2$ and $P_3$, are started with run times 5, 10 and 20 milliseconds, respectively.
How long will it take to complete the execution of these programs? Assume all three programs are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.
“The next obvious step is to replicate not only the functional units, but also some of the control logic. The Pentium 4 ... have this property, called multithreading or hyperthreading ... . To a first approximation, what it does is allow the CPU to hold the state of two different threads and then switch back and forth on a nanosecond time scale. ... Multithreading does not offer true parallelism. Only one process at a time is running, but thread switching time is reduced to the order of a nanosecond.”
Let’s solve the problem

Problem data:
3 “Programs” that would take, in isolation, 3 amounts of CPU time, and CPU time ONLY:

P0 uses 5 ms.  P1 uses 10 ms.  P3 uses 20 ms.

2 CPUs each with two threads, so:
Up to 2 programs can run AT THE SAME TIME.
Up to 2 programs can be worked on by each CPU.
But if a CPU is working on 2 programs, the time it spends is multiplexed (means SHARED).

What’s your answer?

1. 20 milliseconds?
2. 25 milliseconds?
3. 30 milliseconds?
All 3 answers are right. It depends on the program-to-CPU assignment made by the OS.
All 3 answers are right. It depends on the program-to-CPU assignment made by the OS.
Correctly illustrate the operation of concurrent and/or parallel threads, processes or other activities using a UML-style sequence diagram, and make correct analyses about timing, and (eventually) race conditions, locking, etc.
Quiz:
Draw the sequence diagram that shows how 30 milliseconds can be taken. Reminder: P0(5ms)  P1(10ms)  P2(20ms)
C Language

1. Variables: HOLD, HAVE, STORE, RETAIN, KEEP TRACK OF the current VALUE. Most variables are implemented by chunks of RAM
   BUT: The compiler’s optimizer might choose to use a REGISTER instead
2. You won’t go too wrong if you think that a variable is an object.
1. Each variable also has a (data) type. Data types: Interpretations for data stored as raw bits in the computer. What operations are legal.

2. Java syntax is based on C syntax.

3. Imperative: An executable statement commands to computer to do something.

To build and run a C application

1. Write or edit the source file with a text editor, and name it with the .c suffix, say ASourceFile.c
2. Compile and link using one or more commands.
   Example: gcc -c ASourceFile.c
   gcc -o myprogram ASourceFile.o
   YOU give these COMMANDS one by one to a shell; OR (better!) type them into a Makefile once and for all.
3. Run the linked executable file from the command line
   Example: ./printArg1 TheFirstArgString
OS students use the shell’s command line interface
The shell is a program that prompts for and accepts typewritten commands. When an executable file’s name is given as a command, the shell runs (or executes) that file.

Where a process comes from
The shell (1) commands the OS to create a new process (child) and (2) makes the new process run the printArg1 program. Meanwhile, the shell waits for the child process to exit.
The C program from Lab 02

#include "racer.h"

#ifdef CONFIG_VOLATILE
volatile
#endif
int ring = 0;

void * racer(void *tid)
{
/* More to come.... */
The C program from Lab 02

/*Maybe a volatile here..*/ int ring = 0;

void * racer(void *tid)
{

    int count;
    for(count = nLoops; count > 0; count--)
    {
        ring = ring + 1;
    }
}

tid and count are LOCAL, (automatic) STACK-ALLOCATED variables belonging to the racer function.
ring is a GLOBAL, STATICALLY ALLOCATED, sharable among C-functions and threads, in the .data or .bss section.
nLoops is a MYSTERY.
```c
#include "racer.h"

#ifdef CONFIG_VOLATILE
volatile
#endif
int ring = 0;

void * racer(void *tid)
{
    int count;
    for(count = nLoops; count > 0; count--)
    {
        ring = ring + 1;
    }
}
```
#include "config.h"
#define nLoops 20000000

#ifdef CONFIG_VOLATILE
volatile
#endif
int ring;
/* ring has static lifetime.
   ring is initialized in racer.c
   All the new threads run racer so
   they all share ring.
*/

void * racer( void * arg );
See on paper the actual input to preprocessing and compiling AFTER the two `#includes` are processed.
What’s hard about C?

