COMSW 1003-1

Introduction to Computer Programming in C

Lecture 25

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html
Review
Variables and Types
Variables and types

• **Variables** are placeholders for values

• They can have any name we choose

• In C, variables are divided into **types**, according to how they are **represented in memory** (always represented in binary)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
</tr>
</tbody>
</table>
Integer division

vs

Floating point division

```c
int i = 3, j;
float x = 3, y;

j = i/2;    // j = 1
i = x/2;    // j = 1
y = x/2;    // y = 1.5
y = i/2;    // y = 1
```

Implicit cast!
Binary Logic

• 1 = true, 0 = false

• Decimal to binary conversion

\[ 6_{10} = 110_2 \]

Most significant bit   Least significant bit

• Binary to decimal conversion

\[ 11001_2 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 25 \]

• AND
  \[ v = x \& y \]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
  \end{array}
  \]

• OR
  \[ v = x \mid y \]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
  \end{array}
  \]

• NOT
  \[ v = \neg x \]
  \[
  \begin{array}{cc}
  x & v \\
  0 & 1 \\
  1 & 0 \\
  \end{array}
  \]

• EXOR
  \[ v = x \oplus y \]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
  \end{array}
  \]
Arrays and Strings
Arrays

• “A set of consecutive memory locations used to store data” [PCP, Ch 5]

\[
\text{int } \text{m}[4]; // a vector containing 4 integers
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n+ 4</td>
<td>n + 8</td>
<td>n + 12</td>
</tr>
</tbody>
</table>

• Indexing starts at 0!

\[
\text{m}[0] = 3;
\]

\[
\text{m}[2] = 7;
\]

• Be careful not to access uninitialized elements!

\[
\text{int } x = \text{m}[7];
\]

gcc will not complain about this, but the value of \( x \) is going to be random!
Arrays

• Multidimensional arrays

```c
int p[4][3]; // a matrix containing 4x3 = 12 integers
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p[0][0]</td>
<td>p[0][1]</td>
<td>p[0][2]</td>
</tr>
<tr>
<td>p[1][0]</td>
<td>p[1][1]</td>
<td>p[1][2]</td>
</tr>
<tr>
<td>p[3][0]</td>
<td>p[3][1]</td>
<td>p[3][2]</td>
</tr>
</tbody>
</table>

• Indexing starts at 0!

```c
p[0][0] = 1;
p[3][1] = 7;
```

• Initialize arrays

```c
int a[4] = { 3, 6, 7, 89};
int b[2][4] = { {19, 2, 6, 99}, {55, 5, 555, 0} };
int c[] = { 3, 6, 77};
```

This automatically allocates memory for an array of 3 integers
Strings

- Strings are arrays of `char`
- ‘\0’ is a special character that indicates the end of a string

```c
char s[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

We need 6 characters because there is ‘\0’

```
<table>
<thead>
<tr>
<th>'H'</th>
<th>'e'</th>
<th>'l'</th>
<th>'l'</th>
<th>'o'</th>
<th>'\0'</th>
</tr>
</thead>
</table>
```

```c
char s[10] = "Hello";
```

```
<table>
<thead>
<tr>
<th>'H'</th>
<th>'e'</th>
<th>'l'</th>
<th>'l'</th>
<th>'o'</th>
<th>'\0'</th>
</tr>
</thead>
</table>
```

- Difference between string and char

```c
char c = 'a';
```

```
| 'a' |
```

```c
char s[2] = "a";
```

```
| 'a' | '\0' |
```

Strings functions – recap (<string.h>)

```c
char s1[] = "Hello"; char s2[] = "He";
int x; char c;

• `strcmp(s1, s2)`
  
x = strcmp(s1, s2) // x != 0

• `strcpy(s1, s2)`
  
strcpy(s2, s1); // s2 = "Hello"

• `strcat(s1, s2)`
  
strcat(s2, s1); // s2 = "HelloHello"

• `strlen(s)`
  
x = strlen(s1); // x = 5;

• `sizeof(s)`
  
x = sizeof(s1); // x = 6;

• `fgets(s, sizeof(s1), stdin)`
  
fgets(s1, sizeof(s1), stdin);
  User enters “7R”

• `sscanf(s, "%d", &var)`
  
sscanf(s1, "%d%c", &x, &c);
  // x = 7; c = ‘R’;
```
Pointers
## Pointers vs. Arrays

<table>
<thead>
<tr>
<th>Arrays</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1D array of 5 int</strong></td>
<td><strong>int x[5];</strong></td>
</tr>
<tr>
<td><strong>2D array of 6 int</strong></td>
<td><strong>int y[2][3];</strong></td>
</tr>
<tr>
<td><strong>2D array of 4 int</strong></td>
<td><em><em>int</em> z[2] = {{1, 2}, {2, 1}};</em>*</td>
</tr>
<tr>
<td><strong>1D array of 5 char</strong></td>
<td><strong>char c[] = “mike”;</strong></td>
</tr>
</tbody>
</table>

- Space has been allocated in memory only for the pointers variables, **NOT** for the arrays they will point to.
- The DIMENSIONS of the arrays are **UNKNOWN**.
Multidimensional Arrays

2x3 matrix of double

double  M0[2][3];
double  *M1[2] = M0;
double  **M = M0;

double **

double *

double
Dynamic Memory Allocation
Dynamic Memory Allocation

Example: create an array of 10 integers

```c
int myArr[10];
```

- Malloc()

  Example
  ```c
  int *myArr = (int *) malloc( 10 * sizeof(int) );
  ```

- Calloc()

  Example
  ```c
  int *myArr = (int *) calloc( 10 , sizeof(int) );
  ```
Dynamic Memory Allocation

Functions related to DMA are in the library `stdlib.h`

```c
void *realloc(void *ptr, size_t size)
```

Changes the size of the allocated memory block pointed by `ptr` to `size`

Returns a pointer to the allocated memory on success, or NULL on failure

```c
void free(void *ptr)
```

De-allocates (frees) the space in memory pointed by `ptr`
Advanced Types - Struct

We can initialize a struct variable at declaration time, just like with arrays

```c
struct student {
    char name[100];
    int age;
    double grade;
};

struct student st1 = {“mike”, 22, 77.4};
```

The initialization fields must be consistent with the fields types!

```c
struct student *ptr = &st1;
ptr->age = 33;
```
Advanced types - Typedef

typedef is used to define a new type

```c
struct student {
    char name[100];
    int age;
    double grade;
};

struct student st1, st2;

st1.age = 3;
st2.age = st1.age - 10;
```

```c
typedef struct student stud;
stud st1, st2;

st1.age = 3;
st2.age = st1.age - 10;
```
Advanced Types - Union

• Similar to struct, but all fields share same memory
• Same location can be given many different field names

```c
struct value{
    int iVal;
    float fVal;
};
```

```c
union value{
    int iVal;
    float fVal;
};
```

We can use the fields of the union only one at a time!
Advanced Types - Enum

• Designed for variables containing only a limited set of values

• Defines a set of **named integer constants**, starting from 0

```c
enum name{ item1, item2, ... , itemN};
```

```c
dwarf { BASHFUL, DOC, DOPEY, GRUMPY, HAPPY, SLEEPY, SNEEZY};

enum dwarf myDwarf = SLEEPY;

myDwarf = 1 + HAPPY;    // myDwarf = SLEEPY = 5;

int x = GRUMPY + 1;    // x = 4;

printf("dwarf %d\n",BASHFUL);  // 'dwarf 0'
```
**Advanced Types - const**

**Regular variables**

`const` defines a variable whose value cannot be changed

```c
double r;
const double PI;
```

r++; ✅

PI++; ❌

r = PI; ✅

**Pointers**

When we declare a pointer to be a constant, it means that the value at the address in memory it points cannot be modified

```c
int x = 7, y = 3;
const int *ptr = &x;
```

*ptr = 11; ❌

x = 8; ✅

ptr = &y; ✅

*ptr = 9; ❌
Advanced Types - void

Regular variables

`void` defines emptiness. It could be used for a function that does not return anything and does not take any input argument.

```c
void printArrow(void)
{
    printf("---\n");
    return;
}
```

```c
int main()
{
    printArrow();
    return 0;
}
```

Pointers

`void *` means a pointer of ANY type. This allows the programmer to specify the type of pointer to use at invocation time.

```c
void *pointElement(void *A, int i){
    return( A+i );
}
```

```c
int main()
{
    int M[3] = {1, 2, 3};
    int *M2 = (int *) pointElement( (int *) M, sizeof(int) * 2);
    return(0);
}
```
Functions
Functions

Function declaration

Specifies:
• return type
• number and type of the arguments

After declaration, function can be invoked in code

```c
int mean(float, float);
```

Function Definition

Actual implementation

Declaration can be embedded in definition

```c
int mean(float n1, float n2){
    return( (n1+n2)/2 );
}
```
Functions

Passing arguments by value/reference

- **Pass by value**: the value of the variable used at invocation time is copied into a local variable inside the function.

- **Pass by reference**: a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function.
Functions - Recursion

• What if a function calls itself? Recursion

/* Fibonacci value of a number */
int fib ( int num ) {

    switch(num) {
        case 0:
            return(0);

        case 1:
            return(1);

        default: /* Including recursive calls */
            return(fib(num - 1) + fib(num - 2));
    }
}
Pointers to functions

```c
int (*f_ptr)(); // pointer to function that returns an int

Parentheses are important! Without parentheses, f_ptr looks like it returns a pointer to an int.

int (*f_ptr)(int, int);
int greater_than(int a, int b);
f_ptr = greater_than;
```

```c
int *ptr;
int x[2];
ptr = x;
```
Input / Output
Reading strings

Use functions from library `stdio.h`

- `fgets()` : get string from standard input (command line)

```
fgets( name , sizeof(name) , stdin);
```

```c
char s1[100];
fgets( s1, sizeof(s1), stdin);
```

Reads a maximum of `sizeof(name)` characters of a string from stdin and saves them into string `name`

NOTE: `fgets()` reads the newline character `\n`, so we should substitute it with `\0`;

```
name[strlen(name)-1] = '\0';
```

```
`H` `e` `l` `l` `o` `\n`
```
```
`H` `e` `l` `l` `o` `\0`
```

- `sizeof()` : returns the size (number of bytes occupied in memory) of a variable (for strings it counts the number of elements, including `\0`)

```c
char s1[100];
fgets( s1, sizeof(s1), stdin);
```
Reading numbers

• First, read a string
• Then, convert string to number
• `sscanf()` : get string from standard input (command line)

```c
sscanf( string, "format", &var1, ..., &varN);
```

```c
char s1[100];
int x, y;
printf("Please enter two numbers separated by a space\n")
fgets( s1, sizeof(s1), stdin);
sscanf( s1, "%d %d", &x, &y );

// x = 3; y = 18;
```

User enters:  3 18
Files I/O

- Files have a special type of variable associated with them: `FILE *`

- In order to read/write to a file, we must first OPEN it

- After we are done, we must CLOSE the file

```c
FILE *fVar;

fVar = fopen( fileName, mode);
/* read, write or append */
fclose(fVar);
```
## Summary of Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf()</td>
<td>formatted text + args</td>
<td>file</td>
</tr>
<tr>
<td>printf()</td>
<td>formatted text + args</td>
<td>stdout</td>
</tr>
<tr>
<td>sprintf()</td>
<td>formatted text + args</td>
<td>string</td>
</tr>
<tr>
<td>fputc(), fgets()</td>
<td>char,string</td>
<td>file</td>
</tr>
<tr>
<td>fscanf()</td>
<td>file</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>scanf()</td>
<td>stdin</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>sscanf()</td>
<td>string</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>fgetc(), fgets()</td>
<td>file</td>
<td>(char) int, string</td>
</tr>
</tbody>
</table>
Preprocessor
C Preprocessor

Preprocessor is a facility to handle:

- **Header files**
  - `#include <nameofHeader.h>` for standard C libraries
  - `#include "nameofHeader.h"` for user defined headers

- **Macros**
  - Object like macros: `#define SIZE 10`
  - Function like macros: `#define SQR(x) ((x) * (x))`

- **Conditional compilation**
  - `#ifdef var`  
  - `#ifndef var`  
  - `#else`  
  - `#endif`  
  - `#undef var`
Compiling
gcc

gcc -option -o executable source1 source2...

Options:
- Wall : show warnings
- E : print source code after preprocessor
- g : compile with debugging information
- c : compile sources without main() function.
  Used to compile modules
- lm : link <math.h> library
Main Function Arguments
# Makefile for UNIX system
# using a GNU C compiler (gcc)

CC=gcc
CFLAGS=-Wall

mainProgram : mainProgram.c  calculator.o
  $(CC) $(CFLAGS) -o mainProgram mainProgram.c  calculator.o

calculator.o : calculator.c  calculator.h

clean:
  rm -f calculator.o.o mainProgram
Command Line Arguments

- Input parameters of the function main()
- `argc, argv`

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

**argc**
- Integer
- Specifies the number of arguments on the command line (including the program name)

**argv**
- Array of strings
- Contains the actual arguments on the command line
- First element is the name of the program
Data Structures
Linked Lists

- A chain of elements
- First element is called HEAD
- Each element (called NODE) points to the next
- The last node does not point to anything
- Like a treasure hunt with clues leading one to another
**Linked Lists**

- Structure declaration for a node of a linked list

```c
struct ll_node {
    int value;
    struct ll_node *next;
};

typedef struct ll_node node;
```

Diagram:
- Node 1
  - `value`
  - `next` connected to Node 2
- Node 2
  - `value`
  - `next` connected to Node 3
- Node 3
  - `value`
  - `next` connected to Node 4
- Node 4
  - `value`
  - `next` connected to Node 5
- Node 5
  - `value`
  - `next` connected to NULL

```c
linkedList.c
```
Doubly linked lists

- Pointer to next AND previous node
- Faster backtracking

```c
struct dll_node {
  int value;
  struct dll_node *prev;
  struct dll_node *next;
};
```
Binary Trees

- Left has a value < current node
- Right has a value > current node

```c
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```
Trees Definitions

- **Root**: node with no parents. **Leaf**: node with no children
- **Depth (of a node)**: path from root to node
- **Level**: set of nodes with same depth
- **Height or depth (of a tree)**: maximum depth
- **Size (of a tree)**: total number of nodes
- **Balanced binary tree**: depth of all the leaves differs by at most 1.

Height of tree = 3
Size = n = 15
Complexity analysis and big-O notation
Measuring Algorithms

• In Computer Science, we are interested in finding a function that defines the quantity of some resource consumed by a particular algorithm

• This function is often referred to as a complexity of the algorithm

• The resources we usually investigate are
  – running time
  – memory requirements
Big–O : Relationship among common cases

O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(a^n)

Example: big-O when a function is the sum of several statements

```c
int i=0;
for(i=0 ; i < n; i++){
    for(j=0 ; j < n; j++){
        if( (i!=j) && arr[i] == arr[j])
            increment i
            increment j
            check i!=j
            check arr[i]==arr[j]
            dup[i][j] = 1;
    }
}
```

RT = O(4n^2+n) = O(n^2)

Longest operation dominates (worst case)
Sorting
Sorting

- Given a set of $N$ elements, put them in order according to some criteria

- Compare pairs of elements

- Many algorithms, some of the most famous are:
  - Bubble sort \( \text{Complexity} = O(n^2) \)
  - Selection sort \( \text{Complexity} = O(n^2) \)
  - Merge sort \( \text{Complexity} = O(n \log(n)) \)
  - Counting sort \( \text{Complexity} = O(k+n) \)
C++
C++

• Main factors differentiating C++ from C:
  – Slightly different syntax, contains type `bool`
  – Functions overloading
  – Object oriented
Modular Programming

- Multiple files
- Functions of similar logical goal grouped into *modules*
- Different data manipulated inside functions in modules
Object Oriented Programming

• Based on **objects** interacting with each other
• Objects exchange **messages**, but maintain their state and data
• Usually associated also with modular programming
Object oriented programming

• Classes
• Objects
• Inheritance
• Polymorphism