LAB –I: PROGRAMMING IN C ++
CODE:(502105)

Dr.RM. Vidhyavathi
Assistant Professor
Dept. of Bioinformatics
Alagappa University
Karaikudi-4.

Outline

- History of C and C++
- Sample Program
- Data Types & Variables
- printf()
- Arithmetic & Logical Operations
- Conditionals
- Loops
- Arrays & Strings
- Pointers
- Functions
- Command-Line Argument
- Data Structure
- Memory Allocation Programming Tips
- C vs. C++

UNIT-I

History of C and C++

- It is developed by Dennis Ritchie in Bell Laboratories
- It is evolved from two other programming languages such as BCPL and B
- It has Added data typing and other features.
- Development language of UNIX
- Hardware independent
- Portable programs
  - ✓ 1989: ANSI standard
  - ✓ 1990: ANSI and ISO standard published
    - ✓ *ANSI/ISO 9899: 1990*

History of C++

- Extension of C
- It is developed by Bjarne Stroustrup in 1980 at Bell Laboratories
- Provides capabilities for object-oriented programming
  - ✓ *Objects*
  - ✓ *Object*
- Hybrid language
  - ✓ *C-like style*
  - ✓ *Object-oriented style*
  - ✓ *Both*

Sample Program

```
#include <stdio.h>
   int main()
{
        printf("Hello World\n");
        return(0);
}
```

Hello World Program

- How to compile?
  $ gcc hello.c –o hello

gcc	compiling command
hello.c	source file
hello	compiler-generated executable file

Note: the default output filename is “a.out”

Hello World Program

- How to execute?

  ./hello

  "./ " indicates the following file "hello" resides under the current directory.

Data types

Name	Description	Size	Range
char	Character or small integer	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	Floating point number	4 bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number	8 bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number	8 bytes	1.7e +/- 308 (15 digits)

Variable types

- **Local variable**
  - ✓ Declared within the body of a function
  - ✓ It can be used within that function, only.
- **Static variable**
  - ✓ One of the local variable.
  - ✓ It is specified by the keyword static in the variable declaration.
  - ✓ The static variable is not destroyed on exit from the function.
- **Global variable**
  - ✓ The global variable is located outside any of the program's functions.
  - ✓ It is accessible to all functions.

Variable Definition Vs Declaration

Definition	Tell the compiler about the variable: its type and name, as well as allocated a memory cell for the variable
Declaration	Describe information ``about'' the variable, doesn’t allocate memory cell for the variable.

printf() Function

- The printf() function can be instructed to print integers, floats and string properly.
- The general syntax is

```
printf( "format", variables);
```

- An example

```
int stud_id = 5200;

char * name = "Mike";

printf("%s 's ID is %d \n", name, stud_id);
```

- Format Identifiers

%d	decimal integers
%x	hex integer
%c	character
%f	float and double number
%s	string
%p	pointer

- How to specify display space for a variable?

printf("The student id is %5d \n", stud_id);

The value of stud_id will occupy 5 characters space in the print-out.

- Why “\n”

  It introduces a new line on the terminal screen.

Escape Sequence

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\’	single quote
\n	newline	\”	double quote
\r	carriage return	\000	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

Arithmetic Operations

Operator Name	Symbol
Multiplication	*
Division	/
Modulus	%
Addition	+
Subtraction	-

Arithmetic Assignment Operators

Long Hand	Short Hand
X= X * Y;	X *= Y;
X = X / Y;	X /= Y;
X = X % Y;	X % =Y;
X = X + Y;	X +=Y;
X = X - Y;	X -=Y;

Increment and Decrement Operators

Awkward	Easy	Easiest
x = x+1;	x += 1	x++
x = x-1;	x -= 1	x--

Example

- **Arithmetic operators**

```
int i = 10;

int j = 15;

int add = i + j; //25

int diff = j - i; //5

int product = i * j; // 150

int quotient = j / i; // 1

int residual = j % i; // 5

i++;      //Increase by 1

i--;      //Decrease by 1
```

- **Comparing them**

```
int i = 10;

int j = 15;

float k = 15.0;


j / i = ?

j % i = ?

k / i = ?

k % i = ?
```

- **The Answer**

```
j / i  = 1;

j % i = 5;

k / i  = 1.5;

k % i   It is illegal.
```

Note: For %, the operands can only be integers.

Logical Operations

- What is “true” and “false” in C

  In C, there is no specific data type to represent “true” and “false”. C uses value “0” to represent “false”, and uses non-zero value to stand for “true”.

- Logical Operators

```
A && B =>      A and B
A || B  =>     A  or   B
A == B  =>     Is A equal to B?
A != B  =>     Is A not equal to B?
```

A > B	=>	Is A greater than B?
A >= B	=>	Is A greater than or equal to B?
A < B	=>	Is A less than B?
A <= B	=>	Is A less than or equal to B?

- **Don't be confused**

  && and || have different meanings from & and |.

  & and | are bitwise operators.

Boolean conditions

- **Comparison operators**

==	equal
!=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

- **Boolean operators**

&&	and
||	or
!	not

Short Circuiting

- Short circuiting means that we don't evaluate the second part of an AND or OR unless we really need to.

```
int i = 10; int j = 15; int k = 15; int m = 0;
if( i < j && j < k)   =>
if( i != j || k < j)   =>
if( j<= k || i > k)   =>
if( j == k && m)   =>
if(i)                  =>
if(m || j && i )   =>
```

```
int i = 10; int j = 15; int k = 15; int m = 0;

        if( i < j && j < k)   =>   false

        if( i != j || k < j)   =>   true

        if( j<= k || i > k)   =>   true

        if( j == k && m)   =>   false

        if(i)                  =>   true

        if(m || j && i )       =>   true
```

Did you get the correct answers?

Input Statements

cin >> variable-name;

- ✓ Meaning: read the value of the variable called <variable-name> from the user

Example:

```
cin >> a;

cin >> b >> c;

cin >> x;

cin >> my-character;
```

Output Statements

cout << variable-name;

Meaning: print the value of variable <variable-name> to the user

cout << “any message “;

Meaning: print the message within quotes to the user

cout << endl;

Meaning: print a new line

Example:

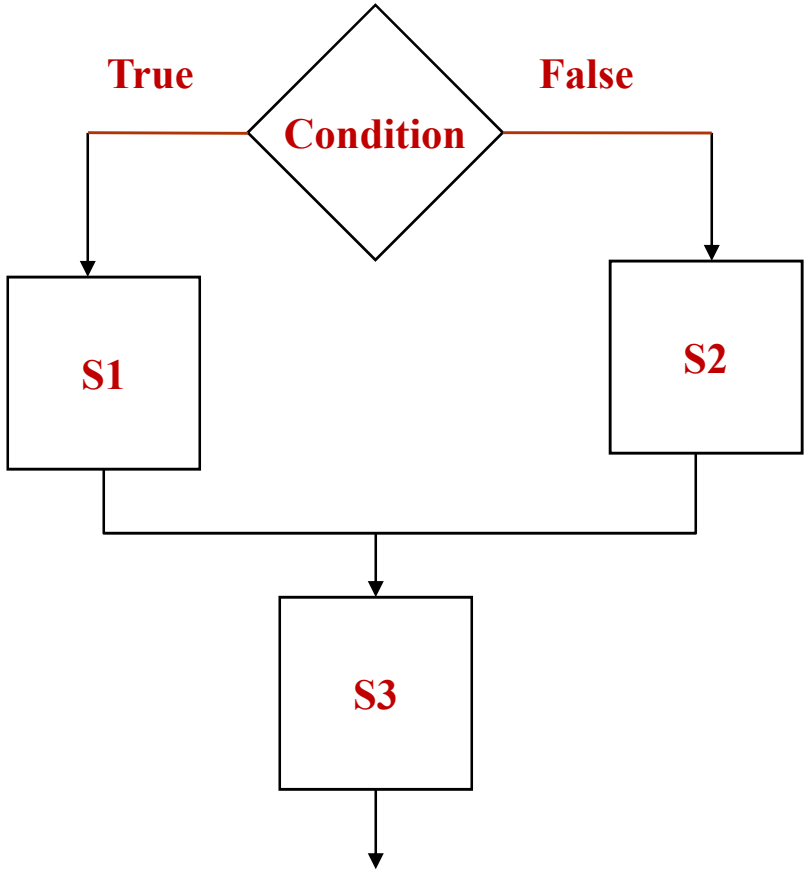
```
cout << a;

cout << b << c;

cout << “This is my character: “ << my-character << “ he he he”
     << endl;
```

If statements

```
if (condition) {
   S1;
}
else {
   S2;
}
S3;
```



Boolean conditions

..are built using

- Comparison operators

  == equal

  != not equal

  < less than

  > greater than

  <= less than or equal

  >= greater than or equal

- Boolean operators

  && and

  || or

  ! not

Conditional Statements

Conditionals

- if statement

  Three basic formats,

  if (*expression*){

  statement ...

  }

  if (*expression*) {

  statement ...

  }else{

  statement ...

  }

```
if (expression) {
   statement…
} else if (expression) {
   statement…
} else{
   statement…
}
```

- **An example**

```
if(score >= 90){
        a_cnt ++;
}else if(score >= 80){
        b_cnt++;
}else if(score >= 70){
        c_cnt++;
}else if (score>= 60){
        d_cnt++
}else{
        f_cnt++
}
```

- **The switch statement**

```
switch (expression)
{
        case item1:
                statement;
                break;
        case item2:
                statement;
                break;
        default:
                statement;
                break;
}
```

Loops

- for statement

  for (*expression1*; *expression2*; *expression3*){

  statement…

  }

  expression1 initializes;

  expression2 is the terminate test;

  expression3 is the modifier;

for <==> while

```
for (expression1; expression2; expression3)
{
        statement…
}
```

equals

```
expression1;
while (expression2)
{
statement…;
expression3;
}
```

- An example

```
int x;
for (x=0; x<3; x++)
{
        printf("x=%d\n",x);
}
```

First time: x = 0;
Second time: x = 1;
Third time: x = 2;
Fourth time: x = 3; (don’t execute the body)

- The while statement

  while (*expression*) {

  statement …

  }

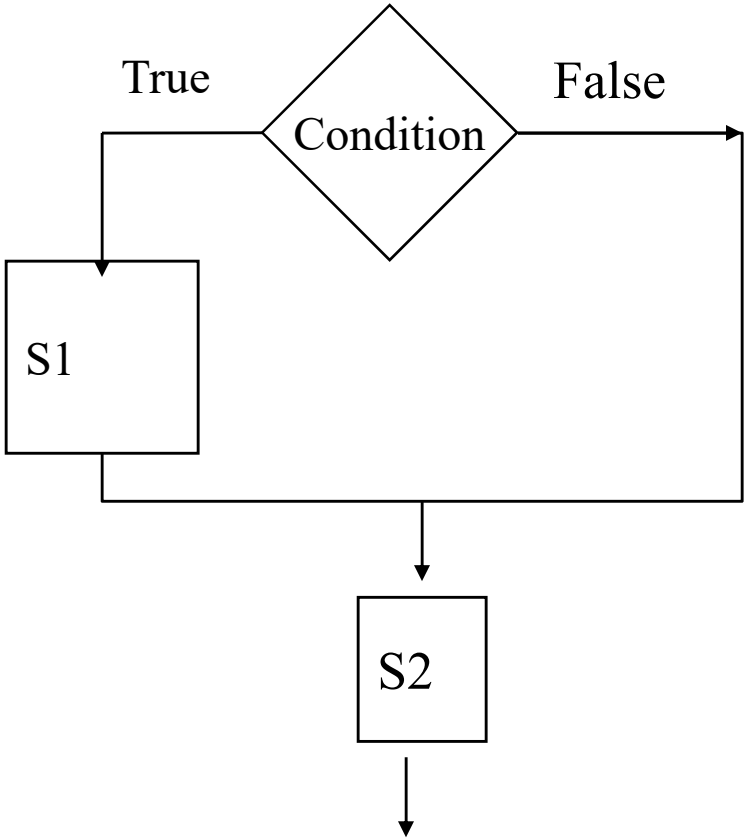
  while loop exits only when the expression is false.

- An example

```
int x = 3;
while (x>0) {
        printf("x=%d n",x);
        x--;
}
```

While statements

```
while (condition) {
   S1;
}
S2;
```



While example

```
//read 100 numbers from the user and output their sum
#include <iostream.h>

void main() {
int i, sum, x;
sum=0;
i=1;
while (i <= 100) {
    cin >> x;
    sum = sum + x;
    i = i+1;
}
cout << “sum is “ << sum << endl;
}
```

UNIT-II

Introduction to Functions

Functions

- Examples
- Function definition
- Function prototypes & Header files
- Pre- and post-conditions
- Scope and storage class
- Implementation of functions
- Recursive functions

Definition – Function

- A fragment of code that accepts zero or more argument values, produces a result value, and has zero or more side effects.

- A method of encapsulating a subset of a program or a system
  - To hide details
  - To be invoked from multiple places
  - To share with others

UNIT-II
INTRODUCTION TO FUNCTIONS

Functions

- Examples
- Definition
- Function prototypes & Header files
- Pre- and post-conditions
- Scope and storage class
- Implementation of functions
- Recursive functions

Definition – Function

- A fragment of code that accepts zero or more argument values, produces a result value, and has zero or more side effects.