- You must understand Virtual Memory in addition to application program data structures, logic and design.
- IT GETS WORSE: The virtual memory and code execution model that C programmers understand is MUCH SIMPLER than HOW IT IS IMPLEMENTED by REAL MEMORY and other HARDWARE together with the OS!
- The OS/Hardware combination is designed to implement the C memory and code execution model.
- (A model is an “abstraction”.)
The Simple C Abstraction Implemented by Complicated HW/OS Combination

Memory for a C application:

- Almost all of a 32-bit (4 Gigabyte) address space.
- Same addresses, starting from 0, used each time it runs.
- User doesn’t worry much about how much physical memory the current computer has.
- Network and file data accessed through function calls.

What bare hardware has

- Some fixed amount of memory; the same address space shared among all programs currently running.
- Some of the address space is reserved to access devices.
- Some address space is reserved for “supervisor mode”.
- Fixed number of network, disk, etc. devices.
Why knowing C is important

We must understand what it is that the HW/Operating System combination implements.
Most OS code is written in C.
C operations correspond closely to hardware operations.
The MOST IMPORTANT ABSTRACTION for OS
A PROCESS is (1) One Virtual Memory together with (2) One (or more for multithreading) Virtual CPU(s) that together are CURRENTLY implementing ONE instance of a program in execution and (3) some allocated resources (files, network connections, ... )

In other words, a process is
One virtual machine that is currently holding one running instance of one program.
Today’s systems (all but the smallest) provide multiprogramming.

Multi-...
Multiprogramming is many processes in a system at once. Multiprocessing is multiprogramming with multiple CPUs so several processes can run truly in parallel, i.e., actually at the same time.
Virtual Memory Contents Simplified

Running C Program’s and Other Processes’ Abstraction of Memory

© 2007 Matt Welsh – Harvard University
int N1 = 38;
...
scanf("%d", N1) /* Parameter passed is 38 */

38 is usually not legal user process address, so after you type in a number, scanf tries to copy a 32-bit integer into memory locations 38, 39, 40, and 41. The system takes a Segmentation fault

int N1 = 38;
...
scanf("%d", &N1) /* The parameter passed is the address of variable N1 */ /* scanf overwrites the initial value 38 of variable N1 */
What I demonstrated

1. Configuring software by editing a header file.
2. Compiling the configuration that will crash.
3. Demonstrating a crash: (1) Process BLOCKS for input. (2) When `scanf` tries to store data into memory, it CRASHES.
4. Demonstrating warnings when compiling with `-Wall`
5. Re-configuring so it doesn’t crash.

Note: make was NOT USED.
ALL parameters are passed BY VALUE

Java
In Java, all parameters are passed BY VALUE too, just like C.

//This is Java.
Rectangle r = new Rectangle(0,100,20,20);
someMethod( r );

The body of someMethod can change the width of the rectangle object because the parameter is the ADDRESS OF that object, called REFERENCE in Java.

The C++ world
But C++ DOES LET YOU DECLARE reference parameters!
Kernel programmers/students/hackers BEWARE!!

With C preprocessor macros, code to refer to the address of a variable can be hidden!

#define GETINT(x) scanf("%d", &x )

int N1;
GETINT(N1);

I believe this a bad programming style; but similar things are done in kernel and other system codes FOR GOOD REASONS. Example: To hide the implementation details of something like a mutex, semaphore, wait queue, etc.
C Statements

Executable statements

1. Expression Statement: Ends with a semicolon.
2. Compound Statement: \( \text{if( ... )} \) \textit{One statement}
3. Block: Delimited by matching curly-braces
   \{ \textit{statements} ... \}

Functions (Synonyms: Subroutines, Procedures, Methods.)

