



INDIAN INSTITUTE OF  
INFORMATION  
TECHNOLOGY

# Basic Programming and Data Structures

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur



Indian Institute of Technology Indore  
भारतीय प्रौद्योगिकी संस्थान इंदौर



PDPM

Indian Institute of Information Technology,  
Design and Manufacturing, Jabalpur

The  
Alan Turing  
Institute

# Goals of the Course

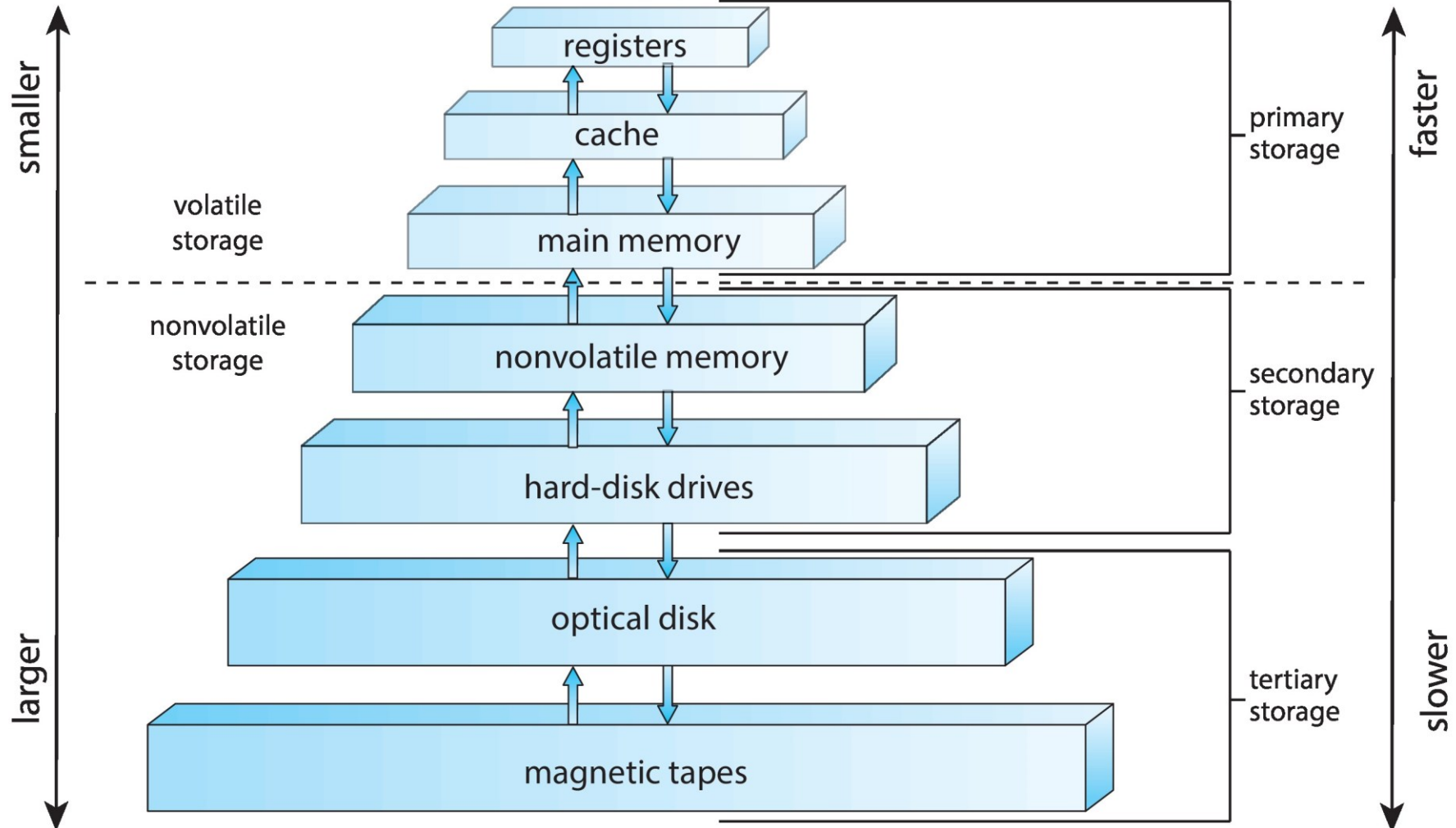
- To become familiar with Data structures and Algorithms
  - To develop ability of problem solving skill in programming
  - To understand that Data structures are the building blocks of larger softwares
  - To develop ability for analyzing existing Data structures and Algorithms
  - To develop skills with the C, C++, Java for Data structures and Algorithms
- 
- "Get your data structures correct first, and the rest of the program will write itself."  
- David Jones