- A method of encapsulating a subset of a program or a system
  - To hide details
  - To be invoked from multiple places
  - To share with others

Common Functions

#include <math.h>

- sin(x) // radians
- cos(x) // radians
- tan(x) // radians
- atan(x)
- atan2(y,x)
- exp(x) // e^x
- log(x) // $\log_e x$
- log10(x) // $\log_{10} x$
- sqrt(x) // $x \geq 0$
- pow(x, y) // x^y
- ...

#include <stdio.h>

- printf()
- fprintf()
- scanf()
- scanf()
- ...

#include <string.h>

- strcpy()
- strcat()
- strcmp()
- strlen()
- ...

Functions in C

resultType **functionName(**type$_1$ **param1,** type$_2$ **param2, …) {**

…

body

…

}

- If no result, resultType should be **void**
  - Warning if not!
- If no parameters, use **void** between **()**

Functions in C

resultType **functionName(**type$_1$ **param1,** type$_2$ **param2, …) {**

…

body

…

} **// functionName**

- If no result, resultType should be **void**
  - Warning if not!
- If no parameters, use **void** between **()**

It is good style to always end a function with a comment showing its name

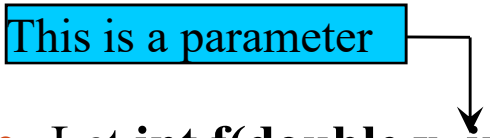
Using Functions

- Let **int f(double x, int a)** be (the beginning of) a declaration of a function.

- Then **f(expr$_1$, expr$_2$)** can be used in *any* expression where a *value* of type **int** can be used – e.g.,

N = f(pi*pow(r,2), b+c) + d;

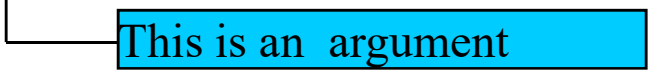
Using Functions (continued)

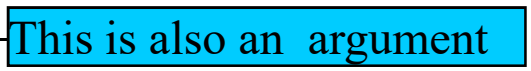


- Let **int f(double x, int a)** be (the beginning of) a declaration of a function.

- Then **f(expr$_1$, expr$_2$)** can be used in *any* expression where a value of type **int** can be used – e.g.,

N = f(pi*pow(r,2), b+c) + d;





Definitions

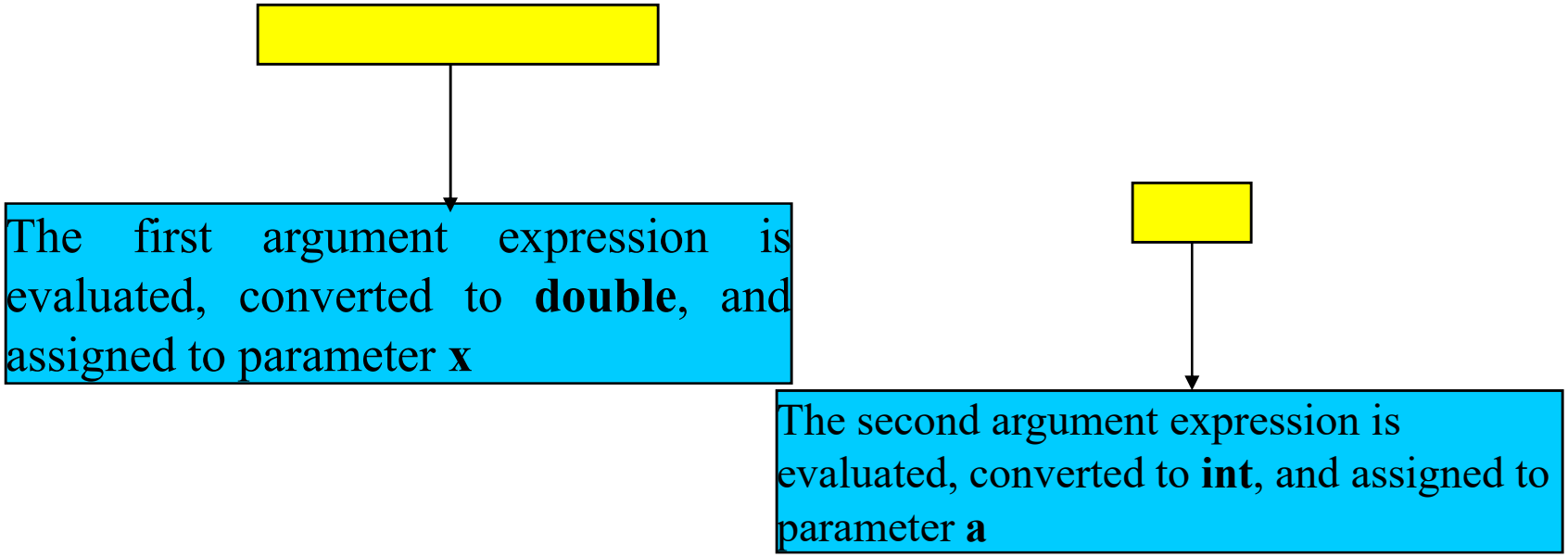
- ***Parameter:–*** a declaration of an identifier within the **'()'** of a function declaration
  - Used within the body of the function as a *variable* of that function
  - Initialized to the value of the corresponding *argument*.
- ***Argument***:– an expression passed when a function is *called*; becomes the initial value of the corresponding parameter

Using Functions (continued)

- Let **int f(double x, int a)** be (the beginning of) a declaration of a function.

- Then **f(expr$_1$, expr$_2$)** can be used in *any* expression where a value of type **int** can be used – e.g.,

N = f(pi*pow(r,2), b+c) + d;

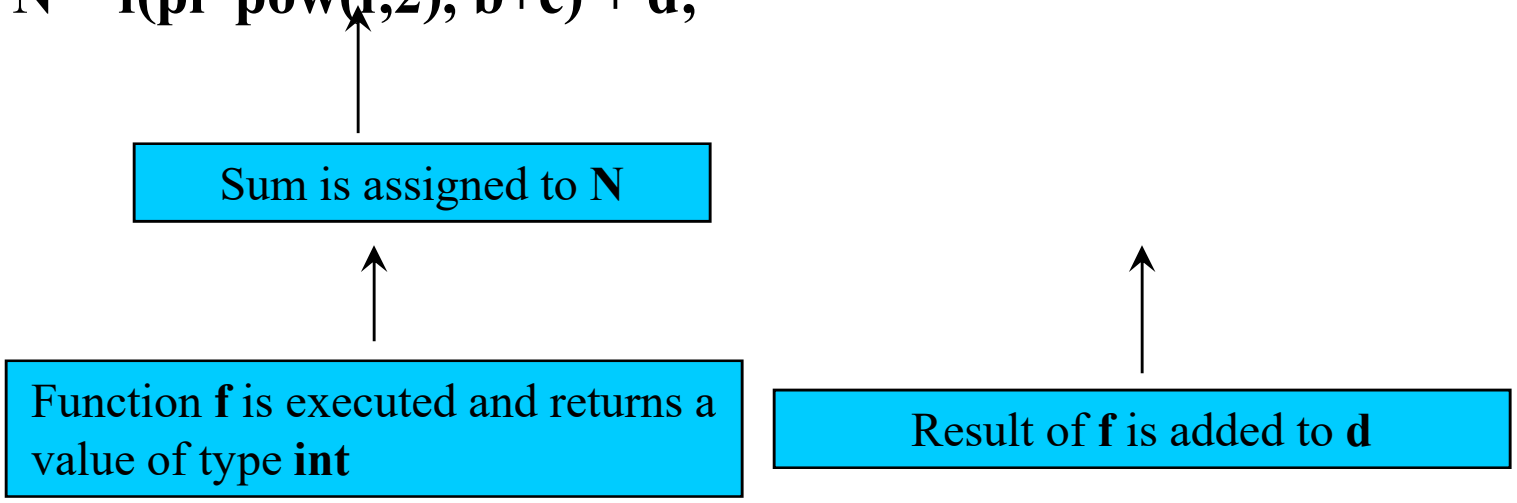


Using Functions (continued)

- Let **int f(double x, int a)** be (the beginning of) a declaration of a function.

- Then **f(expr$_1$, expr$_2$)** can be used in *any* expression where a value of type **int** can be used – e.g.,

  N = f(pi*pow(r,2), b+c) + d;



Functions

Functions are easy to use; they allow complicated programs to be broken into small blocks, each of which is easier to write, read, and maintain. This is called modulation.

- How does a function look like?

```
returntype function_name(parameters…)
{
        local variables declaration;
        function code;
        return result;
}
```

Function Definition

- Every function definition has the form

  return-type function-name (parameter declarations) {

  definitions and statements

  }

- For practical purposes, code between {} (inclusive) is a compound statement

Note

- Functions in C do not allow other functions to be declared within them
  - Like C++, Java
  - Unlike Algol, Pascal
- All functions defined at “top level” of C programs
  - (Usually) visible to linker
  - Can be linked by any other program that knows the function prototype

Examples

- double sin(double radians) {

  …

  } // sin

- unsigned int strlen (char *s) {

  …

  } // strlen

Note on **printf**, etc.

- **int printf(char *s, ...) {**
  body
  } // **printf**

- In this function header, "**…**" is *not* a professor's place-holder
  - (as often used in these slides)
- …but an actual sequence of three dots (no spaces between)
  - Meaning:– the number and types of arguments is indeterminate
  - Use **<stdarg.h>** to extract the arguments

Function Prototypes

- There are many, many situations in which a function must be used separate from where it is defined –
  - before its definition in the same C program
  - In one or more completely separate C programs
- This is actually the normal case!
- Therefore, we need some way to declare a function separate from defining its body.
  - Called a Function Prototype

Function Prototypes (continued)

- **Definition:– a Function Prototype** in C is a language construct of the form:–

return-type function-name (parameter declarations) ;

- I.e., exactly like a function definition, except with a **';'** instead of a body in curly brackets.

Purposes of Function Prototype

- So compiler knows how to compile calls to that function, i.e.,
  - number and types of arguments
  - type of result
- As part of a "contract" between developer and programmer who uses the function
- As part of hiding details of how it works and exposing what it does.
- A function serves as a "black box."

- **Sample function**

```
int addition(int x, int y)
{
        int add;
        add = x + y;
        return add;
}
```

- **How to call a function?**

```
int result;
int i = 5, j = 6;
result  = addition(i, j);
```

POINTERS

Pointers

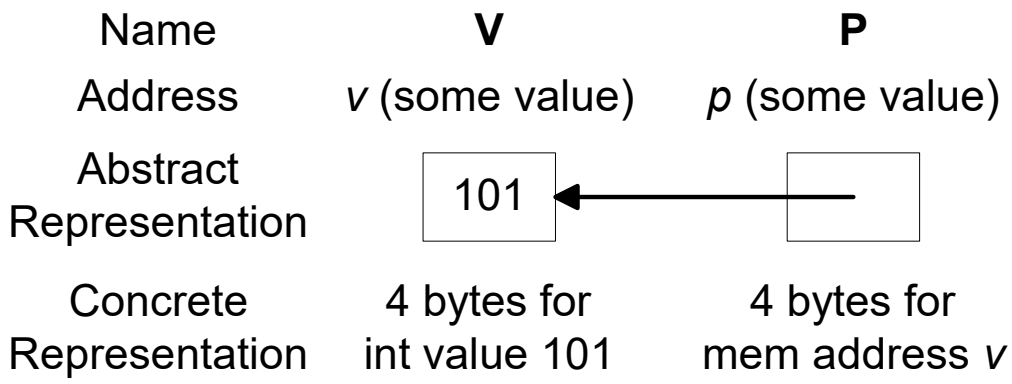
- ✓ Pointer is the most beautiful (ugliest) part of C, but also brings most trouble to C programmers. Over 90% bugs in the C programs come from pointers.
- ✓ A pointer is a variable which contains the address in memory of another variable.
- ✓ In C we have a specific type for pointers.

Pointer Basics

- ✓ Variables are allocated at addresses in computer memory (address depends on computer/operating system)
- ✓ Name of the variable is a reference to that memory address
- ✓ A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):

```
int V = 101;

int *P = &V;
```



Pointer Variable Definition

Basic syntax: Type *Name

Examples:

```
int *P;          /* P is var that can point to an int var */
float *Q;        /* Q is a float pointer */
char *R;         /* R is a char pointer */
```

Complex example:

```
int *AP[5];    /* AP is an array of 5 pointers to ints */
```

- more on how to read complex declarations later

- **Declaring a pointer variable**

  int * pointer;

  char * name;
- How to obtain the address of a variable?

  int x = 0x2233;

  pointer = &x;

  where & is called address of operator.

