211: Computer Architecture
Summer 2016

Liu Liu

Topic:
- C Programming
Recap

Structure:
- header files
- global / local variables
- main()
- macro

Basic Units:
- basic data types
- arithmetic / logical / bit operators
- expressions

Basic syntax:
- control flow:
  - condition(if-then-else, switch case)
  - loop (for, while, do-while)
  - goto (break, continue)

functions:
- function signature (parameters, return type)
- function calls (return statement)
Today’s Topic

- Memory model (heap / stack)

- I / O

- **Pointers**

- Arrays (pointer to Array / Array of Pointers)

- Structures (pointer to Structure)

- Static / Dynamic Memory Allocation
Input and Output

Variety of I/O functions in C Standard Library

#include <stdio.h>

printf("%d\n", counter);
- String contains characters to print and formatting directives for variables
- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)

scanf("%d", &startPoint);
- String contains formatting directives for parsing input
- This call says to read a decimal integer and assign it to the variable startPoint. (Don't worry about the & yet.)
#include <stdio.h>

int printf( const char * arg1, ...); //goes directly into stdout
return # of chars being written.

e.g.
char name[10] = “Liu Liu”;
printf(“hello, I’m%s
”, name);
int num = printf(“hello, I’m%s
”, name);
printf(“%d chars written to stdout
”, num);

//copy and paste to editor
Input: fprintf

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

int printf(FILE *file, const char * arg1, ...); //goes directly into stdout
return # of chars being written.
```

e.g.
```
File * newFile = fopen("file","w");
char* name = "Liu Liu";
int num = fprintf(newFile, "hello, I\'m%s\n", name);
```

//copy and paste to editor
Memory Model

C’s memory model matches the underlying (virtual) memory system
- Array of addressable bytes
C’s memory model matches the underlying (virtual) memory system

- Array of addressable bytes

Variables are simply names for contiguous sequences of bytes

- Number of bytes given by type of variable

Compiler translates names to addresses

- Typically maps to smallest address
- Will discuss in more detail later
Pointers

A pointer is just an address in memory

Can have variables of type pointer
  - Hold addresses as values
  - Used for indirection

When declaring a pointer variable, need to declare the type of the data item the pointer will point to
  - int *p;       /* p will point to a int data item */

Pointer operators
  - De-reference: *
    • *p gives the value stored at the address pointed to by p
  - Address: &
    • &v gives the address of the variable v
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;

store the value 4 into the memory location associated with i

```
i
4
4304

ptr
?
4300
```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;

store the address of i into the memory location associated with ptr
int i;
int *ptr;
i = 4;
ptr = &i;
*ptr = *ptr + 1;

store the result into memory at the address stored in ptr

read the contents of memory at the address stored in ptr
Example Use of Pointers

What does the following code produce? Why?

```c
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}

... 
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```
Parameter Pass-by-Reference

Now what does the code produce? (swap orig data) Why?

```c
void Swap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}

... int fv = 6, sv = 10;
    int* fv_ptr = &fv;
    Swap(&fv, &sv);
    Swap(fv_ptr, &sv);
    printf("Values: (%d, %d)\n", fv, sv);
```
Null Pointer

Sometimes we want a pointer that points to nothing.

In other words, we declare a pointer, but we’re not ready to actually point to something yet.

```c
int *p;
p = NULL;  /* p is a null pointer */
```

**NULL** is a predefined constant that contains a value that a non-null pointer should never hold.

- Often, NULL = 0, because address 0 is not a legal address for most programs on most platforms.
Type Casting

C is NOT strongly typed

Type casting allows programmers to dynamically change the type of a data item
Arrays

Arrays are contiguous sequences of data items

- All data items are of the same type
- Declaration of an array of integers: “int a[20];”
- Access of an array item: “a[15]”

Array index always start at 0

The C compiler and runtime system do not check array boundaries

- The compiler will happily let you do the following:
  - int a[10]; a[11] = 5; //warnings but no errors
Array Storage

Elements of an array are stored sequentially in memory

```
char grid[10];
```

First element (grid[0]) is at lowest address of sequence

Knowing the location of the first element is enough to access any element
Arrays & Pointers

An array name is essentially a pointer to the first element in the array

1. char word[10];
2. char *cptr;
3. cptr = word; /* points to word[0] */

Difference:

- Line 1 allocates space for 10 char items
- Line 2 allocates space for 1 pointer
- Can change value of cptr(value of cptr = *(&cptr)) whereas cannot change value of word(value of word = &word[0])
  - Can only change value of word[i]
Arrays & Pointers (Continued)

Given

```c
char word[10];
char *cptr;
cptr = word;
```

Each row in following table gives equivalent forms

<table>
<thead>
<tr>
<th>cptr</th>
<th>word</th>
<th>&amp;word[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>cptr + n</td>
<td>word + n</td>
<td>&amp;word[n]</td>
</tr>
<tr>
<td>*cptr</td>
<td>*word</td>
<td>word[0]</td>
</tr>
<tr>
<td>*(cptr + n)</td>
<td>*(word + n)</td>
<td>word[n]</td>
</tr>
</tbody>
</table>
Pointer Arithmetic

Be careful when you are computing addresses

Address calculations with pointers are dependent on the size of the data the pointers are pointing to

Examples:

- int *i; ...; i++; /* i = i + 4 */
- char *c; ...; c++; /* c = c + 1 */
- double *d; ...; d++; /* d = d + 8 */

Another example:

double x[10];
double *y = x;
*(y + 3) = 13; /* x[3] = 13 */
Passing Arrays as Arguments

Arrays are passed by reference (Makes sense because array name ~ pointer)

Array items are passed by value (No need to declare size of array for function parameters)

```c
#include <stdio.h>

int *bogus;

void foo(int seqItems[], int item)
{
    seqItems[1] = 5;
    item = 5;
    bogus = &item;
}

int main(int argc, char **argv)
{
    int bunchOfInts[10];

    bunchOfInts[0] = 0;
    bunchOfInts[1] = 0;

    foo(bunchOfInts, bunchOfInts[1]);

    printf("%d, %d\n", bunchOfInts[0], bunchOfInts[1]);
    printf("%d\n", *bogus);
}
```
Common Pitfalls with Arrays in C

Overrun array limits

- There is no checking at run-time or compile-time to see whether reference is within array bounds.

```c
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

Declaration with variable size (important)

- Size of array must be known at compile time.

```c
void SomeFunction(int num_elements) {
    int temp[num_elements];
    ...
}
```
Structures

A struct is a mechanism for grouping together related data items of different types.

Example: we want to represent an airborne aircraft

```c
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

We can use a struct to group these data items together
Defining a Struct

We first need to define a new type for the compiler and tell it what our struct looks like.

```c
struct flightType {
    char flightNum[7];  /* max 6 characters */
    int altitude;        /* in meters */
    int longitude;       /* in tenths of degrees */
    int latitude;        /* in tenths of degrees */
    int heading;         /* in tenths of degrees */
    double airSpeed;     /* in km/hr */
};
```

This tells the compiler how big our struct is and how the different data items are laid out in memory

- But it does not allocate any memory
- Memory is only allocated when a variable is declared
Declaring and Using a Struct

To allocate memory for a struct, we declare a variable using our new data type.

```c
struct flightType plane;
```

Memory is allocated, and we can access individual members of this variable:

```c
plane.altitude = 10000;
plane.airSpeed = 800.0;
foo(&(plane.airSpeed);
/* pass the address of plane.airSpeed */
```
Array of Structs

Can declare an array of struct items:

- struct flightType planes[100];

Each array element is a struct item of type “struct flightType”

To access member of a particular element:

- planes[34].altitude = 10000;

Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:

- (planes[34]).altitude = 10000;
We can declare and create a pointer to a struct:

```c
struct flightType *planePtr;
planePtr = &planes[34];
```

To access a member of the struct addressed by `dayPtr`:

```c
(*planePtr).altitude = 10000;
```

Because the . operator has higher precedence than *, this is NOT the same as:

```c
*planePtr.altitude = 10000;
```

C provides special syntax for accessing a struct member through a pointer:

```c
planePtr->altitude = 10000;
```
Passing Structs as Arguments

Unlike an array, a struct item is passed by value.
Most of the time, you’ll want to pass a pointer to a struct.

```c
int Collide(struct flightType *planeA, struct flightType *planeB)
{
    if (planeA->altitude == planeB->altitude) {
        ...
    } else
        return 0;
}
```
Break
Memory Management

When a function call is performed in a program, the run-time system must allocate resources to execute it
  - Memory for any local variables, arguments, and result

The same function can be called many times (Example: recursion)
  - Each instance will require some resources

The state associated with a function is called an activation record
Allocating Space for Variables

Activation records are allocated on a call stack

- Function calls leads to a new activation record pushed on top of the stack
- Activation record is popped off the stack when the function returns

Let’s see an example
Allocating Space for Variables

\texttt{summation(5);} \\
\texttt{n=5; result = ?} \\
Stack
Allocating Space for Variables

\texttt{summation(5);}
\texttt{summation(4);}

\begin{align*}
\text{n=5; result = ?} \\
\text{n=4; result = ?}
\end{align*}
Allocating Space for Variables

```
summation(5);
summation(4);
... 
summation(0);
```

```
n=5; result = ?
n=4; result = ?
n=0; result = ?
```
Allocating Space for Variables

As functions return, their activation records are removed

**CRITICAL:** The state in a function call can be accessed safely only so long as its activation record is still active on the stack.
Dynamic Allocation

What if we want to write a program to handle a variable amount of data?

- E.g., sort an arbitrary set of numbers
- Can’t allocate an array because don’t know how many numbers we will get
  - Could allocate a very large array
  - Inflexible and inefficient

Answer: dynamic memory allocation

- Similar to “new” in Java
Another area region of memory exists, it is called the **heap**

Dynamic request for memory are allocated from this region

Managed by the run-time system (actually, just a fancy name for a library that’s linked with all C code)
Dynamic Space Allocation

After function returns, memory is still allocated

Request for dynamic chunks of memory performed using a call to the underlying runtime system (a system call).

- Commands: `malloc` and `free`
malloc

The Standard C Library provides a function for dynamic memory allocation

```c
void *malloc(int numBytes);
```

`malloc()` (and `free()`) manages a region of memory called the heap

- We’ll explain what a heap is later on and how it works

`malloc()` allocates a contiguous region of memory of size `numBytes` if there is enough free memory and returns a pointer to the beginning of this region

- Returns NULL if insufficient free memory

Why is the return type `void*`?
Dynamic Space Allocation

\[
x = \text{malloc}(4)
\]
Dynamic Space Allocation

\[ x = \text{malloc}(4) \]

System provides this chunk of memory to foo()

NOTE: This allocation is NOT part of the activation record

foo()’s activation record

0x3456

Stack

0xFFFF

0x0000

instructions

global data

0x3456

x = 0x3456
Using malloc

How do we know how many bytes to allocate?

Function

\[
\text{sizeof(type)} \\
\text{sizeof(variable)}
\]

Allocate right number of bytes, then cast to the right type

\[
\text{int *numbers = (int *)malloc(sizeof(int) * n);}
\]
free

Once a dynamically allocated piece of memory is no longer needed, need to release it

- Have finite amount of memory
- If don’t release, will eventually run out of heap space

Function:

```c
void free(void*);
```
Example

```c
int airbornePlanes;
struct flightType *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes = (struct flightType*)malloc(sizeof(struct flightType) * airbornePlanes);
if (planes == NULL) {
    printf("Error in allocating the data array.\n");
    ...
}
planes[0].altitude = ...  // Note: Can use array notation or pointer notation.
...
free(planes);
```

If allocation fails, `malloc` returns NULL.
Summary

Two ways to “get memory”

- Declare a variable (just covered)
  - Placed in global area or stack
  - Either “lives” forever or “live-and-die” with containing function
  - Size must be known at compile time
- Ask the run-time system for a “chunk” of memory dynamically
typedef

typedef is used to name types (for clarity and ease-of-use)

- typedef <type> <name>;

Examples:

- typedef int Color;
- typedef struct flightType WeatherData;
- typedef struct ab_type {
  int a;
  double b;
} ABGroup;

- struct ab_type a;
- ABGroup a;
Preprocessor

C compilation uses a preprocess called cpp

The preprocessor manipulates the source code in various ways before the code is passed through the compiler

- Preprocessor is controlled by directives
- cpp is pretty rich in functionality

Our use of the preprocessor will be pretty limited

- #include <stdio.h>
- #include “myHeader.h”
- ifndef _MY_HEADER_H
  #define _MY_HEADER_H
  ...
  #endif /* _MY_HEADER_H */
Much useful functionality provided by Standard C Library

- A collection of functions and macros that must be implemented by any ANSI standard implementation
  - E.g., I/O, string handling, etc.
- Automatically linked with every executable
- Implementation depends on processor, operating system, etc., but interface is standard

Since they are not part of the language, compiler must be told about function interfaces

Standard header files are provided, which contain declarations of functions, variables, etc.

- E.g., stdio.h
- Typically in /usr/include