# Basic Program Execution on CPU and Memory

# Data Storage Device Hierarchy

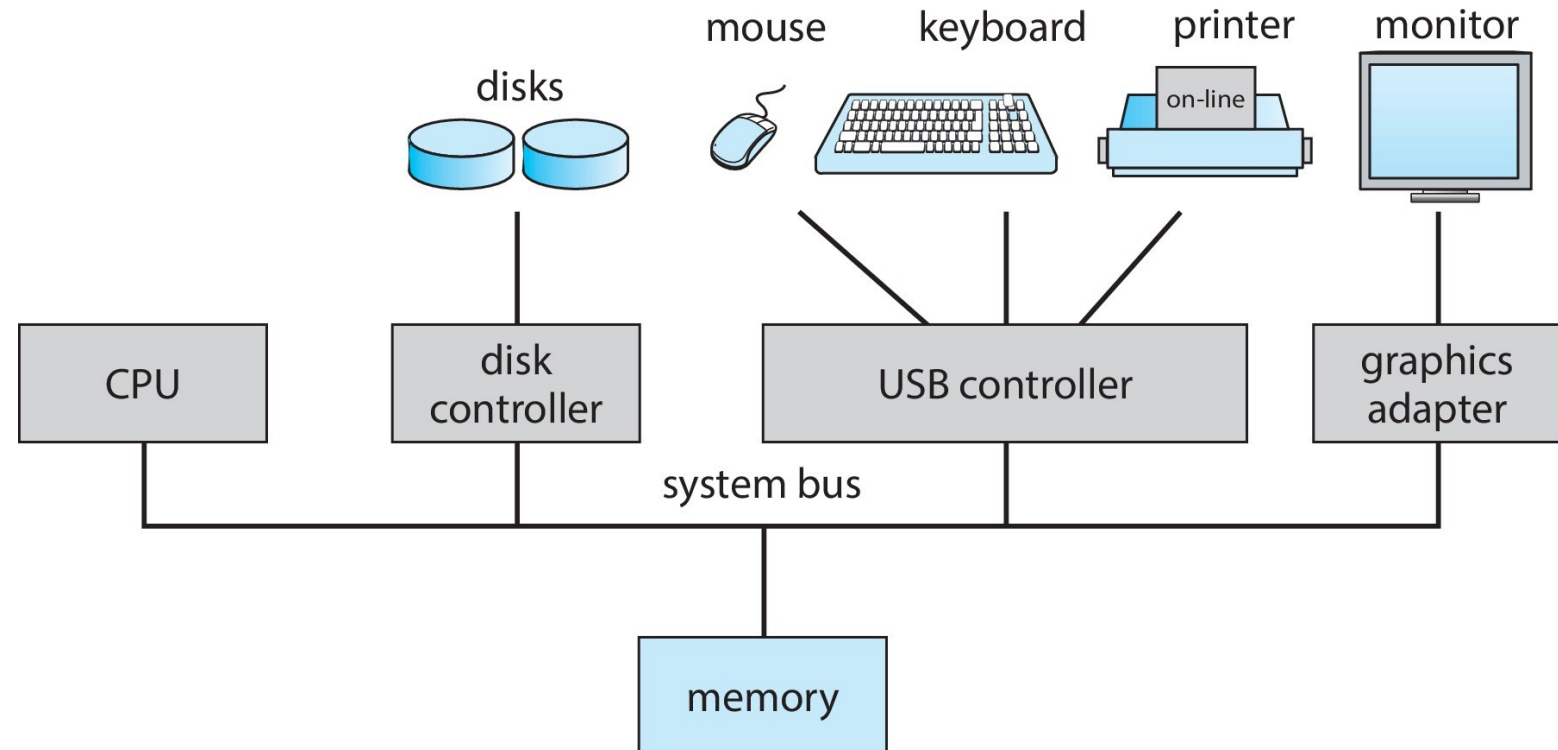
storage capacity

access time



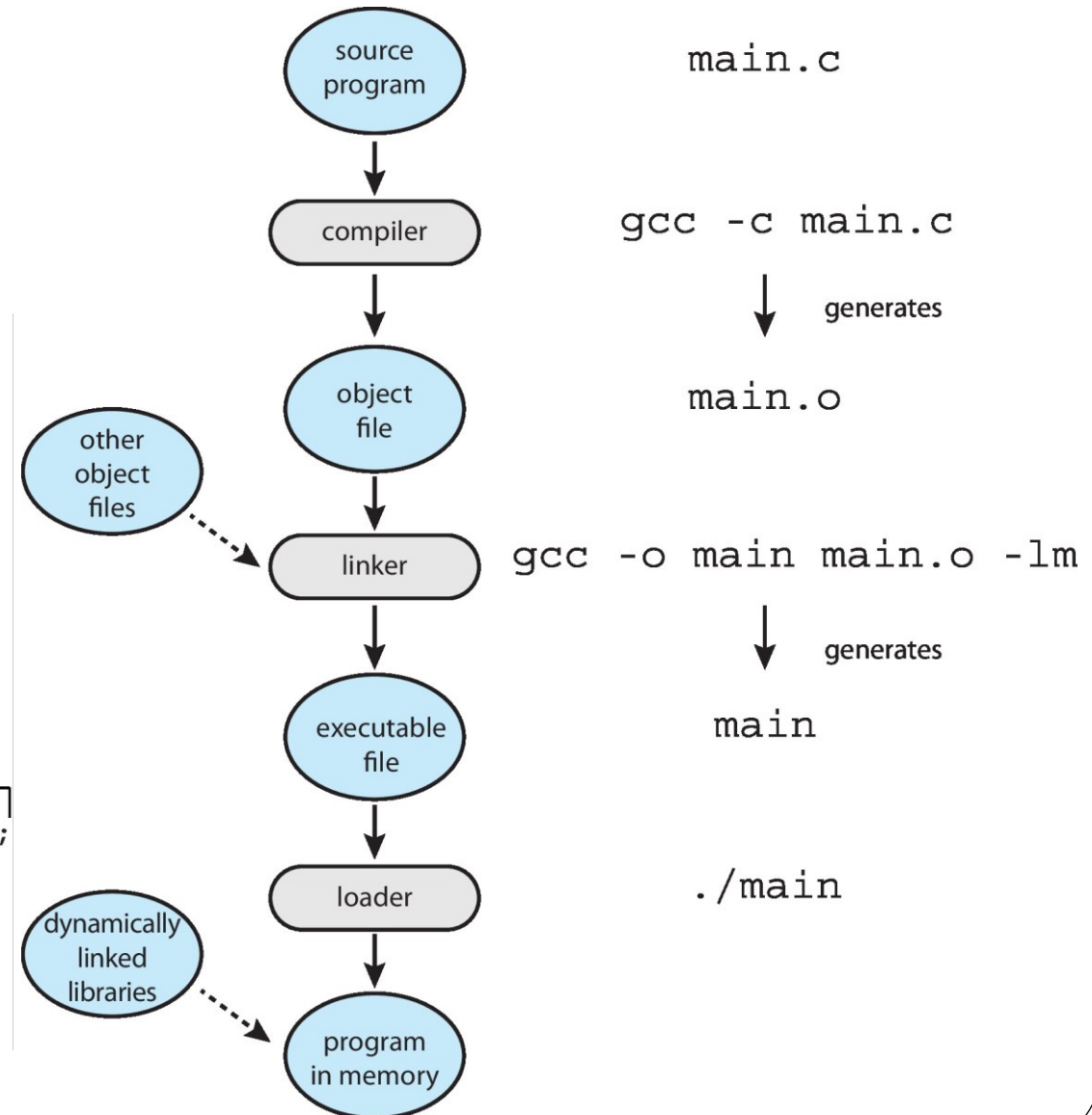
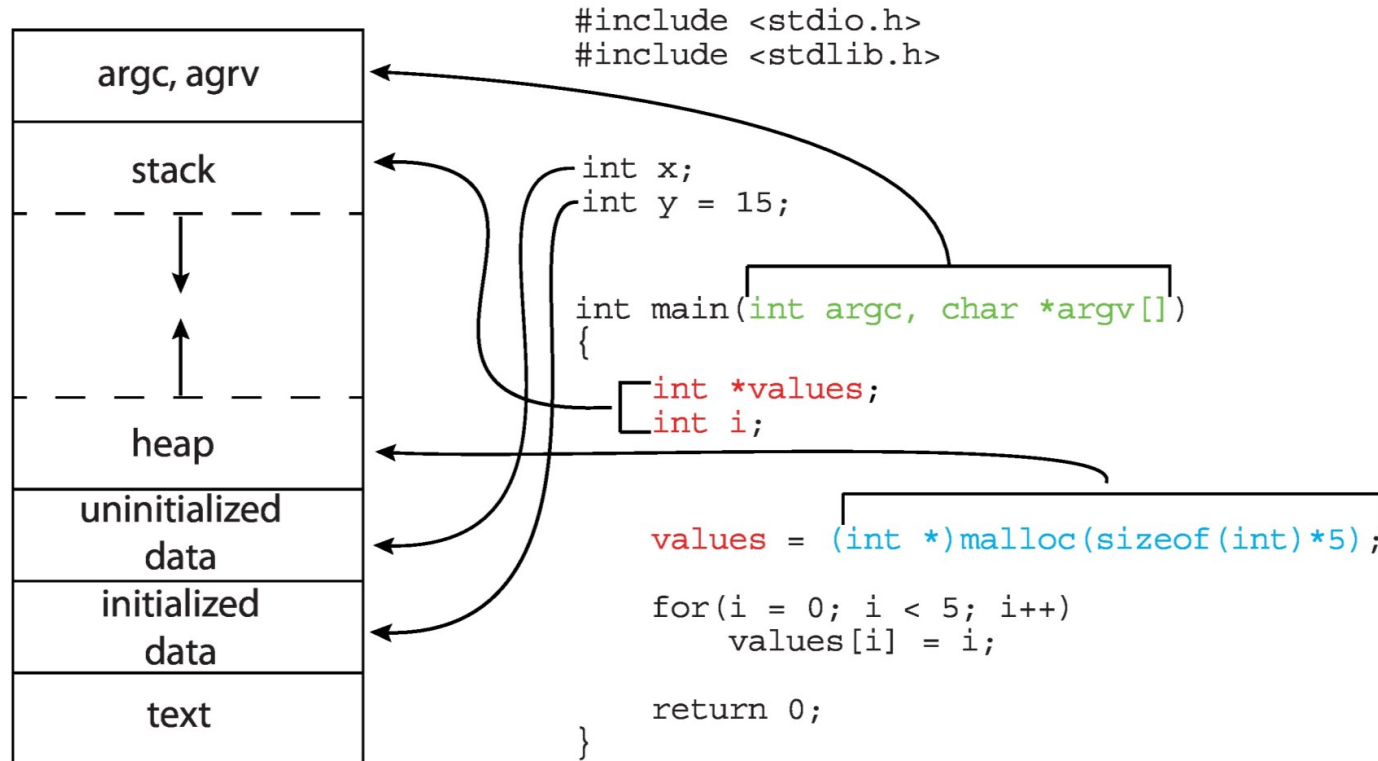
# Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