- How to get the value of the variable indicated by the pointer?

  int y = *pointer;

Address (&) Operator

- The address (&) operator can be used in front of any variable object in C - the result of the operation is the location in memory of the variable
- **Syntax:** &VariableReference
- **Examples:**

```
int V;
int *P;
int A[5];
```

&V - memory location of integer variable V

&(A[2]) - memory location of array element 2 in array A

&P - memory location of pointer variable P

Pointer Variable Initialization/Assignment

NULL - pointer lit constant to non-existent address:

- used to indicate pointer points to nothing can initialize/assign pointer vars to NULL or use the address (&) op to get address of a variable.
- variable in the address operator must be of the right type for the pointer (an integer pointer points only at integer variables)

Examples:

```
int V;
int *P = &V;
int A[5];
P = &(A[2]);
```

Indirection (*) Operator

A pointer variable contains a memory address

To refer to the contents of the variable that the pointer points to, we use indirection operator

Syntax: *PointerVariable

Example:

```
int V = 101;
int *P = &V;
/* Then *P would refer to the contents of the variable V (in this
   case, the integer 101) */
printf("%d",*P);  /* Prints 101 */
```

Pointer Sample

```
int A = 3;
int B;
int *P = &A;
int *Q = P;
int *R = &B;

printf("Enter value:");
scanf("%d",R);
printf("%d %d\n",A,B);
printf("%d %d %d\n",
  *P,*Q,*R);
```

```
Q = &B;
if (P == Q)
  printf("1\n");
if (Q == R)
  printf("2\n");
if (*P == *Q)
  printf("3\n");
if (*Q == *R)
  printf("4\n");
if (*P == *R)
  printf("5\n");
```

Reference Parameters

- To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)
- Then we use indirection operator inside the function to change the value the parameter points to:

```
void changeVar(float *cvar)
{
  *cvar = *cvar + 10.0;
}
float X = 5.0;
changeVar(&X);
printf("%.1f\n",X);
```

Pointer Return Values

A function can also return a pointer value:

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}
void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%.1f %.1f\n",*maxA,A[4]);
}
```

Pointers to Pointers

- A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V; /* P points to int V */
int **Q = &P;          /* Q points to int pointer P */
printf("%d %d %d\n",V,*P,**Q); /* prints 101 3 times */
```

Pointer Types

- Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is inappropriate to assign an address of one type of variable to a different type of pointer

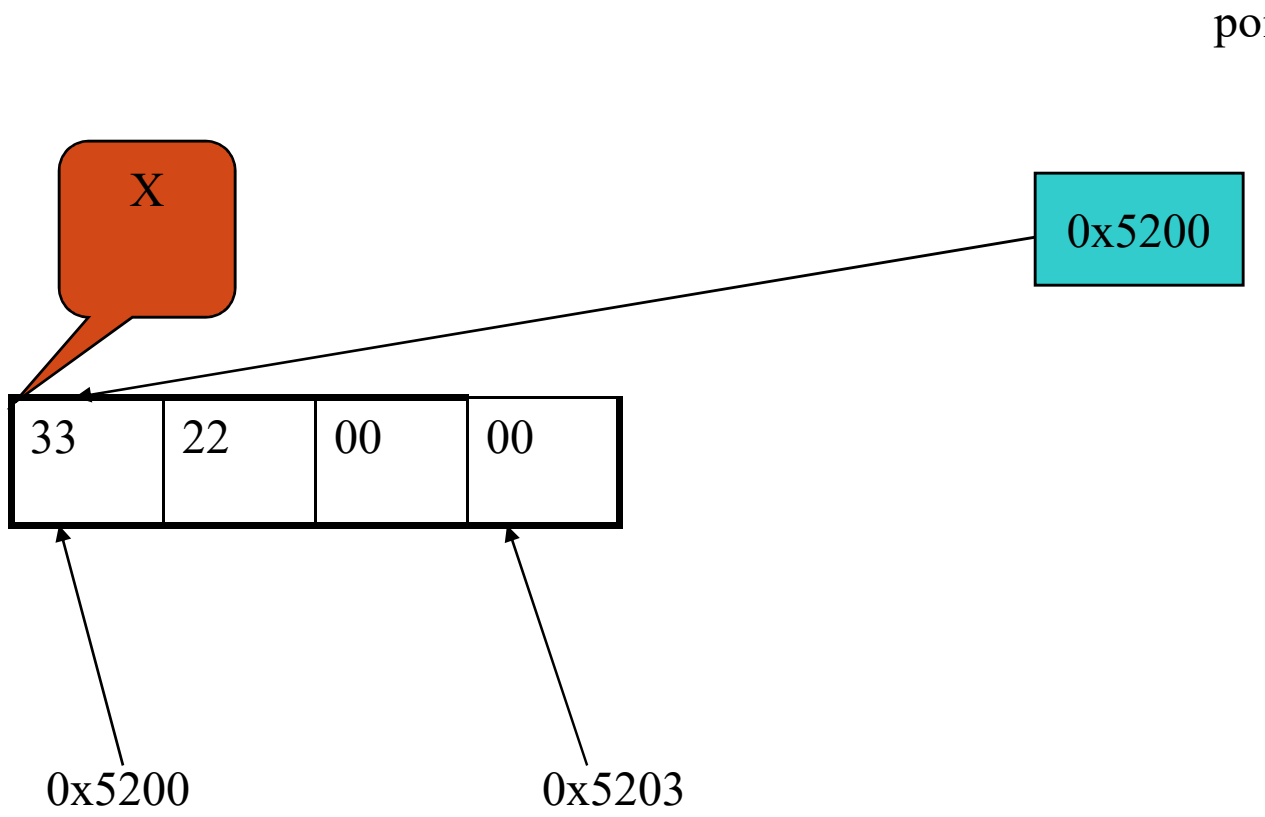
Example:

```
int V = 101;
float *P = &V; /* Generally results in a Warning */
```

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

- What happens in the memory?

Suppose the address of variable x is 0x5200 in the above example, so the value of the variable pointer is 0x5200.



swap the value of two variables

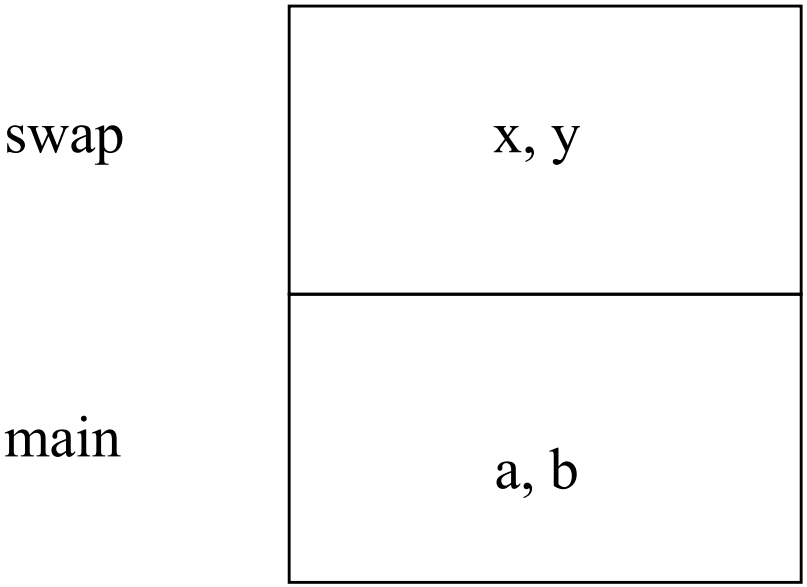
```
void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
void swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

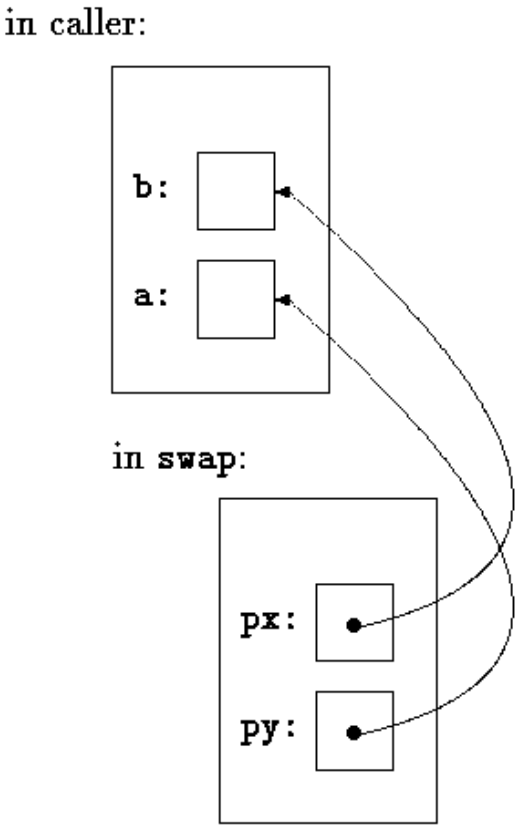
Why is the left one not working?



x, y, a, b are all local variables

call swap(a, b) in main

Why is the right one working?



Pointers and Arrays

Pointers and arrays are very closely linked in C.

Array elements arranged in consecutive memory locations

- Accessing array elements using pointers

  int ids[50];

  int * p = &ids[0];

  p[i] <=> ids[i]

- Pointers and Strings

  A string can be represented by a char * pointer.

Dynamic Memory Allocation

- Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically).
- **Previous example:** ask the user how many numbers to read, then allocate array of appropriate size.
- **Idea:** user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done memory allocated in the *Data Heap*

Memory Management Functions

Calloc - routine used to allocate arrays of memory

Malloc - routine used to allocate a single block of memory

Realloc-routine used to extend the amount of space allocated previously

Free - routine used to tell program a piece of memory n longer needed.

note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Array Allocation with calloc

prototype: **void * calloc(size_t num, size_t esize)**

size_t is a special type used to indicate sizes, generally an unsigned int num is the number of elements to be allocated in the array esize is the size of the elements to be allocated,generally use sizeof and type to get correct value,an amount of memory of size num*esize allocated on heap.

calloc - returns the address of the first byte of this memory generally we cast the result to the appropriate type if not enough memory is available, calloc returns NULL.

calloc Example

```
float *nums;
int N;
int I;
printf(“Read how many numbers:”);
scanf(“%d”,&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf(“Please enter number %d: “,I+1);
  scanf(“%f”,&(nums[I]));
}
/* Calculate average, etc. */
```

Releasing Memory (free)

prototype: void free(void *ptr)

- memory at location pointed to by ptr is released (so we could use it again in the future) program keeps track of each piece of memory allocated by where that memory starts
- if we free a piece of memory allocated with calloc, the entire array is freed (released)
- results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

free Example

```
float *nums;
int N;
printf(“Read how many numbers:”);
scanf(“%d”,&N);
nums = (float *) calloc(N, sizeof(float));
/* use array nums */
/* when done with nums: */
free(nums);
/* would be an error to say it again - free(nums) */
```

The Importance of free

```
void problem() {
  float *nums;
  int N = 5;

  nums = (float *) calloc(N, sizeof(float));

  /* But no call to free with nums */
} /* problem ends */
```

- When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is) Problem called memory leakage.

Array Allocation with malloc

prototype: void * malloc(size_t esize)

- similar to calloc, except we use it to allocate a single block of the given size esize as with calloc, memory is allocated from heap NULL returned if not enough memory available memory must be released using free once the user is done can perform the same function as calloc if we simply multiply the two arguments of calloc together

  malloc(N * sizeof(float)) is equivalent to

  calloc(N,sizeof(float))

Increasing Memory Size with realloc

prototype: void * realloc(void * ptr, size_t esize)

- ptr is a pointer to a piece of memory previously dynamically allocated
- esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already) program allocates memory of size esize, then it copies the contents of the memory at ptr to the first part of the new piece of memory, finally, the old piece of memory is freed up.

realloc Example

```
float *nums;
int I;
nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */
for (I = 0; I < 5; I++)
  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */
nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated, the first 5 floats
   from the old nums are copied as the first 5 floats of the new nums,
   then the old nums is released */
```

Command-Line Argument

In C you can pass arguments to main() function.

- main() prototype

  int main(int argc, char * argv[]);

  argc indicates the number of arguments

  argv is an array of input string pointers.

- How to pass your own arguments?

  ./hello 10

- **What value is argc and argv?**

Let's add two printf statement to get the value of argc and argv.