1. Defining Functions: 
   \textit{prototype like} \int \textit{fun(char *p) then body like} \{ \textit{return} p+1; \}
2. Declaring Functions: 
   \textit{prototype only, followed by semicolon;}
   Usually in header files.
   So function \textbf{defined elsewhere} can be called.
3. Calling Functions, function return, return value.
Every variable must be declared

Variable = Object
Variable = Memory Chunk = Object
It has a location, sometimes a name, and a CURRENT STATE or VALUE.

- int retlen; declares retlen to be a local variable of type integer.
- char *pch (in function definition header!) declares pch to be of type “pointer to character”

The declaration specifies the variable’s type.
The type is essentially the same thing as class: Determines what the values can be and what operations are available.
We illustrate how, in Unix, the “ragged array” data structure is used to pass command line arguments to a C program. when this program is run by the command:

`.myProgm  argU1 583`

the command line arguments appear this way:

```
main( int argc, char * argv[] )
```

```
. / m y P r o g m \0
a r g U 1 \0
5 8 3 \0
```
Pointer to an array of C strings 1

Understanding C declarations. Go inside out.

```c
char * argv []
```
Understanding C declarations. Go inside out.

`char * argv[]

argv` is declared

argv
Understanding C declarations. Go inside out.

`char * argv[]`

`argv` is declared to be an array.
Understanding C declarations. Go inside out.

`char * argv []` 

`argv` is declared to be an array of pointers
Pointer to an array of C strings

Understanding C declarations. Go inside out.

```
char * argv[]
```

`argv` is declared to be an array of pointers to characters.
Pointer to an array of C strings

Understanding C declarations. Go inside out.

`char * argv[]`

`argv` is declared to be an array of pointers to characters

```
argv
```

```
./myProgram
argU1
583
0
```
Array parameters to functions ARE POINTERS

Understanding C declarations. Go inside out.

```c
int main(int n, char * argv[])
```

`argv` is declared to be an array of pointers to characters.
How Virtual Memory of One Process is used, typically.

\[ 2^{32} - 1 = 0xFFFFFFFF \]

Virtual Mem. used by (Linux) kernel, illegal for a user process to access.

command–line arguments, envir. vars.
pointer arrays and C strings.

Stack pointer in `%esp`

(Boundaries of legality in the gap are set by syscalls `brk` and/or `mmap`)

Address 0x0

Managed by malloc/free.

uninitialized data, initialized to 0 by exec.

data initialized from executable file

executable code copied from exe. file

stuff used for dynamic linking

(The gap. Mostly illegal. Might be used for thread stacks, shared memory or dynamically linked libraries.)
pointer in C = reference in Java

Similarities
Both a C pointer and a Java reference value is an address, which
(if not null or illegal) REFERS TO an object in memory.

Differences

- Numerical arithmentic (binary) CAN be done on C pointers,
  but CANNOT be done on Java references.

- A number (say a literal 0x1FFF0000) CAN be used as a C
  pointer but CANNOT be used as a Java reference.

  temp = * ((int *) (0x1FFF0000));
The 3 storage classes of C

“Local Lifetime”
Local variables and parameters reside in the stack frame created when the function is called.
Those variables go away when the function returns.
Examples: fun(int LV1){ int LV2; int LV3; ... }

“Static Lifetime”
Reside in the “bss” and “data” sections of the process’s virtual memory.
They exist throughout the entire lifetime of the process.
Ways to declare:
▶ OUTSIDE of any function body.
▶ static qualifier inside a function or block.
▶ C string literals "Like This"
Local, Static, and (3) "Dynamic Lifetime"

Analogous to objects allocated by `new` in Java and C++. A "heap" area is used. C applications call `p=malloc(no. bytes requested)` to allocate memory and `free(p)` to recycle it. Kernels have their own dynamic memory allocation and freeing functions.
```c
#include <unistd.h>

static int strlen(char *pch)
{
    int len = 0;
    while( *pch != 0 ) {
        len++;
        pch++;
    }
    return len;
}

int main( int argc, char* argv[] ) {
    if( argc != 2 ) {
        write(1,"Wrong num. Args\n", 17);
        return 1;
    }
    else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
```
C Elements Used

