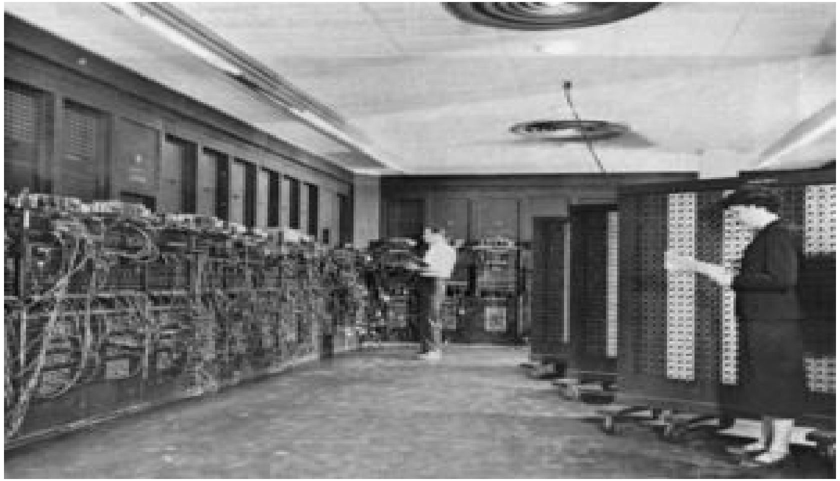
PROGRAMMING

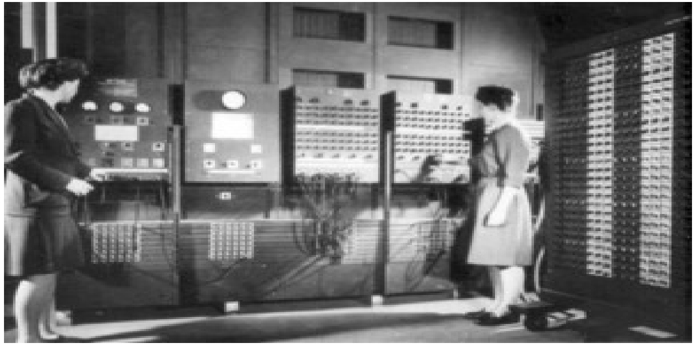
WEEK 1

MODULE 1: INTRODUCTION

Shankar Balachandran, IIT Madras

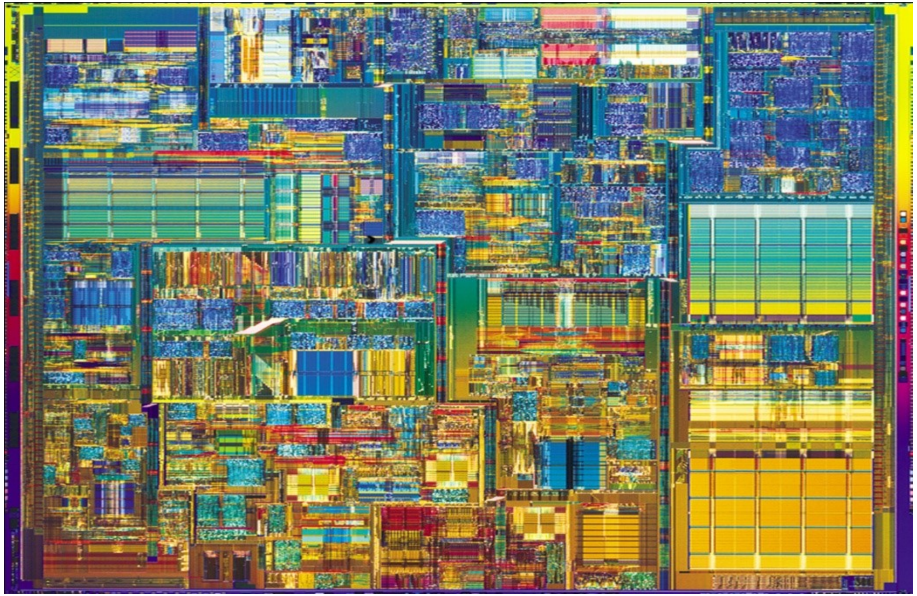
ENIAC (1940s)





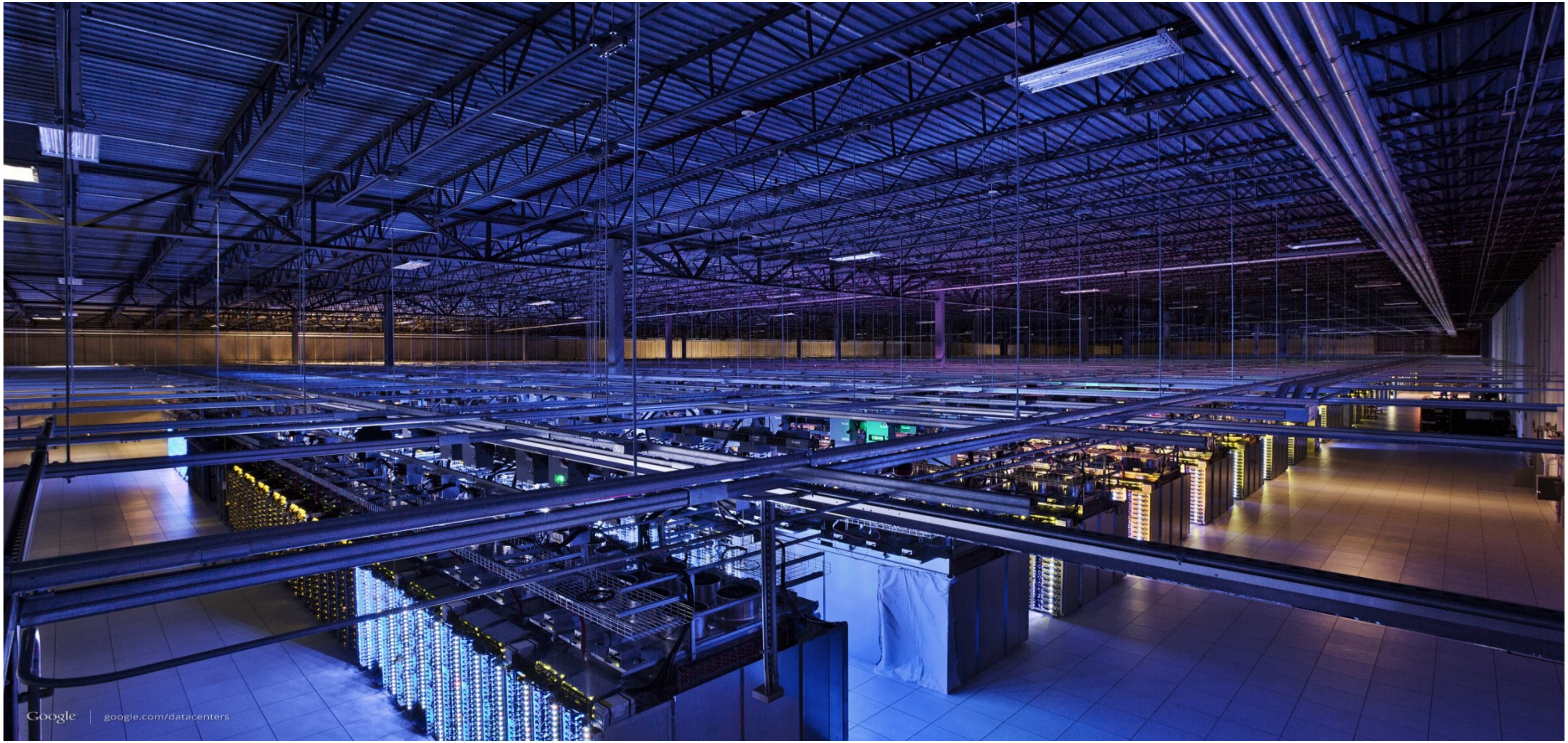
- ENIAC was massive compared to modern PC standards.
  - 17,468 vacuum tubes,
  - 5 million hand-soldered joints.
  - It weighed 27 tons
  - Took up 167 m^2, and consumed 150 kW of power.

Pentium 4(2000)

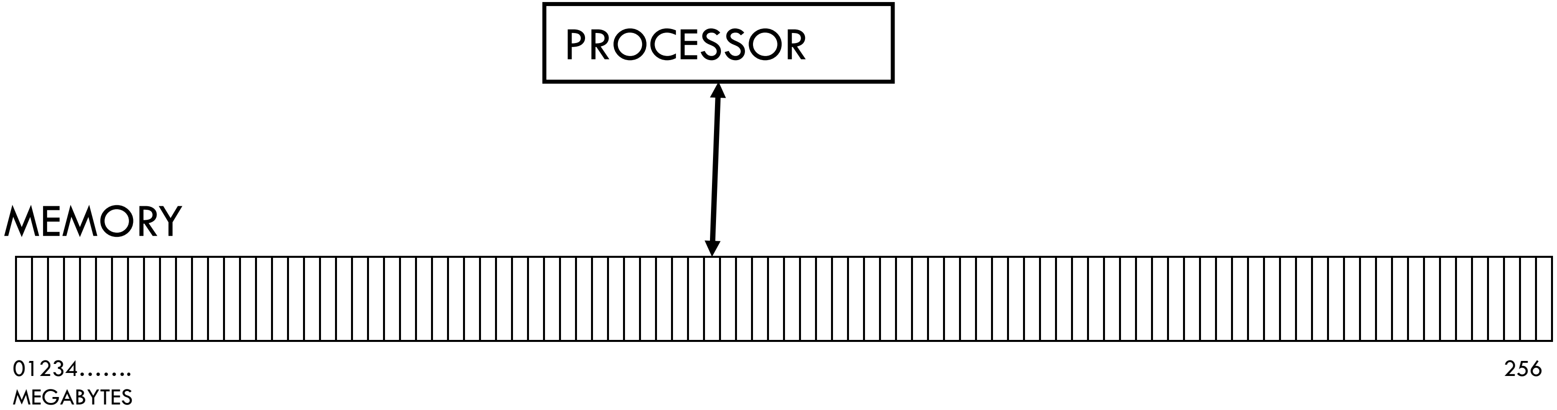


- Intel Pentium 4 Processor
  - Clock speed:
    1.5 GHz
  - #Transistors:
    42 million
  - Technology:
    0.18μm CMOS

Google Data Center

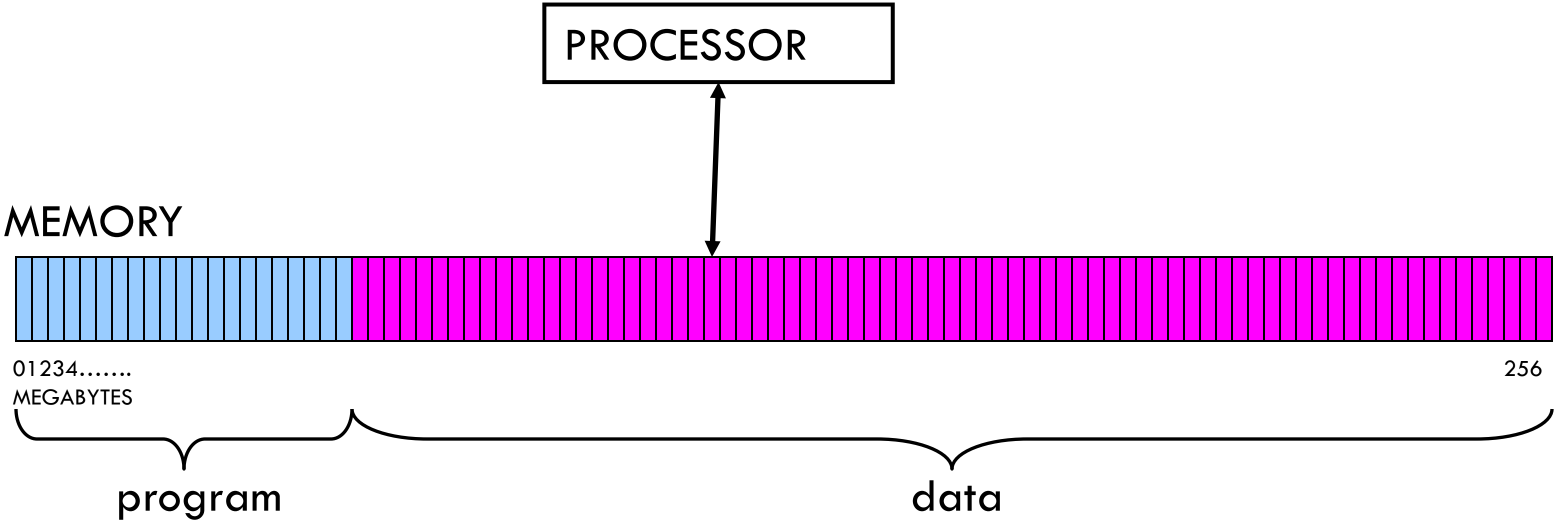


The Computing Machine



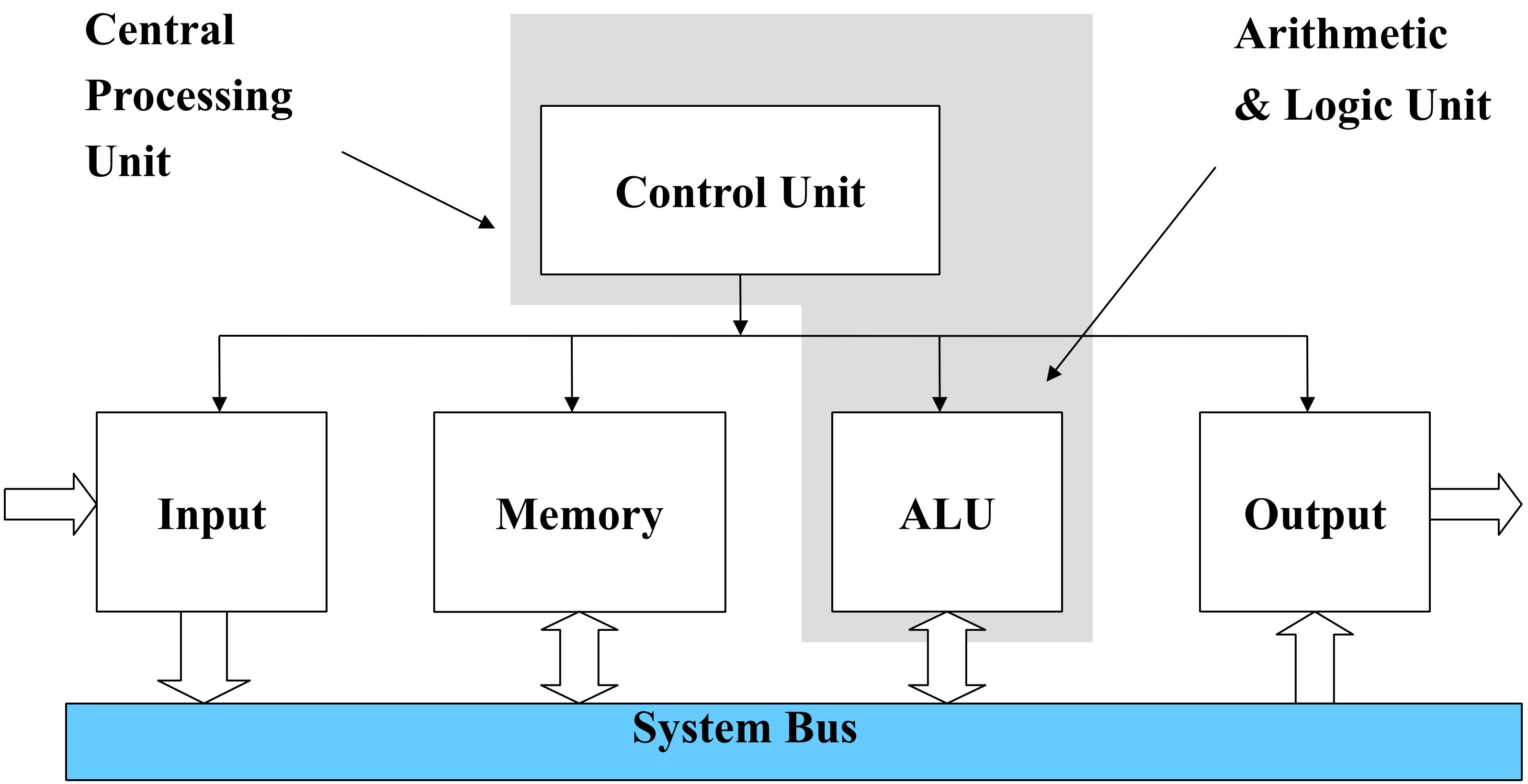
- The computer is made up of a *processor* and a *memory*.
- Memory can be thought of as a series of *locations* to store information.

The Computing Machine



- A *program* is a sequence of *instructions* assembled for some given task
- Most instructions operate on *data*
- Some instructions *control* the flow of the operations

Building Blocks



The CPU

- Can *fetch* an instruction from memory
- *Execute* the instruction
- *Store* the result in memory
- Program – A sequence of instructions
- An instruction has the following structure
  - *Operation, operands, destination*
- A simple operation

add a, b *Adds the contents of memory locations a and b and stores the result in location a*

Assembly Language

- An x86/IA-32 processor can execute the following binary instruction as expressed in machine language:

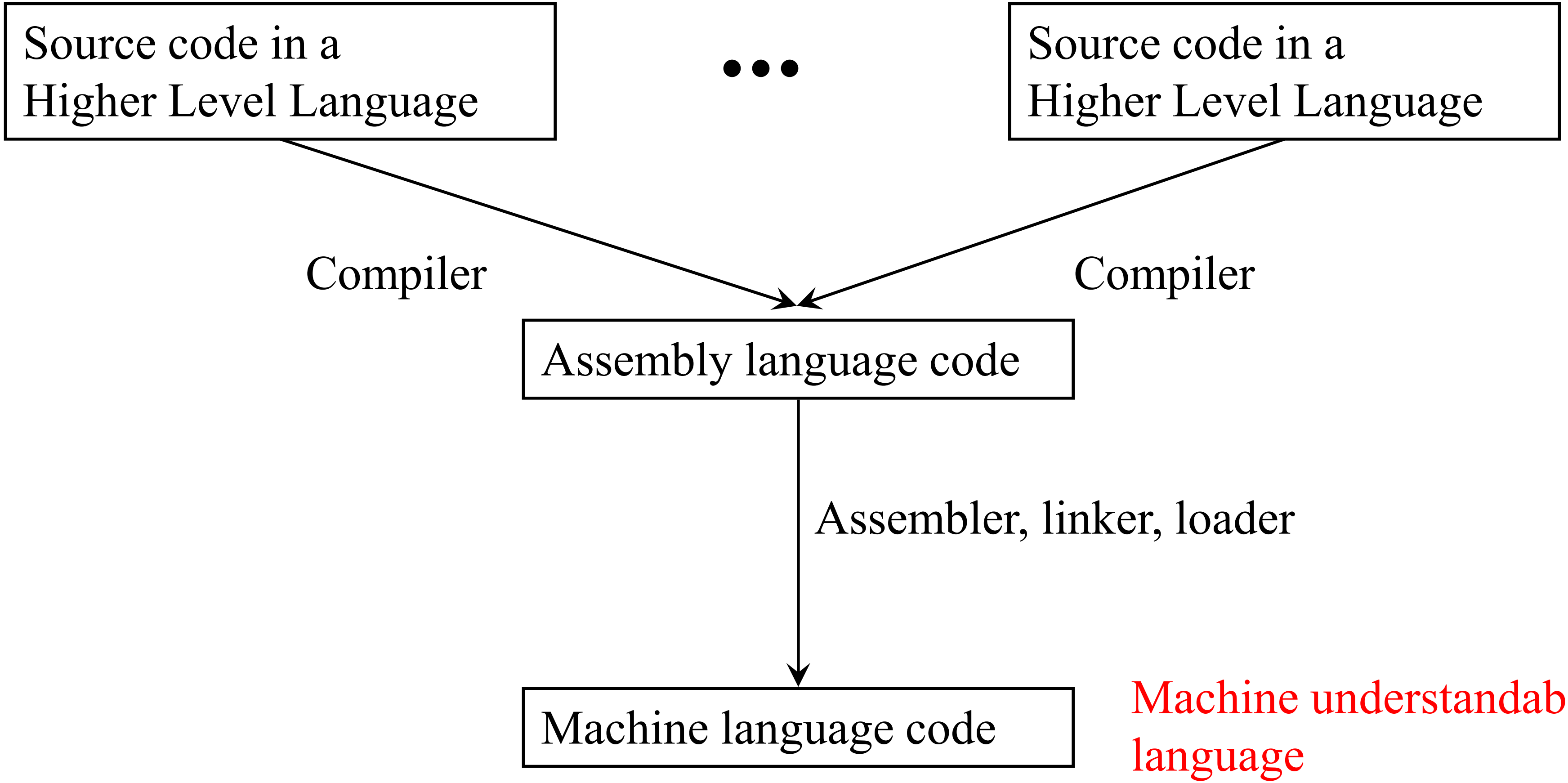
```
Binary: 101 10000  01100001
           mov      al,      061h
```

  - Move the hexadecimal value 61 (97 decimal) into the processor register named "al".
  - assembly language representation is easier to remember (*mnemonic;* e.g. - MVI AL, Val)

High Level Languages

- Higher level statement
  - Many assembly instructions
- For example “X = Y + Z” could require the following sequence
  - Fetch into R1 contents of Y
  - Fetch into R2 contents of Z
  - Add contents of R1 and R2 and store it in R1
  - Move contents of R1 into location named X

Compilers



Source code in a Higher Level Language

…

Source code in a Higher Level Language

Compiler

Compiler

Assembly language code

Assembler, linker, loader

Machine language code

Machine understandable language

Programs = Solutions

- A program is a sequence of instructions
  - *This is from the perspective of the machine or the compiler!*

- A program is a (frozen) solution
  - *From the perspective of a human, a program is a representation of a solution* devised by the human.
  - *Once frozen (or written and compiled) it can be executed by the* computer
    - *Much faster*
    - *As many times as you want.*

Programming = Problem Solving

- Software development involves the following
  - A study of the problem (requirements analysis)
  - A description of the solution (specification)
  - *Devising the solution (design)*
  - Writing the program (coding)
  - Testing

- The critical part is the solution design. One must work out the steps of solving the problem, analyze the steps, and then code them using a programming language.