```
#include <stdio.h>

 int main(int argc, char * argv[]);)

{                    int i=0;

        printf("Hello World\n");

        printf("The argc is %d \n", argc);

        for(i=0; i < argc; i++){

           printf("The %dth element in argv is %s\n", i, argv[i]);

        }

        return(0);

}
```

- **The output**

  The argc is 2

  The 0th element in argv is ./hello

  The 1th element in argv is 10

  The trick is the system always passes the name of the executable file as the first argument to the main() function.

- **How to use your argument?**

  Be careful. Your arguments to main() are always in string format.

  Taking the above program for example, the argv[1] is string “10”, not a number. You must convert it into a number before you can use it.

Storage Classes

Meaning of Storage Class

- Each variable declared in C contains not only its data but also it has a storage class specified with it.
- If user do not specify the storage class of a variable , the compiler will assume its storage class as default i.e. automatic .

Purpose of Storage Class

The storage class of a variable tells us about :-

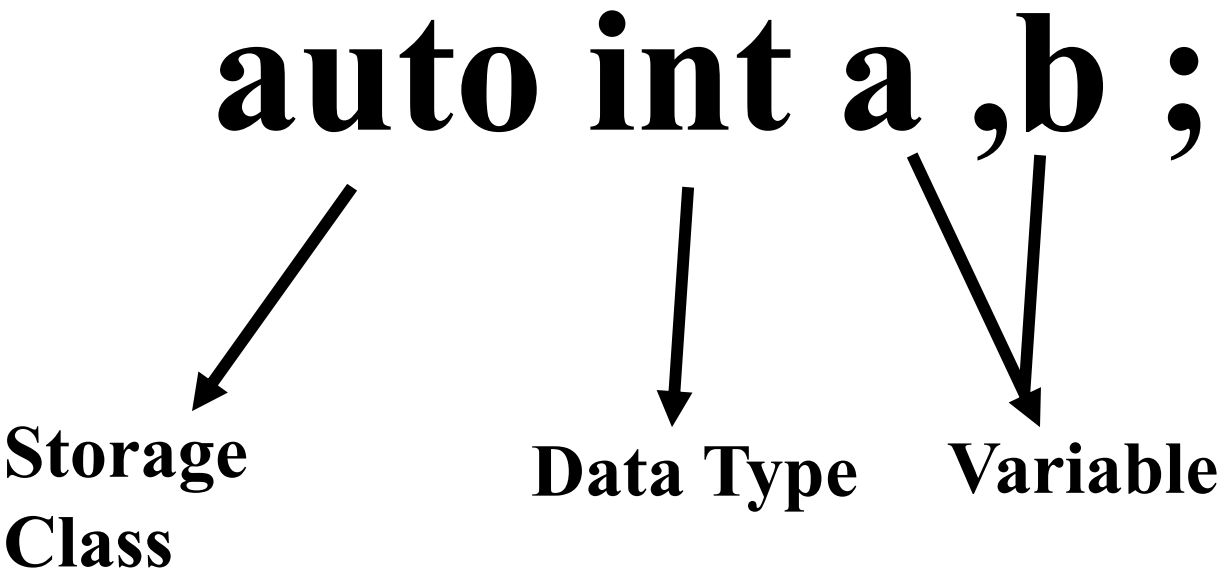
I. Storage Place of Variable i.e. **Memory or CPU Registers**

II. Initial value of Variable

III. Scope of Variable

IV. Lifetime of Variable i.e. **how long variable exists.**

How Storage Class Declared ????



Four Type of Storage Class in C Language :-

- Automatic Storage Class (Local Variables)
- Register Storage Class
- Static Storage Class
- External Storage Class

Automatic Storage Class

This is the default storage class for all the variable . It always reinitialize the value of variable .It is declared as : - auto int a ;

Characteristics	Meaning
Storage	Memory
Initial Value	Garbage Value i.e. An Unpredictable Value.
Scope or Visibility	Local or Visible in the Block or function in which it is declared.
Life Time	It retains its value till it is in the block in which it is declared.

Use of Automatic Storage Class

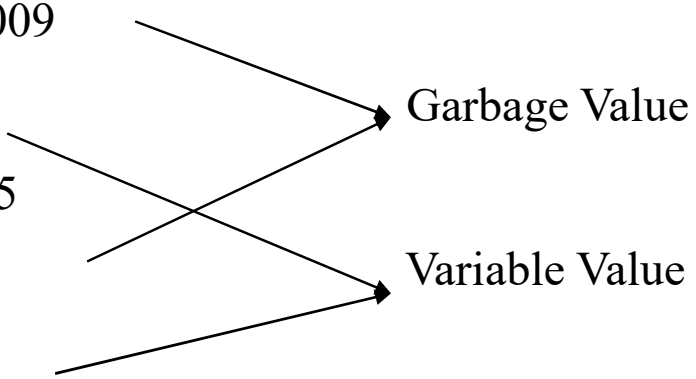
```
Void main ( )
{
        auto int i , j = 5 ;
        int k  ,m =10 ;        // By Default Automatic Storage Class
        printf ( " value of  i  = % d   \n value of  j = % d  " , i , j ) ;
        printf ( " value of  k  = % d   \n value of m = % d  " , i , j ) ;
}
```

Value of i = 2009

Value of j = 5

Value of k = 1005

Value of m = 10



Use of Automatic Storage Class

```
void main( )
{
        void ck() ; //function prototype
        clrscr ( ) ;
        ck ( ) ;
        ck ( ) ;
        ck ( ) ;
        getch( ) ;
}
```

```
void ck ( )
{
        int i = 0 ;
        printf ( "\n\n Value of I ..%d",i ) ;
        i + + ;
}
```

Output

0

0

0

Register Storage Class

In this storage class , variable is stored in C P U Registers , just for the sake of increase the execution speed of some variable of program. It is declares as :-

register int a ;

Characteristics	**Meaning**
Storage	C P U Registers
Initial Value	Garbage Value
Scope or Visibility	Local to the Block in which it is declared
Life Time	It retains its value till the control remains in the block

Use of Register Storage Class

```
Void main ( )
{
        register int i ;
        for ( i = 1 ; i < = 100 ; i + + )
        {
                printf ( " \n % d " , i ) ;
        }
}
```

Use of scanf with register store class variable is invalid Program give error because register is not associated with any memory address.

```
#include<stdio.h>
int main()
{
register int num; //register variable its scope lies with in the main       //function and
can not use with scanf
clrscr();
scanf("%d",&num); //  Error is must take address of memory location
getch();
return 0;
}
```

Static Storage Class

This storage class is used when a user want that a variable should retain its value even after the execution of the function in which it is declared, then this storage class is used . It is declared as follow :-

static int a ;

Characteristics	Meaning
Storage	Memory
Initial Value	Zero (0)
Scope or Visibility	Local to the block in which it is declared
Life Time	It retains its value between the different function calls.

Use of Static Storage Class

```
void main( )
{
        void ck ( ) ;
        clrscr ( ) ;
        ck ( ) ;
        ck ( ) ;
        ck ( ) ;
        getch( ) ;
}
```

```
void ck ( )
{
        static int i = 0 ;
        printf ( "\n\n Value of I ..%d",i ) ;
        i + + ;
}
```

Output

0

1

2

External Storage Class

External variables are declared outside all functions i.e, at the **beginning** of the program. **Global variables** should be available to all the functions with the help of **extern specifier**. It is declared as follow : -

extern int a ;

Characteristics	**Meaning**
Storage	Memory
Initial Value	Zero (0)
Scope or Visibility	Global (Visible in all the Program)
Life Time	It retains its value through out the whole program

Use of External Storage Class

```
#include<stdio.h>
#include<conio.h>
extern int a = 10 ;
void ck ( ) ;
void main( )
{       int a = 5 ;
        printf ( " %d " , a) ;
        ck ( ) ;
        getch ( ) ;
}
```

```
void ck ( )
{
        a = a + 10 ;
        printf ( "\n\n Value of a ..%d",a ) ;
}
```

Output

5

20

```
#include<stdio.h>
#include<conio.h>
int x=21;
int main()
{
int y;
 clrscr();
 printf("%d \t %d",x,y);
getch();
return 0;

}
int y=31;
```

```
#include<stdio.h>
#include<conio.h>
int x=21;
int main()
{
Extern int y; //external variable
declare
 clrscr();
 printf("%d \t %d",x,y);
getch();
return 0;

}
int y=31; // extern variable is defined
```

LHS Program Output is

21 Garbage value

RHS Program Output is

21 31

C PREPROCESSOR

Introduction

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of symbolic constants and macros
  - Conditional compilation of program code
  - Conditional execution of preprocessor directives
- Format of preprocessor directives
  - Lines begin with #
  - Only whitespace characters before directives on a line

The #include Preprocessor Directive

- **#include**
  - Copy of a specified file included in place of the directive
  - **#include <filename>**
    - Searches standard library for file
    - Use for standard library files
  - **#include "filename"**
    - Searches current directory, then standard library
    - Use for user-defined files
  - **Used for**:
    - Programs with multiple source files to be compiled together
    - Header file – has common declarations and definitions (classes, structures, function prototypes)
      - **#include** statement in each file

The #define Preprocessor Directive: Symbolic Constants

- **#define**
  - Preprocessor directive used to create symbolic constants and macros
  - Symbolic constants
    - When program compiled, all occurrences of symbolic constant replaced with replacement text
  - Format

    #define identifier replacement-text
  - Example:

    #define PI 3.14159
  - Everything to right of identifier replaces text

    #define PI = 3.14159
    - Replaces "**PI**" with "**= 3.14159**"
  - Cannot redefine symbolic constants once they have been created

The #define Preprocessor Directive: Macros

- Macro
  - Operation defined in **#define**
  - A macro without arguments is treated like a symbolic constant
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

    #define CIRCLE_AREA(x) (PI * (x) * (x))

  would cause

    area = CIRCLE_AREA(4);

  to become

    area = (3.14159 * (4) * (4));

The #define Preprocessor Directive: Macros

- Use parenthesis
  - Without them the macro

    #define CIRCLE_AREA(x) PI * (x) * (x)

  would cause

    area = CIRCLE_AREA(c + 2);

  to become

    area = 3.14159 * c + 2 * c + 2;

- Multiple arguments

    #define RECTANGLE_AREA(x, y) ((x) * (y))

  would cause

    rectArea = RECTANGLE_AREA(a + 4, b + 7);

  to become

    rectArea = ((a + 4) * (b + 7));

The #define Preprocessor Directive: Macros

- **#undef**
  - Undefines a symbolic constant or macro
  - If a symbolic constant or macro has been undefined it can later be redefined

Conditional Compilation

- Conditional compilation
  - Control preprocessor directives and compilation
  - Cast expressions, **sizeof**, enumeration constants cannot be evaluated in preprocessor directives
  - Structure similar to **if**

```
#if !defined( NULL )
  #define NULL 0
#endif
```

    - Determines if symbolic constant **NULL** has been defined
      - If **NULL** is defined, **defined(NULL)** evaluates to **1**
      - If **NULL** is not defined, this function defines **NULL** to be **0**
  - Every **#if** must end with **#endif**
  - **#ifdef** short for **#if defined(name)**
  - **#ifndef** short for **#if !defined(name)**

Conditional Compilation

- Other statements
  - **#elif** – equivalent of **else if** in an **if** structure
  - **#else** – equivalent of **else** in an **if** structure
- "Comment out" code
  - Cannot use /* ... */
  - Use

```
#if 0
    code commented out
#endif
```

  - To enable code, change **0** to **1**

Conditional Compilation

- Debugging

```
#define DEBUG  1

#ifdef DEBUG
    cerr << "Variable x = " << x << endl;
#endif
```

  - Defining **DEBUG** to **1** enables code
  - After code corrected, remove **#define** statement
  - Debugging statements are now ignored

The #error and #pragma Preprocessor Directives

- **#error** tokens
  - Tokens are sequences of characters separated by spaces
    - **"I like C++"** has **3** tokens
  - Displays a message including the specified tokens as an error message
  - Stops preprocessing and prevents program compilation
- **#pragma** tokens
  - Implementation defined action (consult compiler documentation)
  - Pragmas not recognized by compiler are ignored

The # and ## Operators

- Causes a replacement text token to be converted to a string surrounded by quotes
- The statement

  #define HELLO(x) printf(“Hello, ” #x “\n”);

would cause

HELLO(John)

to become

printf(“Hello, ” “John” “\n”);

- Strings separated by whitespace are concatenated when using **printf**

The # and ## Operators

- Concatenates two tokens
- The statement

#define TOKENCONCAT(x, y) x ## y

would cause

TOKENCONCAT(O, K)

to become

OK

Line Numbers

- **#line**
  - Renumbers subsequent code lines, starting with integer value
  - File name can be included
  - **#line 100 "myFile.c"**
    - Lines are numbered from **100** beginning with next source code file
    - Compiler messages will think that the error occurred in **"myfile.C"**
    - Makes errors more meaningful
    - Line numbers do not appear in source file

Predefined Symbolic Constants

- Five predefined symbolic constants
  - Cannot be used in **#define** or **#undef**

Symbolic constant	Description
__LINE__	The line number of the current source code line (an integer constant).
__FILE__	The presumed name of the source file (a string).
__DATE__	The date the source file is compiled (a string of the form **"Mmm dd yyyy"** such as **"Jan 19 2001"**).
__TIME__	The time the source file is compiled (a string literal of the form **"hh:mm:ss"**).

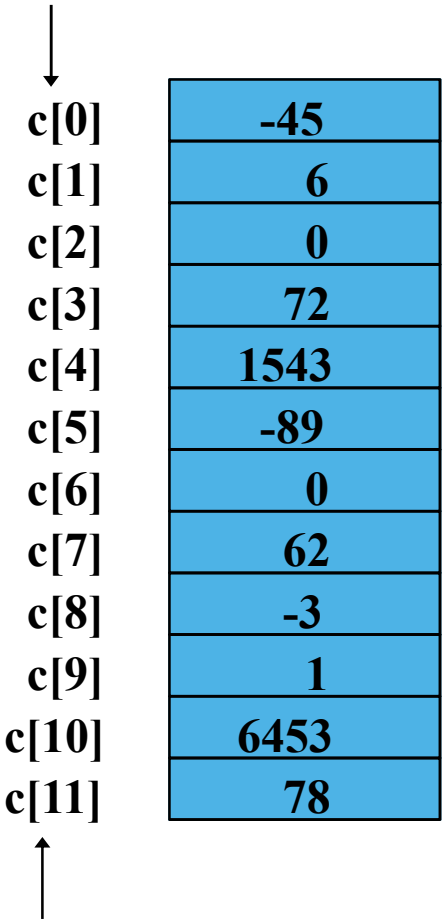
UNIT-III-ARRAYS

Introduction

- Arrays
  - Structures of related data items
  - Static entity – same size throughout program

- Array
  - Group of consecutive memory locations
  - Same name and type
- To refer to an element, specify
  - Array name
  - Position number
- Format:

  arrayname[*position number*]
  - First element at position **0**
  - **n** element array named **c**:
    - **c[0], c[1]...c[n – 1]**



Array Applications

- Given a list of test scores, determine the maximum and minimum scores.
- Read in a list of student names and rearrange them in alphabetical order (sorting).
- Given the height measurements of students in a class, output the names of those students who are taller than average.

Array Declaration

- Syntax:

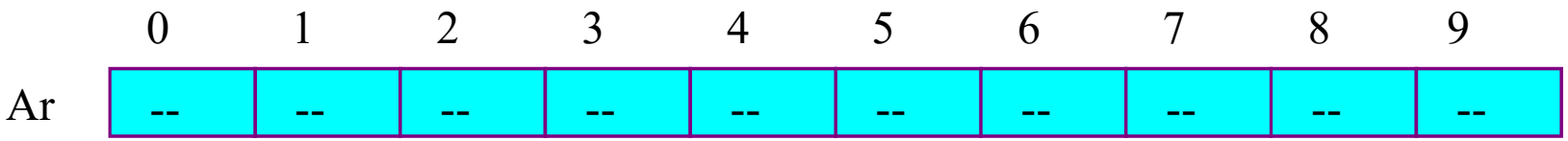
  <type> <arrayName>[<array_size>]

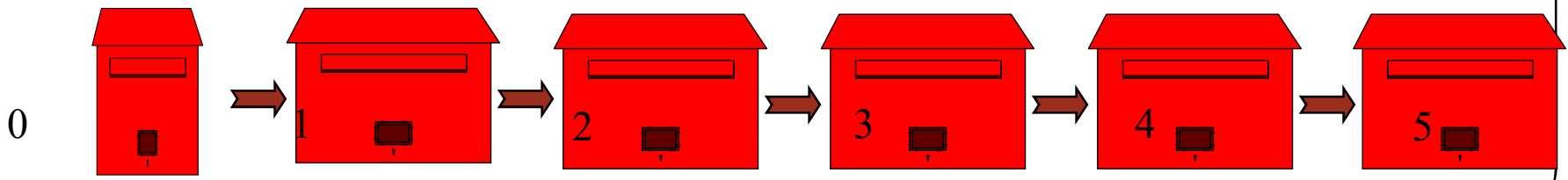
  Ex. int Ar[10];

- The array elements are all values of the type **<type>.**
- The size of the array is indicated by **<array_size>**, the number of elements in the array.
- **<array_size>** must be an **int** constant or a constant expression. Note that an array can have multiple dimensions.

Array Declaration