# Program to Process

- When you run an exe file, the OS creates a process = a running program

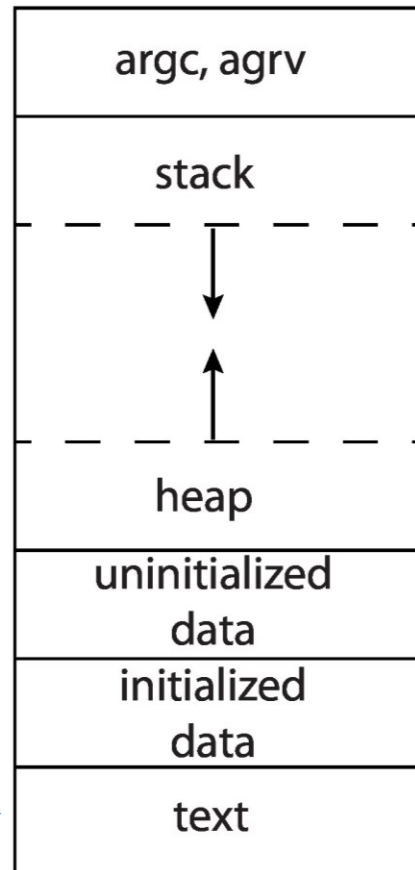


# Motivation: Program Code to Memory

- **Abstraction** of complex usage Program as a Memory (RAM or Cache).
- Conversion of **High level language to Low level language**
- Static/global variables are allocated in the executable

- Local variables of a function on Stack
- Dynamic allocation with malloc on the heap

Process as  
a Memory



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

```
int x;
int y = 15;
```

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

```
    int *values;
    int i;
```

```
    values = (int *)malloc(sizeof(int)*5);
```

```
    for(i = 0; i < 5; i++)
        values[i] = i;
```

```
    return 0;
```

```
}
```

Program as a Code

# Program to Process

- Virtual address space is setup by OS during process creation

Simplified OS: places entire memory image in one chunk

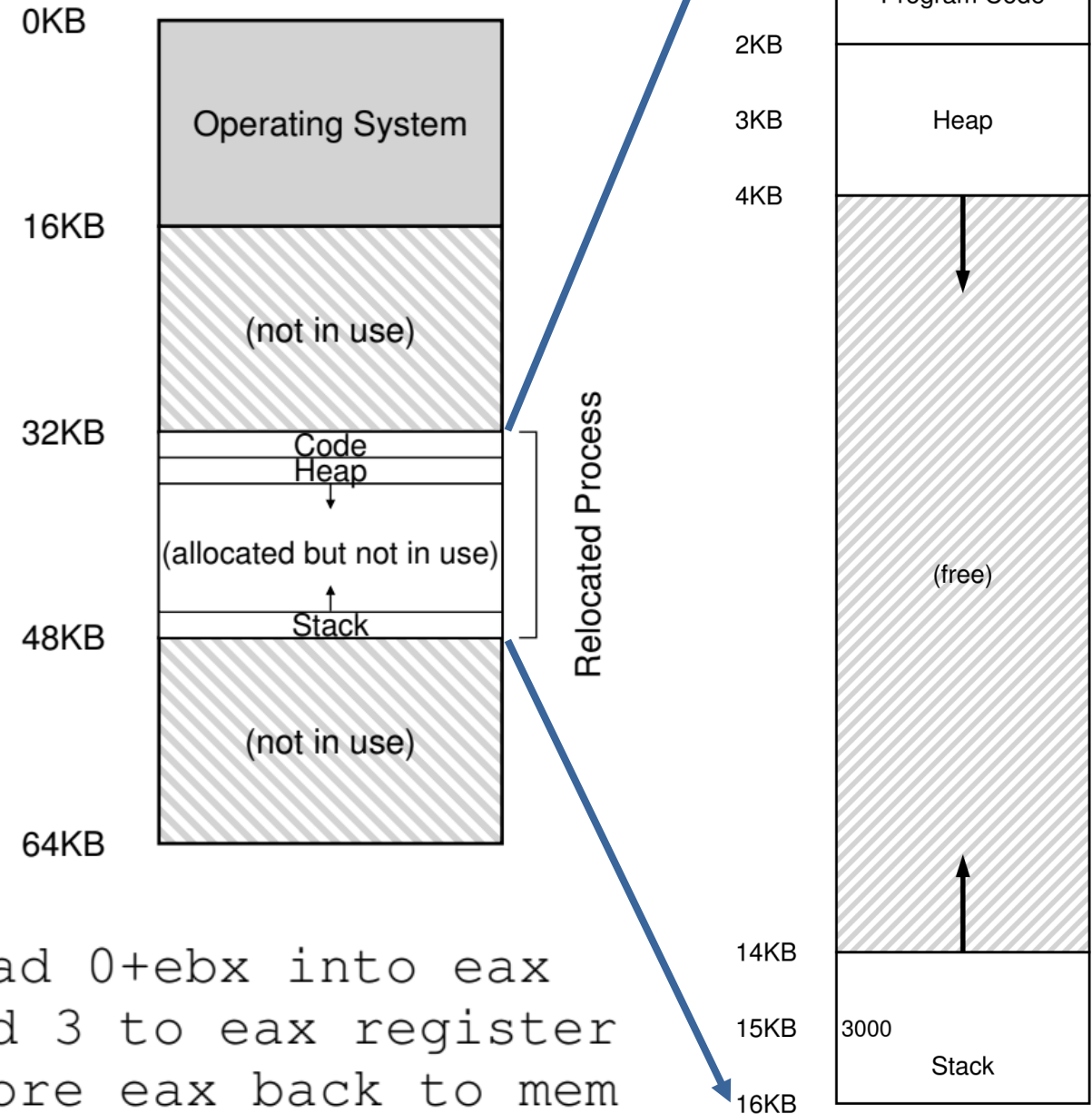
```
void func() {  
    int x = 3000;  
    x = x + 3;  
    ...
```



Compiler

```
128: movl 0x0(%ebx), %eax  
132: addl $0x03, %eax  
135: movl %eax, 0x0(%ebx)
```

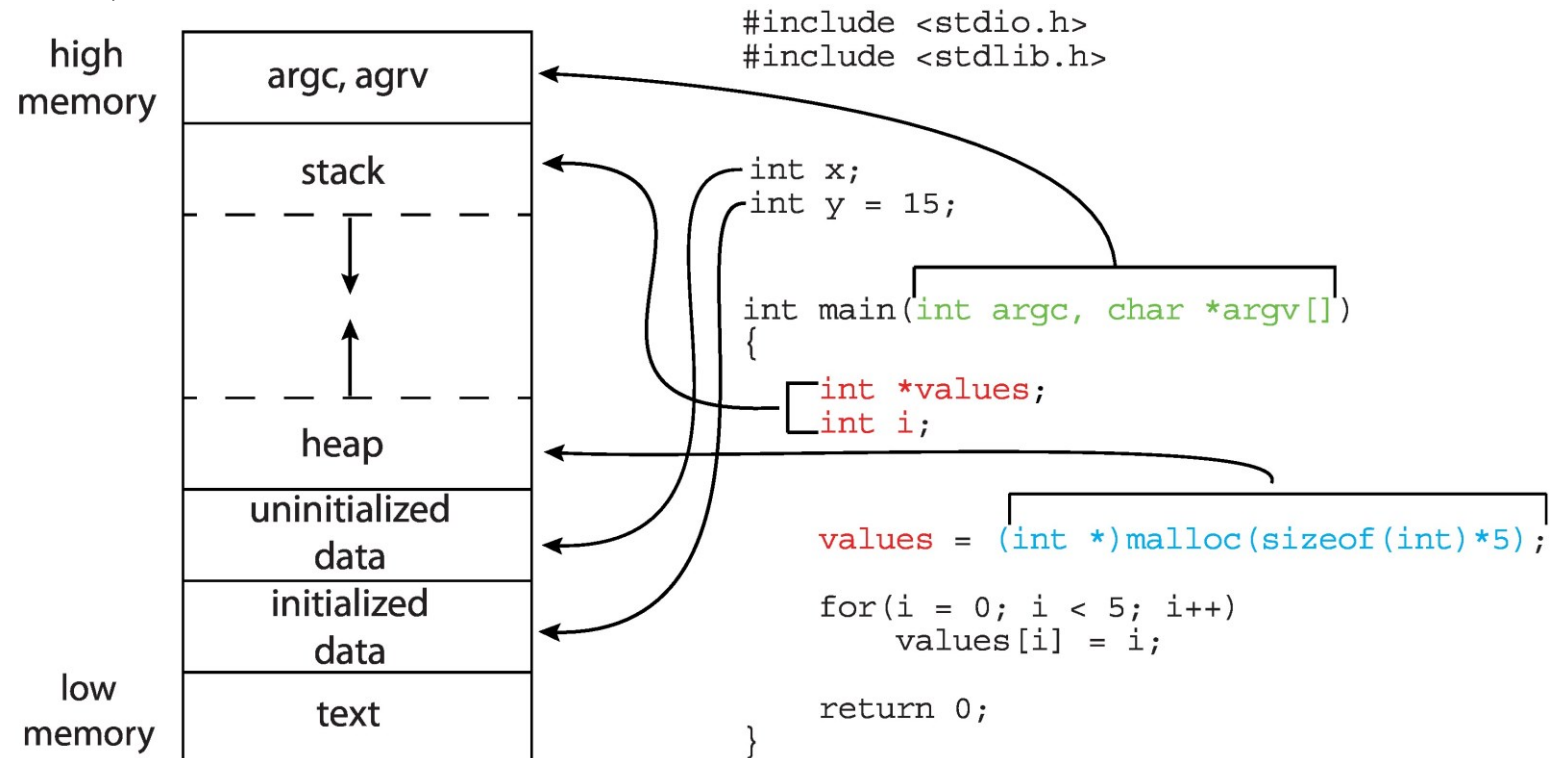
```
;load 0+ebx into eax  
;add 3 to eax register  
;store eax back to mem
```



