Computer Science I

Felix G. Hamza-Lup

fhamza@cs.ucf.edu

School of CS

University of Central Florida

Basic Information

Who and Where?

- Instructor:

  Felix G. Hamza-Lup

- Meeting Times and Place:

  Tu,Tr 9:00 to 10:15 AM in BA 119

- Office Hours:

  Tu: *10:30 AM to 12:00 PM, CSB 204*

  Tr: *10:30 AM to 12:00 PM, CSB 204*

- Text:

  “Data Structures, Algorithms & Software Principles in C”

  by Thomas A. Standish

  ISBN: 0-201-59118-91995

- Web Page:

  http://www.cs.ucf.edu/courses/cop3502/spr2005/section1/

Other C Learning Resources

- *Programming Abstractions in C*, Eric S. Roberts, Addison-Wesley.
- *C by Dissection: The Essentials of C Programming* , (4th Edition), Al Kelley, Ira Pohl, 2001.
- *Problem Solving and Program Design in C* , (4th Edition) J. R. Hanly and E. B. Koffman, Addison-Wesley, 2001.

Course Objectives

- **Provide an introduction to the field of computing:** The central concept that underlies computer science is the algorithm and thus algorithms are made the central object of study.

- **Provide Conceptual Content and Software Skills:** The lecture component focuses on conceptual tools for constructing and analyzing algorithms, while the lab component focuses on programming applications.

- **Introduction to elementary data structures:** Arranging data in arrays, linked lists, stacks and queues, binary trees. Study of searching and sorting techniques. Study of recursion.

Prerequisites

- Knowledge of core syntax of C as presented in COP 3223.
- Programming experience as would be required to successfully complete assignments in COP 3502.
- I will provide a review of C programming basics.

Class Structure

- **Lecture Classes:** The sequence of lecture topics given above is tentative and may be altered without notice. Some of the topics covered will go beyond the depth of coverage provided in the textbook. Thus each class attendance is important.

- **Recitation Sections :** Students must enroll in one of the recitation sections of COP 3502 . In the recitation sections, the TA will either present material related to recent lectures , or may go beyond that covered in the lecture section, or will provide assistance in solving problems.

Evaluation

- Programming assignments 30%
- 2 Tests 20% each.
- Final 30%

- Programming assignments:
  - Adhere to Deadlines
  - Work independently (be fair)
  - TA support (read and try to understand before you ask, you will need to show proof of trials)

Some Guiding Principles

Programming in the Small

- Code is developed by one programmer
  - Maybe a small close-knit group
- One person can understand the entire system
  - To them the system is self-documenting
  - The creator is the maintainer
- Designed to solve a particular (singular) problem
- System does not have a long life cycle
- The biggest problem is getting it done on time

Programming in the Large

- Developed by a large team of programmers with a lot of input from management
- Nobody really knows what is really going on inside all parts of the system
  - Everybody has their own little piece
- The system is designed to solve a “systems” level problem
- Ideally, the system will be around for a long time
- Communication (developers to customers & developers to developers) is the biggest problem

Desirable System Qualities

- Useful – meets needs
- Timely – shipped when expected
- Reliable – within tolerable limits
- Easy to use – by intended audience
- Efficient – including good algorithms
- *Maintainable* – flexible, simple, readable
  - Over 40% of maintenance is change in user requirements
- *Reusable* – by you and others
  - Modularity

Maintainability Factors

- Flexibility
  - Easily changeable; limited impact from changes
- Simplicity
  - All complex systems have errors; dividing a system into simpler components reduces or at least manages complexity
- Readability
  - Clarity is a great aid to maintainability; simple, consistent style aids understandability

Software Phases

- Conceptualization
  - Requirements – informal description of what is needed; often includes prototype to help resolve issues
  - Specifications – formal requirements; can be legal document; often includes walkthrough with client, moderated by QA staff
- Analysis and Modeling – identifying the software components
- Design – designing the components and their relationships
- Implementation – making it real
- Integration – putting the parts together with each other and existing components
- Maintenance – fixing, improving, adapting

Focus of Course

- We will focus on
  - *Data Types & Operations*
  - *Implementation*
  - *Algorithms*
  - *Analysis*

Basic Vocabulary

- Abstraction
  - Selective examination of certain aspects of a problem
  - The goal is to isolate those aspects that are important for some purpose and suppress those that are unimportant
  - Many different abstractions of the same thing are possible, depending on the purposes for which they are made
- Model
  - An abstraction used to understand or reason about some real object
  - Omits nonessential details and is at granularity appropriate for its purposes
- Methodology (in software engineering)
  - A process for the organized production of software using a collection of predefined techniques and notional conventions

Modeling

- Designing a system is about modeling the problem space (real world).