```
// array of 10 uninitialized ints
int Ar[10];
```





Subscripting

- Declare an array of 10 integers:

  ```
  int Ar[10];    // array of 10 ints
  ```

- To access an individual element we must apply a subscript to array named **Ar.**
  - A subscript is a bracketed expression.
    - The expression in the brackets is known as the index.
  - First element of array has index 0.

    Ar[0]
  - Second element of array has index 1, and so on.

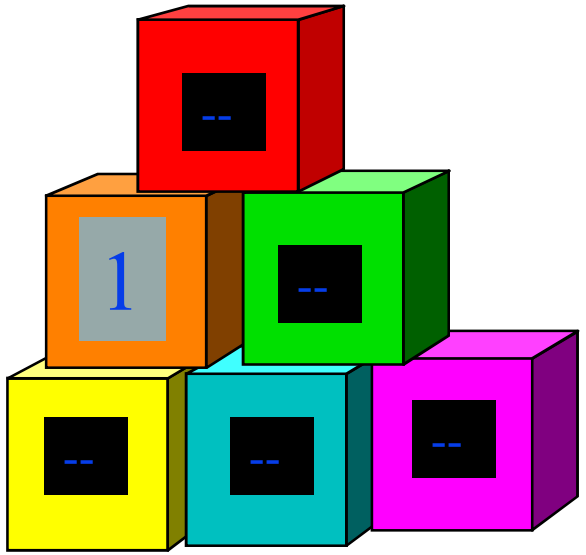
    Ar[1], Ar[2], Ar[3],…
  - Last element has an index one less than the size of the array.

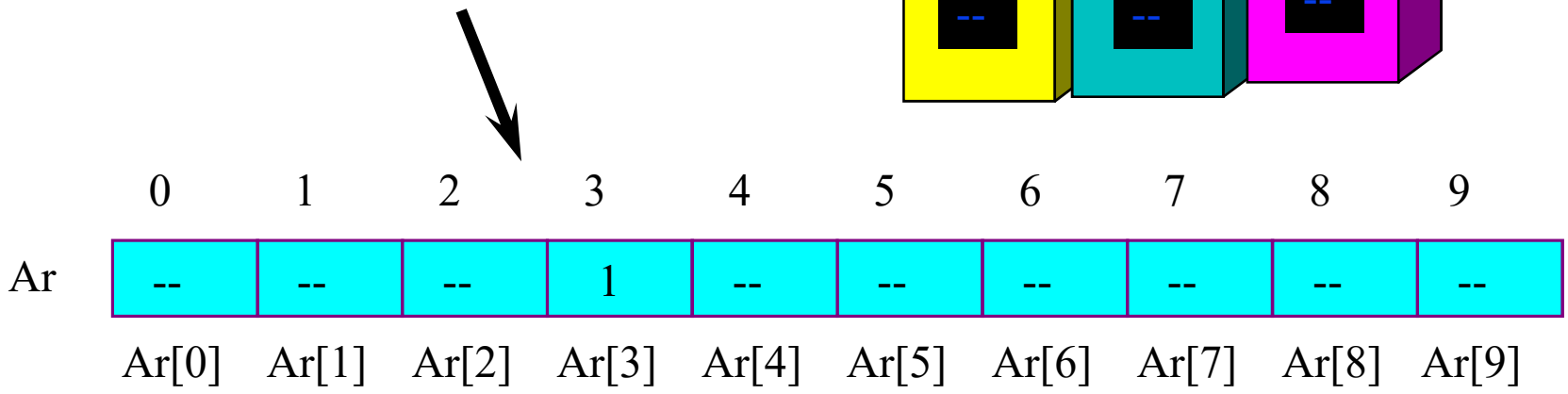
    Ar[9]
- Incorrect indexing is a common error.

Subscripting

```
// array of 10 uninitialized ints
int Ar[10];

Ar[3] = 1;
int x = Ar[3];
```





Arrays

- Array elements are like normal variables

```
c[ 0 ] =  3;
printf( "%d", c[ 0 ] );
```

  - Perform operations in subscript. If **x** equals **3**

```
c[ 5 - 2 ] == c[ 3 ] == c[ x ]
```

Examples Using Arrays

- Character arrays
  - String **“first”** is really a static array of characters
  - Character arrays can be initialized using string literals

    char string1[] = "first";

    - Null character **'\0'** terminates strings
    - **string1** actually has 6 elements
      - It is equivalent to

char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };

  - Can access individual characters

    string1[3] is character ‘s’

  - Array name is address of array, so & not needed for scanf

    scanf("%s", string2);

    - Reads characters until whitespace encountered
    - Can write beyond end of array, be careful

Passing Arrays to Functions

- Passing arrays
  - To pass an array argument to a function, specify the name of the array without any brackets

    int myArray[24];

    myFunction(myArray, 24);

    - Array size usually passed to function
  - Arrays passed call-by-reference
  - Name of array is address of first element
  - Function knows where the array is stored
    - Modifies original memory locations
- Passing array elements
  - Passed by call-by-value
  - Pass subscripted name (i.e., **myArray[3]**) to function

Passing Arrays to Functions

- Function prototype

  void modifyArray(int b[], int arraySize);

  - Parameter names optional in prototype
    - **int b[]** could be written **int []**
    - **int arraySize** could be simply **int**

Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
- Bubble sort (sinking sort)
  - Several passes through the array
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change
    - If decreasing order, elements exchanged
  - Repeat
- Example:
  - original: 3 4 2 6 7
  - pass 1: 3 2 4 6 7
  - pass 2: 2 3 4 6 7
  - Small elements "bubble" to the top

Searching Arrays: Linear Search and Binary Search

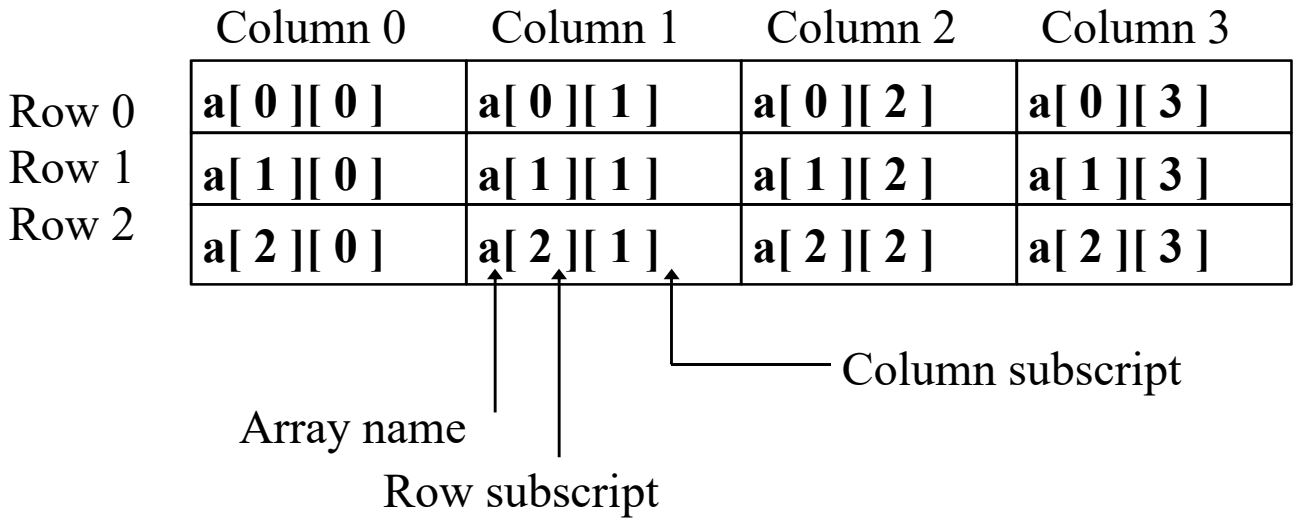
- Search an array for a key value
- Linear search
  - Simple
  - Compare each element of array with key value
  - Useful for small and unsorted arrays

Searching Arrays: Linear Search and Binary Search

- Binary search
  - For sorted arrays
  - Compares **middle** element with **key**
    - If equal, match found
    - If **key < middle**, looks in first half of array
    - If **key > middle**, looks in last half
    - Repeat
  - Very fast; at most n steps, where 2^n > number of elements
  - 30 element array takes at most 5 steps
    - $2^5 > 30$ so at most 5 steps

Multiple-Subscripted Arrays

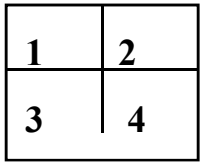
- Multiple subscripted arrays
  - Tables with rows and columns (**m** by **n** array)
  - Like matrices: specify row, then column

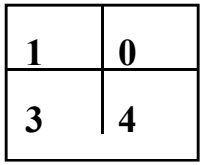


Multiple-Subscripted Arrays

- Initialization
  - **int b[2][2] = { { 1, 2 }, { 3, 4 } };**
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

    int b[2][2] = { { 1 }, { 3, 4 } };





- Referencing elements
  - Specify row, then column

    printf("%d", b[0][1]);

STRING MANIPULATION

Introduction:

- Strings are nothing but array of characters ended with a null character.
- The null character indicates the end of the string.
- Strings are always enclosed by double quotes. Whereas , character is enclosed by single quotes in C.

Examples for C strings:

- char sting[20]={‘h’,’e’,’l’,’l’,’o’,’\0’};

Or

- char string[ ]=“hello”;
- When we declare char as string[20],20 bytes of memory space is allocated for holding the string value.
- When we declare char as string[ ],memory space will be allocated as per the requirement during execution of the program.

Program for C string:

```
#nclude<stdio.h>

int main()

{

char string[20]=“helloworld”;

printf(“The string is:%s\n”,strig);

return 0;

}
```

Output:

The string is : helloworld.

C String functions:

- String.h header supports all the string functions.

Strcat()functions:

- It concatenates two given strings.
- It concatenates source string at the end of destination string.

Syntax:

```
char *strcat(char *destination,const  char *source);
```

Program:

```
#include<stdio.h>
#include<string.h>
int main()
{
char source[]="fresh2refresh.com";
char target[]="C tutorial";
printf("\n Source string= %s", string);
printf("\n Target string= %s", target);
strcat ( target , source);
printf("\n Target string after strcat ( )= %s", target);
}
```

Output:

source string=fresh2refresh

target string=C tutorial

target string after strcat()=C tutorialfresh2refresh

Strcpy( ) function:

❖Copies the content of one sting into another string.

Syntax:

char *strcpy(char *destination,const char *source);

Example:

strcpy(str1,str2)-It copies contents of str2 into str1.

strcpy(str2,str1)-It copies contents of str1 into str2.

❖If destination string length is less than source string, the entire source string value won't be copied into destination string.

Program:

```
#include<stdio.h>
#include<string.h>
 int main ()
{
  char source[]="fresh2refresh.com";
  char target[20]=" ";
printf("\nSource string=%s", string);
printf("\nTarget string=%s", target);
strcat(target , source);
printf("\nTarget string after strcat ( )=%s", target);
return 0;
}
```

Output:

```
source string=fresh2refresh
target string=
target string after strcat()=fresh2refresh
```

Strlen ( ) function:

- It gives the length of the given string.

Syntax:

size_t strlen(const char *str);

- It counts the number of characters in a given string and returns the integer value.
- It stops counting the character when null character is found

Source Code:

Program:

```
#include<stdio.h>
#include<string.h>
int main ()
{
int len;
char array[20]="goodmorning";
len=strlen(array);
printf("\n string length =%d\n",len);
return 0;
}
```

Output:

string length=11

Strrev ( ) function:

- It reverses a given string in C language.

Syntax:

```
char *strrev(char *string);
```

- It is a non standard function which may not be available in standard library in C.

Program:

```
#include<stdio.h>
#include<string.h>
int main ()
{
char name[30]=“Hello”;
printf(“String before strrev():%s\n”,name);
printf(“String after strrev():%s”,strrev(name));
return 0;
}
```

Output:

```
string before strrev( )=Hello
 string after strrev( )=olleH
```

Strdup ( ) function.

- It duplicates the given string.

Syntax:

```
char  *strdup(const  char *string);
```

- It is a non standard function which may not be available in standard library in C.

Program:

```
#include<stdio.h>
#include<string.h>
int main ()
{
  char *p1="Food";
   char *p2;
  p2=strdup(p1);
printf("Duplicated string is:%s");
return 0;
}
```

Output :

Duplicated string is:Food

Strlwr ( ) function.

- Converts a given string into lowercase.

Syntax:

```
char *strlwr(char *string);
```

- It is a non standard function which may not be available in standard library in C.

Program:

```
#include<stdio.h>
#include<string.h>
int main ()
{
  char str[ ]="MODIFY this string to lower case",
printf("%s\n",strlwr(str));
return 0;
}
```

Output:

modify this string to lower.

Strupr( ) function.

- Converts a given string into uppercase.

Syntax:

char *strlwr(char *string);

- It is a non standard function which may not be available in standard library in C.

Program:

```
#include<stdio.h>
#include<string.h>
int main ()
{
  char str[ ]="MODIFY this string to upper case",
printf("%s\n",strupr(str));
return 0;
}
```

Output:

MODIFY THIS STRING TO LOWER.

Unions

- Like structures, but every member occupies the same region of memory!
  - Structures: members are “and”ed together: “name and species and owner”
  - Unions: members are “xor”ed together

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float xor an int xor a string */
```

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a "type" variable that indicates how to interpret the union value

```
enum TYPE { INT, FLOAT, STRING };
struct VARIABLE {
  enum TYPE type;
  union VALUE value;
};
```

Access type to determine how to interpret value

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes;
    for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

GRAPHICS

Computer Graphics

- Computer Graphics is one of the most powerful and interesting aspect of computers. There are many things we can do in graphics apart from drawing figures of various shapes.
- All video games, animation, multimedia predominantly works using computer graphics.

Graphics in C

- There is a large number of functions in C which are used for putting pixel on a graphic screen to form lines, shapes and patterns.
- The Default output mode of C language programs is “Text” mode. We have to switch to “Graphic” mode before drawing any graphical shape like line, rectangle, circle etc.

Basic color Function

- textcolor() function
- textbackground() function
- setbkcolor() function
- setcolor() function

textcolor() function

Declaration:

void textcolor(newcolor);

Remarks:

- This function works for functions that produce text-mode output directly to the screen (console output functions).
- **textcolor** selects a new character color in text mode.
- This functions does not affect any characters currently on the screen.

textbackground() function

Declaration:

void textbackground(newcolor);

Remarks:

textbackground selects the background color for text mode.

If you use symbolic color constants, the following limitation apply to the background colors you select:

- **You can only select one of the first eight colors (0--7).**

NOTE: If you use the symbolic color constants, you must include conio.h.

setcolor() function

Declaration:

void setcolor(color);

Remarks:

setcolor sets the current drawing color to color, which can range from 0 to getmaxcolor.

setbkcolor() function

Declaration:

void setbkcolor(color);

Remarks:

setbkcolor sets the background to the color specified by color.

Example 1.
txtcolor() & textbackground() functions

```
#include<graphics.h>
#include<stdio.h>
#include<conio.h>
Void main()
{
textcolor(4);          OR        textcolor(4+BLINK)
textbackground(3);
cprintf("NFC-IEFR INSTITUTE ");
getch();
}
```

OUTPUT

NFC-IEFR INSTITUTE

Graphics in C

- There are lot of library functions in C language which are used to draw different drawing shapes.
- For eg.
  - line(x1, y1, x2, y2);
  - putpixel(x, y);
  - circle(xCenter, yCenter, radius);
  - rectangle(x1, y1, x2, y2);
  - ellipse(xCenter, yCenter, start, end, Xradius, Yradius);
  - arc(xCenter, yCenter, start, end, radius);

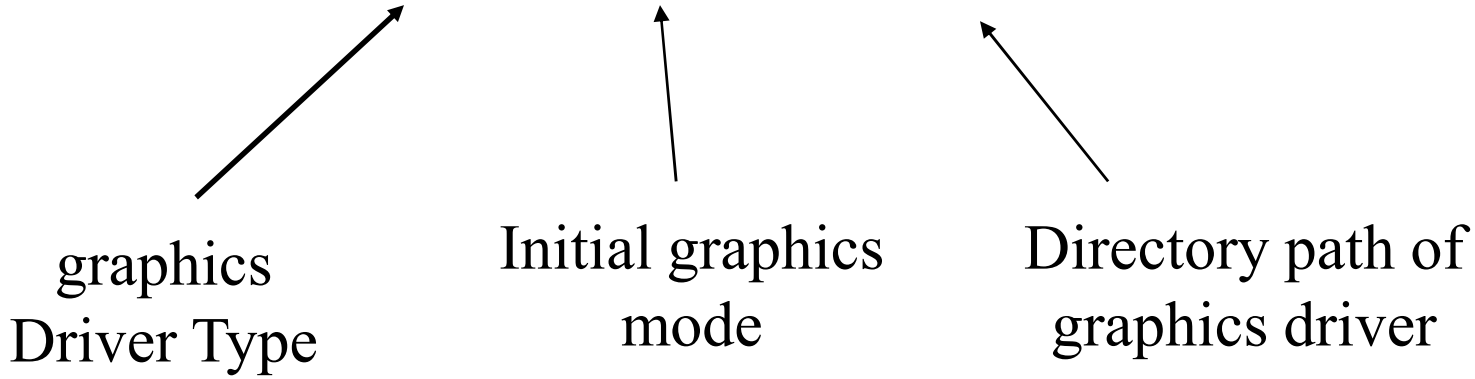
Graphics in C-Example 2

```
#include<graphics.h>
void main(void)
{
   int dr=DETECT, md;
    initgraph(&dr,&md,"c:\\tc\\bgi");
    line(0 , 0, 640, 480);
    getch();
   closegraph();
}
```

Dissecting initgraph(…) Function

- The initgraph function is used to switch the output from text mode to graphics mode.
- The initgraph function takes three arguments.

```
initgraph(&dr , &md , ”c:\\tc\\bgi” );
```



Graphics Drivers

Constant	Value
DETECT	0 (requests auto detection)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Graphics Mode

driver	graphics_modes	Value	Column x Row	Colors
CGA	CGAC0	0	320 x 200	4 colors
	CGAC1	1	320 x 200	4 colors
	CGAC2	2	320 x 200	4 colors
	CGAC3	3	320 x 200	4 colors
	CGAHI	4	640 x 200	2 colors
EGA	EGALO	0	640 x 200	16 colors
	EGAHI	1	640 x 350	16 colors
VGA	VGALO	0	640 x 200	16 colors
	VGAMED	1	640 x 350	16 colors
	VGAHI	2	640 x 480	16 colors

Directory path of graphics driver

- The third argument to initgraph() is the pathname for the graphics drivers. This driver is a file like cga.bgi or egavga.bgi.
- cga.bgi file is used to run programs in CGA modes.
- egavga.bgi file is used to run programs in EGA or VGA modes.
- Other Drivers are available for other display standards such as Hercules and IBM 8514.
- In the current version of Turbo C, these driver files are located in the subdirectory \tc\bgi. So this is the pathname used in the arguments to initgraph().

line() function

- line draws a line between two specified points

Syntax:

```
line(x1, y1, x2,y2);
```

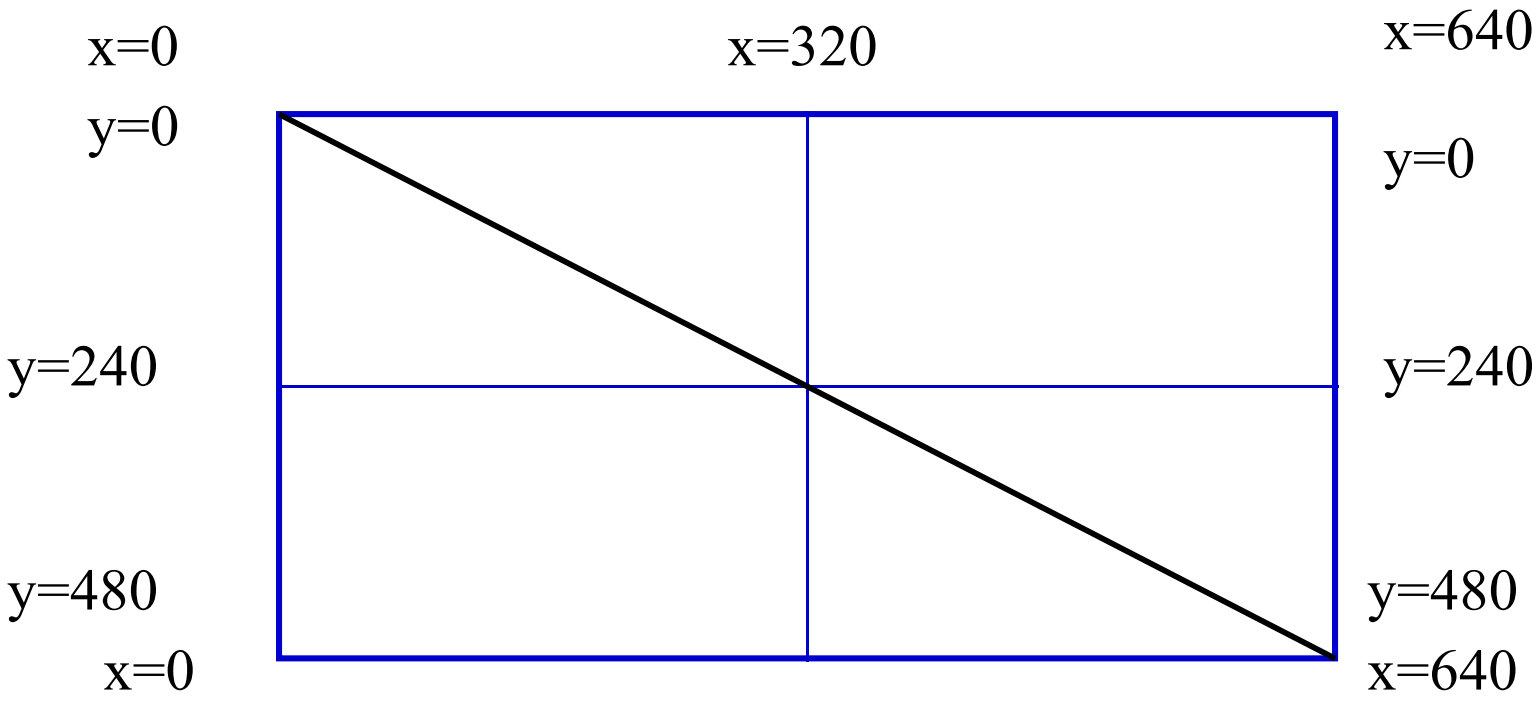
Remarks

line draws a line from (x1, y1) to (x2, y2) using the current color, line style, and thickness.

line() function

- **Example**

```
line(0, 0, 640, 480);
```



setlinestyle() function

Sets the current line style and width or pattern

Syntax:

setlinestyle (linestyle, upattern, thickness);

Remarks:

setlinestyle sets the style for all lines drawn by line, lineto, rectangle, drawpoly

Line Styles

Line Style	Int Value	Pattern
SOLID_LINE	0	
DOTTED_LINE	1	
CENTER_LINE	2	
DASHED_LINE	3	
USERBIT_LINE	4	**User Defined**

upattern, thickness

- U pattern
  - User can define its own pattern.
  - 0 should be used if using predefined pattern, other wise any integer number representing user pattern
- Thickness
  - Thickness of the line in pixels

rectangle() function

Draws a rectangle (graphics mode)

syntax:

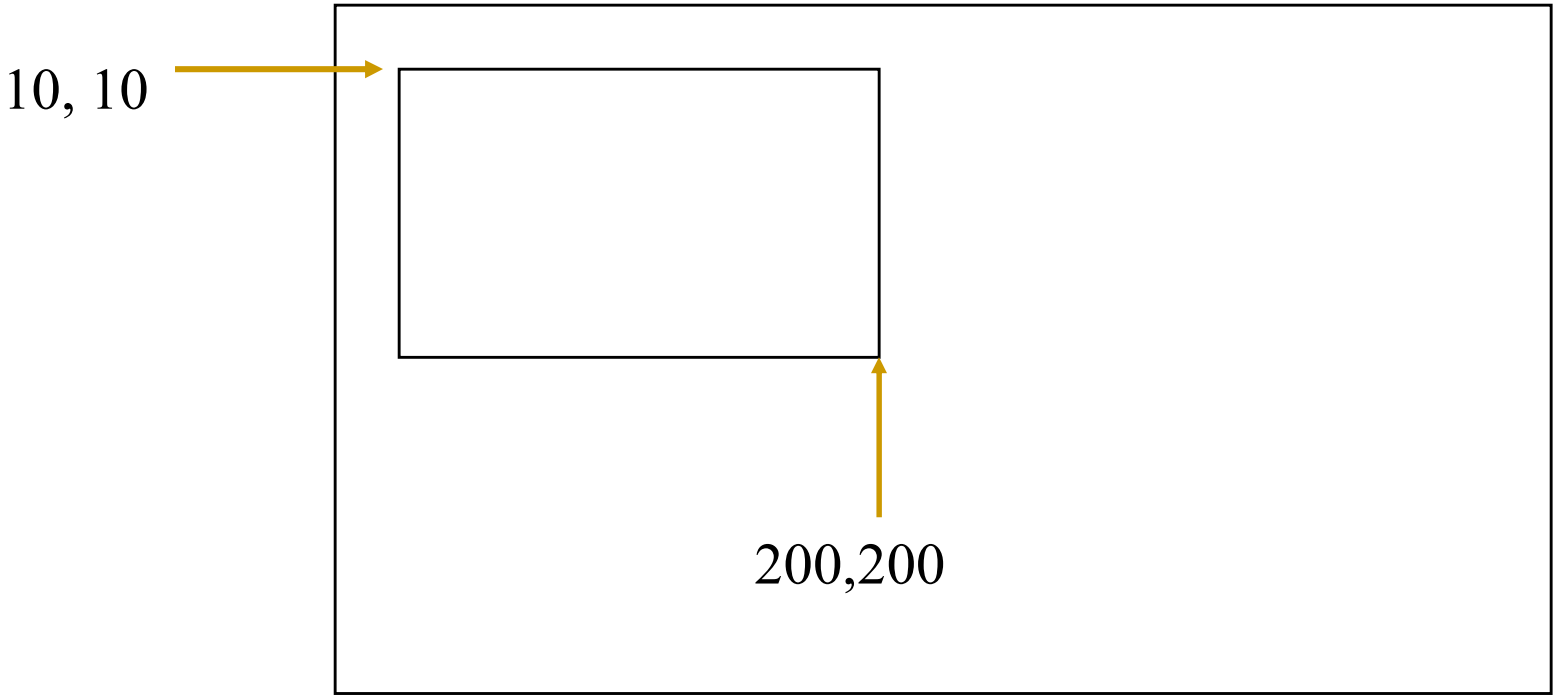
void rectangle(left, top, right, bottom);

Remarks:

- rectangle draws a rectangle in the current line style, thickness, and drawing color.
- (left,top) is the upper left corner of the rectangle, and (right,bottom) is its lower right corner.

Example: rectangle() function

rectangle(10, 10, 200,200);



bar( ) function

bar(..) function Draws a bar

- **Syntax:**

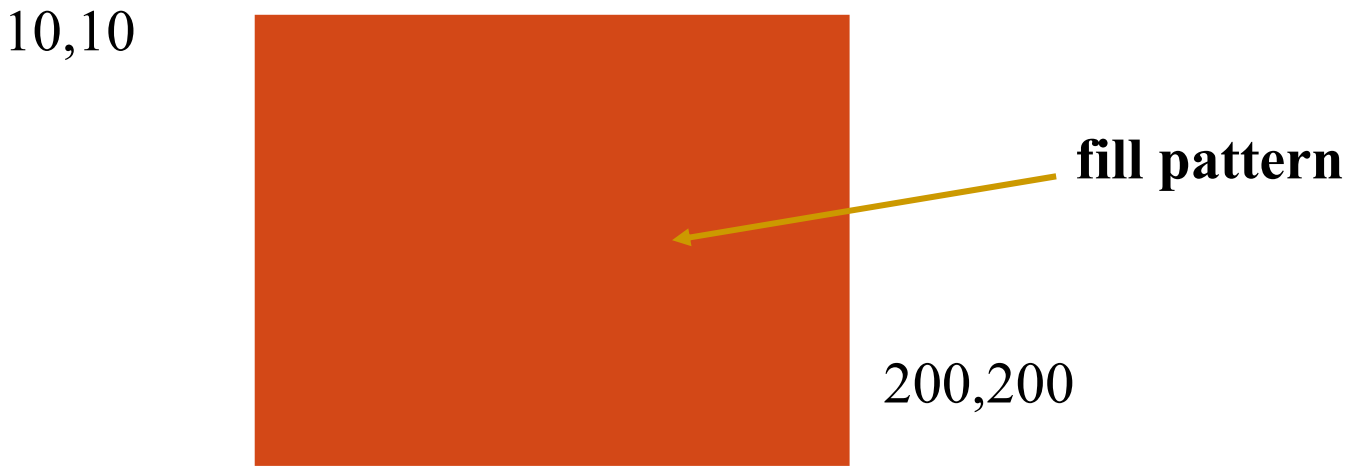
  void bar(left, top, right, bottom);

- **Remarks:**
  - bar draws a filled-in, rectangular, two-dimensional bar.
  - The bar is filled using the current fill pattern and fill color. bar does not outline the bar.
  - To draw an outlined two-dimensional bar, use bar3d with depth = 0.

Example: bar( ) function

Usage:

bar(10,10,200,200);



bar3d() function

Declaration:

void bar3d(left,top,right, bottom, depth,topflag);

Remarks:

bar3d draws a three-dimensional rectangular bar, then fills it using the current fill pattern and fill color. The three-dimensional outline of the bar is drawn in the current line style and color.

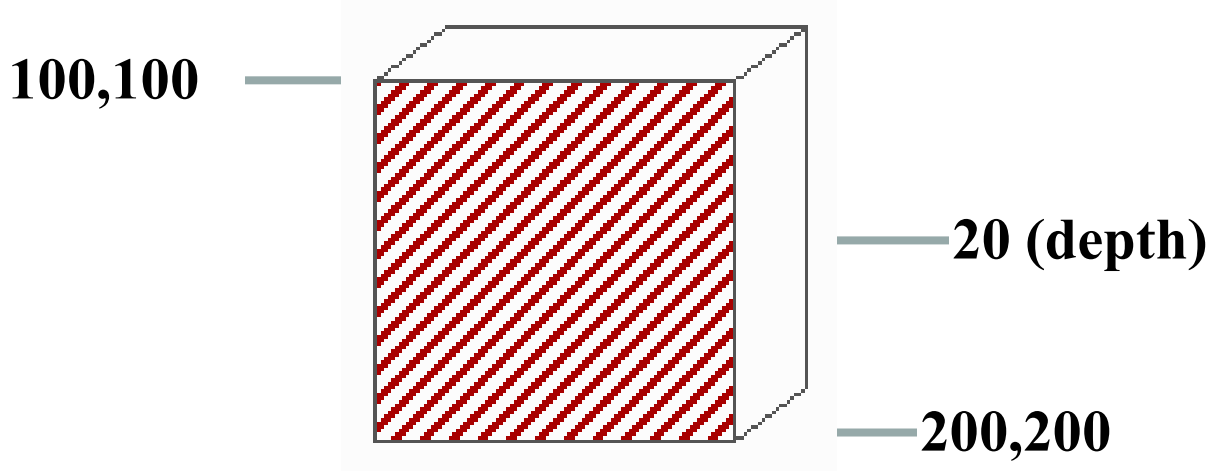
Parameter	What It Is/Does
depth	Bar's depth in pixels
topflag	Governs whether a three-dimensional top is put on the bar
(left, top)	Rectangle's upper left corner
(right, bottom)	Rectangle's lower right corner

eg: bar3d() function

Usage:

```
setfillstyle(4, 4);
bar3d(100, 100, 200, 200, 20, 1);
```

OUTPUT



circle() function

Declaration:

void circle(x,y,radius);

Remarks:

circle draws a circle in the current drawing color.

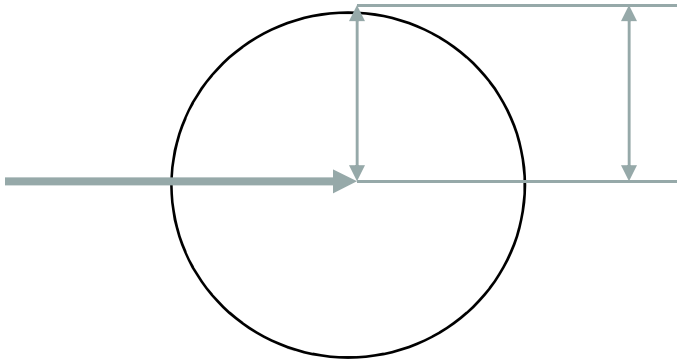
Argument	What It Is/Does
(x,y)	Center point of circle
radius	Radius of circle

eg: circle() function

Usage:

circle(320,240,50);

OUTPUT



arc() function

Declaration:

void arc(x,y,stangle,endangle radius);

Remarks:

arc draws a circular arc in the current drawing color.

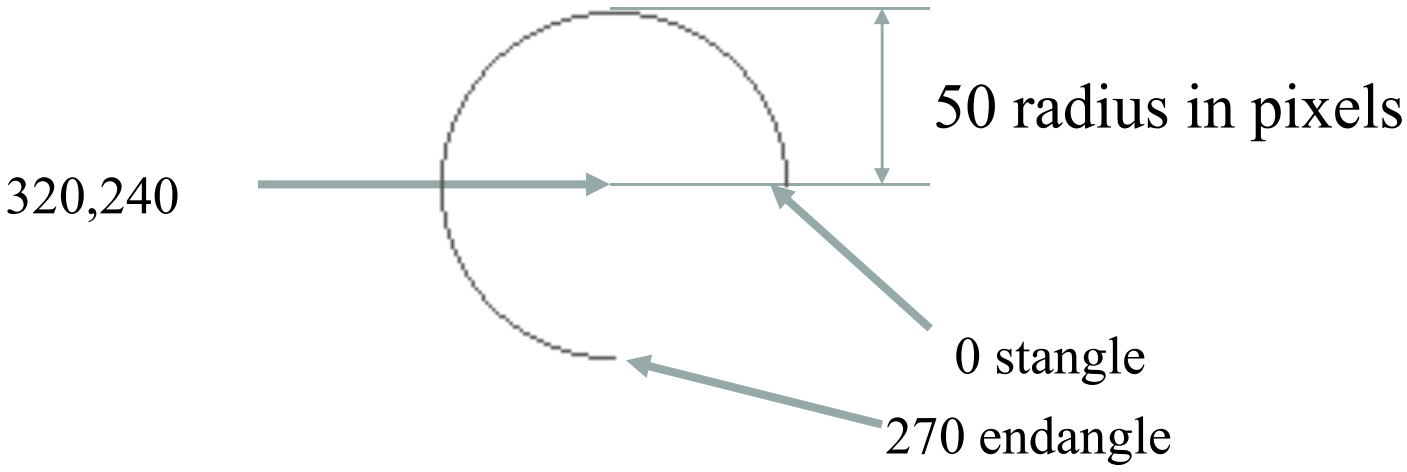
Argument	What It Is/Does
(x,y)	Center point of arc
stangle	Start angle in degrees
endangle	End angle in degrees
radius	Radius of circle

eg: arc() function

Usage:

arc(320, 240, 0, 270, 50);

OUTPUT



ellipse() function

Declaration:

void ellipse(x, y,stangle,endangle,xradius,yradius);

Remarks:

ellipse draws an elliptical arc in the current drawing color.

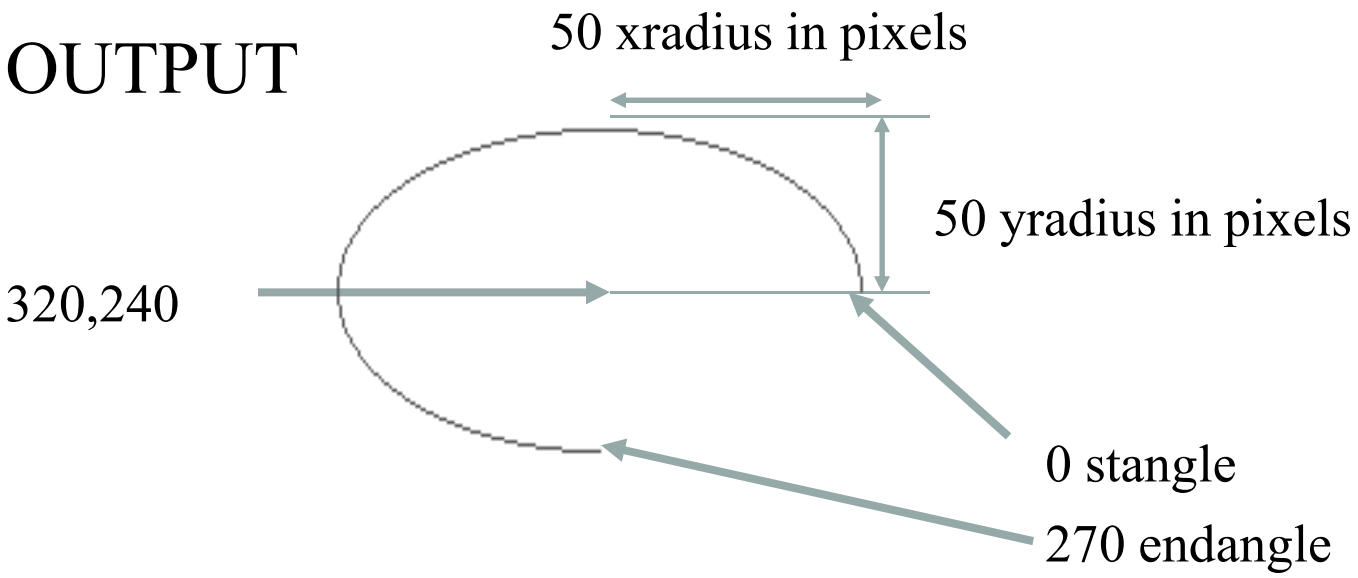
Argument	What It Is/Does
(x,y)	Center point of ellipse
stangle	Start angle in degrees
endangle	End angle in degrees
xradius	Horizontal axis
yradius	Vertical axis

eg: ellipse() function

Usage:

ellipse(320, 240, 0, 270, 100,50);

OUTPUT



fillellipse() function

Declaration:

void far fillellipse(x, y,xradius, yradius);

Remarks:

fillellipse draws an ellipse, then fills the ellipse with the current fill color and fill pattern.

Argument	What It Is/Does
(x,y)	Center point of ellipse
xradius	Horizontal axis
yradius	Vertical axis

e.g: fillellipse() function

Declaration:

void far fillellipse(x, y, xradius, yradius);

Remarks:

fillellipse draws an ellipse, then fills the ellipse with the current fill color and fill pattern.

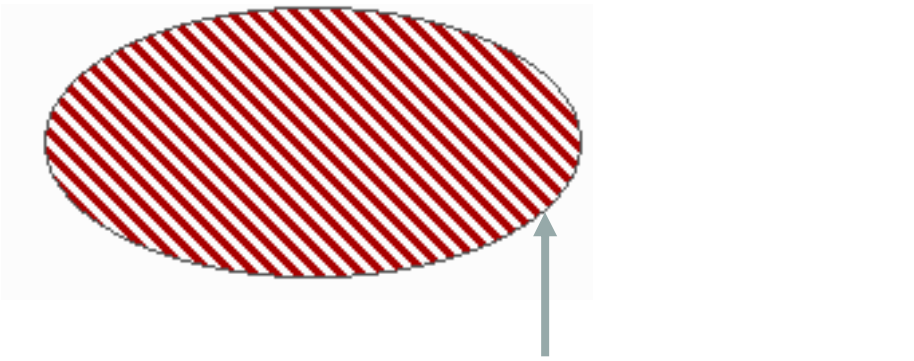
Argument	What It Is/Does
(x,y)	Center point of ellipse
xradius	Horizontal axis
yradius	Vertical axis

eg: fillellipse() function

Usage:

```
setfillstyle(5, 4);
fillellipse(320, 240, 0, 270, 100,50);
```

OUTPUT



e.g: setfillstyle() function

Declaration:

void setfillstyle(pattern, color);

Remarks:

setfillstyle sets the current fill pattern and fill color.

e.g: setfillstyle() function

Fill patterns

Names	Value	Means Fill With	Pattern
EMPTY_FILL	0	Background color	
SOLID_FILL	1	Solid fill	
LINE_FILL	2	---	
LTSLASH_FILL	3	///	
SLASH_FILL	4	///, thick lines	
BKSLASH_FILL	5	\\\, thick lines	
LTBKSLASH_FILL	6	\\\	
HATCH_FILL	7	Light hatch	
HATCH_FILL	8	Heavy crosshatch	
INTERLEAVE_FILL	9	Interleaving lines	
WIDE_DOT_FILL	10	Widely spaced dots	
CLOSE_DOT_FILL	11	Closely spaced dots	
USER_FILL	12	User-defined fill pattern	User Defined

putpixel() function

Declaration:

void putpixel(x, y, color);

Remarks:

putpixel plots a point in the color defined by color at (x,y)

Viewports

- **Viewports** provide a way to restrict to an arbitrary size the area of the screen used for drawing.
- We can draw an image that would ordinary occupy the entire screen but if a view port is in use, only part of the image will be visible.
- The **View Ports** don't scale the image; that is, the image isn't compressed to fit the view port, Rather, the parts of the image that don't fit in the view port are simply not visible

setviewport() Funciton

Sets the current viewport for graphics output

- **Declaration:**

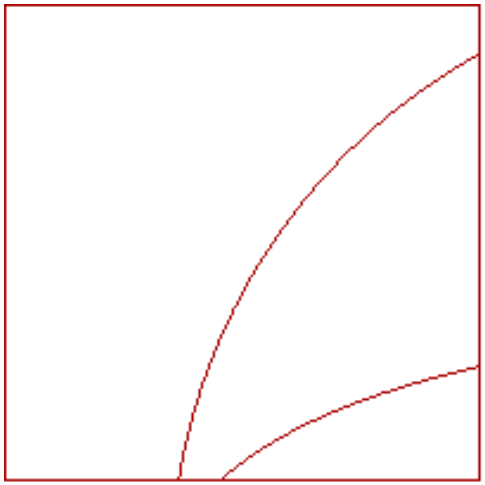
  void far setviewport(left,top,right,bottom,clip);

- **Remarks:**
  - setviewport establishes a new viewport for graphics output.
  - The viewport's corners are given in absolute screen coordinates by (left,top) and (right,bottom).
  - The clip argument determines whether drawings are clipped (truncated) at the current viewport boundaries. If clip is non-zero, all drawings will be clipped to the current viewport.

e.g: setviewport() Function

```
setviewport(0,0,200,200,1);
    rectangle(0,0,200,200);
    circle(320,240,250);
    ellipse(320,240,0,360,250,100);
```

OUTPUT:



clearviewport() Function

- Clear the current viewport.
- example:

```
circle(320,240,260);

setviewport(0,0,200,200,1);

rectangle(0,0,200,200);

circle(320,240,250);

ellipse(320,240,0,360,250,100);

getch();

clearviewport();
```

Text with Graphics

- There are functions in C language that draw text characters in graphics mode.
- These functions can be used to mix text and graphics in the same image.
- These functions also make it possible to change text font and very the size of text.

outtext() function

- outtext displays a string in the viewport (graphics mode)
- **Declaration:**

  void outtext(far *textstring);

- **Remarks:**

  outtext display a text string, using the current justification settings and the current font, direction, and size. outtext outputs textstring at the current position (CP).

outtextxy() Function

- **outtextxy** displays a string at the specified location (graphics mode)
- **Declaration:**

  void outtextxy(x, y, far *textstring);

- **Remarks:**

  outtextxy() display a text string, using the current justification settings and the current font, direction, and size.(CP)

  outtextxy() displays textstring in the viewport at the position (x, y)

settextstyle( ) Function

Sets the current text characteristics

- Declaration:

  void settextstyle(font, direction, charsize);

- Remarks:
  - settextstyle() sets the text font, the direction in which text is displayed, and the size of the characters.
  - A call to settextstyle() affects all text output by outtext and outtextxy.

settextstyle( ) Function

Direction

- Font directions supported are horizontal text (left to right) and vertical text (rotated 90 degrees counterclockwise).
- The default direction is HORIZ_DIR.

Name	Value	Direction
HORIZ_DIR	0	Left to right
VERT_DIR	1	Bottom to top

settextstyle( ) Function

Charsize

- The size of each character can be magnified using the charsize factor.
- If charsize is non-zero, it can affect bit-mapped or stroked characters.
- A charsize value of 0 can be used only with stroked fonts.

settextstyle( ) Function

- Fonts

  There are currently five fonts available . But it is easy to add other to the systems. These are,

Value	Constant	File	Comment
0	DEFAULT_FONT	Built in	Bit-mapped, 8x8
1	TIPLEX-FONT	TRIP.CHR	Stroked (Times Roman style)
2	SMALL_FONT	LITT.char	Stroked (for small lette4rs)
3	SANS_SERIF_FONT	SANS.CHR	Stroked(sans_serif style)
4	GOTHIC_FONT	GOTHIC.CHR	Stroked (gothic style)

- Strings

Strings are defined as arrays of characters.

The only difference from a character array is, a symbol “\0” is used to indicate the end of a string.

For example, suppose we have a character array, char name[8], and we store into it a string “Dave”.

Note: the length of this string 4, but it occupies 5 bytes.

D	a	v	e	\0			

UNIT-IV
INTRODUCTION TO C++

C++ program

- It include headers; these are modules that include functions that you may use in your program; we will almost always need to include the header that defines cin and cout; the header is called iostream.h

```
#include <iostream.h>

int main() {

//variable declaration

//read values input from user

//computation  and print output to user

return 0;}
```

- ✓ After you write a C++ program you compile it; that is, you run a program called **compiler** that checks whether the program follows the C++

syntax

- if it finds errors, it lists them
- If there are no errors, it translates the C++ program into a program in machine language which you can execute

Example Program

- ✓ When learning a new language, the first program people usually write is one that salutes the world :)

  Here is the Hello world program in C++.

```
#include <iostream.h>

int main() {

    count << “Hello world!”;

     return 0;

}
```

The Task of Programming

- Programming a computer involves writing instructions that enable a computer to carry out a single task or a group of tasks
- A computer programming language requires learning both vocabulary and syntax
- Programmers use many different programming languages, including BASIC, Pascal, COBOL, RPG, and C++
- The rules of any language make up its syntax
- Machine language is the language that computers can understand; it consists of 1s and 0s

✓A translator (called either a compiler or an interpreter) checks your program for syntax errors

✓A logical error occurs when you use a statement that, although syntactically correct, doesn't do what you intended

✓You run a program by issuing a command to execute the program statements

✓You test a program by using sample data to determine whether the program results are correct

Programming Universals

- All programming languages provide methods for directing output to a desired object, such as a monitor screen, printer or file
- Similarly, all programming languages provide methods for sending input into the computer program so that it can be manipulated
- In addition, all programming languages provide for naming locations in computer memory
- These locations commonly are called variables (or attributes)

- The type determines what kind of values may be stored in a variable
- Most computer languages allow at least two types: one for numbers and one for characters
- Numeric variables hold values like 13 or -6
- Character variables hold values like ‘A’ or ‘&’
- Many languages include even more specialized types, such as integer (for storing whole numbers) or floating point (for storing numbers with decimal places)

- The type determines what kind of values may be stored in a variable
- Most computer languages allow at least two types: one for numbers and one for characters
- Numeric variables hold values like 13 or -6
- Character variables hold values like ‘A’ or ‘&’
- Many languages include even more specialized types, such as integer (for storing whole numbers) or floating point (for storing numbers with decimal places)

A main( ) Function in C++

- C++ programs consist of modules called functions
- Every statement within every C++ program is contained in a function
- Every function consists of two parts:
  - A function header is the initial line of code in a C++ which always has three parts:
    - Return type of the function
    - Name of the function
    - Types and names of any variables enclosed in parentheses, and which the function receives

  A function body

Creating a main( ) Function

- A C++ program may contain many functions, but every C++ program contains at least one function, and that function is called main( )
- If the main function does not pass values to other programs or receives values from outside the program, then main( ) receives and returns a void type
- The body of every function in a C++ program is contained in curly braces, also known as curly brackets

Working with Variables

- In C++, you must name and give a type to variables (sometimes called identifiers) before you can use them.
- Names of C++ variables can include letters, numbers, and underscores, but must begin with a letter or underscore.
- No spaces or other special characters are allowed within a C++ variable name.
- Every programming language contains a few vocabulary words, or keywords, that you need in order to use the language.

A C++ keyword cannot be used as a variable name

Each named variable must have a type

C++ supports three simple types:

Integer — Floating point — Character

An integer is a whole number, either positive or negative

An integer value may be stored in an integer variable declared with the keywor

can also declare an integer variable using short int and long int

- Comments are statements that do not affect the compiling or running of a program
- Comments are simply explanatory remarks that the programmer includes in a program to clarify what is taking place
- These remarks are useful to later program users because they might help explain the intent of a particular statement or the purpose of the entire program
- C++ supports both line comments and block comments

Object oriented programming concept

- Object oriented programming is a programming style that is associated with the concept of class , object , and various other concepts revolving around these ,

- Like inheritance , polymorphism , abstraction , encapsulation .

- Computer programs are designed by making them out of objects that interact with one another .

- Object oriented programming languages typically share low level feature with high fundamental tools that can be used to construct a program include ,

- Variables which can store information formatted in a small number of built –in data types like integers and alphanumeric characters .

Object

- Objects are the basic run-time entities in an object-oriented system.
- When a program is executed, objects interact with each other by sending messages.
- Different objects can also interact with each other without knowing the details of their data or code.
- An object is an instance of a class .
- More than one instance of the same class can be in existence at any one time.

Class

- A class is a collection of objects of a similar type.
- Once a class is defined, any number of objects can be created which belong to that class.
- A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

Instance

- The instance is the actual object created at runtime.
- One can have an instance of a class or a particular object.

State

- The set of values of the attributes of a particular object is called its state.
- The object consists of state and the behaviour that's defined in the object's class.

Method

- Method describes the object's ability.
- A Bird has ability to Fly. So Fly() is one of the method of the Bird class.

Message Passing

- The process by which an object sends data to another object or asks the other object to invoke a method.
- Message passing corresponds to "method calling".

Abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes.

Encapsulation

- It is the mechanism that binds together code and data in manipulates, and keeps both safe from outside interference and misuse.
- In short, it isolates a particular code and data from all other codes and data.
- A well-defined interface controls the access to that particular code and data.
- The act of placing data and the operations that perform on the data in the same class.
- