Programming and Data Structure CS13002

Pabitra Mitra

Dept. of Computer Science & Engineering

pabitra@cse.iitkgp.ernet.in

Room No: 310

Objective of the Course

- To learn programming
  - The logic
  - Style
  - Method

- C Language is being chosen and used just as a medium of expression

About the Course

- L-T-P rating of 3-1-0.
- There is a ***separate*** laboratory 0-0-3
- Individual practice and performance is required – the laboratory will complement the theory classes
- Class attendance is mandatory
- Random checks – may lead to deregistration from the course
- Evaluation in the theory course:
  - Mid-semester (30 %) – 25% + 5 % (regularity and performance)
  - End-semester (50%) – 45% + 5 % (regularity and performance)
  - Two class tests (20%)

Course Materials

- The course materials are available as PowerPoint slides.
- How to get them?
  1. A copy will be kept at the xerox centre so it will be available to all the students. You may choose to bring the handouts to the class and take notes on them

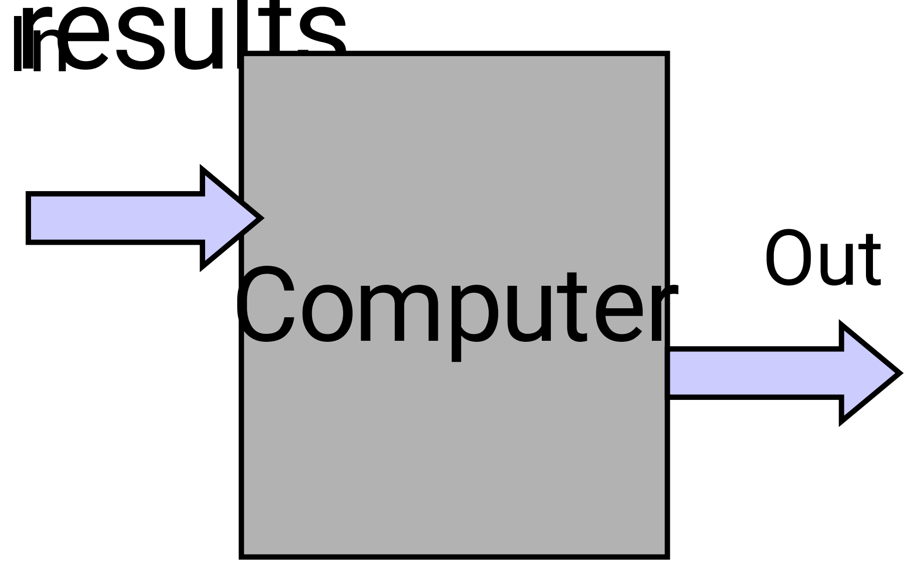
  2. For students having access to Internet, the slides would be available on-line at http://facweb.iitkgp.ernet.in/~pds

Reference Books

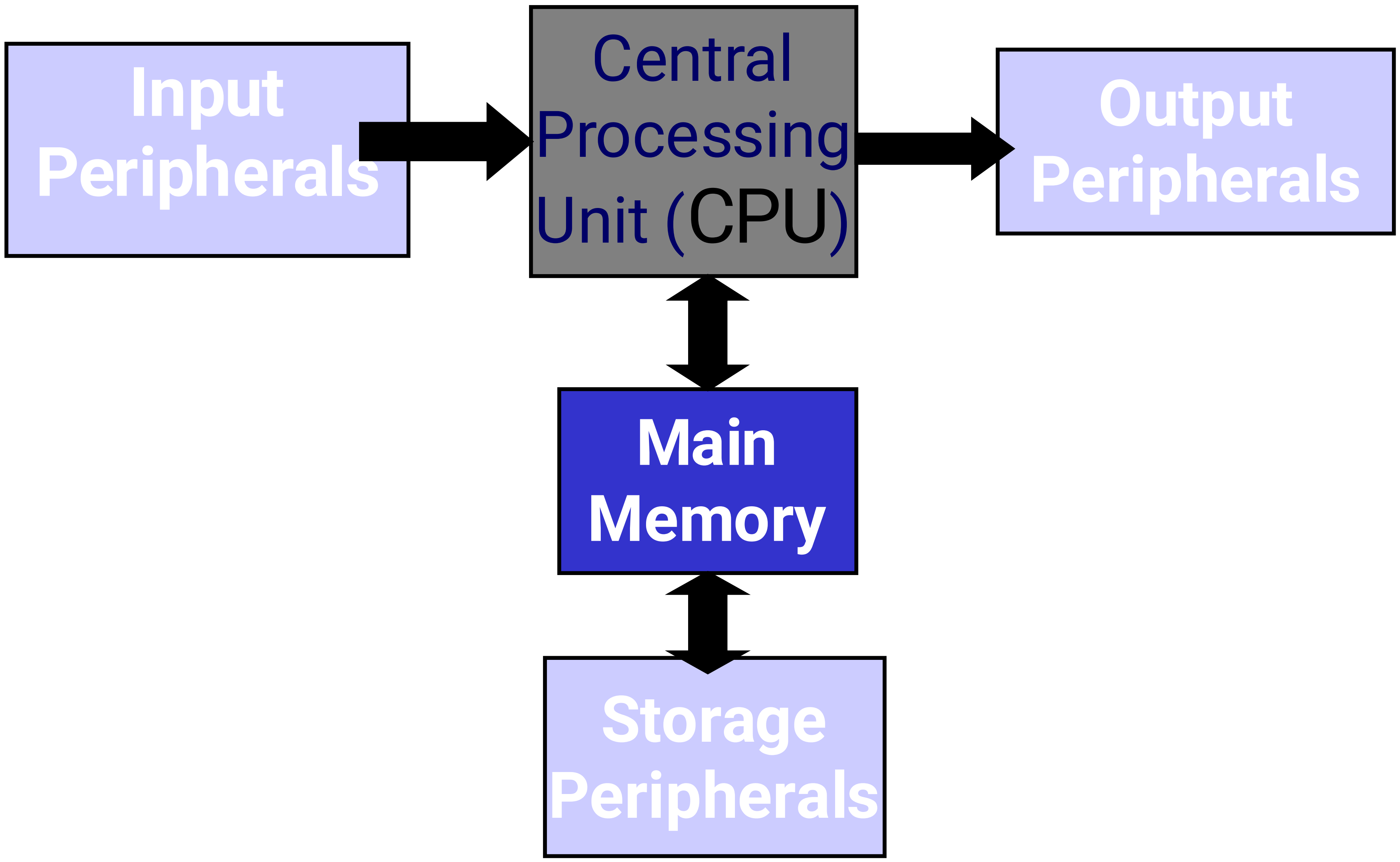
- **Programming With C**
  - **B.S. Gottfried, Schaum's Outline Series, Tata McGraw-Hill.**
- **The C Programming Language,**
  - **B. W. Kernighan & D. M. Ritchie, Prentice Hall**
- A Book on C
  - Al Kelley & Ira Pohl, 4th Edition, Pearson Education, Asia

What is a computer ?

- A computer is a machine which can accept data, process the data and supply results



A computer


Input Peripherals
Central Processing Unit (CPU)
Output Peripherals
Main Memory
Storage Peripherals

Input Devices

- Keyboard
- Mouse
- Joystick
- Scanners (OCR)
- Bar code readers
- Microphones / Sound digitizers
- Voice recognition devices

Output Devices

- VDU / Monitor
- Printers
- Plotters
- Sound cards
- Film and video
- Robot arms

Storage Peripherals

- Magnetic Tape
  - Data stored sequentially (back ups)
- Magnetic Disks
  - Direct (random) access possible
  - Types
    - Hard Disks
    - Floppy Disks
- Optical Disks
  - CDROM
  - CD-RW
- Flash memory – Pen Drives

Typical Configuration of a PC

CPU:	Pentium 4, 2.8GHz
Main Memory:	256 MB
Hard Disk:	40 GB
Floppy Disk:	1.44 MB
CDROM:	52X
Input Device:	Keyboard, Mouse
Output Device:	Color Monitor (17 inch)

How does a computer work?

- Stored program
- A program is a coded form of an Algorithm
- A program is a set of instructions for carrying out a specific task.
- Programs are stored in secondary memory, when created.
- Programs are in main memory during execution.

CPU

- Central Processing Unit (CPU) is where computing takes place in order for a computer to perform tasks.
- CPU’s have large number of registers which temporarily store data and programs (instructions).
- The CPU receives stored instructions, interprets them and acts upon them.

Computer Program

- A program is ultimately
  - a sequence of numeric codes stored in memory which is converted into simple operations (instructions for the CPU).

  This type of code is known as machine code.
- The instructions are retrieved from
  - consecutive (memory) locations

  unless the current instruction tells it otherwise (branch / jump instructions).

Programming Languages

- Machine language
- Assembly Language
  - Mnemonics (opcodes)
- Higher level languages
  - Compiled languages:
    - C, C++, Pascal, Fortran
    - Converted to machine code using compilers
  - Interpreted Languages:
    - Basic,
    - Lisp

Instruction Set

- Start
- Read M
- Write M
- Load Data, M
- Copy M1, M2
- Add M1, M2, M3
- Sub M1, M2, M3
- Compare M1, M2, M3
- Jump L
- J_Zero M, L
- Halt

Program

```
0: Start
1: Read 10
2: Read 11
3: Add 10, 11, 12
4: Write 12
5: Halt
```

Examples of Software

- Read an integer and determine if it is a prime number.
- A Palindrome recognizer
- Read in airline route information as a matrix and determine the shortest time journey between two airports
- Telephone pole placement problem
- Patriot Missile Control
- A Word-processor
- A C language Compiler
- Windows 2000 operating system
- Finger-print recognition
- Chess Player
- Speech Recognition
- Language Recognition
- Discovering New Laws in Mathematics
- Automatic drug discovery

Programming Languages

- Machine language
  - Only the machine understands.
  - Varies from one class of computers to another.
  - Not portable.
- High-level language
  - Easier for the user to understand.
  - Fortran, C, C++, Java, Cobol, Lisp, etc.
  - Standardization makes these languages portable.
    - For example, C is available for DOS, Windows, UNIX, Linux, MAC platforms.

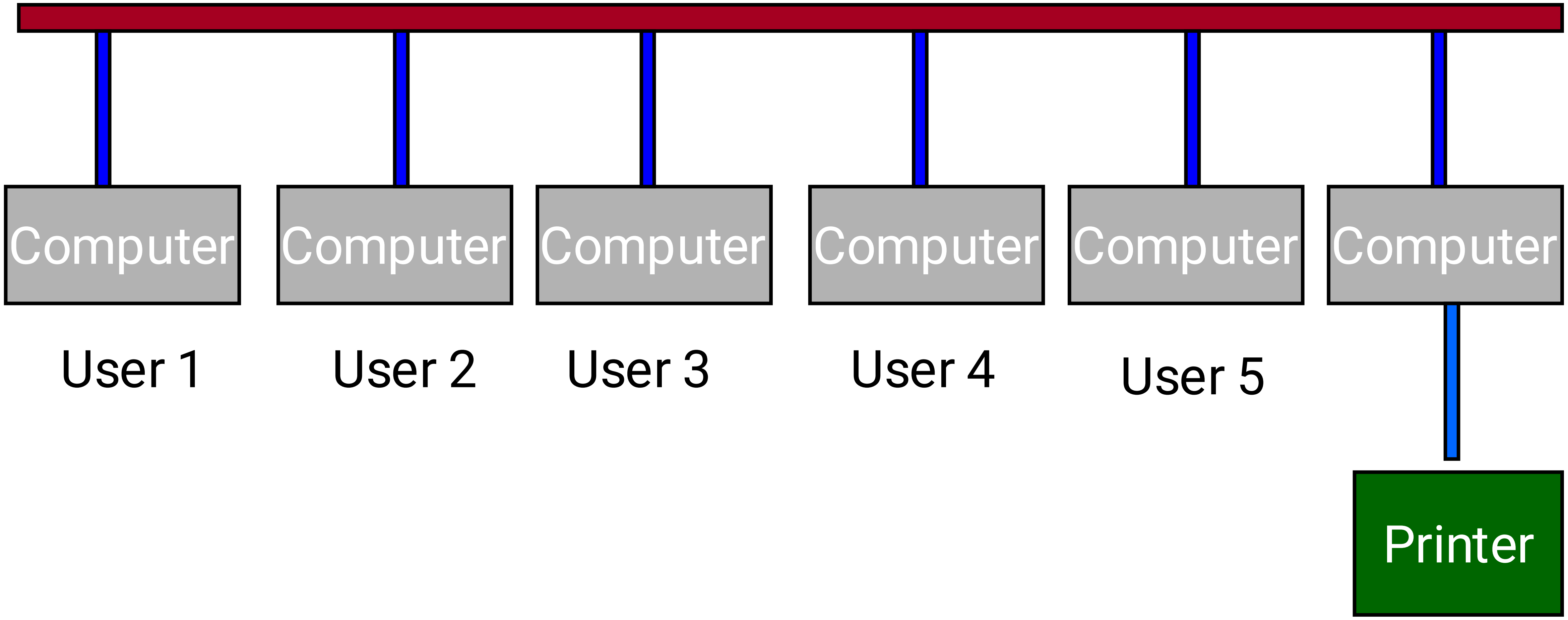
Operating Systems

- Makes the computer easy to use.
  - Basically the computer is very difficult to use.
  - Understands only machine language.
- Categories of operating systems:
  - Single user
  - Multi user
    - Time sharing
    - Multitasking
    - Real time

- DOS is a single-user operating system.
- Windows 95 is a single-user multitasking operating system.
- Unix is a multi-user operating system.
  - Linux is a version of Unix
- Question
  - How multiple users can work on the same computer?

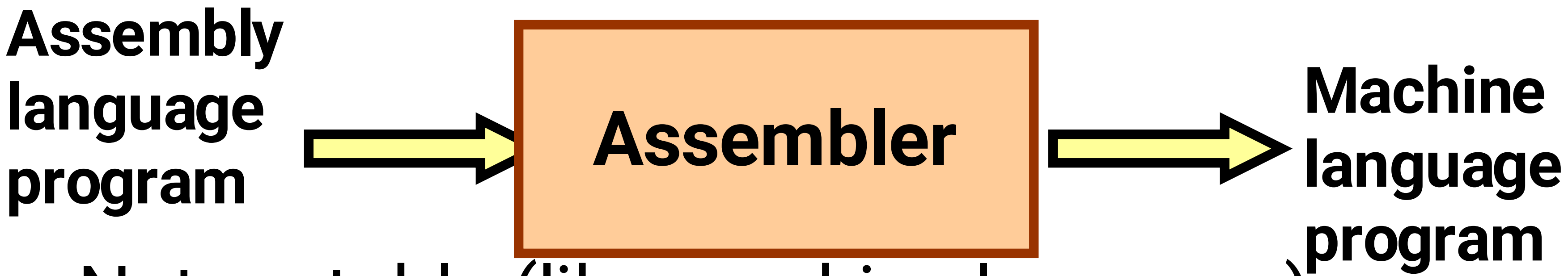
- Computers are often connected in a network.
- Many users may work on a computer.
  - Over the network.
  - At the same time.
  - CPU and other resources are shared among the different programs.

Multi-user environment



Contd.

- Assembly Language
  - Mnemonic form of machine language.
  - Easier to use as compared to machine language.
    - For example, use "ADD" instead of "10110100".



  - Not portable (like machine language).
  - Requires a translator program called assembler.

Contd.

- Assembly language is also difficult to use in writing programs.
  - Requires many instructions to solve a problem.
- Example: Find the average of three numbers.

```
MOV   A,X     ;  A = X
ADD   A,Y     ;  A = A + Y
ADD   A,Z     ;  A = A + Z
DIV   A,3     ;  A = A / 3
```

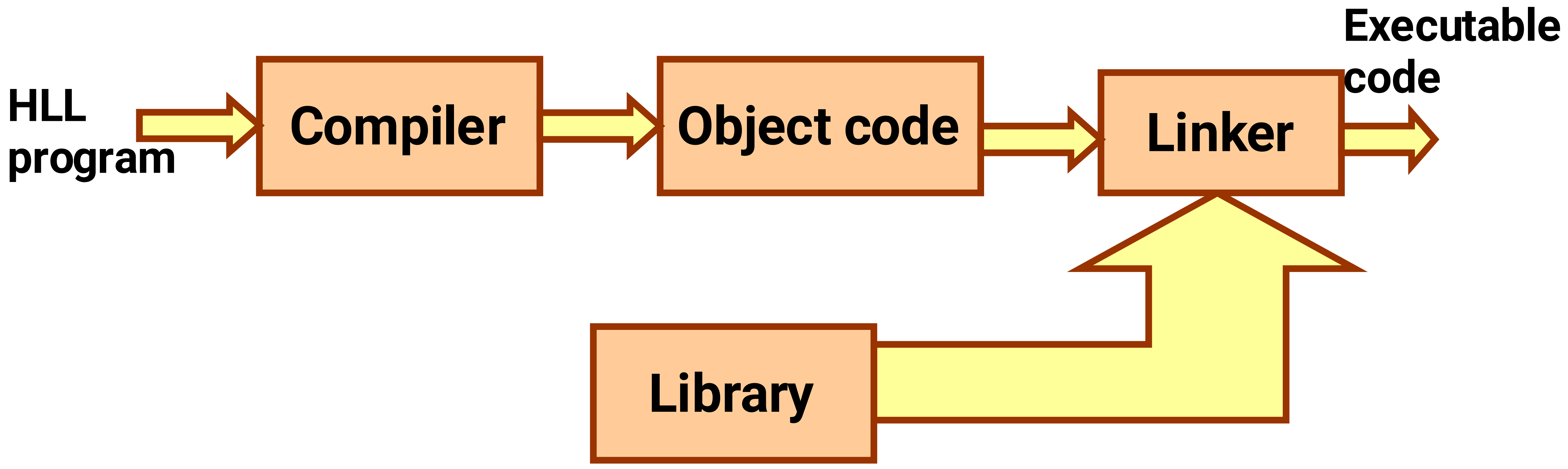
In C,

RES = (X + Y + Z) / 3

High-Level Language

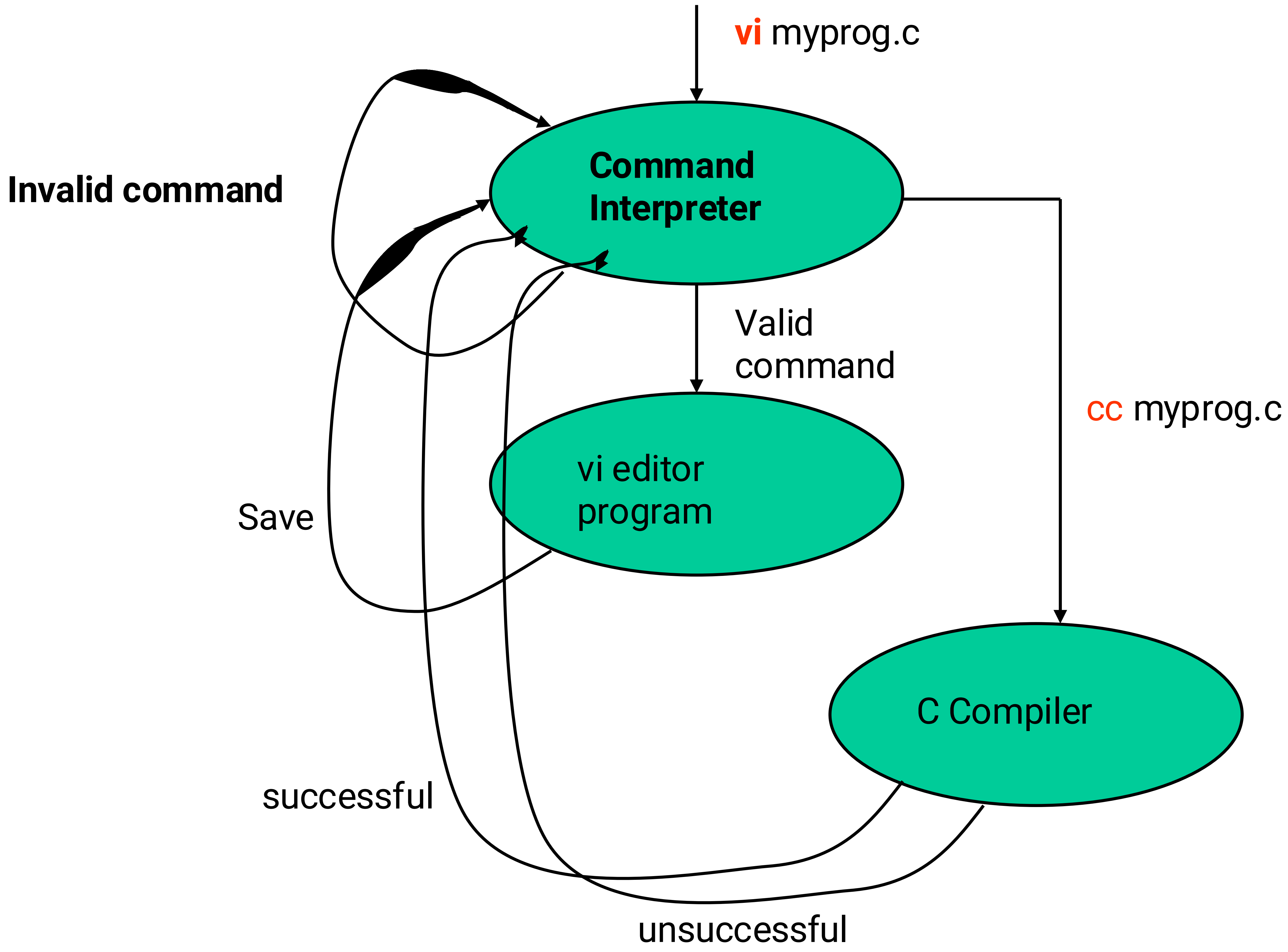
- Machine language and assembly language are called low-level languages.
  - They are closer to the machine.
  - Difficult to use.
- High-level languages are easier to use.
  - They are closer to the programmer.
  - Examples:
    - Fortran, Cobol, C, C++, Java.
  - Requires an elaborate process of translation.
    - Using a software called compiler.

Contd.


HLL program
Compiler
Object code
Linker
Executable code
Library

Role of Operating System

- Accept command
- Initiate relevant system programs if the command is valid


vi myprog.c
Invalid command
Command Interpreter
Valid command
cc myprog.c
vi editor program
Save
C Compiler
successful
unsuccessful

Lab Environment

Number System Basics

Number System :: The Basics

- We are accustomed to using the so-called decimal number system.
  - Ten digits :: 0,1,2,3,4,5,6,7,8,9
  - Every digit position has a weight which is a power of 10.
- Example:

$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 +$
$6 \times 10^{-1} + 7 \times 10^{-2}$

- A digital computer is built out of tiny electronic switches.
  - From the viewpoint of ease of manufacturing and reliability, such switches can be in one of two states, ON and OFF.
  - A switch can represent a digit in the so-called binary number system, 0 and 1.
- A computer works based on the binary number system.

Digital Information

- Computers store all information digitally:
  - Numbers
  - Text
  - Graphics and images
  - Audio
  - Video
  - Program instructions
- In some way, all information is *digitized* – broken down into pieces and represented as numbers

Binary Numbers

- Once information is digitized, it is represented and stored in memory using the binary number system
- A single binary digit (0 or 1) is called a bit.
- A collection of 8 bits is called a byte.
  - 00110010
- Word: Depends on the computer
  - 4 bytes
  - 8 bytes

- An k-bit decimal number
  - Can express unsigned integers in the range

    0 to $10^k - 1$
  - For k=3, from 0 to 999.

- An k-bit binary number
  - Can express unsigned integers in the range

    0 to $2^k - 1$

Variables, Constants, Memory

Variables and constants

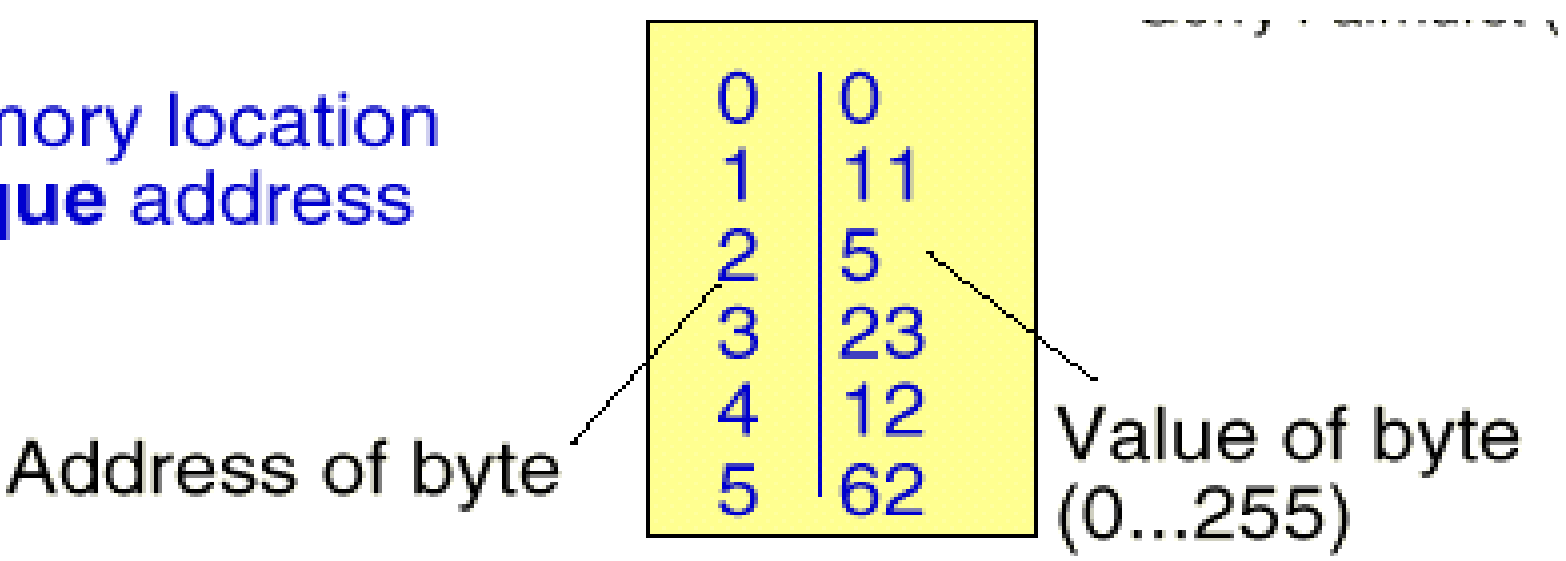
- All temporary variables are stored in variables and constants.
  - The value of a variable can be changed.
  - The value of a constant does not change.
- Variables and constants are stored in main memory.

Memory

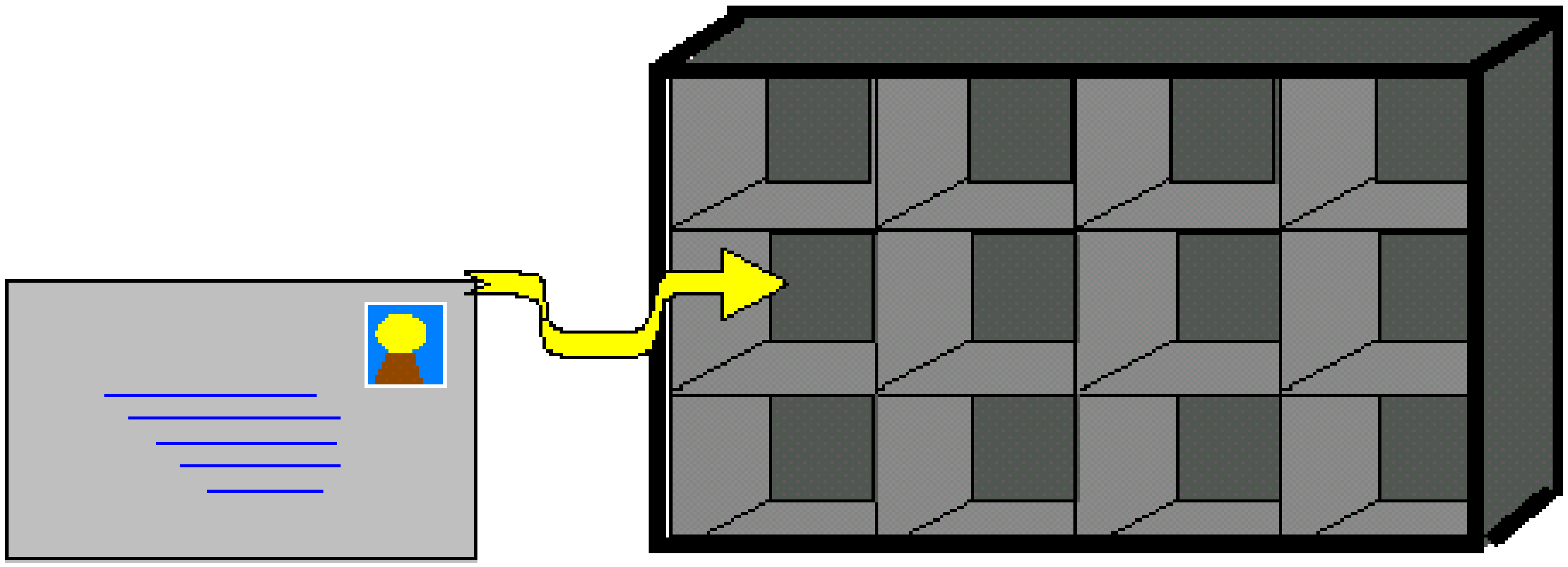
- How does memory look like ?
  - A list of storage locations, each having a unique address
  - Variables and constants are stored in these storage locations.
  - A variable is like a house. The name of the variable is the address of the house.

Address and Values

Every memory location
has a **unique** address

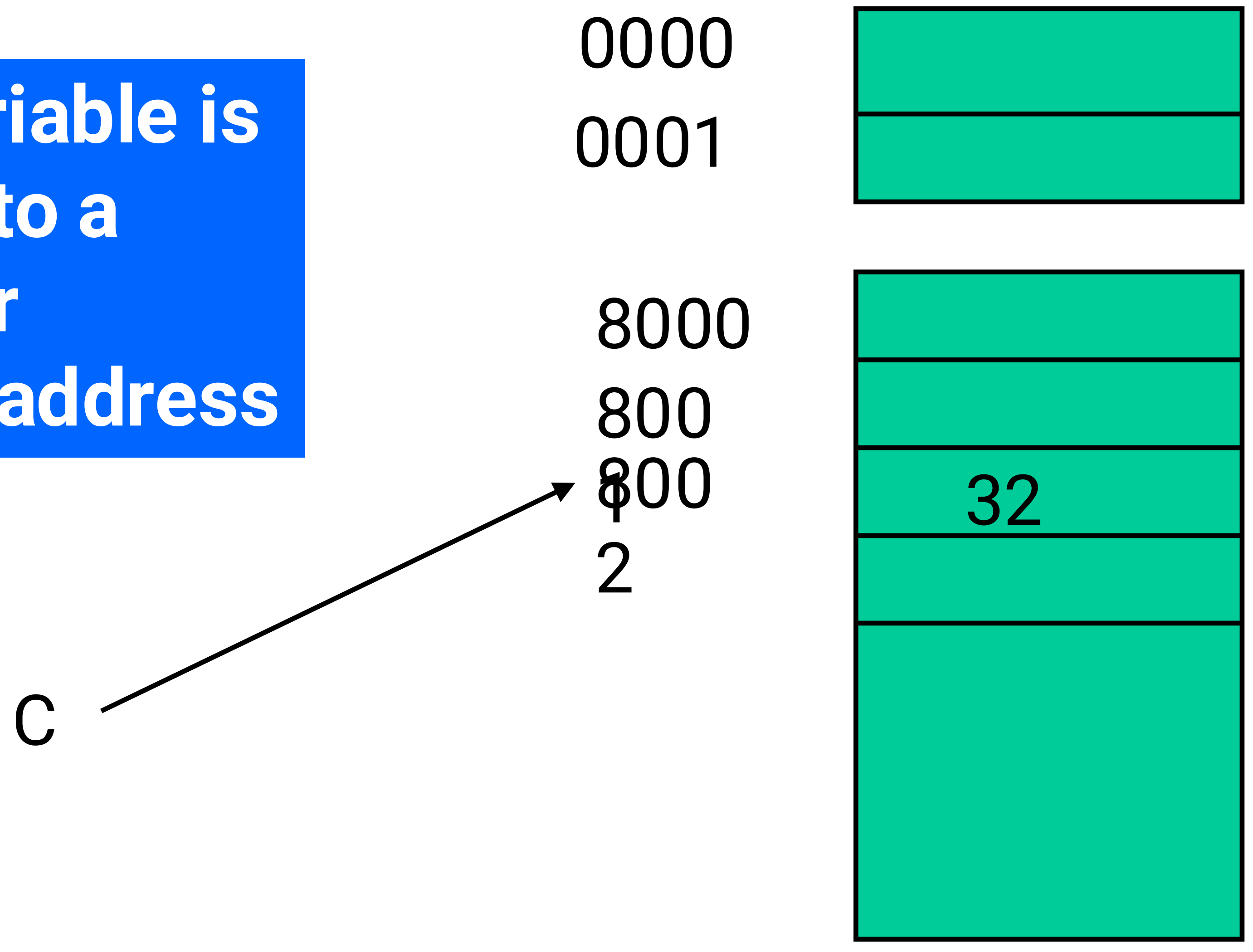


0	0
1	11
2	5
3	23
4	12
5	62

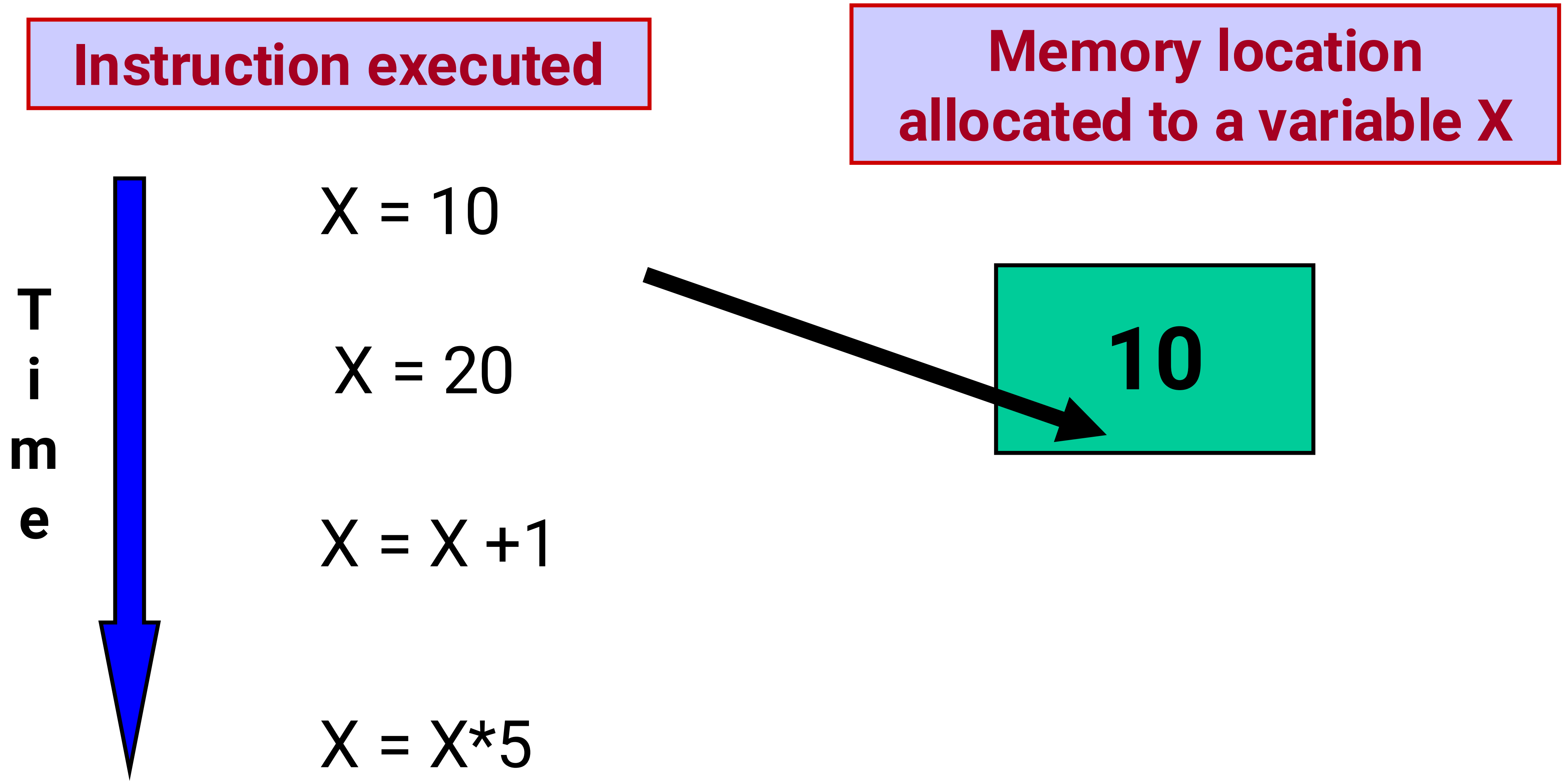


Memory Map

Every variable is mapped to a particular memory address



Variables in Memory



Variables in Memory

Instruction executed

Memory location allocated to a variable X

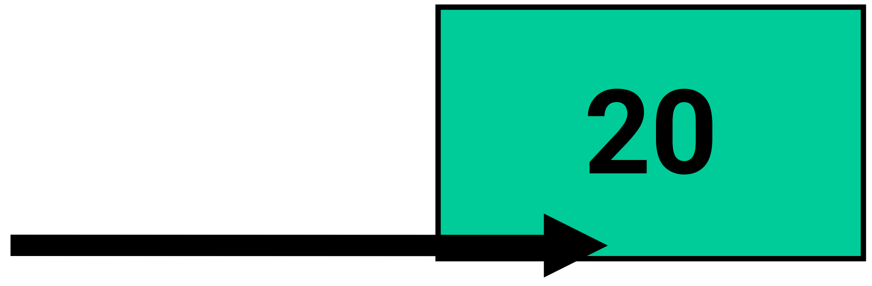
Time

X = 10

X = 20

X = X +1

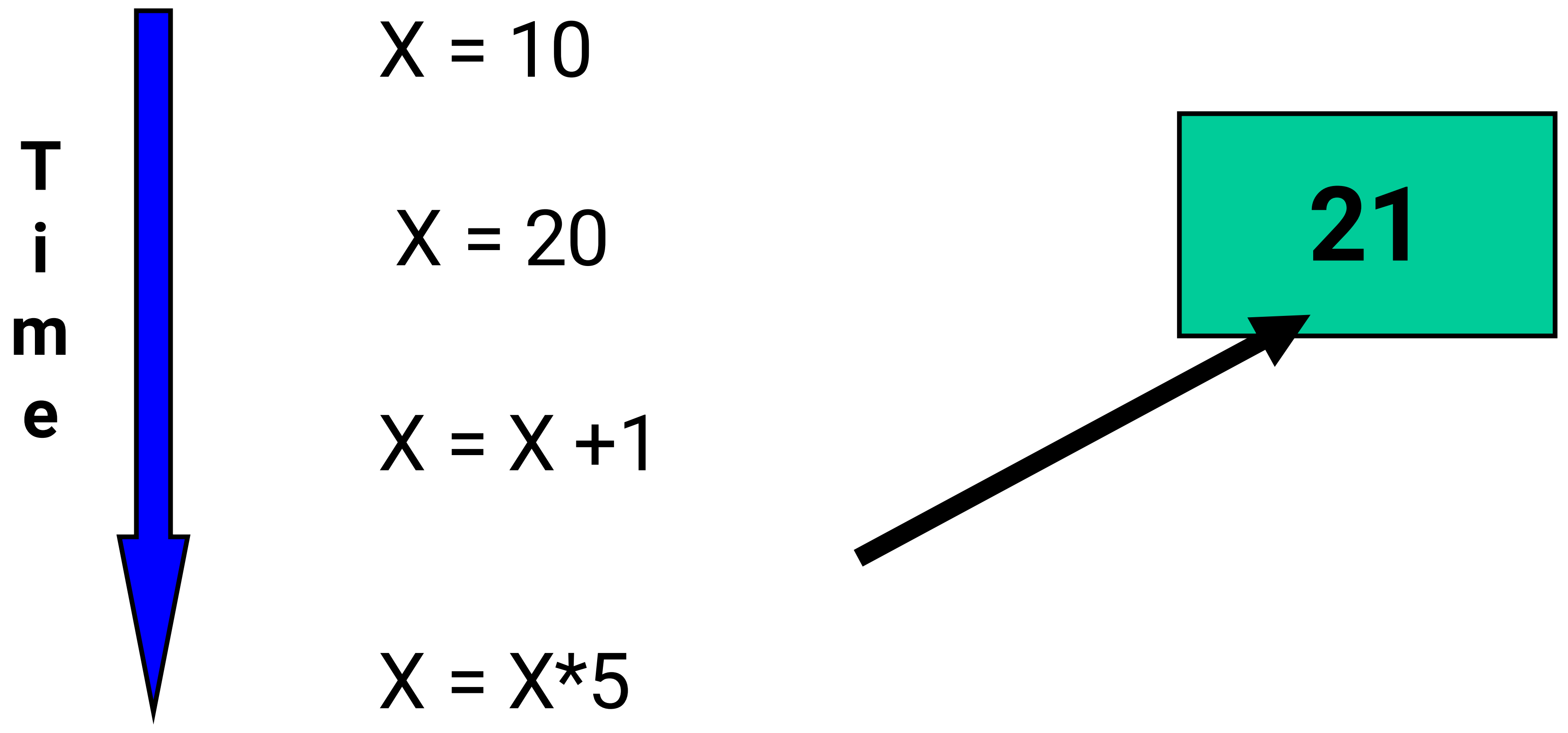
X = X*5



Variables in Memory

Instruction executed

Memory location allocated to a variable X



Variables in Memory

Instruction executed

Memory location allocated to a variable X

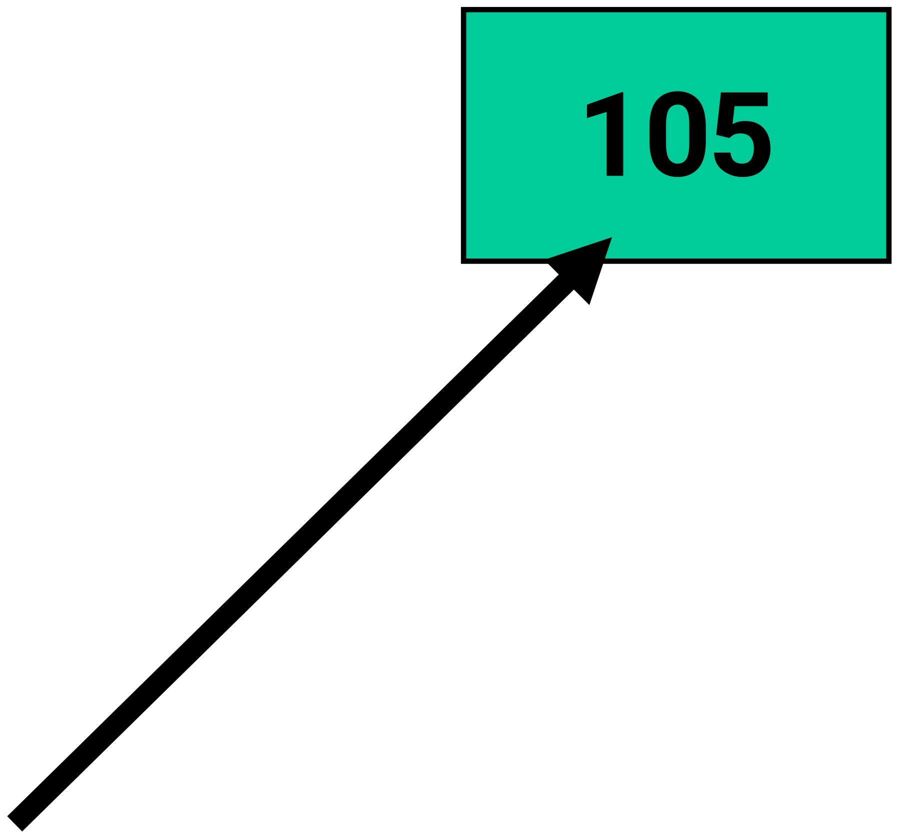
T i m e

X = 10

X = 20

X = X +1

X = X*5



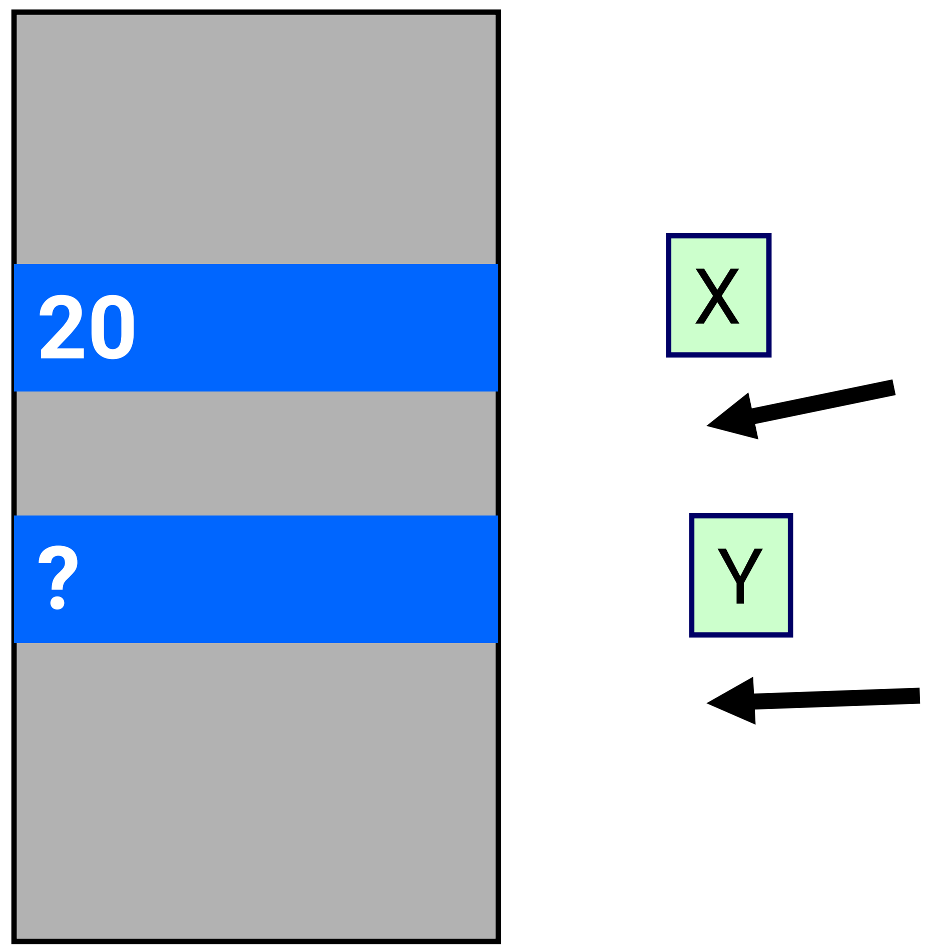
Variables (contd.)

```
X = 20

Y=15

X = Y+3

 Y=x/6
```



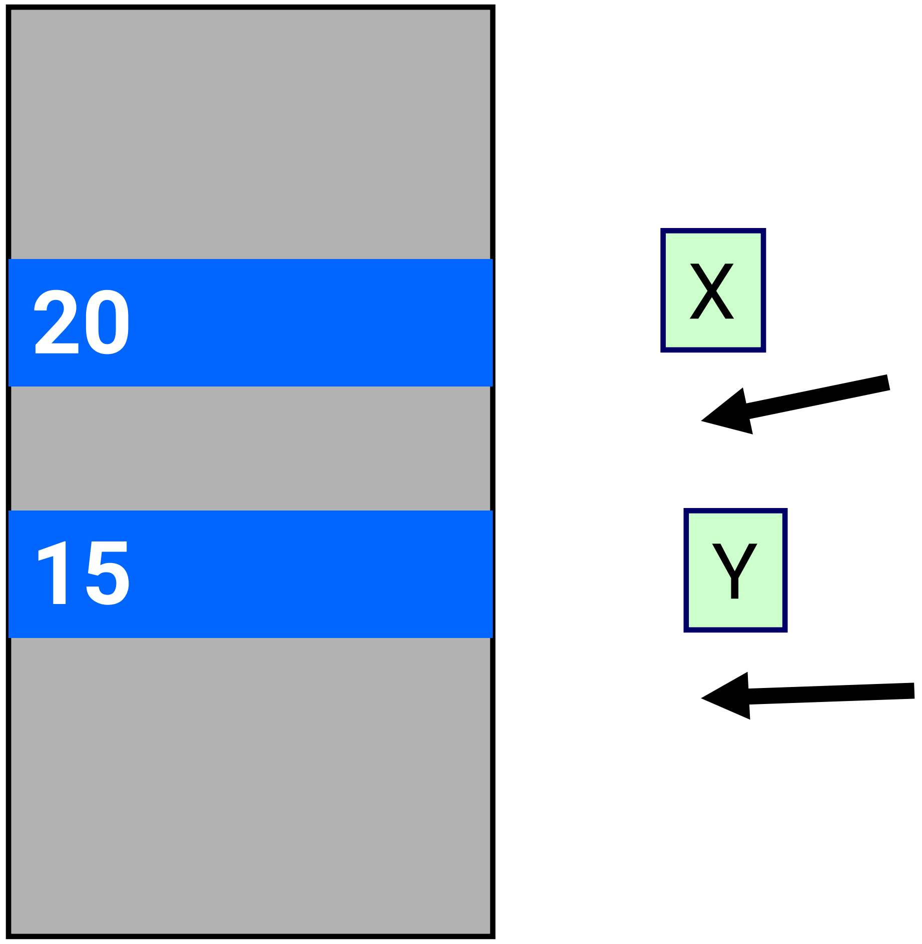
Variables (contd.)

```
X = 20

Y=15

X = Y+3

Y=x/6
```



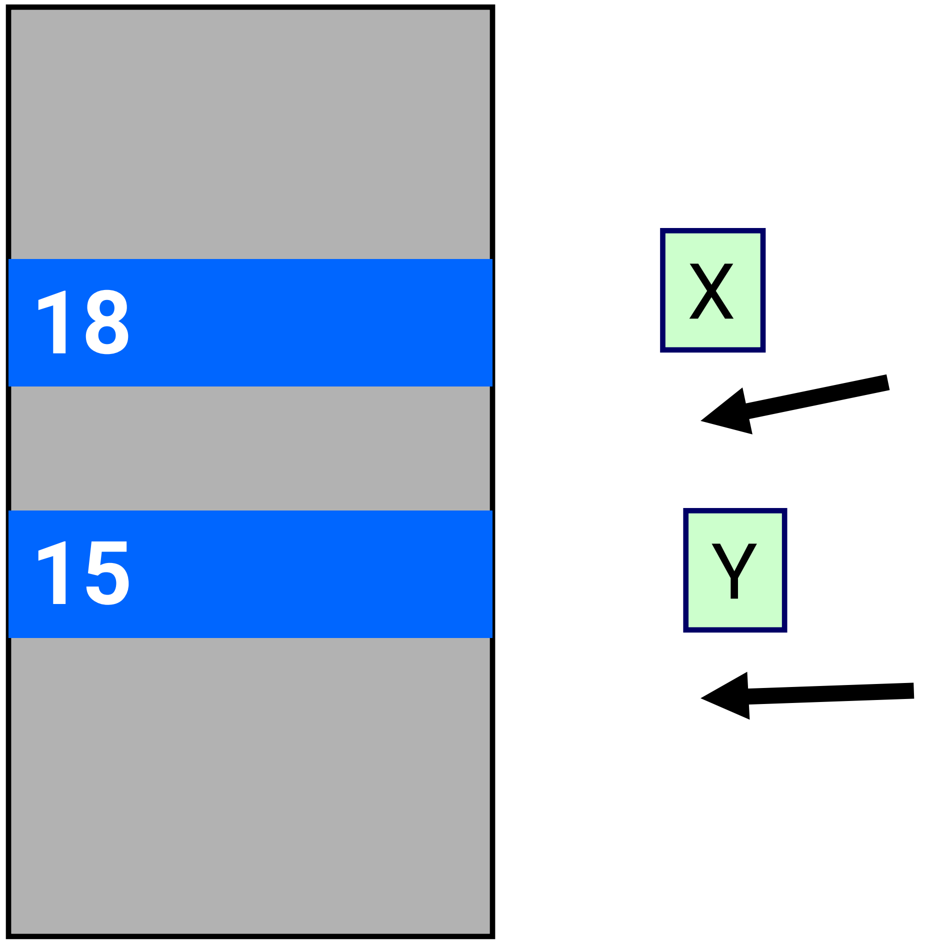
Variables (contd.)

```
X = 20

Y=15

X = Y+3

 Y=x/6
```



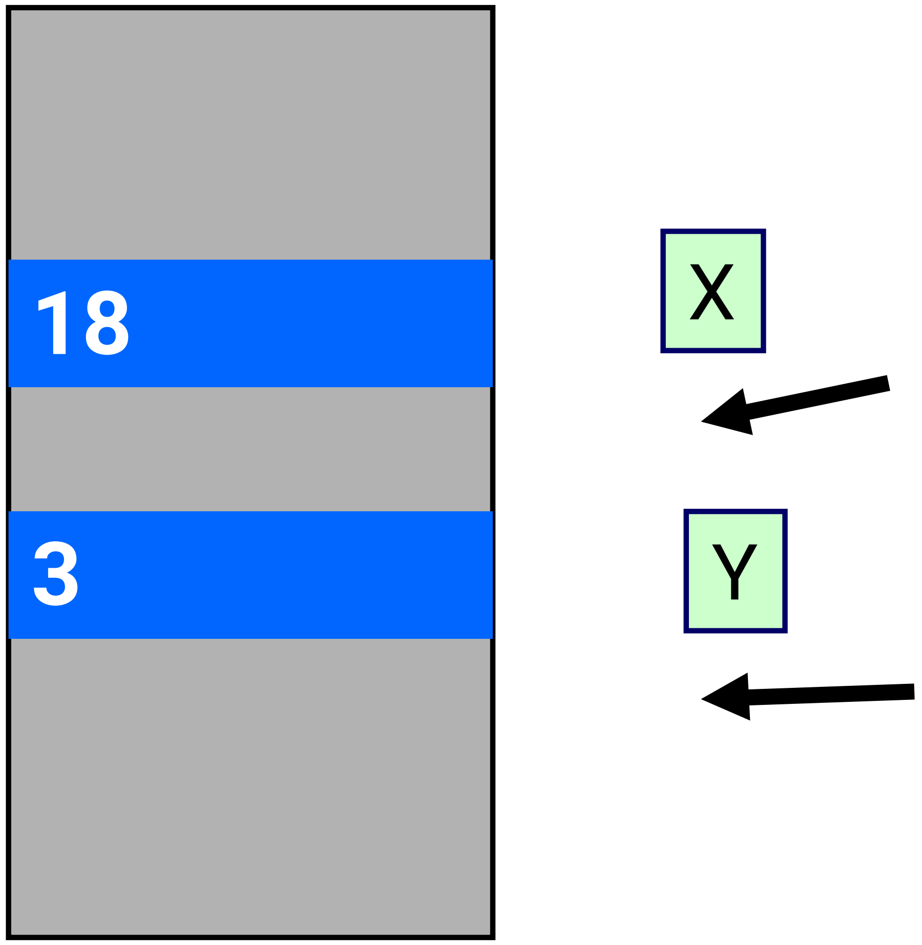
Variables (contd.)

```
X = 20

Y=15

X = Y+3

Y=X/6
```



High-Level Programs

```
Variables x, y;
Begin
Read (x);
Read (y);
If (x >y) then Write (x)
          else  Write (y);
End.
```

```
0: Start
1: Read 20
2: Read 21
3: Compare 20, 21, 22
4: J_Zero 22, 7
5: Write 20
6: Jump 8
7: Write 21
8: Halt
```

Multiplying two integers

```
→   0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	
13	
14	
15	
16	

Multiplying two integers

```
      0: Start
→     1: Read 12
      2: Read 13
      3: Load 0, 14
      4: Load 1, 15
      5: J_Zero 13, 9
      6: Add 12, 14, 14
      7: Sub 13, 15, 13
      8: Jump 5
      9: Write 14
      10: Halt
```

11	
12	5
13	
14	
15	
16	

Multiplying two integers

```
   0: Start
   1: Read 12
→  2: Read 13
   3: Load 0, 14
   4: Load 1, 15
   5: J_Zero 13, 9
   6: Add 12, 14, 14
   7: Sub 13, 15, 13
   8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	6
14	
15	
16	

Multiplying two integers

```
   0: Start
   1: Read 12
   2: Read 13
→3: Load 0, 14
   4: Load 1, 15
   5: J_Zero 13, 9
   6: Add 12, 14, 14
   7: Sub 13, 15, 13
   8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	6
14	0
15	
16	

Multiplying two integers

```
   0: Start
   1: Read 12
   2: Read 13
   3: Load 0, 14
→  4: Load 1, 15
   5: J_Zero 13, 9
   6: Add 12, 14, 14
   7: Sub 13, 15, 13
   8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	6
14	0
15	1
16	

Multiplying two integers

```
       0: Start
       1: Read 12
       2: Read 13
       3: Load 0, 14
       4: Load 1, 15
→      5: J_Zero 13, 9
       6: Add 12, 14, 14
       7: Sub 13, 15, 13
       8: Jump 5
       9: Write 14
       10: Halt
```

11	
12	5
13	6
14	0
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
→ 6: Add 12, 14, 14
  7: Sub 13, 15, 13
  8: Jump 5
  9: Write 14
  10: Halt
```

11	
12	5
13	6
14	5
15	1
16	

Multiplying two integers

```
   0: Start
   1: Read 12
   2: Read 13
   3: Load 0, 14
   4: Load 1, 15
   5: J_Zero 13, 9
   6: Add 12, 14, 14
→  7: Sub 13, 15, 13
   8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	5
14	5
15	1
16	

Multiplying two integers

```
   0: Start
   1: Read 12
   2: Read 13
   3: Load 0, 14
   4: Load 1, 15
   5: J_Zero 13, 9
   6: Add 12, 14, 14
   7: Sub 13, 15, 13
→  8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	5
14	5
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
→   5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	5
14	5
15	1
16	

Multiplying two integers

```
      0: Start
      1: Read 12
      2: Read 13
      3: Load 0, 14
      4: Load 1, 15
      5: J_Zero 13, 9
→     6: Add 12, 14, 14
      7: Sub 13, 15, 13
      8: Jump 5
      9: Write 14
      10: Halt
```

11	
12	5
13	5
14	10
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
→   7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	4
14	10
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
→   8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	4
14	10
15	1
16	

Multiplying two integers

```
        0: Start
        1: Read 12
        2: Read 13
        3: Load 0, 14
        4: Load 1, 15
→       5: J_Zero 13, 9
        6: Add 12, 14, 14
        7: Sub 13, 15, 13
        8: Jump 5
        9: Write 14
        10: Halt
```

11	
12	5
13	4
14	10
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
→   6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	4
14	15
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
→   7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	3
14	15
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
→   8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	3
14	15
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
→   5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	3
14	15
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
→   6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	3
14	20
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
→   7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	2
14	20
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
  6: Add 12, 14, 14
  7: Sub 13, 15, 13
→ 8: Jump 5
  9: Write 14
  10: Halt
```

11	
12	5
13	2
14	20
15	1
16	

Multiplying two integers

```
   0: Start
   1: Read 12
   2: Read 13
   3: Load 0, 14
   4: Load 1, 15
→  5: J_Zero 13, 9
   6: Add 12, 14, 14
   7: Sub 13, 15, 13
   8: Jump 5
   9: Write 14
   10: Halt
```

11	
12	5
13	2
14	20
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
→ 6: Add 12, 14, 14
  7: Sub 13, 15, 13
  8: Jump 5
  9: Write 14
  10: Halt
```

11	
12	5
13	2
14	25
15	1
16	

Multiplying two integers

```
     0: Start
     1: Read 12
     2: Read 13
     3: Load 0, 14
     4: Load 1, 15
     5: J_Zero 13, 9
     6: Add 12, 14, 14
→    7: Sub 13, 15, 13
     8: Jump 5
     9: Write 14
     10: Halt
```

11	
12	5
13	1
14	25
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
    7: Sub 13, 15, 13
→   8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	1
14	25
15	1
16	

Multiplying two integers

```
      0: Start
      1: Read 12
      2: Read 13
      3: Load 0, 14
      4: Load 1, 15
→     5: J_Zero 13, 9
      6: Add 12, 14, 14
      7: Sub 13, 15, 13
      8: Jump 5
      9: Write 14
      10: Halt
```

11	
12	5
13	1
14	25
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
→   6: Add 12, 14, 14
    7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	1
14	30
15	1
16	

Multiplying two integers

```
    0: Start
    1: Read 12
    2: Read 13
    3: Load 0, 14
    4: Load 1, 15
    5: J_Zero 13, 9
    6: Add 12, 14, 14
→   7: Sub 13, 15, 13
    8: Jump 5
    9: Write 14
    10: Halt
```

11	
12	5
13	0
14	30
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
  6: Add 12, 14, 14
  7: Sub 13, 15, 13
→ 8: Jump 5
  9: Write 14
  10: Halt
```

11	
12	5
13	0
14	30
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
→ 5: J_Zero 13, 9
  6: Add 12, 14, 14
  7: Sub 13, 15, 13
  8: Jump 5
  9: Write 14
  10: Halt
```

11	
12	5
13	0
14	30
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
  6: Add 12, 14, 14
  7: Sub 13, 15, 13
  8: Jump 5
→ 9: Write 14
  10: Halt
```

11	
12	5
13	0
14	30
15	1
16	

Multiplying two integers

```
  0: Start
  1: Read 12
  2: Read 13
  3: Load 0, 14
  4: Load 1, 15
  5: J_Zero 13, 9
  6: Add 12, 14, 14
  7: Sub 13, 15, 13
  8: Jump 5
  9: Write 14
→ 10: Halt
```

11	
12	5
13	0
14	30
15	1
16	

The C Programming Language

Why learn C ?

- "Least common denominator" - good building block for learning other languages
  - Subset of C++
  - Similar to JAVA
- Closeness to machine allows one to learn about system-level details
- Portable - compilers available for most platforms
- Very fast

```
/* Program Name : countdown
  Description  : This program prompts the user to type
in a positive number and counts down from that
number to 0, displaying each number      */
#include <stdio.h>
#define STOP 0
main ()
{
      int counter ;      /* Holds intermediate count value
*/
      int startPoint ;  /* Starting point for countdown */
          /* Prompt the user for input */
      printf (" Enter a positive number : ") ;
      scanf ("%d", &startPoint) ;
      for (counter=startPoint; counter >=STOP;counter--
)
               printf ("%d\n", counter) ;
}
```

```
$ cc t1.c
$ ./a.out
Enter a positive number : 6
6
5
4
3
2
1
0
$
```

The first C program

```
#include <stdio.h>
void main ()
{
    printf ("Hello, World! \n") ;
}
```

All programs run from the main function
printf is a function in the library stdio.h
To include any library use #include

Second C program

```
#include <stdio.h>
void main()
{
    int x = 1, y;
    int sum;
    y = 3;
    sum = x + y;    /*    adds x to y, places
                         value in variable sum */
    printf( “%d plus %d is %d\n”, x, y, sum );
}
```

Comments

- Any string of symbols placed between the delimiters /* and */.
- Can span multiple lines
- Can’not be nested! Be careful.
- /* /* /* Hi */   is an example of a comment.
- /* Hi */ */ is going to generate a parse error

Keywords

- Reserved words that cannot be used as variable names
- OK within comments . . .
- Examples: *break, if, else, do, for, while, int, void*
- Exhaustive list in any C book

Identifiers

- A token (word) composed of a sequence of letters, digits, and underscore (_) character. (NO spaces.)
  - First character cannot be a digit
  - C is case sensitive, so beware (e.g. printf ¹ Printf)
- Identifiers such as printf normally would not be redefined; be careful
- Used to give names to variables, functions, etc.
- Only the first 31 characters matter

Constants

0, 77, 3.14 examples.

- Strings: double quotes. “Hello”
- Characters: single quotes. ‘a’ , ‘z’
- Have types implicitly associated with them
- 1234567890999 too large for most machines

Simple Data Types

- Void
- Integer types (signed or unsigned): char, short int, int, long int
  - char is an 8 bit (=1 byte) number
- Floating-point types: float, double, long double
- No boolean types
  - Use 0=False and anything else(usually 1)=True

Input and Output

- **printf** : performs output to the standard output device (typically defined to be the monitor)
  - It requires a format string to which we can provide
    - The text to print out
    - Specifications on how to print the values

    printf ("The number is %d.\n", num) ;

    The format specification %d causes the value listed after the format string to be embedded in the output as a decimal number in place of %d.

Input

- **scanf :** performs input from the standard input device, which is the keyboard by default.
  - **It requires a format string and a list of variables into which the value received from the input device will be stored.**
    - **scanf ("%d", &size) ;**
    - **scanf ("%c", &nextchar) ;**
    - **scanf ("%f", &length) ;**

Variables

- Variables hold the values upon which a program acts. They are the symbolic reference to values.
- The following declares a variable that will contain an integer value.

```
int  num_of_students ;
```

The compiler reserves an integer's worth of memory for num_of_students

In C, all variables must be declared before they can be used.

- A variable declaration conveys three pieces of information
  - the variable's identifier
  - its type
  - its scope - the region of the program in which the variable is accessible.
    (implicitly specified by the place in the code where the declaration occurs.)

C Program # 3

-
  ```
  #include <stdio.h>
  main ()
  {
          int num_of_students ;
          scanf ("%d", &num_of_students) ;
          printf ("%d  \n", num_of_students) ;
  }
  ```

Sample C program #4

```
#include <stdio.h>
#define   PI   3.1415926

/* Compute the area of a circle */
main()
   {
       float   radius, area;
       float   myfunc (float radius);

       scanf (“%f”,   &radius);
       area = myfunc (radius);
       printf (“\n Area is %f \n”,
area);
   }
```

```
float   myfunc (float
r)
   {
       float   a;
       a = PI * r * r;
       /* return result
*/
       return (a);
}
```

Operators and Expressions

Operators

- Operators are used to manipulate variables.
- They perform
  - arithmetic
  - logic functions
  - comparisons between values

```
int x = 6 ;
int y = 9;
int z, w;
z = x + y ;    w = x * y ;
```

Expressions and statements

- Expressions : combine constants and variables with operators
  - x * y
- Expressions can be grouped to form statements
  - z = x * y ;

  Semicolons terminate statements
- One or more simple sentences can be grouped to form a compound sentence or a block by enclosing within { }

Assignment operator

int x = 4 ;

x = x + 9 ;

1. The right hand side is evaluated.
2. The left hand side is set to the value of the right hand side.

All expressions evaluate to a value of a particular type.

x + 9 evaluates to the integer value of 13.

Arithmetic operators

+ : addition

- : subtraction

* : multiplication

/ : division

% : modulus operator

- distance = rate * time ;
- netIncome = income - tax ;
- speed = distance / time ;
- area = PI * radius * radius
- y = a * x * x + b*x + c;
- quotient = dividend/divisor;
- remainder=dividend %divisor;

C Program # 5

```
/* FIND THE LARGEST OF THREE NUMBERS */
main()
{
      int   a, b, c;
       scanf (“%d %d %d”, &a, &b, &c);
       if  ((a>b) && (a>c))     /* Composite condition
    check*/
          printf (“\n Largest is %d”, a);
       else
          if  (b>c) /* return result */

             printf (“\n Largest is %d”, b);
          else
             printf (“\n Largest is %d”, c);
 }
```

Structure of a C program

- Every C program consists of one or more functions.
  - One of the functions must be called main.
  - The program will always begin by executing the main function.

Function

- Each function must contain:
  - A function heading, which consists of the function name, followed by an optional list of arguments enclosed in parentheses.
  - A list of argument declarations.
  - A compound statement, which comprises the remainder of the function.

Function

- Each function must contain:
  - A function heading, which consists of the function name, followed by an optional list of arguments enclosed in parentheses.
  - A list of argument declarations.
  - A compound statement, which comprises the remainder of the function.

Compound Statement

- Each compound statement is enclosed within a pair of braces (‘{‘ and ‘}’).
  - The braces may contain combinations of elementary statements and other compound statements.
- Comments may appear anywhere in a program, enclosed within delimiters
- ‘/*’ and ‘*/’.

Compound Statement (or block)

```
{
        definitions-and-declarations (optional)
        statement-list
}
```

Used for grouping, as function body, and to restrict identifier visibility

Desirable programming style

- Clarity
  - The program should be clearly written.
  - It should be easy to follow the program logic.
- Meaningful variable names
  - Make variable/constant names meaningful to enhance program clarity.
    - ‘area’ instead of ‘a’
    - ‘radius’ instead of ‘r’

Program Documentation

- Insert comments in the program to make it easy to understand.
- Put a comment for each function.
- Put comments for the important variables.
- Do not give too many comments.

Program indentation

- Use proper indentation.
- C has standard indentation conventions.
  - Followed by any book on C
  - Followed in the class

Identifiers

- Identifiers
  - Names given to various program elements (variables, constants, functions, etc.)
  - May consist of letters, digits and the underscore ('_') character, with no space in between.

- First character must be a letter.
- An identifier can be arbitrary long.
  - Some C compilers recognize only the first few characters of the name (16 or 31).
- Case sensitive
  - ‘area’, ‘AREA’ and ‘Area’ are all different
- Examples : number, simple_interest, List
- Non-examples :1stnum, simple interest, no-of-students

Keywords

- Reserved words that have standard, predefined meanings in C.
- Cannot be used as identifiers.
- OK within comments.
- Standard C keywords:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	

Data Types in C

- int : signed integer, typically 2 / 4 bytes

  int numberOfStudents ;

- char: character, typically 1 byte

  char lock; char key = 'Q' ;

- float: floating point number (4 bytes)

  float averageTemp ;

- double: double precision floating point (8 bytes)

  double electrondPerSecond ;

Variations of these types

- short int, longed int, unsigned int

```
short int age;
long int worldPopulation;
unsigned int numberOfDays;
```

- long double

```
long double particlesInUniverse;
```

Values of Data Types

- 2 byte int :
  - -32768 to +32767 (-2^{15} to 2^{15}-1)
- 4 byte int :
  - -2147483648 to +2147483647
- 2 byte unsigned int :
  - 0 to 65535 (2^{16}-1)
- char : 0 to 255
  - 'a', 'A', '+', '=', ...
- float : -2.34, 0.0037, 23.0, 1.234e-5

E or e means "10 to the power of"

Constants

- integer constants:
  - 0, 1, 648, 9999
- floating point constants:
  - 0.2, 12.3, 1.67E+8, 1.12E-12
- character constants:
  - ‘C’, ‘x’, ‘ ‘,
- string constants:
  - “Welcome aboard”, “Rs. 89.95”, “Bye \n”

Ascii value	Character
000	NUL
032	blank
036	$
038	&
043	+
048	0
049	1
057	9
065	A
066	B
090	Z
097	a
098	b
122	z

Escape Sequences: Certain non-printing characters can be expressed in terms of escape sequences:

‘\n’	:	new line
‘\t’	:	horizontal tab
‘\v’	:	vertical tab
‘\\’	:	backslash
‘\”’	:	double quote
‘\0’	:	null

Variables

- It is an identifier
  - used to represent a specified type of information
  - within a designated portion of the program
- The data item must be assigned to the variable at some point of the program
- It can be accessed later by referring to the variable name
- A given variable can be assigned different data items at different places within the program.

```
int a, b, c ;
char  d;
a = 3;
b = 5;
c = a+b;
d = ‘a’ ;
a = 4;
b = 2;
c = a-b;
d = ‘D’ ;
```

a	b	c	d
?	?	?	?
3	?	?	?
3	5	?	?
3	5	8	?
3	5	8	97
4	5	8	97
4	2	8	97
4	2	2	97
4	2	2	68

Declaration of Variables

data-type variable-list

```
int a, b, c;
float root1, root2;
char flag, response;
```

Declaration :

1. specifies the name of the variable
2. Specifies what type of data the variable will hold.

A First Look at Pointers

- A variable is assigned a specific memory location.
  - For example, a variable speed is assigned memory location 1350.
  - Also assume that the memory location contains the data value 100.
  - When we use the name speed in an expression, it refers to the value 100 stored in the memory location.

```
distance = speed * time;
```

- Thus every variable has an address (in memory), and its contents.

Contd.

- In C terminology, in an expression
  - speed refers to the contents of the memory location.
  - &speed refers to the address of the memory location.

- Examples:
  - printf (“%f %f %f”, speed, time, distance);
  - scanf (“%f %f”, &speed, &time);

An Example

```
#include <stdio.h>
main()
{
        float  speed, time, distance;

        scanf (“%f %f”, &speed, &time);
        distance = speed * time;
        printf (“\n The distance traversed is: \n”
,distance);
 }
```

Assignment Statement

- Used to assign values to variables, using the assignment operator (=).
- General syntax:

  variable_name = expression;

- Examples:
  - velocity = 20;
  - b = 15; temp = 12.5;   /* Multiple assign on same line */
  - A = A + 10;
  - v = u + f * t;
  - s = u * t + 0.5 * f * t * t;

Contd.

- A value can be assigned to a variable at the time the variable is declared.
  - int   speed = 30;
  - char flag = ‘y’;
- Several variables can be assigned the same value using multiple assignment operators.
  - a = b = c = 5;
  - flag1 = flag2 = ‘y’;
  - speed = flow = 0.0;

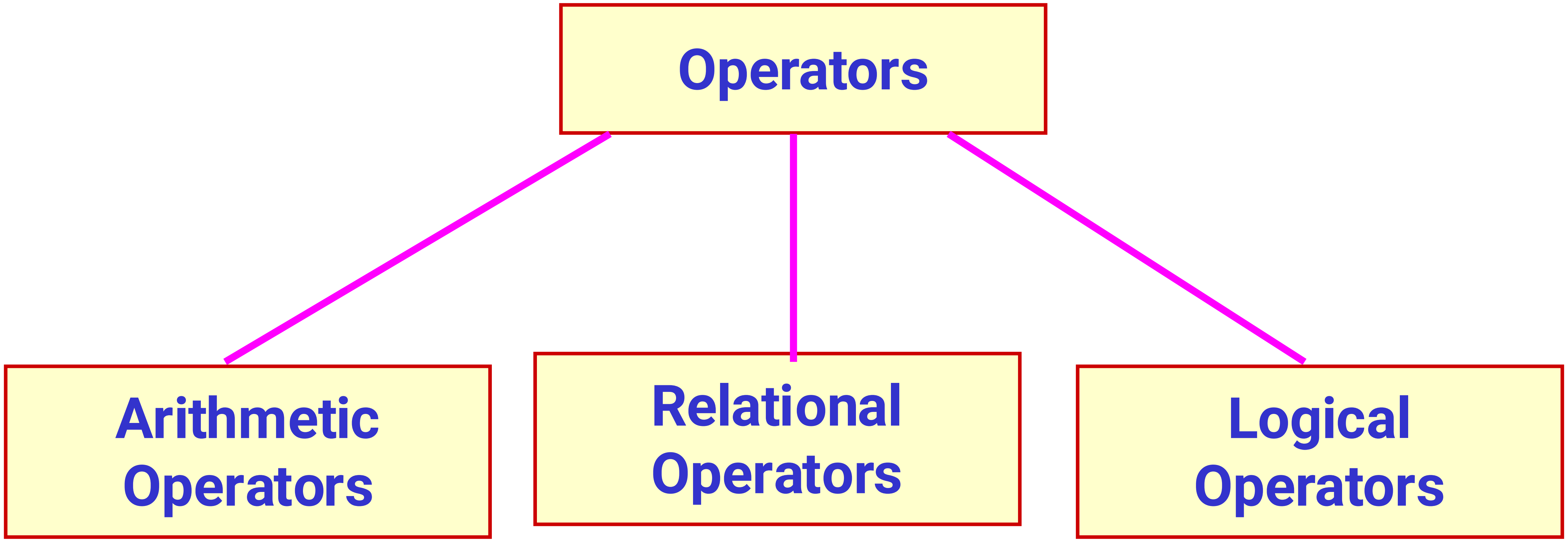
Assignment Statement

- Used to assign values to variables, using the assignment operator (=).
- General syntax:

  variable_name = expression;

- Examples:
  - velocity = 20.5;
  - b = 15; temp = 12;   /* Multiple assign on same line*/
  - A = A + 10;
  - v = u + f * t;
  - s = u * t + 0.5 * f * t * t;

Operators in Expressions



Operator Precedence

- In decreasing order of priority
  1. Parentheses :: ( )
  2. Unary minus :: -5
  3. Multiplication, Division, and Modulus
  4. Addition and Subtraction
- For operators of the same priority, evaluation is from left to right as they appear.
- Parenthesis may be used to change the precedence of operator evaluation.

Examples: Arithmetic expressions

- a + b * c – d / e → a + (b * c) – (d / e)
- a * -b + d % e – f → a * (-b) + (d % e) – f
- a – b + c + d → (((a – b) + c) + d)
- x * y * z → ((x * y) * z)
- a + b + c * d * e → (a + b) + ((c * d) * e)

Integer Arithmetic

- When the operands in an arithmetic expression are integers, the expression is called integer expression, and the operation is called integer arithmetic.
- Integer arithmetic always yields integer values.

Real Arithmetic

- Arithmetic operations involving only real or floating-point operands.
- Since floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the final result.
  - 1.0 / 3.0 * 3.0 will have the value 0.99999 and not 1.0
- The modulus operator cannot be used with real operands.

Mixed-mode Arithmetic

- When one of the operands is integer and the other is real, the expression is called a mixed-mode arithmetic expression.
- If either operand is of the real type, then only real arithmetic is performed, and the result is a real number.
  - 25 / 10 ⊠ 2
  - 25 / 10.0 ⊠ 2.5
- Some more issues will be considered later.

Relational Operators

- Used to compare two quantities.

<	**is less than**
>	**is greater than**
<=	**is less than or equal to**
>=	**is greater than or equal to**
==	**is equal to**
!=	**is not equal to**

Examples

- 10 > 20 is false
- 25 < 35.5 is true
- 12 > (7 + 5) is false

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared.
  - a + b > c – d is the same as (a+b) > (c+d)

Logical Operators

- Logical operators act upon logical expressions
  - && : and (true if both operands are true)
  - || : or (true if either or both operands true
  - ! : negates the value of the logical expression
- Example
  - (n >= lo_bound) && (n <= upper_bound)
  - ! (num > 100)

Example: Logical Operators

```
int main ()   {
      int i, j;
      for (i=0; i<2; i++)    {
            for (j=0; j<2; j++)
                  printf (“%d AND %d = %d,
                        %d OR %d=%d\n”,
                                    i,j,i&&j, i,j, i||j) ;
         }
}
```

```
$ ./a.out
0 AND 0 = 0          0 OR 0 = 0
0 AND 1 = 0          0 OR 1 = 1
1 AND 0 = 0          1 OR 0 = 1
1 AND 1 = 1          1 OR 1 = 1
$
```

```
int main ()  {
        int amount ;   /* The no of bytes to be transferred */
        int rate ;        /* The average network transfer rate */
        int time;         /* The time, in seconds, for the transfer */
        int hours, minutes, seconds; /* The no of hrs,mins,secs for the tr

        printf  ("How many bytes of data to be transferred ?\n") ;
        scanf ("%d", &amount) ;
        printf ("What is the average transfer rate in bytes/sec ?\n") ;
        scanf ("%d", &rate) ;
        time = amount / rate ;
        hours = time / 3600 ;
        minutes = (time % 3600) / 60 ;
        seconds = ((time % 3600) % 60) /60 ;
        printf ("The expected time is %dh %dm %ds\n",
                    hours,minutes,seconds);
}
```

C’s special operators

- ++ and -- : a trademark of C programming
- ++ : increments a variable ;
- -- : decrements a variable
- x++
  - In an expression, value of this expression is the value of x prior to increment
- ++x
  - In an expression, value of this expression is the value of x after the increment.

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

y=4, x=5 after evaluation

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

y=5, x=5 after evaluation

x += 5	equivalent to	x = x + 5
h %= f	equivalent to	h = h%f
product *= num	equivalent to	product = product * num

```
void main ()
{
        int x = 10;
        printf (“ x = %d\n”, ++x) ;
        printf (“x = %d\n”, x++) ;
}
```

Question : What will get printed ?

Exercise

- Suppose your program contains two integer variables, x and y which have values 3 and 4 respectively, Write C statements that will exchange the values in x and y such that after the statements are executed, x is equal to 4 and y is equal to 3.
  - First, write this routine using a temporary variable for storage.
  - Now re-write this routine without using a temporary variable for storage.

Control Constructs

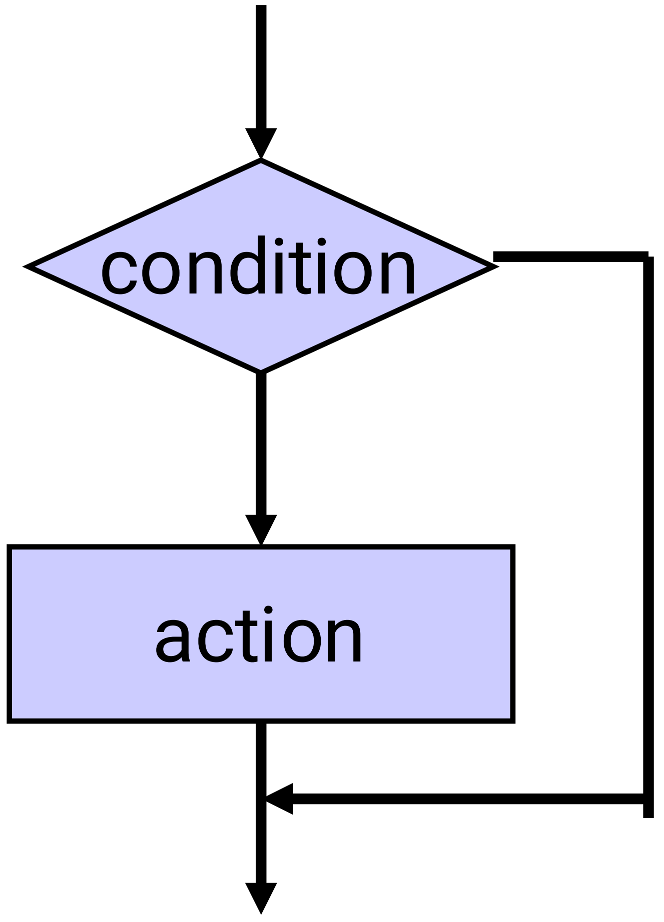
Control Structures: conditional constructs

```
if (x <= 10)
        y = x * x + 5;
```

```
if (x <= 10)
        y = x * x + 5;
        z = (2 * y)/4 ;
```

```
if (condition)
    action ;
```

```
if (x <= 10)      {
        y = x * x + 5;
        z = (2 * y)/4 ;
}
```


condition
action