EC 413
Computer Organization

C/C++ Language Review

Prof. Michel A. Kinsky
Hardware Mechanics for Bridging

- Our physical world is analog and computing is interacting with the physical world:
  - So we need to convert physical signals to digital then back to analog to communicate with the real world
Software Mechanics for Bridging

- The Art of Abstraction

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>Programming Language</th>
<th>Operating System/Virtual Machine</th>
<th>Instruction Set Architecture (ISA)</th>
<th>Microarchitecture</th>
<th>Register-Transfer Level (RTL)</th>
<th>Circuits</th>
<th>Devices</th>
<th>Physics</th>
</tr>
</thead>
</table>
Another View of the Abstraction

Applications & Algorithms
  Programming Language
  Compiler
  Operating System
  Firmware

ISA
  Processor
  Memory organization
  I/O system

Datapath & Control

Digital Design

Circuit Design

Layout
Pre-von Neumann Architecture

- Memoryless architecture
von Neumann Architecture

- Stored program architecture

Diagram:

- Central Processing Unit
  - Arithmetic/Logic Unit
  - Control Unit
- Memory
- Inputs
- Outputs
von Neumann Architecture

- Stored program architecture

![Diagram of von Neumann Architecture](image)
Computing Process

Computation ideas & Solutions

Central Processing Unit

Arithmetic/Logic Unit

Control Unit

Memory

The Computer

Inputs

Outputs

Reduction Process

Presentation Process

111111010000000100000 00100010011 000000110000000100000 10000010011 ...

111111010000000100000 00100010011 000000110000000100000 10000010011 ...

...
Bridging/Compiling Process

- **High-Level Language**

  - **C/C++/Java program**
  - **Compiler**
  - **Assembly Code**
  - **Assembler**
  - **Object Code**
  - **Linker**
  - **Library Routines**
  - **Executable**
  - **Loader**
  - **Memory**

Human Readable

Machine Code
The Overall Organization!

- The modern computer system has three major functional hardware units: CPU (Processing Engine), Main Memory (Storage) and Input/Output (I/O) Units.
Application Compiling Process

- C Language

Human Readable

- C program
  - compiler
  - assembly code
  - assembler

Machine Code

- object code
  - library routines
  - linker
  - executable
  - loader
  - memory
Programming Languages

- There are many programming languages available: Pascal, C, C++, Java, Ada, Perl and Python
- All of these languages share core concepts
- By focusing on these concepts, you are better able to learn any programming language
Programming Languages

- There are many programming languages available: Pascal, C, C++, Java, Ada, Perl and Python
- All of these languages share core concepts
- Hence, by learning C, you are poised to learn other languages, such as Java or Python
  - In this class, we will learn core programming concepts through the powerful C language
  - Why C? It runs nearly as fast as assembly language code
Programming Process

Write Program

Compile Program

Run Program

Debug Program
Introduction to C

- Developed in 1972 by Dennis Ritchie at Bell Labs
- It is imperative programming language
- It provides:
  - Efficiency, high performance and high quality software
  - Flexibility and power
  - Many high-level and low-level operations
Introduction to C

- Developed in 1972 by Dennis Ritchie at Bell Labs
- It is imperative programming language
- It provides:
  - Stability and small size code
  - Provide functionality through rich set of function libraries
  - Gateway for other professional languages like C++ and Java
Introduction to C

- It is used:
  - System software, Compilers, Editors, embedded systems, application programs
  - Data compression, graphics and computational geometry, utility programs
  - Databases, operating systems, device drivers, system level routines

- The real world still runs on C
  - Most of legacy code in use are in C
  - Many other programming languages are based on C
Introduction to C


Original comic is available here: http://xkcd.com/519/
Basic C variable types

- There are five basic data types in C
  - Char: ‘a’
    - A single byte capable of holding one character in the local character set
  - Int: 3
    - An integer of unspecified size
  - Float: 3.14
    - Single-precision floating point
  - Double: 3.1415926
    - Double-precision floating point
  - Void: Valueless special purpose type
### Basic C variable types

<table>
<thead>
<tr>
<th>Type (32 bit)</th>
<th>Smallest Value</th>
<th>Largest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>-32,768(-2^{15})</td>
<td>32,767(2^{15}-1)</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65,535(2^{16}-1)</td>
</tr>
<tr>
<td>Int</td>
<td>-2,147,483,648(-2^{31})</td>
<td>2,147,483,648(2^{31}-1)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long int</td>
<td>-2,147,483,648(-2^{31})</td>
<td>2,147,483,648(2^{31}-1)</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>
Variable assignment

- In C variables must be declared
- They are given values through assignments
- Assignment is done with the '==' operator

**Declarations**

```c
int number_of_students;
float average_gpa;
```

**Assignments**

```c
number_of_students = 12;
average_gpa = 3.9;
```
Variable assignment

- In C variables must be declared
- They are given values through assignments
- Assignment is done with the '==' operator

Declarations

```c
int number_of_students;
float average_gpa;
```

Assignments

```c
number_of_students = 12;
average_gpa = 3.9;
```
A Simple C Program

```c
#include <stdio.h> /* Header files */

int main(void) {
    printf ("Hello World!\n" );
    return 0;
}
```
C Program compilation

Compile: gcc –o myhello hello.c

Run: ./myhello
C Program Analysis

- `#include <stdio.h> /* Header files */`
  - It is a preprocessor directive
  - It tells computer to load contents of the file
  - It allows standard input/output operations

- Comments are used to describe program
  - Text surrounded by /* and */ is ignored by computer
  - Lines starting with // are also ignored
C Program Analysis

- `int main (void)`
  - C programs contain one or more functions, exactly one of which must be main
  - Parenthesis used to indicate a function
  - `int` means that main "returns" an integer value

- Braces ({ and }) indicate a block
  - Bodies of all functions must be contained in braces

- `printf ("Hello World!\n")`
  - `printf` and `scanf` functions
C Program Analysis

- `printf`
  - Sends output to standard out
  - General form
    - `printf(format descriptor, var1, var2, ...);`
    - `printf(“%s
”, “Hello world”);`
      - Translation: Print hello world as a string followed by a newline character
    - `printf(“%d\t%f\n”, j, k);`
      - Translation: Print the value of the variable j as an integer followed by a tab followed by the value of floating point variable k followed by a new line
C Program Analysis

- scanf
  - Gets inputs from user
  - General form
    - scanf(format descriptor, &var1, &var2, …);
    - scanf("%f", &i);
      - Translation: Get floating point input i from user
    - scanf("%d %f\n", &j, &k);
      - Translation: Get the value of the variable j as an integer followed by the value of floating point variable k from user
      - Blocks program until user enters input
C Program Analysis

- Some special characters are not visible directly in the output stream
- These begin with an escape character (\);
  - \n  newline
  - \t  horizontal tab
  - \a  alert bell
  - \v  vertical tab
C Program Operations

- Arithmetic operators
  - + "plus"
  - - "minus"
  - * "times"
  - / "divided by"

```c
#include <stdio.h> /* Header files */
int number1, number2, number3;

int main(void) {
    scanf("Enter number1: %d", &number1);
    scanf("Enter number2: %d", &number2);
    number3 = number1 + number2;
    printf ("\n Number1 + number2 = %d\n", number3 );

    number3 = number1 - number2;
    printf ("\n Number1 - number2 = %d\n", number3 );

    number3 = number1 * number2;
    printf ("\n Number1 * number2 = %d\n", number3 );

    number3 = number1 / number2;
    printf ("\n Number1 / number2 = %d\n", number3 );
    return 0;
}
```
C Program Comparators

- Relational operators:
  - `==` "is equal to"
  - `!=` "is not equal to"
  - `>` "greater than"
  - `<` "less than"
  - `>=` "greater than or equal to"
  - `<=` "less than or equal to"
C Program Logical Operators

- There are two logical operators in C
- `||` "logical or"
  - An expression formed with `||` evaluates to true if any one of its components is true
- `&&` "logical and"
  - An expression formed with `&&` evaluates to true if all of its components are true
Advance Data types

- In C
  - Arrays (a list of data (all of the Same Data Type!))
    - `int grades[] = {94, 78, 88, 90, 93, 87, 59};`
  - Structures (a collection of named data referring to a single entity)
    ```c
    struct Student {
        char Name [50];
        int id;
        float GPA;
        char major [25];
    };
    ```
Advance Data types

- Pointers in C
  - Pointers are memory addresses
  - Every variable has a memory address
  - Symbol & means “take the address of” e.g., &x
  - Symbol * means “take the value of” e.g., *p
  - Symbol * is also used to denote a pointer type e.g., int *q;
Advance Data types

- Pointers in C
  - Declaration of integer pointers and an integer number
    - int * pointer1, * pointer2;
    - int number1;
  - Setting pointer1 equal to the address of number1
    - pointer1 = &number1;
  - Setting pointer2 equal to pointer1
    - pointer2 = pointer1;
Abstraction Data Types in C++

- Data structures
- A set of operations
- The interface is the only access mechanism to the data structures.
Abstraction Data Type
Object-Oriented Programming

- Example of an abstract data structure

```cpp
#include <iostream>
using namespace std;

class Robot_Rectangle {
    private:
        int width;
        int length;
```
Object-Oriented Programming

- Example of an abstract data structure

```cpp
#include <iostream>
using namespace std;

class Robot_Rectangle {
    private:
        int width;
        int length;
};
```
Object-Oriented Programming

- Example of an abstract data structure

```cpp
#include <iostream>
using namespace std;

class Robot_Rectangle
{
    private:
        int width;
        int length;

    Make them less visible from the outside
```
Object-Oriented Programming

- Example of an abstract data structure

```cpp
#include <iostream>
using namespace std;

class Robot_Rectangle
{
    private:
        int width;
        int length;

    public:
        Robot_Rectangle (void) {
            width = 0; length = 0;
        }
        void set(int w, int l) {
            width = w; length = l;
        }
        int get_width(void) {
            return width;
        }
};
```
Object-Oriented Programming

- Example of an abstract data structure

```c
int get_length(void) {
    return length;
}

int area(void) {
    int calculated_area = 0;
    calculated_area = width * length;
    return calculated_area;
}
```
Conditional execution

- Conditional constructs provide the ability to control whether a statement list is executed

- If statement
  - If
  - if (Expression)
  - Action
Conditional execution

- Conditional constructs provide the ability to control whether a statement list is executed

- If statement
  - If
    - if (Expression)
  - Action

```c
int num1;
int num2;
printf("Enter two integers: ");
scanf("%d %d", &num1, &num2);
if (num1 > num2) {
    int remember_num1 = num1;
    num1 = num2;
    num2 = remember_num1;
}
printf("Inputs in sorted order: %d \t %d", num1, num2);
```
Conditional execution

- Conditional constructs provide the ability to control whether a statement list is executed.
- If statement
  - if-else
    - if (Expression)
      - Action1
    - else
      - Action2

Expression

true

false

Action1

Action2
Conditional execution

- Conditional constructs provide the ability to control whether a statement list is executed

- If statement
  - if-else
    - if (Expression)
      - Action1
    - else
      - Action2

```c
int num1;
int num2;
printf("Enter two integers: ");
scanf("%d %d", &num1, &num2);

int max;
if (num1 > num2) {
    max = num1;
} else {
    Max = num2
}
printf("Maximum number is: %d", max);
```
Conditional execution

- For loop
- While loop
- Do while loop
Iteration

- For loops
  - The general format when using for loops is
    - for ( initialization; loopContinuationTest; increment )

```
for ( initialization; loopContinuationTest; increment )
Statement
```

![Flowchart for for loops]

- Initialize variable
- Test the variable
  - true: statement
  - false: increment variable
Iteration

- For loops
  - The general format when using for loops is
    - for ( initialization; loopContinuationTest; increment )
      Statement

```c
int counter;
for(counter = 1; counter <= 10; counter++)
  printf("Current counter value %d\n", counter);
```
Iteration

- While loop
  - while loop repeated until condition becomes false
    - Initialization;
      while (loopContinuationTest){
        statement
        increment;
      }

Iteration

- While loop
  - while loop repeated until condition becomes false
    - Initialization;
      while (loopContinuationTest){
        statement
        increment;
      }

```c
int counter = 1;    //initialization
while (counter <= 10){ //repetition
  printf("Current counter value \%d\n", counter);
  counter++;        //increment
}
```
Iteration

- **Do while**
  - do/while repetition structure is similar to the while structure
  - Condition for repetition tested after the body of the loop is executed
    - do {
      - statement
    } while ( condition );
Functions

- A Definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program or/and receives values(s) from the calling program.

- There are two types of function:
  - Predefined functions
    - Standard libraries like stdio.h, math.h
  - User-defined functions
    - Programmer created functions for specialized tasks
    - e.g., `int fibonacci(int n)`
Functions

- Characteristics of a function
- Function header: It has a return type, a unique name, and list of parameters with their types

```plaintext
Return_type function_name (type1 parameter1, type2 parameter2 ...) {
    variable declaration(s)
    statement(s)
}
```

Examples

- `void function1 (int x, float y, char z)`
- `float function2 (float x, double y)`
- `int function3 (long size)`
- `void function4 (void)`
Functions

- The rules govern the use of variables in functions:
  - To use a variable in a function, it must be declared either in the function header or the function body.
  - For a function to obtain a value from the calling program (caller), the value must be passed as an argument (the actual value) unless it is a global value.

```c
/* declare and define */
int exponential (int x)
{
    int result = 1;
    int i;
    for (i = 0, i < x, i++)
        result *= 2;
    return result;
}

int main()
{
    /* function call */
    int y;
    y = exponential(3);
}
```
Functions

- The rules govern the use of variables in functions:
  - For a calling program (caller) to obtain a value from function, the value must be explicitly returned from the called function (callee) unless it is updated through a global variable.

```c
/* declare and define */
int exponential (int x)
{
    int result = 1;
    int i;
    for (i = 0, i<x, i++)
        result *= 2;
    return result;
}

int main()
{
    /* function call */
    int y;
    y = exponential(3);
}
```
Recursion

- Often it is difficult to express a problem explicitly
  - For example the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  - It is difficult to follow the logic of this sequence

- However, a recursive definition consisting of expressing higher terms in the sequence in terms of lower terms
  - Recursive definition for \( \{f_n\} \):
    - Initialization: \( f_0 = 0, f_1 = 1 \)
    - Recursion: \( n = f_{n-1} + f_{n-2} \) for \( n > 1 \)
Recursion

- Sometimes the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type
- The technique ends up with functions that call themselves (recursive functions)
Logic of recursive functions

- Recursive definition and inductive proofs are complement each other
- A recursive function has two parts
  - Initialization – analogous to induction base cases
  - Recursion – analogous to induction step
    - Recursive definition for \( \{f_n\} \):
    - Initialization: \( f_0 = 0, f_1 = 1 \)
    - Recursion: \( n = f_{n-1} + f_{n-2} \) for \( n > 1 \)
Recursion

- Factorial function
  - Iterative implementation

```c
int Factorial(int n)
{
    int count;
    int fact = 1;
    for(count = 2; count <= n; count++)
        fact = fact * count;
    return fact;
}
```
Recursion

- Factorial function
  - Recursive implementation

```c
int Factorial(int n)
{
    if (n==0) // base case
        return 1;
    else
        return n * Factorial(n-1);
}
```
Next Class

- Stack and Heap structures