Computes length of a C-string

- Formal parameters argc, argv are local variables.
- Value pch is the address of the first string character.
- len, temp are also local variables.
- Only one library function, write is used. It is a Unix system call.
- temp, declared “char *temp” is a pointer variable. *temp refers to the byte-sized variable containing the character in memory whose address is the current value of temp.
- ( *temp != 0 ) tests if that character is “the null character”.
Types

**Primative**
short int, long int, int, char, float, double
pointers

**Composite: Structures**
Like classes in Java, but only with data members.

**Composite: Arrays**

**Composite: Unions**
Storage Lifetimes in Assembly Language

Static
Example

```c
int AStaticVar;
int ASInited = 14;

char *fun()
{
    return
    "A Lit Str";
}

Compiled with
```
gcc -S Data.c
```

```asm
.file "Data.c"
.globl ASInited
.type ASInited, @object
.size ASInited, 4
ASInited:
    .long 14
    .align 4
    .section .rodata
.LC0:
    .string "A Lit Str"
.globl fun
.type fun, @function
fun:
    pushl %ebp
    movl %esp, %ebp
    movl $.LC0, %eax
    popl %ebp
    ret
.size fun, .-fun
.comm AStaticVar,4,4
.ident "GCC: (GNU) 4.2.4"
.section .note.GNU-stack
```
Top Level Objectives: Demonstrated Abilities

1. Explain what a complex software/hardware system does during normal and abnormal operational scenarios, and “drill down” to details of those subsystems that were studied. Which one? Modern general purpose computer operating environment, emphasis on Linux/Unix.

2. Do intermediate (1st yr. grad., 4th sem. undergrad. semester) complexity C and some i386 Asm. projects that utilize and/or simulate technologies listed in the course descriptions.

3. Solve problems: Analyze scenarios for each topic, including identification and comparing consequences of alternative choices. Utilize numbers, graphs, formulas, sequence diagrams, etc., together with accurate logical reasoning.

4. Install Linux systems, build Linux kernels and kernel modules, find and annotate code in Linux sources for given OS functions, run, debug and test kernel modifications and modules. Report OS behavior using system tools and interfaces.
Benefits of Virtualization

Virtual CPU

- Real CPU is timeshared; it’s shared among many running programs at once.
- Ugly instructions (and other interfaces) to control memory and hardware are hidden.
- Code to use ugly interfaces (1) resides in the OS kernel and (2) is invoked ultimately by traps for system calls.

Virtual Memory

- The virtual memories used by different programs are completely separate (usually). Provides protection.
- Memory addresses used by one program are independent of addresses used by another program.
- The OS manages paging data between RAM and disk when RAM is too small.
OS students break out gcc steps

Required for current assignments

1. gcc -S ASourceFile.c Compiles into Assembly Language.
2. gcc -c ASourceFile.s Assembles into an Object File.
3. gcc --static -o printArg1 ASourceFile.o Links (statically) the object file with library objects and writes out executable file named printArg1

Assignments include analyzing assembly language files like ASourceFile.s
NOTE DOWN these commands for future use (soon).
A C Function

```c
#include <unistd.h>
static int strlen(char *pch) {
    int len = 0;
    while(*pch != 0) {
        len++;
        pch++;
    }
    return len;
}
int main(int argc, char* argv[]) {
    if(argc != 2) {
        write(1, "Wrong num. Args\n", 17);
        return 1;
    }
    else {
        write(1, argv[1], strlen(argv[1]));
        write(1, "\n", 1);
        return 0;
    }
}
```
Elements Illustrated

2. Consistent indentation!
3. Include files.
4. Functions.
5. Static Function.
6. Parameter variables.
7. C strings.
8. “Local” variables.
9. While loops.
10. Pointer variables.
12. Increment an int variable.
13. Increment a pointer variable.
14. Test when pointer addresses the end of a C string.
15. Return.
16. The main function.
17. Command line argument parameters of main().
18. Pointer to an array of C strings.
19. The write system call.
20. File descriptor (2) for std. output.
22. Value returned by main() to the shell.