The C Programming Language

- C Language
  - A general-purpose language
  - Extremely effective and expressive
  - Has compact syntax
  - Has a rich a set of operators
  - Extensive collections of library functions
- Been in use for four decades

A Tiny C Program

```
/* A first program in C */
#include <stdio.h>
main( )
{
   printf(“Hello, World! \n”);
}
```

A comment

Library of standard input output functions

Every C program starts execution with this function.

Statement & terminator

Body of the function - enclosed in braces

printf - a function from C Standard library stdio.h

End of Module 1

PROGRAMMING

WEEK 1

MODULE 2: PROBLEM SOLVING

Shankar Balachandran, IIT Madras

- Demo
  - Problem: Find the largest of 3 numbers
  - How to solve the problem?
  - C program
  - Demo of IDE for C programming

PROGRAMMING

WEEK 1

MODULE 3: VARIABLES AND ASSIGNMENTS

Shankar Balachandran, IIT Madras

Problem P1.1: Polynomial Multiplication

- Two polynomials $ax+b$ and $cx+d$
- Product:

$$(ac)x^2+(ad+bc)x+bd$$

Writing a Program for the P1.1

- Steps
  - Declare storage for all coefficients
  - Need to read the coefficients a, b, c and d from the user
  - Perform arithmetic operations and store the results
  - Print the coefficients of the resultant polynomial

Declare Storage

```
//This is a program to multiply two polynomials ax+b and cx+d
#include <stdio.h>
int main()
{
        int a, b, c, d;
        int p2, p1, p0;
```

Read the Inputs

```
printf("Enter a:");
scanf("%d",&a);
printf("Enter b:");
scanf("%d",&b);
printf("Enter c:");
scanf("%d",&c);
printf("Enter d:");
scanf("%d",&d);
```

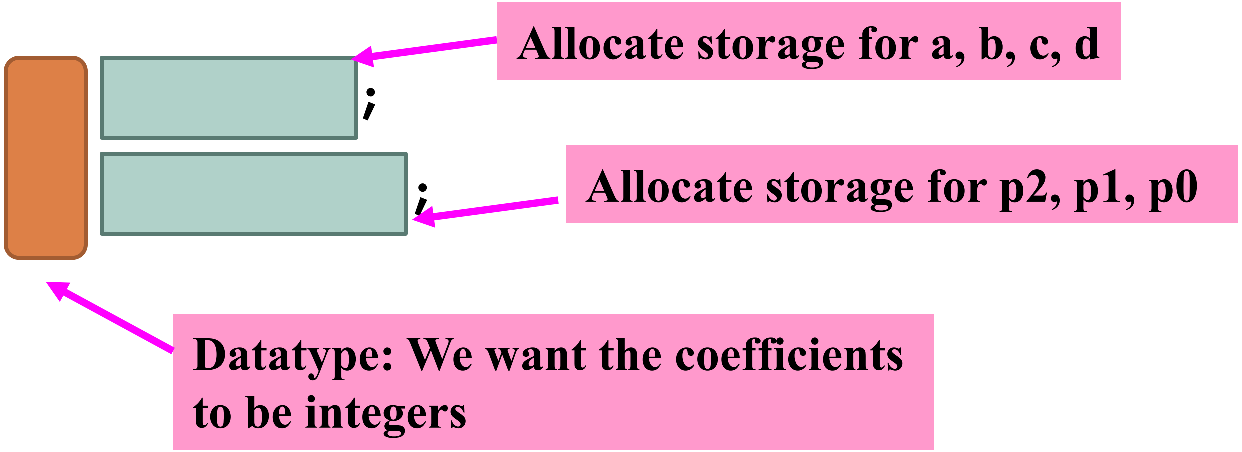
Calculate Coefficients and Print

```
        p2 = a*c;
        p1 = a*d + b*c;
        p0 = b*d;

        printf("The product is: %dx^2+%dx+%d\n",p2,p1,p0);
        return 0;
}
```

Close Look at the Program

```
//This is a program to multiply two polynomials ax+b and cx+d
#include <stdio.h>
int main()
{
```



Allocate storage for a, b, c, d

Allocate storage for p2, p1, p0

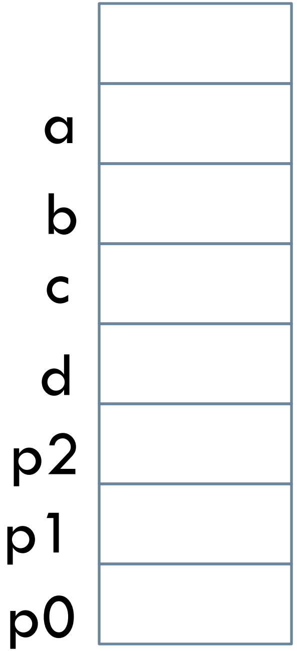
Datatype: We want the coefficients to be integers

From a Memory Point of View

- □ Initially unused

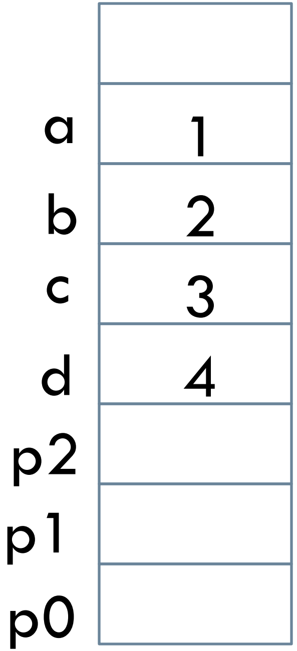
```
int a, b, c, d;

int p2, p1, p0;
```



Reading Inputs: How it changes memory

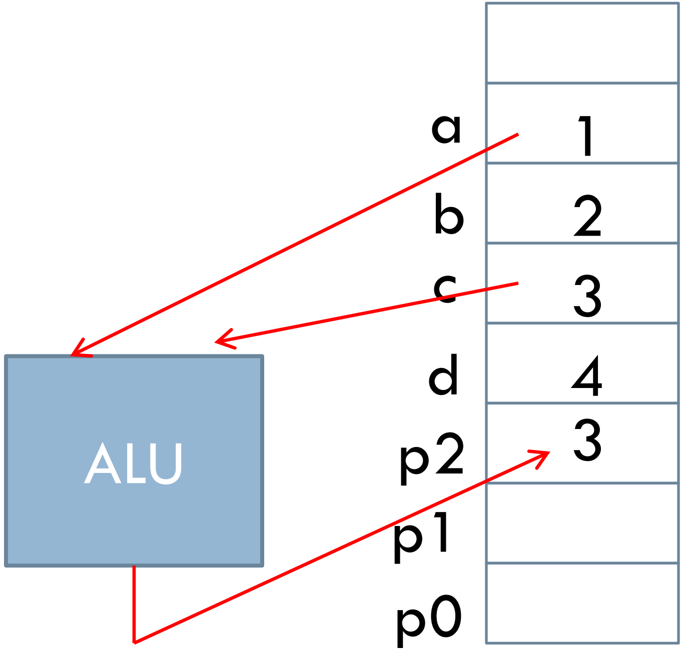
```
printf("Enter a:");
scanf("%d",&a);
printf("Enter b:");
scanf("%d",&b);
printf("Enter c:");
scanf("%d",&c);
printf("Enter d:");
scanf("%d",&d);
```



a	1
b	2
c	3
d	4
p2	
p1	
p0	

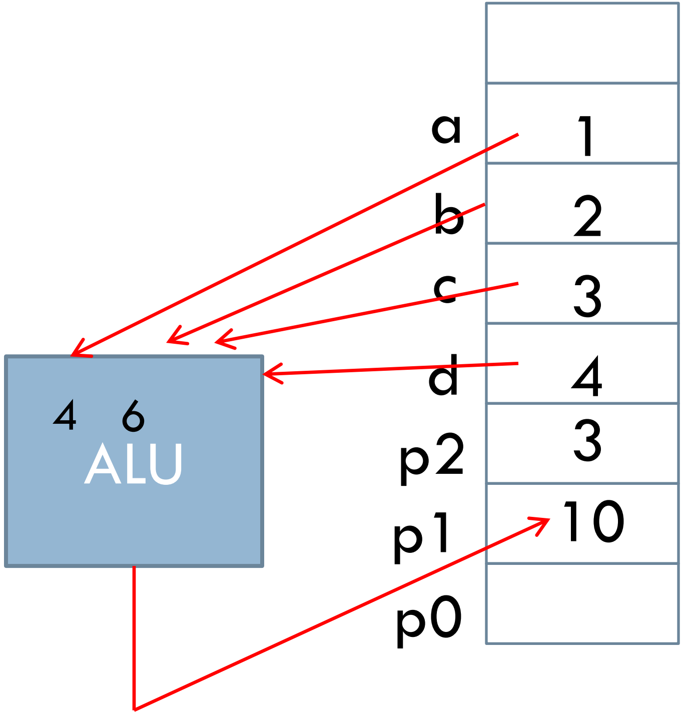
Calculations

```
p2 = a*c;
p1 = a*d + b*c;
p0 = b*d;
```



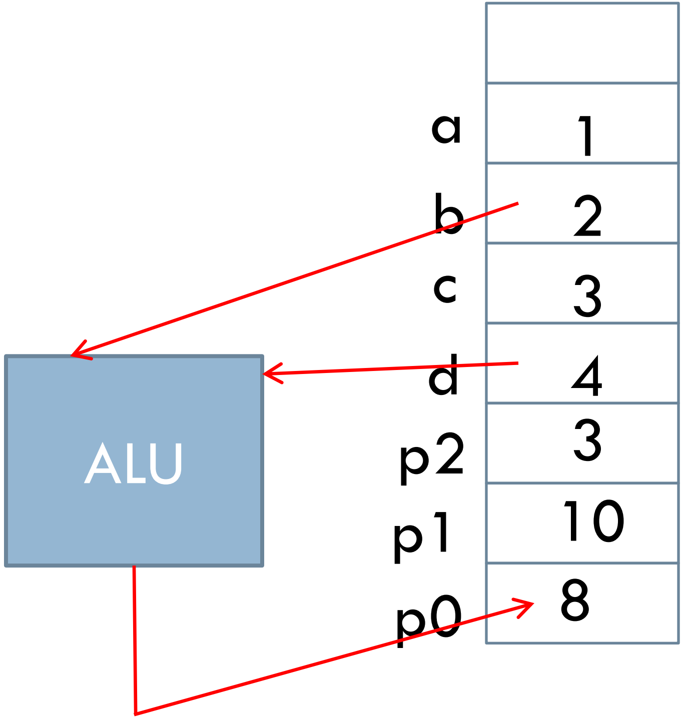
Calculations

```
p2 = a*c;
p1 = a*d + b*c;
p0 = b*d;
```



Calculations

```
p2 = a*c;
p1 = a*d + b*c;
p0 = b*d;
```



a	1
b	2
c	3
d	4
p2	3
p1	10
p0	8

PROGRAMMING

WEEK 1

MODULE 4: VARIABLE DECLARATIONS, OPERATORS AND PRECEDENCE

Shankar Balachandran, IIT Madras

Variables

- Each memory location is given a name
- The name is the *variable* that refers to the data stored in that location
  - Eg: rollNo, classSize, p2, a,…
- Variables have *types* that define the interpretation data.
  - e.g. integers (1, 14, 25649), or characters (a, f, G, H)
- All data is represented as binary strings. That is, it is a sequence of 0's and 1's (bits), of a predetermined size – "word". A *byte* is made of *8 bits*.

Instructions

- Instructions take data stored in variables as arguments.
- Some instructions do some operation on the data and store it back in some variable.
- The instruction “X←X+1” on integer type says: “Take the integer stored in X, add 1 to it, and store it back in (location) X”..
- Other instructions tell the processor to do something. For example, “jump” to a particular instruction next, or to exit

Programs

- A program is a sequence of instructions.
- Normally the processor works as follows,
  - Step A: pick next instruction in the sequence
  - Step B: get data for the instruction to operate upon
  - Step C: execute instruction on data (or "jump")
  - Step D: store results in designated location (variable)
  - Step E: go to Step A

Assignments

- = is the assignment operator
- The value of a variable is modified due to an assignment
- LHS has the variable to be modified
  - RHS is the value to be assigned.
- RHS is evaluated first
  - After completing the operation on RHS, assignment is performed.
- a = 1
- a = c
- a = MAX_PILLAR_RADIUS
- a = a*b + d/e

Variables and Constants

Names

- made up of letters, digits and '_'

  case sensitive: *classSize* and *classsize* are different

  maximum size: 31 letters

- first character must be a letter

- choose meaningful and self-documenting names

PI	a constant
radius	a variable

- keywords are reserved:
  - if, for, else, float, …

Variable Declaration

- Need to declare variables
  - They allocate storage
- Declaration in general

  type <variablename>
- Types: *int, float, char, double*
- *int x;*
  - contents of the location corresponding to x is treated as an integer.
  - Number of bytes assigned to a variable depends on its type.

Modifying Variables

- Each C program is a sequence of modification of variable values
- A modification can happen due to operations like +, -, /, *, etc.
- Also due to some functions/operators provided by the system like *sizeof, sin* etc.
- Also due to some functions (another part of the program) created by the programmer.

Operators in C

Four basic operators

- + , – , *, /
- addition, subtraction, multiplication and division
- applicable to integers and floating point numbers
- integer division - fractional part of result truncated
  - 12/ 5 is 2, 5/9 is 0

modulus operator : %

- x % y : gives the remainder after x is divided by y
- applicable only for integers, not to float/double

Operator Precedence

first	parenthesized subexpessions - innermost first	
second	*, / and %	(associates left to right)
third	+ and –	(associates left to right)

a + b * c * d % e – f / g

4 1 2 3 6 5

(a + (((b * c) * d) % e))– (f / g)

good practice -- use parentheses rather than rely on precedence rules

Precedence – Another Example

Value = a * (b+c) % 5 + x / (3 + p) – r - j

Evaluation order:

1. (b+c) and (3+p) : due to brackets
2. * and % and / have same precedence: a(b+c) is evaluated first, then *mod 5*. Also, x/(3+p).
3. Then, the additions and subtractions are done from the left to right.
4. Finally, the assignment of the RHS to LHS is done.
5. = is the operator that violates the left to right rule

Increment and Decrement Operators

- unusual operators
  - prefix or postfix
  - only to variables
  - can only be in the RHS of =

+ +	adds 1 to its operand
– –	subtracts 1 from its operand
n++	increments n after its use
++n	increments n before its use

- n = 4 ; x = n++; y = ++n;
  - After execution, x would be 4 , y would be 6 and n would be 6

Additional Slides

Calculations

The code sequence

```
 n = 4 ;
x = n++;
y = ++n;
```

is equivalent to

```
n = 4;
x = n;          //assign to x first and then increment n.  x = 4 now
n = n+1;        //n is 5 now
n = n+1;       //first increment n and then assign to y; n becomes 6 now
y = n;         //y is also 6 now
```

PROGRAMMING

WEEK 1

MODULE 5: I/O AND COMPOUND STATEMENTS

Shankar Balachandran, IIT Madras

Output Statement

- Format-string is enclosed in double quotes
- Format string indicates:
  - How many variables to expect
  - Type of the variables
  - How many columns to use for printing them (not very commonly used)
  - Any character string to be printed
    - Sometimes this would be the only output
      - Example : printf(“Hello World!”);

Example

```
int x; float y;
x = 20;  y = – 16.789;
printf(“Value x=%d and value y=%f\n”, x, y);
```

%d : print as integer   %f: print as real value

There are other specifiers too.

The output:

```
Value x=20 and value y=–16.789
```

Some printf Statements We Used

```
printf("Enter three numbers A, B and C: ");
```

- Empty format specification. The text will be printed as it is

```
printf("The product is %d x^2 + %d x +  %d\n",p2,p1,p0);
```

Output:

```
The product is: 3 x^2 + 10 x + 4
```

- %d means print as integer
- The three %d specifiers are matched to p2, p1 and p0 in that order
- Notice the spaces in the specifier as well as the output
- \n moves the cursor to the next line

Input Statement

- Format-string is enclosed in double quotes
- Format string indicates:
  - How many variables to expect
  - Type of the data items to be stored in var_1 etc
  - The symbol ‘&’ is used to specify the memory address where the value is to be stored

Example

```
scanf("%d%d%d",&A,&B,&C);
```

- Read three integers from the user
- Store them in memory locations of A, B and C respectively

```
scanf(“%d%f”,&marks, &averageMarks);
```

If the user keys in 16   14.75

- 16 would be stored in the memory location of marks
- 14.75 would be stored in the memory location of aveMarks
- scanf skips over spaces if necessary to get the next input
- Usually, space, comma, \n etc. are not used in the format specifier string of scanf

Other Format Specifiers

- Most commonly needed are %d and %f
- Character %c
- Exponent form %e
  - Example 1.523e2 = 152.3
- Several modifications to %f and %d are possible for printf
  - They control how much space is taken by the number vs. how much white space is introduced

Statements

- Program:
  - Declaration and one or more statements
    - Assignment statement
    - Function calls etc.
    - Selection statement
    - Repetitive Statements

Simple Statements

- Any statement that is an expression or a function call
- Examples:
  - X = 2;
  - X = 2 + 8;
  - printf(“Hello, World!”);
  - Y = sin(X);
- Generally, all simple statements are terminated by ‘;’

Compound Statements

- A group of declarations and statements collected together
  - Usually to form a single logical unit
  - Surrounded by braces
- Also called a block

Example

```
{
        int max;




}
```

Compound Statements

- Usually come in two varieties
- Conditionals
  - If..else..
  - switch
- Loops
  - for
  - while
  - do..while

PROGRAMMING

WEEK 1

MODULE 6: CONDITIONAL STATEMENTS

Shankar Balachandran, IIT Madras

Selection Statements

- Also called conditional statements
- Three forms
  - Single selection

```
    if(attendance < 75) grade = ‘W’;
```

  - Double selection

```
  if(marks < 40) passed = 0;  /*false = 0*/
  else passed = 1 /*true = 1*/
```

  - Switch
    - Will see later

if Statement

if (<expression>) <stmt1> [***else*** <stmt2>]

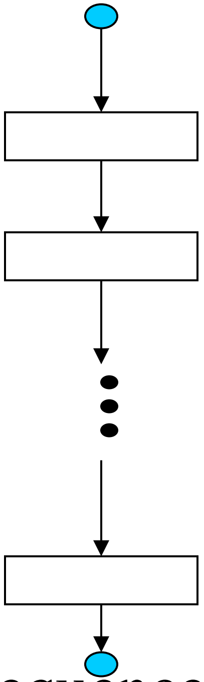
optional

- Semantics
  - If expression evaluates to “true”
    - stmt1 will be executed
  - If expression evaluates to “false”
    - stmt2 will be exectued
- stmt1 and stmt2 are usually blocks

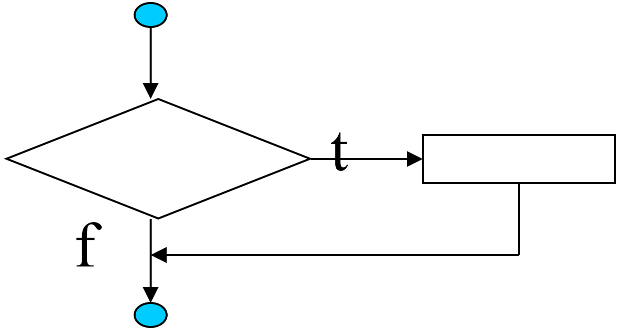
Else part is optional

- If there is no else part in the *if* statement
    - If expression is “true”
        - stmt1 will be executed
    - Otherwise the if statement has no effect

Sequence and Selection Flowcharts



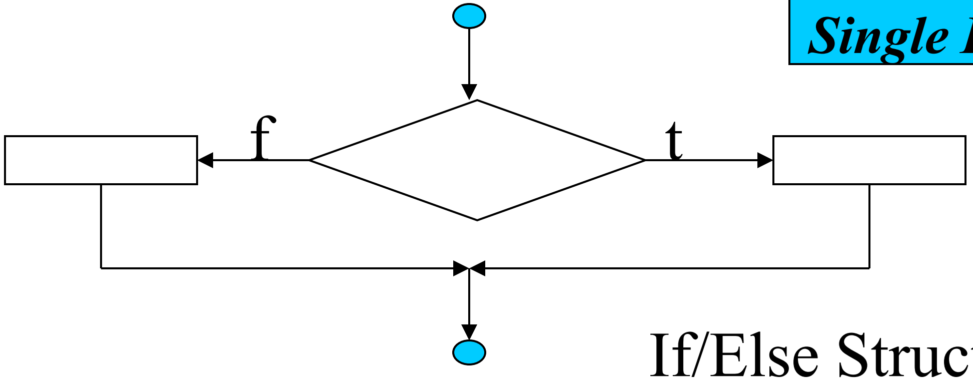
Sequence Structure



If Structure

Single Entry

Single Exit



If/Else Structure

Example 1: No else clause

- Given a number, find out if it is a multiple of 3

Modulo operator

```
if (x %3 == 0)
        printf(“x is a multiple of 3”);
```

== is an operator to check equality

Example 2: No else clause

- If the given number is a multiple of 3, ask for another input

```
if (x %3 == 0) {
        printf(“x is a multiple of 3; Please enter another number”);
        scanf(“%d”,&x);
}
```

Needs a brace because the if statement has two simple statements inside

Be Warned

- If the given number is a multiple of 3, ask for another input

```
if (x %3 == 0)
        printf("x is a multiple of 3; Please enter another number");
        scanf("%d",&x);
```

- *if* block only has printf
- scanf statement is outside the if condition
- Result:
  - User will have to enter a number even if x is not a multiple of 3

Simple Thumbrules

- Use ‘{’ and ‘}’ to enclose if and else blocks
- Will save you several headaches
- Can become slightly unreadable though

Example 3: Else clause

- If the given number is a multiple of 3, ask for another input. Otherwise, thank the user.

```
if (x %3 == 0) {
        printf(“x is a multiple of 3; Please enter another number”);
        scanf(“%d”,&x);
}
else{
        printf(“Thank you!\n”);
}
```

There is no guarantee that the user entered a correct number here though!

Two Useful Structures Involving *if* Statements

Cascading if..else

```
if (expr1)
 …
else if (expr2)
 …
else if (expr…)
.
.
else
…
```

Nested if..else

```
if (expr1){
    …
    if(expr2){
        …
        if(
            .
                .
    }
}
```

Example 4: *cascading if*

Below 50: D; 50 to 59: C ; 60 to 75: B; 75 above: A

```
int marks; char grade;
       ...
       if  (marks <= 50) grade = ‘D’;
       else if (marks <= 59) grade = ‘C’;
               else  if (marks <=75) grade = ‘B’;
                    else grade = ‘A’;

       ...
```

Simple statements;

No braces required

Example 5: *Nested if* (Maximum of 3 Numbers)

```
if(A>B){
        if(A>C){
                printf("A is the largest\n");
        }
        else{
                printf("C is the largest\n");
        }
}
...
```

Caution In Use of *else*

```
if  ( marks  > 40)
    if  ( marks > 75 ) printf(“you got distinction”);
else printf(“Sorry you must repeat  the course”);
```

/* WRONG */

Else* gets matched to the nearest *if* statement without an *else

```
if  ( marks  > 40) {
    if  ( marks > 75 ) printf(“you got distinction”);
}
else printf(“Sorry you must repeat  the course”);
```

/*RIGHT*/

Switch Statement

A multi-way decision statement

Syntax:

```
switch ( expression ) {
    case const-expr : statements
    case const-expr : statements
    …
    [ default: statements ]
}
```

Example 6

```
char c;
scanf(“%c”,&c);
switch (c) {
        case   : printf(“RED”);
        case   : printf(“BLUE”);
        case   : printf(“YELLOW”);
}
```

Choices

Breaks from the switch statement

Example 7:

This example handles both lower and upper case user choices

```
char c;
scanf(“%c”,&c);
switch (c) {
        case ‘R’: case ‘r’: printf(“RED”); break;
        case ‘B’: case ‘b’: printf(“BLUE”); break;
        case ‘Y’: case ‘y’: printf(“YELLOW”);
}
```

Warning : Variables cannot appear as choices

```
char c;
char char1 = ‘r’; char char2 = ‘B’;
scanf(“%c”,&c);
switch (c) {
        case char1: printf(“RED”); break;
        case char2: printf(“BLUE”); break;
        case ‘Y’: case ‘y’: printf(“YELLOW”);
}
```

Warning: Incorrect program segment

Warning : Cannot use ranges

Warning: Incorrect program segment

```
int marks;
scanf(“%d”,&marks);
switch (marks) {
        case 0-49: printf(“D”); break;
        case 50-59: printf(“C”); break;
        case 60-74: printf(“B”); break;
        case 75-100: printf(“A”); break;
}
```

PROGRAMMING

WEEK 1

MODULE 7: REPETITIVE STATEMENTS

Shankar Balachandran, IIT Madras

Loops

- A very important type of statement
  - Iterating or repeating a set of operations
  - Very useful in algorithms
- C offers three iterative constructs
  - The *for* construct
  - The *while*… construct
  - The *do* … *while* construct

Loops

- Two kinds
- Counter controlled
  - Repeat a set of operations for some fixed number of times
  - Use when the number of repetitions is known
- Sentinel Controlled
  - Loop runs until a certain condition is met
    - Example: -1 is entered as input
  - Use when the number of repetitions is a property of the input and not of the problem being solved

For loops

- Ideal for counter controlled repetitions
  - Initial value
  - Modification of counter
    - ++, -- or some other arithmetic operation
  - Final value
- *For* repetition structure lets the programmer specify all of these

The *for* construct

- General Form:

```
for (expr1;  expr2; expr3)
    statement
```

Usually a block of code.

- Semantics:
  - Evaluate expr1 – initialization operations
  - Repeat
    - Evaluate expr2
    - If expr2 is true execute statement and expr3
    - Else stop and exit the loop

Example 1:

Compute the sum of first 20 odd numbers

```
int i, k, sum;
sum = 0;
k =1;
for(i=1; i <= 20; i++){
    sum += k;
    k += 2;
}
```

Set k to the first odd number

Termination condition

Add the ith odd number to sum

i is the loop control variable

Set k to the next odd number

A Small Detour: Relational Operators

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

A Small Detour: Logical Operators

&&	logical AND operator
||	logical OR operator

- Useful for combining conditions
- Example:
  - *if* ((age <= 45) && (salary >= 5000))
  - *if* ((num %2 == 0) || (num %3 == 0))

Example 2: Triangular Number

Find the n^{th} triangular number (the sum of integers from 1 to n)

Code Segment:

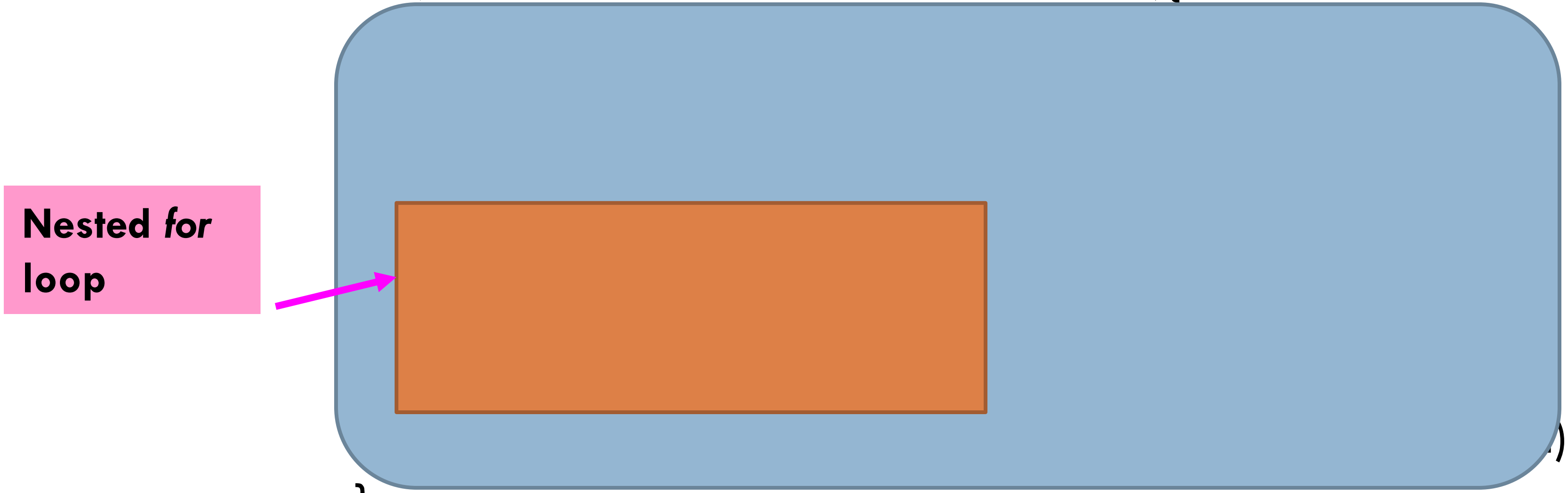
```
int i, number, sum;
printf("What triangular number do you want?");
scanf("%d",&number);
sum = 0;
for(i=1; i <= number; i++){
    sum += i;
}
printf("The %dth triangular number is %d\n",number, sum);
```

Example 3: User wants five such numbers

Can run the program five times. Better, rewrite the program.

Code Segment:

```
int i, number, sum, counter;
for(counter=1; counter <=5; counter++){
```



Nested *for* loop

```
);
}
```

The *while* construct

- General Form:

```
while (expr) <statement>
```

- Semantics:
  - Repeat
    - Evaluate expr
    - If expr is true execute statement
    - Else stop and exit the loop

expr must change inside the loop. Otherwise, you would end up with an infinite loop.

Example 4:

- Simple program: Print the first 5 integers

Code Segment:

```
int count=1;
 while(count <= 5){
         printf(“%d\n”, count);
        count++;
 }
```

Example 5: GCD of Two +ve Numbers

- Idea: if (m>n) gcd(m,n) = gcd(n, m%n)
  - Called the Euclid's algorithm (around 300 B.C.E)

- GCD(43,13)
  - 43 % 13 = 4
  - 13 % 4 = 1
  - 4 % 1 = 0
  - 1 is the GCD

- GCD(96,28)
  - 96 % 28 = 12
  - 28 % 12 = 4
  - 12 % 4 = 0
  - 4 is the GCD

Example 5: GCD of Two +ve Numbers

- Idea: if (m>n) gcd(m,n) = gcd(n, m%n)
  - Called the Euclid's algorithm (around 300 B.C.E)
- Let u, v be the two +ve numbers that user inputs such that u > v
- Code Segment:

```
 int temp;
/* code to read u and v from user here*/
while(v != 0){
        temp = u%v;
        u = v;
        v = temp;
}
printf("GCD is %d\n",u);
```

Example 6: Find the Reverse of a +ve Number

- Reverse of 234 is 432
- Till the number becomes 0
  - Extract the last digit of the number
    - number modulo 10
  - Make it the next digit of the result
    - Multiply the current result by 10 and add the current digit

An Example

x is the given number

y is the number being computed

y = 0	x = 56342
y = 0*10 + 2 = 2	x = 5634
y = 2*10 + 4 = 24	x = 563
y = 24*10 + 3 = 243	x = 56
y = 243*10 + 6 = 2436	x = 5
y = 2436*10 + 5 = 24365	x = 0

Termnation condition: Stop when x becomes zero

Program

```
#include <stdio.h>
void main( ){
    int x = 0; int y = 0;
    printf("input an integer :\n");
    scanf("%d", &x);
    while(x > 0){
        y = 10*y + ( x % 10 );
        x = (x / 10);
     }
     printf("The reversed number is %d \n", y);
}
```

Remember integer division truncates the quotient

The *do..while* construct

- General Form:

```
do <statement> while (expr)
```

- Semantics:
  - Execute the statement and then evaluate expr
  - If expr is true re-execute statement, else exit the loop

***for* and *while* loop are different. They check the condition before even the first execution of the loop body.**

Example 7: Input Numbers of a Specific Kind

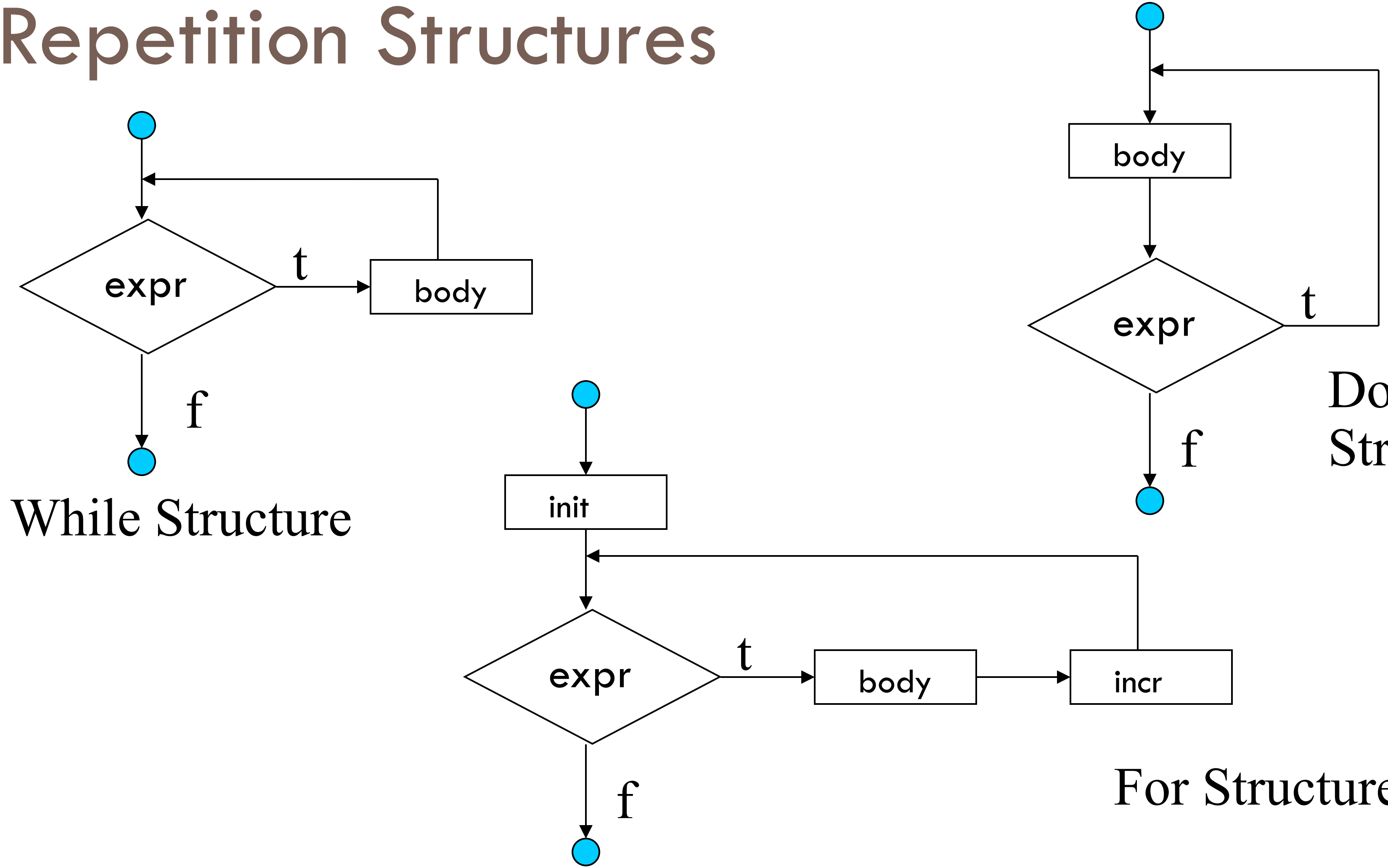
Ask the user for a number that is positive and not a multiple of 3

Code Segment:

Logical OR

```
int x;
do {
    printf (“Enter a positive number that is not a multiple of 3:”);
    scanf(“%d”,&x);
 } while  ( (x < 0) || (x%3 == 0));
```

Repetition Structures



While Structure

Do/While Structure

For Structure

Two Ways to Change The Loop Behavior

- *break*
- *continue*
- More on these later, when we see other examples

Miscellaneous

- Dev C++
  - http://sourceforge.net/projects/orwelldevcpp
  - ~42 MB in size
- Practice the problems given here
  - Write complete programs using the code segments shown here
  - Compile and run the programs
  - Test the programs with your own inputs

End of Week 1