CSC 4304 - Systems Programming
Fall 2008

LECTURE - III

ADVANCED STRUCTURES IN C

Tevfik Koşar

Louisiana State University
September 9th, 2008

Summary of Last Class

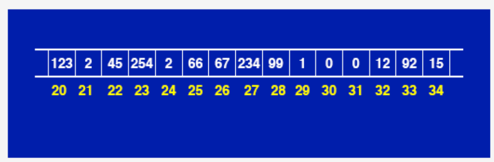
- Basic C Programming:
  - C vs Java
  - Writing to stdout
  - Taking arguments
  - Reading from stdio
  - Basic data types
  - Formatting
  - Arrays and Strings
  - Comparison Operators
  - Loops
  - Functions

In Today's Class

- Advanced Structures in C
  - Memory Manipulation in C
  - Pointers & Pointer Arithmetic
  - Parameter Passing
  - Structures
  - Local vs Global Variables
  - Dynamic Memory Management

Memory Manipulation in C

- To a C program, memory is just a row of bytes
- Each byte has some value, and an address in the memory

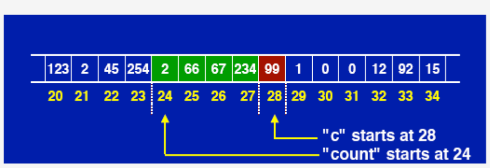


Memory Manipulation in C

- When you define variables:

```
int count;
unsigned char c;
```

- Memory is reserved to store the variables
- And the compiler 'remembers their location'



Memory Manipulation in C

- As a result, each variable has two properties:
  1. The '**value**' stored in the variable
     - If you use the name of the variable, you refer to the variable's value
  2. The '**address**' of the memory used to store this value
     - Similar to a reference in Java (but not exactly the same)
     - A variable that stores the address of another variable is called a **pointer**

- Pointers can be declared using the * character

```
int *ptr;                  /* Pointer to an int */
unsigned char *ch;         /* Pointer to an unsigned char */
struct ComplexNumber *c;   /* Pointer to a  struct ComplexNumber */
int **pp;                  /* Pointer to a  pointer to an int */
void *v;                   /* Pointer to anything (use with care!) */
```

Defining Pointers

- To use pointers, you must give them a value first
  - Like any other variable
- The '&' operator gives you the **memory address** of any variable

```
int i = 8;

int *p;          /* p is a pointer to an int */

p = &i;          /* p contains the address of variable i */

double *d = &i; /* ERROR, wrong pointer type */
```

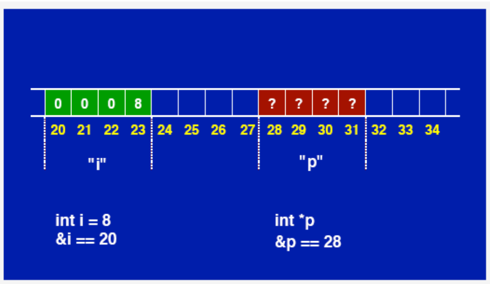

Using Pointers

- Once you have a pointer, you can access the value of the variable being pointed by using '*'

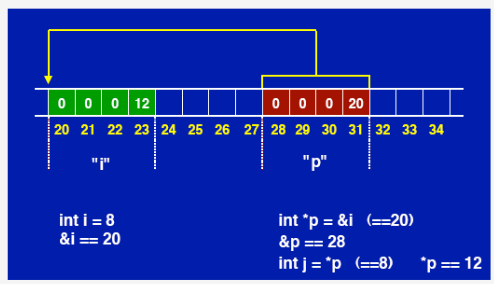
```
int i = 8;
int *p = &i;
int j = *p;
*p = 12;
```

☞ Attention, the '*' sign is used for two different things:
- To **declare** a pointer variable: `int *p;`
- To **dereference** a pointer: `*p=12;`

Using Pointers



Using Pointers



Parameter Passing in C

- In C, function parameters are passed **by value**
  - Each parameter is copied
  - The function can access the copy, not the original value

```
#include <stdio.h>

void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int x = 9;
  int y = 5;
  swap(x, y);
  printf("x=%d y=%d\n", x, y);
  return 0;
}
```


Parameter Passing in C

- In C, function parameters are passed **by value**
  - Each parameter is copied
  - The function can access the copy, not the original value

```
#include <stdio.h>

void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int x = 9;
  int y = 5;
  swap(x, y);
  printf("x=%d y=%d\n", x, y); /* This will print: x=9 y=5 */
  return 0;
}
```