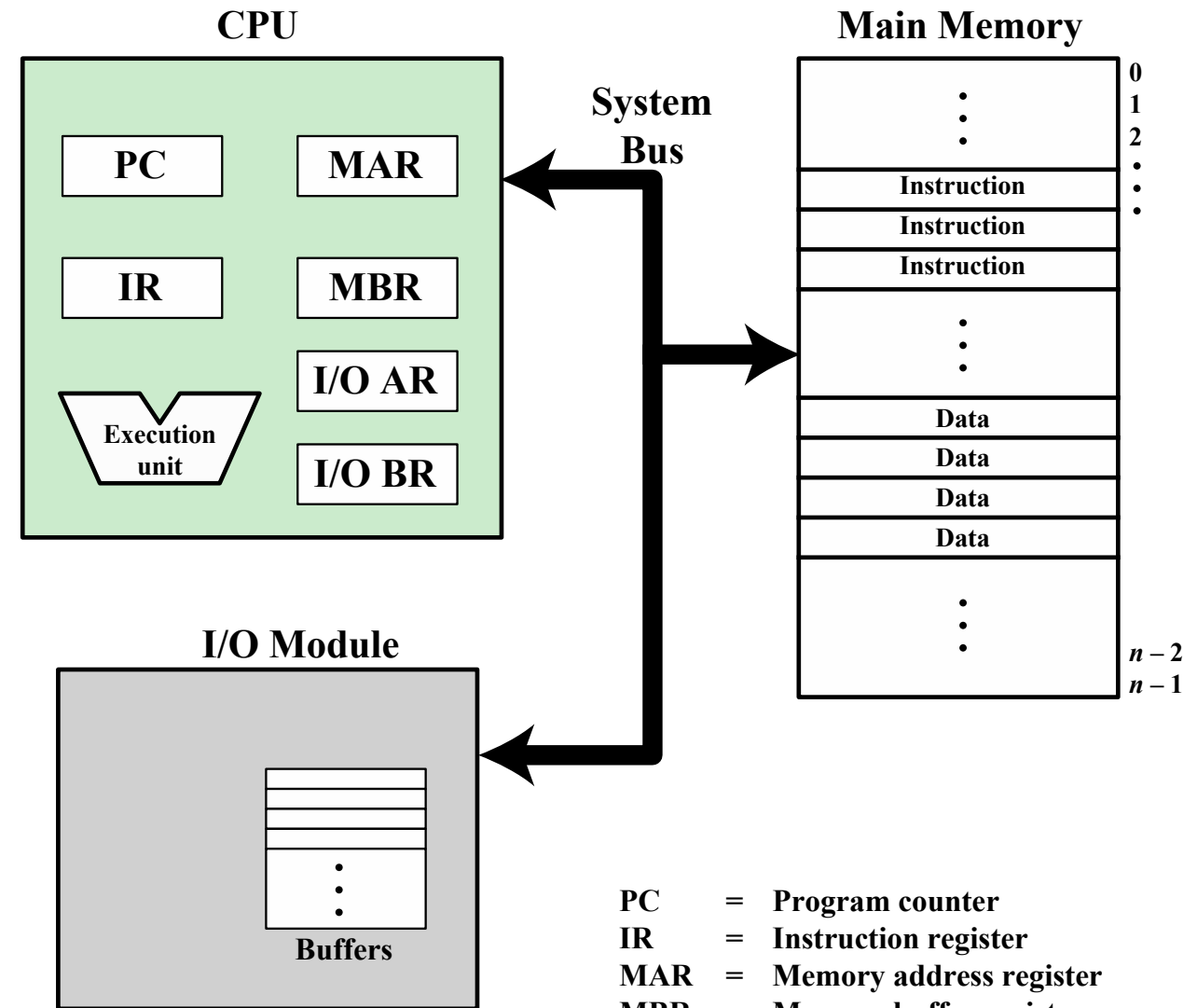
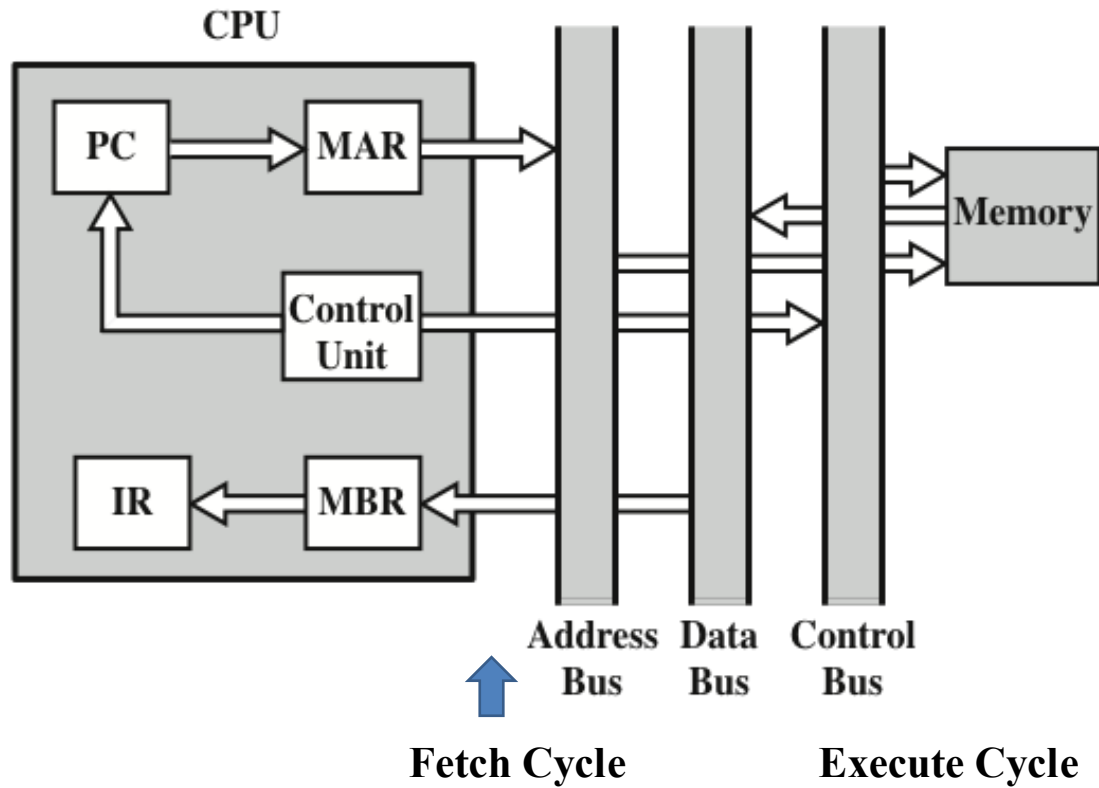