- Issues include
  - What are the components of the real world?
  - How do they behave?
  - How do they interact?
  - What can we ignore?

Algorithms

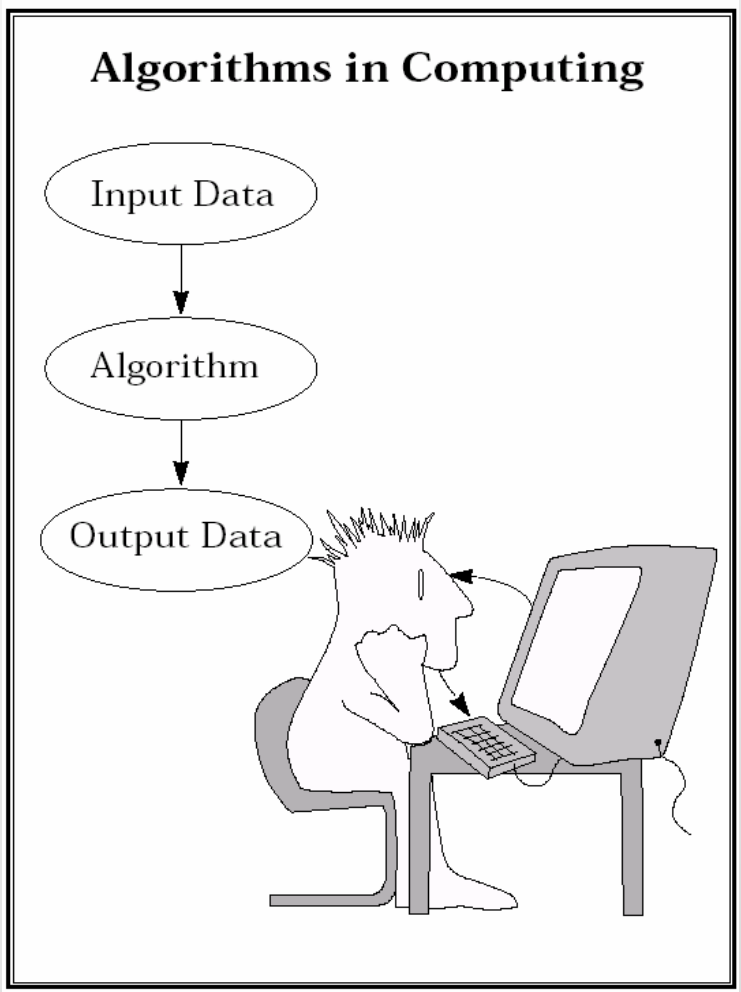
Overview of Algorithms

- Computers are devices that do only one kind of thing:
  - They carry out *algorithms* to process information.

- To computer scientists, the algorithm is the central unifying concept of computing, the mode of thought that is the core of the computing perspective.

Algorithm

- A set of logical steps to accomplish a task.
- A “recipe of action”.
- A way of describing ***behavior***.



Algorithm

- To be useful, an algorithm must
  - accept input data,
  - process that data in some way and
  - output the results.
- Successful algorithms must consider all possible cases presented by acceptable data.
- You will succeed more quickly at constructing algorithms if you make it a habit to
  - think about the problem and its data, then
  - enumerate all the special cases that the algorithm must handle.

Describing Algorithms

In specifying behavior, must be:

- Precise
- Unambiguous
- Complete
- Correct

Ways to describe algorithms:

→ Natural language (English)

→ Pictures

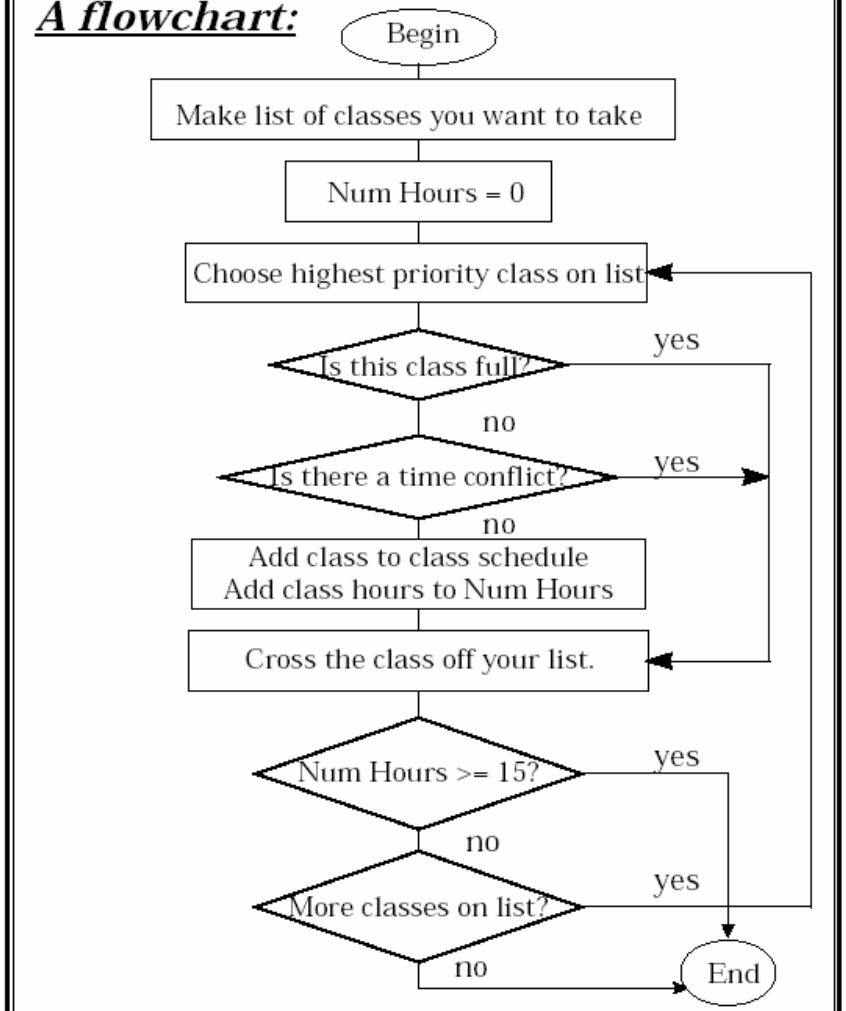
→ Pseudocode or a specific programming language.

Example

Example Algorithm: Register for Classes

1. Make a list of courses you want to register for, in order of priority
2. Start with an empty schedule. Number of hours=0.
3. Choose highest priority class on list.
4. If the chosen class is not full and its class time does not conflict with classes already scheduled, then register for the class:
   - 4a. Add the class to the schedule
   - 4b. Add the class hours to the number of hours scheduled
5. Cross that class off of your list.
6. Repeat steps 3 through 5 until the number of hours scheduled is >= 15, or until all classes have been crossed out.
7. Stop.

A flowchart:



Properties of Good Algorithms

1. Precision

- each step must be clear and unambiguous in its meaning
- the order of execution of the steps must be clear
- the number of steps must be finite
- each step must itself be finite

2. Simplicity

- each step must be simple enough that it can be easily understood
- each step should translate into only a few (or one) computer operation or instruction

3. Levels of abstraction

- the steps in the algorithm should be grouped into related modules or blocks
- you may use one module inside another module
- you may refer to other algorithms by name instead of including all of their steps in the current algorithm

Well Designed Algorithms

- Well-designed algorithms will be organized in terms of abstraction. This means that we can refer to each of the major logical steps without being distracted by the details that make up each one. The simple instructions that make up each logical step are hidden inside ***modules***. These modules allow us to function at a higher level, to hide the details of each step inside a module, and then to refer to that module by name whenever we need it.

- By hiding the details inside appropriate modules, we can understand the main ideas without being distracted. This is a key goal of using levels of abstraction:
  - Each module represents an abstraction. The name of the module describe the idea that the module implements. The instructions hidden within the module specify how that abstraction is implemented.
  - We can see what is being done (the idea) by reading the descriptive name of the module without having to pay attention to how it is being done.
  - If we want to understand how it is being done, we can look inside the module to find out.

Algorithm Components

- Data structures to hold data.
- Data manipulation instructions to change data values.
- Conditional expressions to make decisions
- Control structures to act on decisions.
- Modules to make the abstraction manageable by abstraction.

Data Structures

- Data are the representations of information used by an algorithm. This includes:
  - Input data
  - Output data
  - Any interim data generated by the algorithm for its own internal use.
- Data structures are "*containers*" for data values.
  - Variables
    - e.g. The average of student grades (real number)
    - The number of classes enrolled (integer)
  - Constants
    - e.g. the number pi
    - tax rate
- Complex data structures: special data organization tools that help us solve the problem at hand more efficiently.
  - e.g. lists, records (structures), arrays

Data Structures (cont.)

- For distinguishing and referral purposes we give each data structure a name.

- The names should be *descriptive* of the data.
  - e.g. Total amount to be paid : Balance
  - The number of hours worked: NumHrs

- The descriptive names we use for data structures are called ***identifiers***.

Data Manipulation Instructions

- Instructions allow the algorithm to
  - obtain data values from the world and store them in data structures.
  - manipulate those values via arithmetic operations, copying the contents of one data structure to another, etc.
  - output the resulting values back to the world.

Conditional Expressions

- A computer can make decisions. The ability of an algorithm to make decisions and act on them is what makes algorithms (and computers) powerful.

- All such decisions are based on conditional expressions that are either true or false.
  - e.g. age == 20
  - salary > 100,000
  - grade == ‘A’
  - salary > 100,000 && age == 20

Control Structures

- They allow an algorithm to act on the decisions it makes.

- E.g.

```
if (class is not full)
        Add the class to the schedule

while (there are still data)
        read data
        store data
```

Modules

- Algorithms can become very complex and we don’t want an algorithm to be a big sequential list of steps.
- Simply placing all components of the algorithm together will make them hard to understand, repair and extend.
- Instead *modules* are used to group logically related data and instructions. Modules raise the *level of abstraction*. Thus they allow
  - Clearer thinking
  - Faster repairs
  - Easier modifications

Exercise

- Write an algorithm to find the minimum of three numbers.
  - Problem Understanding
  - Analysis: What are the inputs and outputs?
  - Design: How to solve the problem.
  - Implementation: in C
  - Verification and Testing.

Minimum of 3 Numbers

Input: three numbers
Output: the minimum value
Algorithm1:

1. Read three numbers into a, b and c.
2. Keep a variable to hold the minimum value.
3. if (a < b) then min ← a
   else min ← b
4. if (c < min) then min ← c
5. Print min.

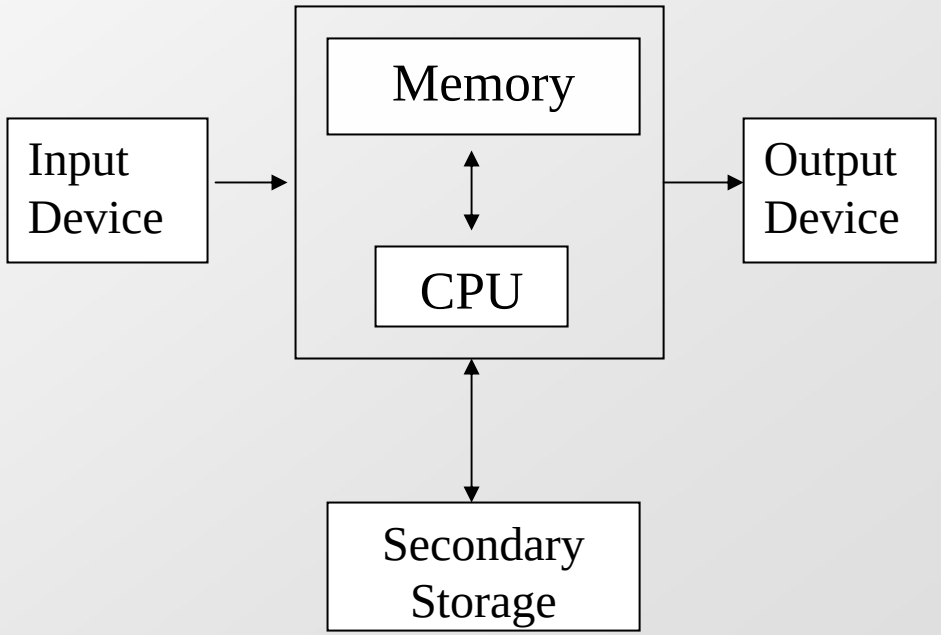
Algorithm2:

1. Read three numbers into a, b and c.
2. if (a < b) and (a < c) then print a
   else if (b < c) then print b
   else print c.

Brief Computer Architecture Review

Computer Architecture

- Computer hardware consists of five main components:
  - Memory
  - Central Processing Unit (CPU)
  - Input Devices
  - Output Devices
  - Secondary Storage



Memory

- Store information (data + instructions)
- A sequence of memory cells.
  - a byte is 8 bits
  - a bit is the smallest unit (0 or 1)
- Store, retrieve, update
  - changing the pattern of 0 and 1s in memory cells
  - copying these patterns into some internal registers
- Stored information in memory is volatile.

CPU (Central Processing Unit)

- Process and manipulate information stored in memory.
- It can be divided into two units: CU (Control Unit) and ALU (Arithmetic Logic Unit)
- CU coordinates activities of the computer and controls other devices of computer.
- ALU processes arithmetical and logical instructions.

Input and Output Devices

- Provide the interface between the user and the computer.
  - Input devices are used to enter instructions or data by the user.
  - Output devices are used to give results of computations.
- Input Devices: keyboard, mouse
- Output Devices: monitor, printer

Secondary Storage

- Computers have limited main memory and information stored in main memory is volatile. i.e. when a computer is switched off, information in its main memory disappears.
- There are additional data storage units, called *secondary storage devices*.
- Data stored in these secondary storage devices are permanent, i.e. data does not disappear when you switch off the computer.
- Some secondary storage units:
  - Floppy Disks, Hard Disks, Tape Drive, Optic Disk (CD Drive)

Data Representation

Data in Computers

- All data (numbers, alphabetic characters, images, sounds, movies, etc.) are stored as *binary numbers* in a computer.

- What are binary numbers?
  - Binary is a base 2 number system. Binary numbers only have two digits to work with, 0 and 1, and make all numbers by grouping one or more of those two digits together. E.g.:
  -

Decimal number	Binary number
5	101
12	1100
156	10011100

  - Decimal number system is base 10. (i.e. it has 10 digits to represent the numbers) e.g. The decimal number 3816 represents a specific value. We can calculate that value as:

    $3 * 1000 + 8 * 100 + 1 * 10 + 6 * 1$

    or as:

    $3 * 10^3 + 8 * 10^2 + 1*10^1 + 6*10^0$

Binary numbers

- Binary numbers are represented in a similar form. e.g. the value of the binary number 1101 can be calculated as:
  - $1*2^3 + 1* 2^2 + 0*2^1 + 0*2^0$ or as: $1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 = 13$
  - The decimal number 3816 can be converted into the binary number 111011101000.

- Notice that we need 12 binary digits to store a 4 digit decimal number. In fact, it takes 20 binary digits to store any 6 digit decimal integer and 30 binary digits to store decimal integers up to 1 billion.

- This means that we must set aside large portions of the computer's memory to have room to store numbers that we will use in calculations.

- It is also important to know that we can find the power of 2 that is equal to a given number. We take the logarithm of the number, base 2.

  e.g. $\log_2 16 = 4$ since $2^4 = 16$.

- You may find it useful to memorize the first few powers of 2 so you don't have to calculate them when you need to convert binary numbers to decimal or find a logarithm base 2.

Storing Other Types of Data

- **Character Codes**
  - Alphabetic characters are represented by specific patterns of binary numbers called character codes. There are a number of different character code standards that have been used in computers, but the most common for personal computers is ASCII (American Standard Code for Information Interchange).

- ASCII is an 8-bit code, i.e. it uses patterns that are 8 bits long to represent a single character. This allows a total of 256 different bit patterns and that is enough for all of the characters found on computer keyboards, plus some special patterns that are used for control codes .

Sample ASCII codes for characters:

Character	Decimal number	Binary representation
‘A’	65	01000001
‘B’	66	01000010
‘a’	97	01100001
‘$’	36	00100100
space	32	00100000

- **Strings**

They are simply stored as a sequence of character codes.

H e l l o !

01001000 01100101 01101100 01101100 01101111 00100001

Program Instructions

- Computers cannot read program instructions written in a high-level programming language. The instructions must be translated into a form that the computer can understand.

- How are computer programs stored?
  - Instructions must also be stored as binary numbers. The specific binary numbers that represent steps in a program are built into the computer's Central Processing Unit (CPU) by the designers. These binary numbers are referred to as the computer's ***machine code*** because it is a code that the machine understands. Machine codes are usually different for each type of CPU, thus different types of computers can't run each other's programs. We need to understand that each machine code number tells the CPU to perform a single, simple instruction like adding two numbers or comparing two numbers to see if they are equal. The program that you write in a language like C is translated into these simple binary codes and they are stored in a disk file so they can be executed later.

How are Instructions Executed ?

- The CPU contains several components that help it execute instructions. It has a number of internal memory locations, called *registers* that are used for temporary storage.
- It has special electrical circuits that can add or subtract numbers, compare the value of two numbers or perform other math-related operations.
- It has two special registers, the *Instruction Register* and the *Program Counter* (or instruction counter). The Instruction Register (IR) is used to hold the machine code instruction that is currently being executed. Each instruction must be copied into the IR so that the CPU can decode it and set switches in its circuits to perform the operation that it specifies. The Program Counter (PC or IC) is used to indicate the next instruction that will be executed. Therefore, each time an instruction is completed, the CPU knows where to get the next one.
- The basic operation of the CPU can be reduced to the following three steps, which are repeated until the program has finished or an error occurs:
  - **fetch** - copy the next instruction to be executed into the IR
  - **increment** - change the PC so that it points to the next instruction to be executed
  - **execute** - decode and perform the instruction in the IR
  - This is called the ***machine cycle*** (or the instruction cycle) and is fundamental to the operation of the computer.

The “*C*” Environment Review

Structure of a C Program

```
/* File: powertab.c
*  This program generates a table comparing values of the functions n^2 and 2^n.
*/
#include <stdio.h>

/* Constants */
#define LowerLimit 0
#define UpperLimit 12

/* Function prototypes */
int RaiseIntPower(int n, int k);

/* Main program */
int main() {
        ….
        RaiseIntPower(n, 2),
}

/* Function: RaiseIntPower :This function returns n to the kth power. */
int RaiseIntPower(int n, int k) {
   int i, result;
   result = 1;
   for (i = 0; i < k; i++){
      result *= n;
   }
   return (result);
}
```

Variables, Values and Types

- A ***variable*** can be thought of as a named box, or cell, in which one or more data values are stored and may be changed by the algorithm.
- The act of creating a variable is called ***declaring the variable***. For every variable that is declared it must be explicitly typed. In other words, each variable has an associated ***data type***.

  Examples:
  - int age;
  - int test_score;
  - float average;
  - double result = 0.0;

- An ***identifier*** is simply the algorithmic terminology for a name that we make-up to "identify" the variable. **Every** variable must be given a ***unique identifier*** so that there will be no ambiguity as to which piece of data we are referencing.

- **Rules for Variable Identifiers (in C)**
  - A sequence of letters, digits, and the special character _.
  - A letter or underscore must be the 1st character of an identifier.
  - C is *case-sensitive*: Apple and apple are two different identifiers.

Data Types

- A data type is defined by two properties:
  - a *domain*, which is a set of values that belong to that type
  - a *set of operations*, which defines the behavior of that type

  e.g.
  - Type int includes all integers (… -2, -1, 0, 1, 2, …) up to the limits established by the hardware of the computer.
  - The set of operations includes the standard arithmetic operations like addition, multiplication.
  - C includes several fundamental types that are defined as part of the language. These types are called ***atomic types***.
- Atomic types can be grouped into 3 categories: integer, floating point and character.
  - Integer Types:
    - short: 2 bytes
    - int: 4 bytes
    - long: 4 bytes
    - unsigned: 4 bytes
  - Floating-point Types:
    - float: 4 bytes
    - double: 8 bytes
    - long double: 8 bytes
    - signed/unsigned
- The range of values for each type depends on the particular computer's hardware.
- **Characters:**
    - **char:** 1 byte

Expressions

- For example:
  - (-b + sqrt(b * b – 4 * a * c)) / (2 * a)

- **Arithmetic Operators**
  - Basic operations include the four basic operations and the modulus operator:
    - **Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%)**
  - x % y produces the remainder when x is divided by y.

    e.g.

    11 % 5 = 1

    20 % 3 = 2

- **Operator Precedence:**
  - **x =** 1 + 2 * 3; (What is the value of x?)
  - x = 1 + (2*3); x is 7?

  OR
  - x= (1+2) * 3; x is 9?

- Associativity: (*left to right*)
  - 10 + 3 + 7 => 20
  - 10 – 3 + 7 => 14
  - 15 / 5 * 2 => 6

Assignment Operator

- *variable* = *expression*

- The expression can simply be a constant or a variable:
  - int x, y;
  - x = 5;
  - y = x;
  - x = 6;
- The expression can be an arithmetic expression:
  - x = y + 1;
  - y = x * 2;
  - x = x + 10;
- Embedded assignments:
  - z = (x = 6) + (y = 7);
  - n1 = n2 = n3 = 0;
- Shorthand assignments:
  - x += y;
  - z -= x;
  - y /= 10;

Boolean Operators

- C defines three classes of operators that manipulate Boolean data.

- Relational operators
  - Greater than >
  - Greater than or equal >=
  - Less than or equal <=
  - Less than <
  - Equal to ==
  - Not equal to !=

- Logical operators
  - AND : &&
  - OR : ||
  - NOT: !
  - ?: operator
    - (*condition*) ? *expr1* : *expr2*;
  - max = (x>y) ? x : y;

Input and Output Operators

- ***I/O Operators:*** allow us to communicate with the "outside world," i.e, the real world beyond the algorithm

  scanf: obtains (reads) an input value; each *read* operation does 2 things:

  1. obtains next value from "outside"
  2. stores it in the specified variable

  e.g. scanf("%d", &num);

  printf: sends out an output value

  e.g. printf("%d", num);

  printf("Hello world!");

  scanf and **printf** may take any number of arguments of the existing types.

General format:

scanf(*format control string*, *input var list*);

printf(*format control string*, *output var list*);

number of conversion characters = number of arguments.

Conversion characters for different types:

%c	char
%d	int
%f	float, double (for printf)
%lf	double (for scanf)
%Lf	long double (for scanf)

Statements

- **Simple Statement**
  - *expression*;
  - The expression can be a function call, an assignment, or a variable followed by the ++ or – operator.

- **Blocks**

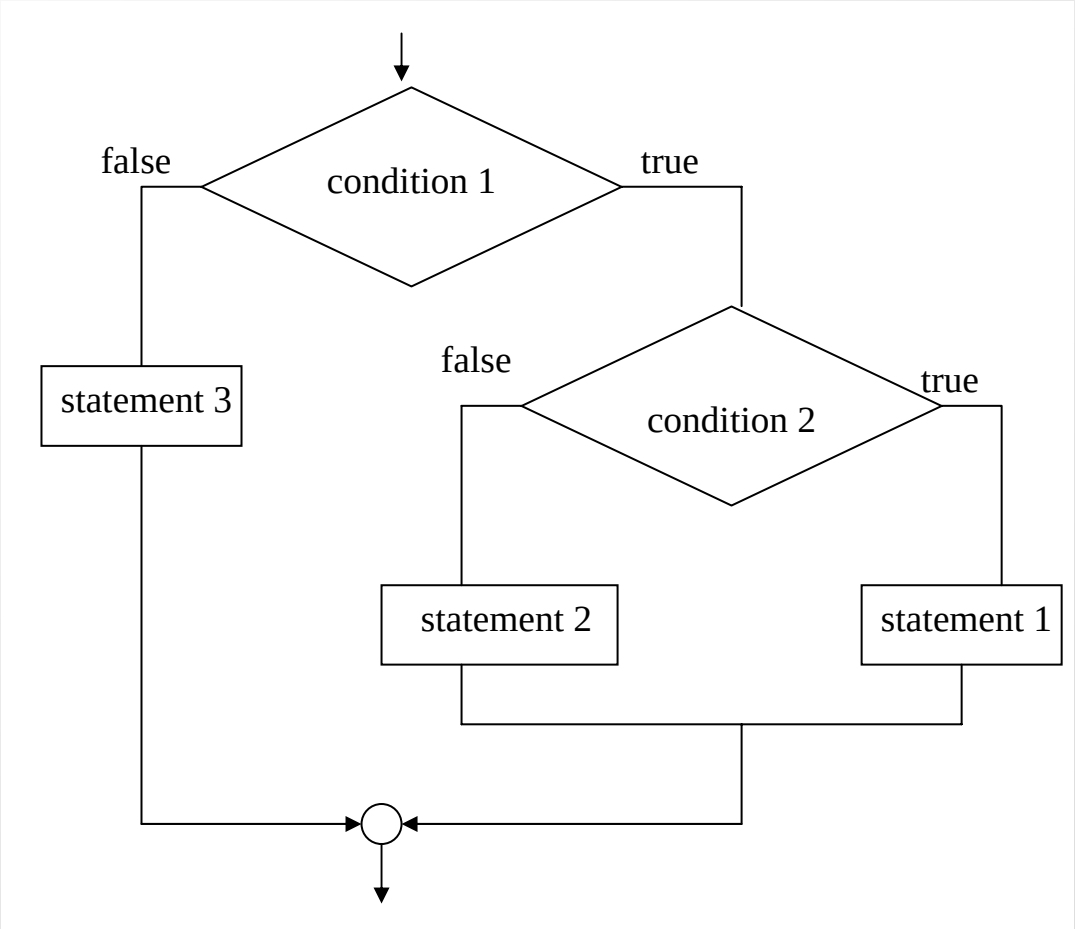
  A block is a collection of statements enclosed in curly braces:

  {
  statement_1
  statement_ 2
  …
  statement_n
  }

The *if* statement

- It comes in two forms:

```
if (condition)
   statement
if (condition)
   statement
else
   statement
```



The ***switch*** statement

switch (*e***) {**
  case *c1* **:**
    statements
    break;
  case *c2* **:**
    statements
    break;
  more case clauses
  default:
    statements
    break;
}

Example:

```
switch (month){
  case 6:
  case 11: return 30;
  case 2:
    return (IsLeapYear(year))? 29: 28);
  default :
    return 31;
}
```

Iterative Statements (***While***)

while (*conditional expression*) {
    statements
}

Example:

```
/* This function computes the sum of the digits in an integer.*/
int DigitSum (int n) {
     int sum;
     sum = 0;
     while (n > 0) {
        sum += n % 10;
        n /= 10;
     }
     return (sum);
}
```

while (cont.)

Many programming problems do not fit easily into the standard while loop structure. The most common example of such problems are those that read in data from the user until some special value, or sentinel, is entered to signal the end of the input.

One way to solve this problem is to use the break statement that has the effect of immediately terminating the innermost enclosing loop:

Example:

```
/* This program add a list of numbers */
#define sentinel 0
main() {
   int value, total =0;
   printf(“This program add a list of numbers.\n”);
   printf(“Use %d to signal the end of list.\n”,  sentinel);
   while (TRUE) {
      printf(“ ? “);
      value = GetInteger();
      if (value == sentinel) break;
      total += value;
   }
   printf(“The total is %d.\n”, total);
}
```

The *for* Statement

for (*initialization*; *loopContinuationTest*; *increment*) {
statements
}
which is equivalent to the while statement:
initialization;
while (*loopContinuationTest*) **{**
statements
increment;
}

Example: Finding the sum 1+3+…+99:

```
int main() {
        int val ;
        int sum = 0;

        for (val = 1; val < 100; val = val+2) {
            sum = sum + val;
        }
        printf("1+3+5+...+99=%d\n",sum);
        return 0;
}
```

The *do/while* Statement

do {
statements
} while (*condition* **);**

Example 1:

```
counter = 10;
do {
printf("%2d\n", counter );
counter -= 1;
} while (counter >= 0);
printf(“Liftoff!\n”);
```

Example 2:

Write a loop to enforce the user to enter an acceptable answer of Yes or No.

```
do {
    printf(“Do you want to continue? (Y/N)”);
    scanf(“%c”,&ans);
} while (ans != ‘Y’ && ans != ‘y’ &&
 ans != ‘N’ && ans != ‘n’);
```