Parameter Passing in C

- To pass parameters by reference, use pointers
  - The pointer is copied
  - But the copy still points to the same memory address

```
#include <stdio.h>

void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int x = 9;
  int y = 5;
  swap(&x, &y);
  printf("x=%d y=%d\n", x, y); /* This will print: x=5 y=9 */
  return 0;
}
```


Arrays and Pointers

- You can use pointers instead of arrays as parameters

```
#include <stdio.h>

void func1(int p[], int size) { }

void func2(int *p, int size) { }

int main() {
  int array[5];
  func1(array, 5);
  func2(array, 5);
  return 0;
}
```


Arrays and Pointers

- You can even use array-like indexing on pointers!

```
void clear(int *p, int size) {
    int i;
    for (i=0;i<size;i++) {
        p[i] = 0;
    }
}

int main() {
  int array[5];
  clear(array, 5);
  return 0;
}
```


Arrays and Pointers

- So a string is in fact just a pointer to a character array:

```
int main() {
   char s1[32] = "Hello, world!\n";
   char *s2;
   char s3[32];
   s2 = s1;            /* s1 and s2 point to the same character array */
   strncpy(s3,s1,31); /* s3 contains a copy of s1 */
```


Pointer Arithmetic

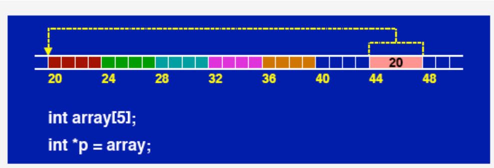
- Pointers are just a special kind of variable
- You can do **calculations** on pointers
  - You can use +, –, ++, –– on pointers
  - This has no equivalent in Java
- Be careful, operators work with the **size** of variable types!

```
int i = 8;
int *p = &i;
p++;  /* increases p with sizeof(int) */

char *c;
c++; /* increases c with sizeof(char) */
```

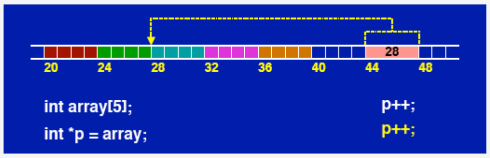

Pointer Arithmetic

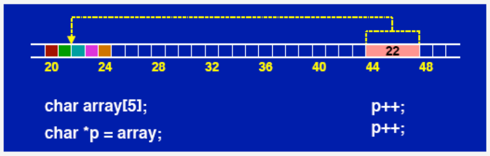
- This is obvious when using pointers as arrays:

```
int i;
int array[5];
int *p = array;

for (i=0;i<5;i++) {
   *p = 0;
   p++;
}
```



Pointer Arithmetic





Structures

- You can build higher-level data types by creating structures:

```
struct Complex {
  float real;
  float imag;
};
struct Complex number;
number.real = 3.2;
number.imag = -2;

struct Parameter {
  struct Complex number;
  char description[32];
};
struct Parameter p;
p.number.real = 42;
p.number.imag = 12.3;
strncpy(p.description, "My nice number", 31);
```


Pointers to Structures

- We very often use statements like:

```
(*pointer).field = value;
```

- There is another notation which means exactly the same:

```
pointer->field = value;
```

- For example:

```
struct data {
   int counter;
   double value;
};

void add(struct data *d, double value) {
   d->counter++;
   d->value += value;
}
```


Enumerations

- **enum** is used to create a number of related constants

```
enum workdays {monday, tuesday, wednesday, thursday, friday };

enum workdays today;
today = tuesday;
today = friday;

enum weekend {saturday = 10, sunday = 20};
```


Variables

- C has two kinds of variables:
  - Local (declared inside of a function)
  - Global (declared outside of a function)

```
int global;

void function() {
  int local;
}
```


Static Local Variables

- Declaring a static variable means it will persist across multiple calls to the function

```
void foo() {
  static int i=0;
  i++;
  printf("i=%d\n",i);  /* This prints the value of i on the screen */
}

int main() {
  int i;
  for (i=0;i<3;i++) foo();
}
```

This program will output this:

```
i=1
i=2
i=3
```

Non-static Local Variables

- If ***i*** is not static, the same example program (from prev. slide) will output:
  - i=1
  - i=1
  - i=1

Global Variables

Global variables have file scope:

```
int i=0;

void foo() {
    i++;
    printf("i=%d\n",i);
}

int main() {
    for (i=0;i<3;i++) foo();
}
```


Dynamic Memory Management

- Until now, all data have been static
  - It is clear by reading the program how much memory must be allocated
  - Memory is reserved at compile time
- But sometimes you want to specify the amount of memory to allocate **at runtime**!
  - You need a string, but you don't know yet how long it will be
  - You need an array but you don't know yet how many elements it should contain
  - Sizes depend on run-time results, user input, etc.

Dynamic Memory Management

- `malloc()` will allocate any amount of memory you want:

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - `malloc` takes a size (in bytes) as a parameter
    - If you want to store 3 integers there, then you must reserve `3*sizeof(int)` bytes
  - It returns a pointer to the newly allocated piece of memory
    - It is of type `void *`, which means "pointer to anything"
    - Do not store it as a `void *`! You should "cast" it into a usable pointer:

```
#include <stdlib.h>
int *i = (int *) malloc(3*sizeof(int));
i[0] = 12;
i[1] = 27;
i[2] = 42;
```


Dynamic Memory Management

- After you have used `malloc`, the memory will remain allocated until you decide to destroy it

```
#include <stdlib.h>
void free(void *pointer);
```

- After you have finished using dynamic memory, **you must release it!**
  - Otherwise it will remain allocated (and unused) until the end of the program's execution

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  /* Use i */
  free(i);
  /* Do something else */
}
```


Dynamic Memory Management

- Unlike arrays, dynamically allocated memory can be returned from a function.

```
int *createIntArrayWrong() {
  char tmp[32];
  return tmp;                        /* WRONG! */
}

int *createIntArray(int size) {
    return (int *) malloc(size*sizeof(int)); /* CORRECT */
}

int main() {
    int *array = createIntArray(10);
    /* ... */
    free(array);
    return 0;
}
```

Memory Leaks

- You must **always** keep a pointer to allocated memory
  - You need this to use it, and free it later
  - If you don't, you've got a **memory leak**
  - Memory leaks will slowly reserve all the machine memory, causing the program (or the machine) to crash eventually!

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  i = 0;      /* Wooops, I lost the pointer to my dynamic memory */
  free(???);  /* It is too late to free my dynamic memory */
```

- If you run out of memory, `malloc` will return `NULL`

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

int main() {
    int *array = (int *) malloc(10*sizeof(int));

    if (array == NULL) {
        printf("Out of memory!\n");
        return 1;
    }

    /* do something useful here */
    return 0;
}
```


malloc Example

```
int main ()
{
   int x = 11;
   int *p, *q;

   p = (int *) malloc(sizeof (int));
  *p = 66;
   q = p;
   printf ("%d %d %d\n", x, *p, *q);
   x = 77;
  *q = x + 11;
   printf ("%d %d %d\n", x, *p, *q);
   p = (int *) malloc(sizeof (int));
  *p = 99;
   printf ("%d %d %d\n", x, *p, *q);
}
```

```
$./malloc
11 66 66
77 88 88
77 99 88
```

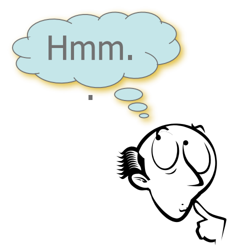

free Example

```
int main ()
{
   int x = 11;
   int *p, *q;
   p = (int *) malloc(sizeof (int));
  *p = 66;
   q = (int *) malloc(sizeof (int));
  *q = *p - 11;
   free(p);
   printf ("%d %d %d\n", x, *p, *q);
   x = 77;
   p = q;
   q = (int *) malloc(sizeof (int));
  *q = x + 11;
   printf ("%d %d %d\n", x, *p, *q);
   p = &x;
   p = (int *) malloc(sizeof (int));
  *p = 99;
   printf ("%d %d %d\n", x, *p, *q);
   q = p;
   free(q);
   printf ("%d %d %d\n", x, *p, *q);
}
```

```
./free
11 ? 55
77 55 88
77 99 88
77 ? ?
```


Summary

- Advanced Structures in C
  - Memory Manipulation in C
  - Pointers & Pointer Arithmetic
  - Parameter Passing
  - Structures
  - Local vs Global Variables
  - Dynamic Memory Management



- Next Week: File I/O in C

- Read Ch.5 & 6 from Kernighan & Ritchie
- HW-1 will be out on Thursday, Sep 11th and due Sep 18th.

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).