# Program and Process

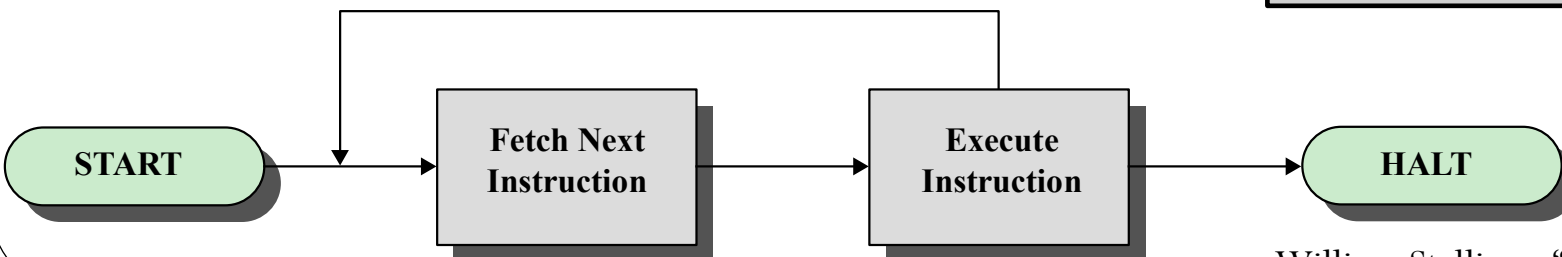
- A unique identifier
- Points CPU program counter to current instruction – Other registers may store operands, return values etc.
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer



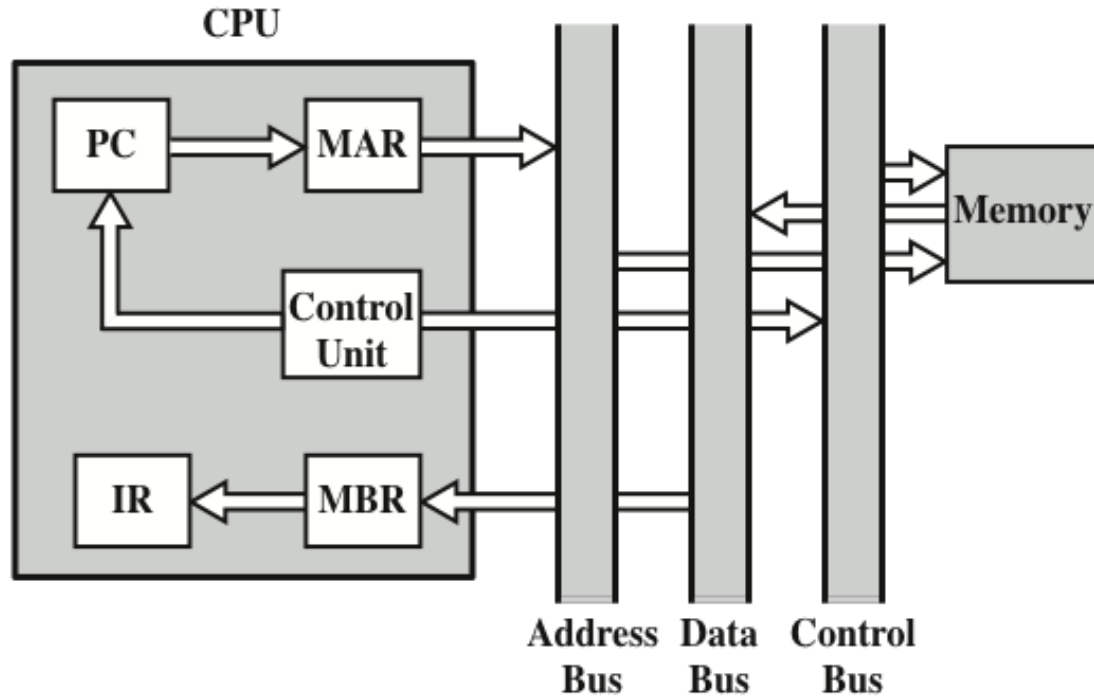
# CPU and Memory



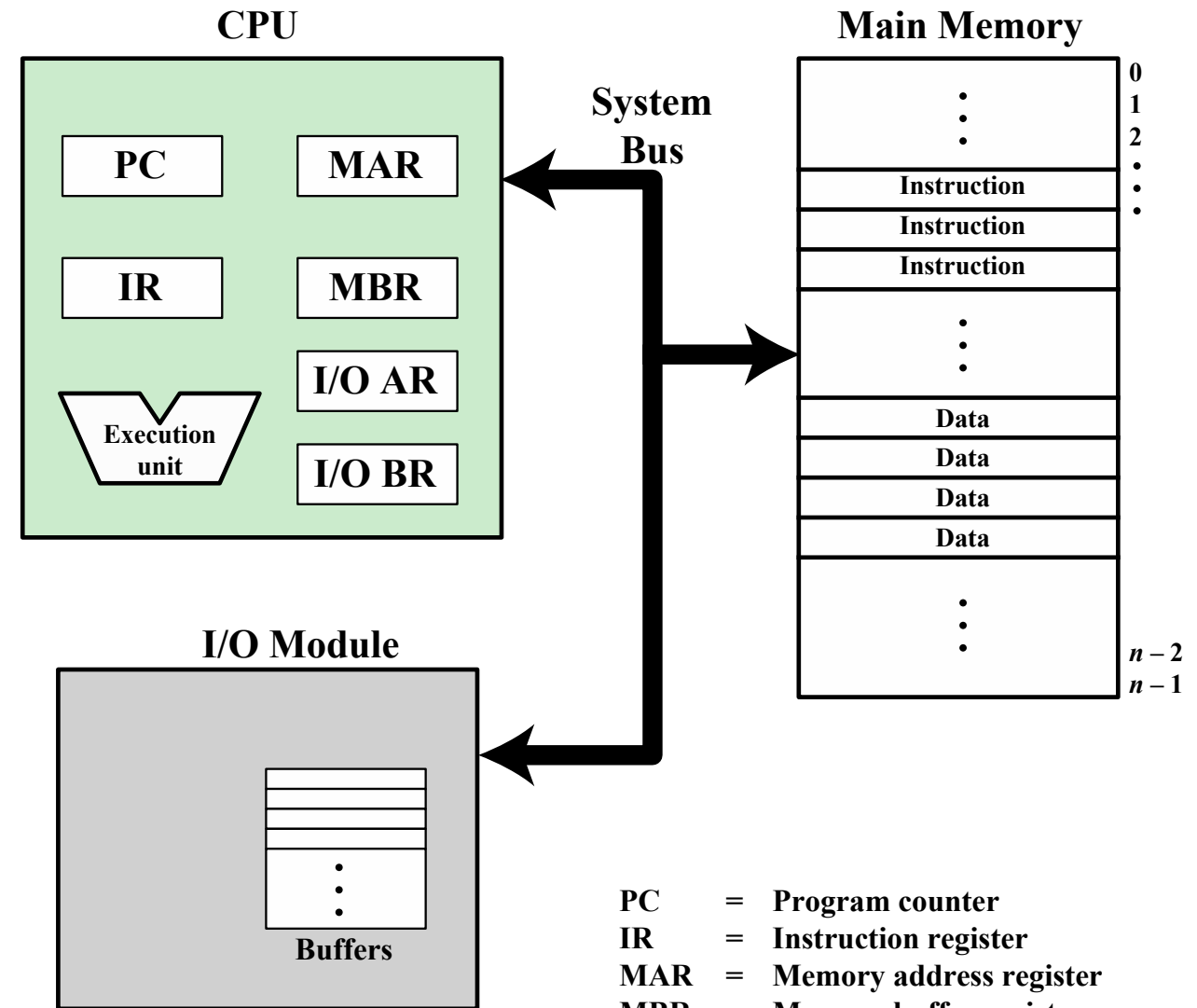
PC = Program counter  
 IR = Instruction register  
 MAR = Memory address register  
 MBR = Memory buffer register  
 I/O AR = Input/output address register  
 I/O BR = Input/output buffer register



# CPU and Memory



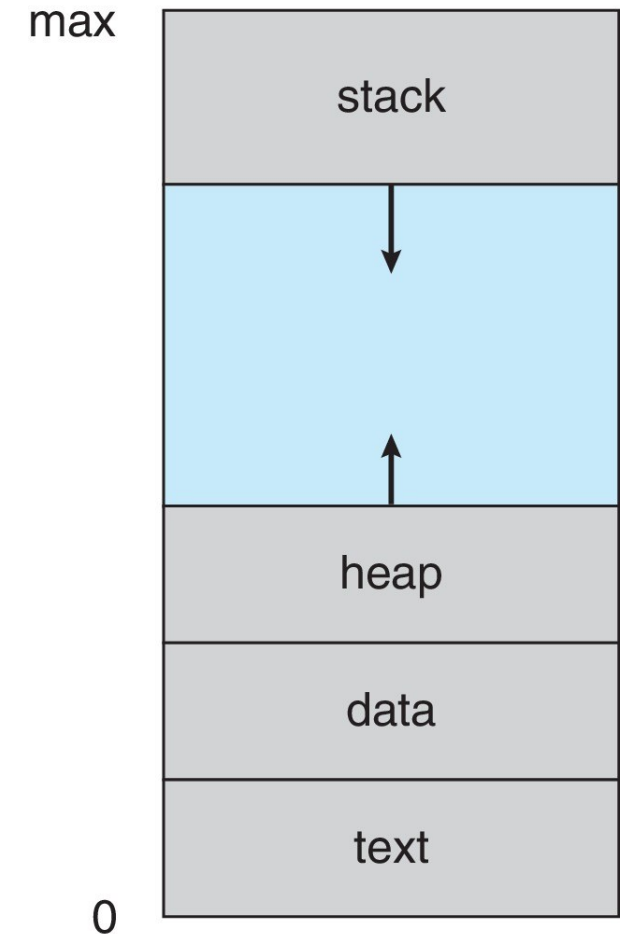
t1 : MAR ← PC  
 t2 : MBR ← MEMORY  
 PC ← (PC) + I  
 t3 : IR ← (MBR)



PC = Program counter  
 IR = Instruction register  
 MAR = Memory address register  
 MBR = Memory buffer register  
 I/O AR = Input/output address register  
 I/O BR = Input/output buffer register

# Program and Process

- OS allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
  - Opens basic files – STD IN, OUT, ERR
  - Initializes CPU registers – PC points to first instruction
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)



# Basic Data Structures

# Programing (Coding)

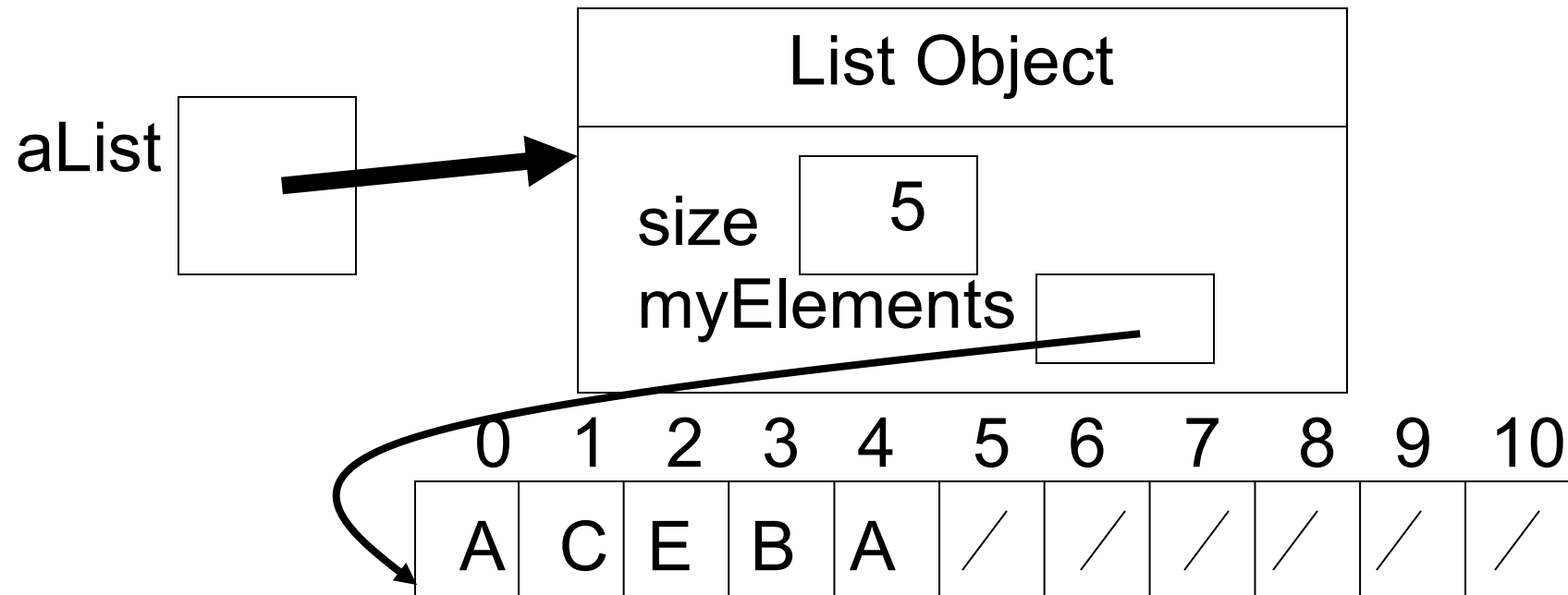
- All programs (or process) has data
  - To *store, display, gather*
  - In the form of *information, numbers, images, sound*
- Programmer decide to structure data
- Several option leads to program development
- Program properties
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

# Code Implementation

- Theoretically
  - abstract base class describes ADT
  - inherited implementations implement data structures
  - can change data structures transparently (to client code)
- Practice
  - different implementations sometimes suggest different interfaces  
(generality vs. simplicity)
  - performance of a data structure may influence form of client code  
(time vs. space, one operation vs. another)

# Data Structures

- A *Data Structure* is:
  - an implementation of an abstract data type *and*
  - "An organization of information, usually in computer memory", for better algorithm efficiency."





# Data structures

Ideal data structure:

fast, elegant, memory efficient

Generates tensions:

- time *vs.* space
- performance *vs.* elegance
- generality *vs.* simplicity
- one operation's performance *vs.* another's

## Dictionary ADT

- list
- binary search tree
- AVL tree
- Splay tree
- Red-Black tree
- hash table

# Data structures

- Data Structures are containers:
  - they hold other data
  - arrays are a data structure
  - ... so are lists
- Other types of data structures:
  - stack, queue, tree, binary search tree, hash table, dictionary or map, set, and on and on
  - [www.nist.gov/dads/](http://www.nist.gov/dads/)
  - [en.wikipedia.org/wiki/List\\_of\\_data\\_structures](http://en.wikipedia.org/wiki/List_of_data_structures)
- Different types of data structures are optimized for certain types of operations



# Data Structures Operations

- Data Structures will have 3 core operations
  - a way to add things
  - a way to remove things
  - a way to access things
- Details of these operations depend on the data structure
  - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do

# Built-in Data Type

- There are five basic data types associated with variables
  - int - integer: a whole number
  - float - floating point value: a number with a fractional part
  - double - a double-precision floating point value
  - char - a single character
  - void - valueless special purpose type
- `int a = 4000;` // positive integer data type
- `float b = 5.2324;` // float data type
- `char c = 'Z';` // char data type
- `long d = 41657;` // long positive integer data type
- `long e = -21556;` // long -ve integer data type
- `int f = -185;` // -ve integer data type
- `short g = 130;` // short +ve integer data type
- `short h = -130;` // short -ve integer data type
- `double i = 4.1234567890;` // double float data type
- `float j = -3.55;` // float data type

# Built-in Data Type

## Advantage

- Simple: Really simple! Only FIVE types of data!!
- Easy to handle: Allocation of memory and operations are already defined
- Built-in support by programming language
  - The C library are there to deal with them

## Limitations

- There is a need for storing and handling variety of data types: Image, text, video, etc.
- Limited range
- Waste of memory
- No flexibility
- Error prone programming

# User-Defined Data Types

- User can define their own data type
- Also, called Custom Data Type, Abstract Data Type (ADT), etc.
- All logically related data can be grouped into a form called structure
- Each member into the group may be a built-in data type or any other user defined data type
- No recursion, that is, a structure cannot include itself
- Examples:

Complex number:  $z = x + i y$

Matrices:  $A_{(m \times n)}$

Date: dd/mm/yy

Date: {int dd, int mm, int yy}

# User-Defined Data Types

## **Advantage**

- It is always convenient for handling a group of logically related data items.
- Examples: Student's record, name, roll number, and marks.
- Elements in a set: Used in relational algebra, database, etc.
- A non non-trivial data structure becomes a trivial.
  - Helps in organizing complex data in a more meaningful way

# Abstraction

- Because details of the implementation are hidden.
- When you do some operation on the list, say insert an element, you just call a function.
- Details of how the list is implemented or how the insert function is written is no longer required.



# Abstract Data Types

- Algorithm: Description of a step-by-step process to solve a problem
  - independent of High Level Language (HLL)
- Data Structure
  - A set of algorithms which implement an Abstract Data Type (ADT)
  - Data Structures: Arrays, Linked lists, Stacks, Queues, Matrices, Trees, Graphs
  - Usage: Searching, Sorting
- Abstract Data Type
  - An opportunity for an acronym
  - Mathematical description of an object and the set of operations on the object

# Abstract Data Types

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - develop a data structure for the ADT
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

# Abstract Data Types

- Abstract Data Types (aka ADTs) are descriptions of how a data type will work without implementation details
- Description can be a formal, mathematical description
- Java interfaces are a form of ADTs
  - some implementation details start to creep in

# Abstract Data Types

- Programming languages usually have a library of data structures
  - [Java collections framework](#)
  - [C++ standard template library](#)
  - .Net framework (small portion of VERY large library)
  - Python lists and tuples
  - Lisp lists

# Structure in C

# Defining a Structure

```
struct tag {  
    member 1;  
    member 2;  
    :  
    member m;  
};
```

- **struct** is the required keyword
- **tag** is the name of the structure
- **member 1**, **member 2**,.. are individual member declarations

# Defining a Structure

```
struct student {  
    char name[30];  
    int roll_number;  
    int total_marks;  
    char dob[10];  
};
```

```
struct student s1, sList[100];
```



A new data-type

- Each member in a structure can be accessed with (.) operator called scope resolution operator

```
s1.name; sList[5].roll_number;
```

# Structures (records)

```
struct {  
    char name[10];  
    int age;  
    float salary;  
} person;
```

```
strcpy(person.name, "james");  
person.age=10;  
person.salary=35000;
```

```
typedef struct human_being {  
    char name[10];  
    int age;  
    float salary;  
};
```

or

```
typedef struct {  
    char name[10];  
    int age;  
    float salary;  
} human_being;
```


```
human_being person1, person2;
```



# Set Manipulation Structure

## Set Manipulation


```
struct nodeS {  
    int element;  
    struct nodeS *next;  
};
```



Structure  
definition

```
typedef struct nodeS set;
```

```
set *union (set a, set b);  
set *intersect (set a, set b);  
set *minus (set a, set b);  
void insert (set a, int x);  
void delete (set a, int x);  
int size (set a);
```



Function  
prototypes

# Complex Numbers Structure

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

```
main()
```

```
{
```

```
    struct complex
```

```
    {
```

```
        float real;
```

```
        float complex;
```

```
    } a, b, c;
```

```
    scanf ("%f %f", &a.real, &a.complex);
```

```
    scanf ("%f %f", &b.real, &b.complex);
```

```
    c.real = a.real + b.real;
```

```
    c.complex = a.complex + b.complex;
```

```
}
```

Structure definition  
and  
Variable Declaration

Reading a member variable

Accessing members

Scope  
restricted  
within  
**main()**

# Complex Numbers Structure

```
struct typeX {  
    float re;  
    float im;  
};
```

Structure  
definition

```
typedef struct typeX complex;
```

```
complex *add (complex a, complex b);  
complex *sub (complex a, complex b);  
complex *mul (complex a, complex b);  
complex *div (complex a, complex b);  
complex *read();  
void print (complex a);
```

Function  
prototypes

# Add Two Complex Numbers Structure

```
#include <stdio.h>
typedef struct complex
{
    float real;
    float imag;
} complex;

int main()
{
    complex n1, n2, temp;

    printf("For 1st complex number \n");
    printf("Enter re & im part respectively:\n");
    scanf("%f %f", &n1.real, &n1.imag);

    printf("\nFor 2nd complex number \n");
    printf("Enter re & im part respectively:\n");
    scanf("%f %f", &n2.real, &n2.imag);

    temp = add(n1, n2);
    printf("Sum = %.1f + %.1fi", temp.real, temp.imag);
    return 0;
}
```

```
complex add(complex n1, complex n2)
{
    complex temp;

    temp.real = n1.real + n2.real;
    temp.imag = n1.imag + n2.imag;

    return(temp);
}
```

## OUTPUT

For 1st complex number  
Enter re & im part respectively: 2.3  
4.5

For 2nd complex number  
Enter re & im part respectively: 3.4  
5  
Sum = 5.7 + 9.5i

# Comparing Dates Structure

```
#include <stdio.h>
struct date { int dd, mm, yy;    };

int date_cmp(struct date d1, struct date d2);
void date_print(struct date d);

int main(){
    struct date d1 = {7, 3, 2015};
    struct date d2 = {24, 10, 2015};

    int cmp = date_cmp(d1, d2);
    date_print(d1);
    if (cmp == 0)
        printf(" is equal to");
    else if (cmp > 0)
        printf(" is greater, i.e., later than ");
    else printf(" is smaller, i.e., earlier than");

    date_print(d2);
    return 0;
}
```

```
/* compare given dates d1 and d2 */

int date_cmp(struct date d1, struct date d2) {

    if (d1.dd == d2.dd && d1.mm == d2.mm && d1.yy == d2.yy)
        return 0;
    else if (d1.yy > d2.yy || d1.yy == d2.yy && d1.mm > d2.mm ||
            d1.yy == d2.yy && d1.mm == d2.mm && d1.dd > d2.dd)
        return 1;
    else return -1;
}

/* print a given date */

void date_print(struct date d) {
    printf("%d/%d/%d", d.dd, d.mm, d.yy);
}
```

## OUTPUT:

7/3/2015 is smaller, i.e., earlier than 24/10/2015

# Sparse Matrices – Data Structure

# 1-D Array Representation

- Implementation of the abstract list data structure using programming language
  - “Backing” Data Structure
- Arrays are contiguous memory locations with fixed capacity
- Allow elements of same type to be present at specific positions in the array
- Index in a List can be mapped to a Position in the Array
  - Mapping function from list index to array position

# Matrix Multiplication

// Given 2-D arrays: a[n][n], b[n][n]

// Output 2-D array: c[n][n] initialized to 0

for (i = 0; i < N; i++)

    for (j = 0; j < N; j++)

        for (k = 0; k < N; k++)

            c[i][j] += a[i][k] \* b[k][j];



# Symmetric Matrix

- An  $n \times n$  matrix can be represented using 1-D array of size  $n(n+1)/2$  by storing either the lower or upper triangle of the matrix
- Use one of the methods for a triangular matrix
- Optimization: The elements that are not explicitly stored may be computed from those that are stored.

2	4	6	0
4	1	9	5
6	9	4	7
0	5	7	0

# Sparse Matrices

- Only a small subset of items are populated in matrix
  - Students and courses taken, faculty and courses taught
  - Adjacency matrix of social network graph
  - vertices are people, edges are “friends”
- Rows and columns are people, cell has 0/1 value
- Why not use regular 2-D matrix?
  - 1-D representation
  - Array of arrays representation

# Sparse Matrix

- A matrix is **sparse**
  - if many of its elements are zero
- A matrix that is not sparse is **dense**
- The boundary is not precisely defined
  - Diagonal and tridiagonal matrices are sparse
  - We classify triangular matrices as dense
- Two possible representations
  - array
  - linked list

*s p a r s e*

	7				6	
	7	6	3		4	
	4	3				
4	2					
				3	2	4

**Dense Matrix**

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

**Sparse Matrix**

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

# Array Representation of Sparse Matrix

- The nonzero entries may be mapped into a 1D array in row-major order
- To reconstruct the matrix structure, need to record the row and column each nonzero comes from

0	0	8	0	0	0
0	7	0	0	0	0
0	0	0	0	0	5
0	3	0	0	0	0
0	0	0	0	1	0



Rows	Columns	values
5	6	5
0	2	8
1	1	7
2	5	5
3	1	3
4	4	1

```

0 0 0 2 0 0 1 0
0 6 0 0 7 0 0 3
0 0 0 9 0 8 0 0
0 4 5 0 0 0 0 0
    
```

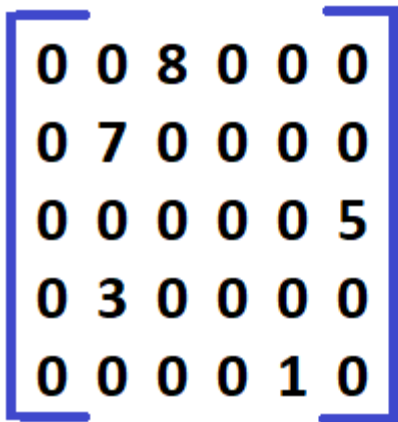
(a) A  $4 \times 8$  matrix

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

(b) Its representation

# Array Representation of Sparse Matrix

```
template<class T>
class Term {
private:
    int row, col;
    T value;
};
```



0	0	8	0	0	0
0	7	0	0	0	0
0	0	0	0	0	5
0	3	0	0	0	0
0	0	0	0	1	0



Rows	Columns	values
5	6	5
0	2	8
1	1	7
2	5	5
3	1	3
4	4	1

```
template<class T>
class sparseMatrix {
private:
    int rows, cols,
    int terms;
    Term<T> *a;
    int MaxTerms;

public:
    // ...
};
```

# Linked Representation of Sparse Matrix

- A shortcoming of the 1-D array of a sparse matrix is that we need to know the number of nonzero terms in each of the sparse matrices when the array is created
- A linked representation can overcome this shortcoming

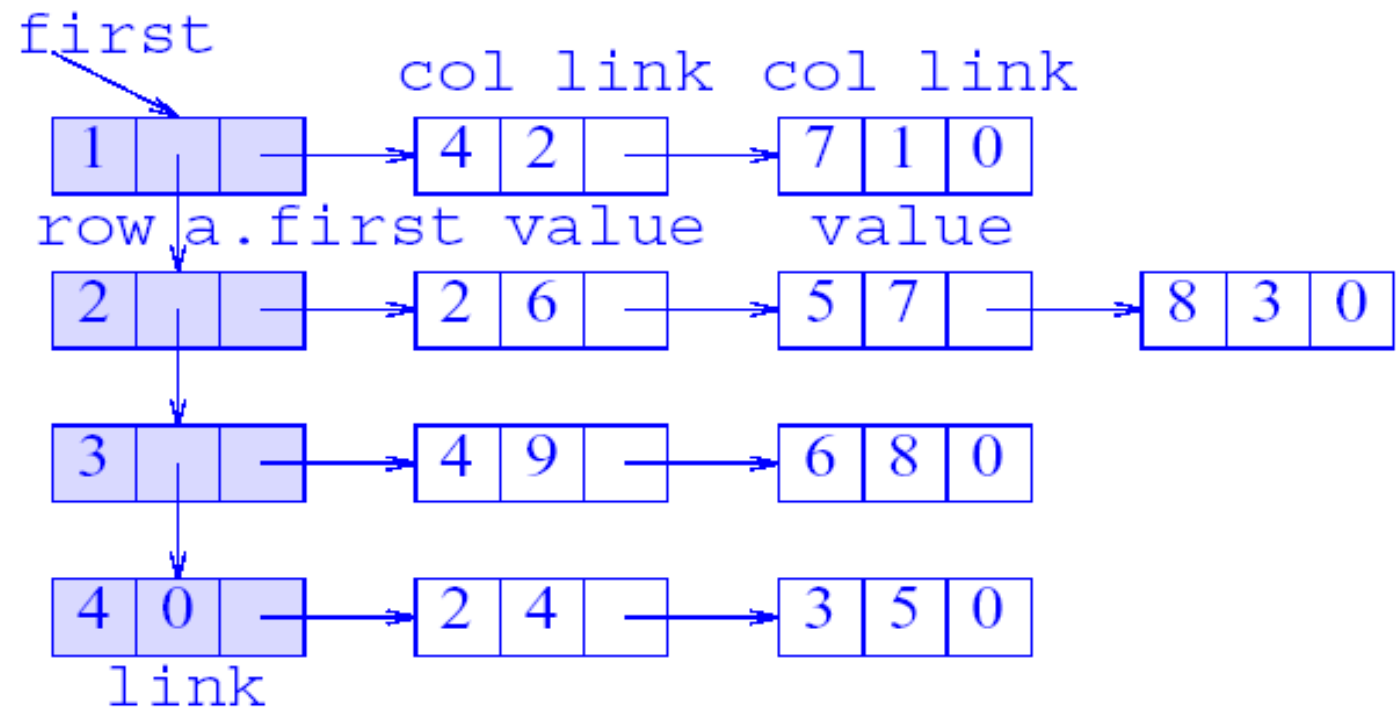
```

0 0 0 2 0 0 1 0
0 6 0 0 7 0 0 3
0 0 0 9 0 8 0 0
0 4 5 0 0 0 0 0
    
```

(a) A  $4 \times 8$  matrix

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

(b) Its representation



# References

- William Stallings. “Computer Organization and Architecture”. 10th Edition
- Mythili Vutukur. Lectures on Operating Systems, Department of Computer Science and Engineering, IIT Bombay, <https://www.cse.iitb.ac.in/~mythili/os/>
- Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts (Tenth Edition). <https://www.os-book.com/OS10/slide-dir/index.html>
- Online textbook [Operating Systems: Three Easy Pieces \(OSTEP\)](#)
- CSE326: Data Structure, Department of Computer Science and Engineering, University of Washington <https://courses.cs.washington.edu/courses/cse326>
- Mike Scott, CS 307 Fundamentals of Computer Science, <https://www.cs.utexas.edu/~scottm/cs307/>
- Debasis Samanta, Computer Science & Engineering, Indian Institute of Technology Kharagpur, Spring-2017, Programming and Data Structures. <https://cse.iitkgp.ac.in/~dsamanta/courses/pds/index.html>

ขอบคุณ

Thai

Grazie  
Italian

תודה רבה  
Hebrew

धन्यवादः  
Sanskrit

ಧನ್ಯವಾದಗಳು  
Kannada

Ευχαριστώ  
Greek

Thank You  
English

Gracias  
Spanish

Спасибо  
Russian

Obrigado  
Portuguese

شكراً  
Arabic

<https://sites.google.com/site/animeshchaturvedi07>

Merci  
French

多謝  
Traditional  
Chinese

धन्यवाद  
Hindi

Danke  
German

多谢  
Simplified  
Chinese

நன்றி  
Tamil

ありがとうございました  
Japanese

감사합니다  
Korean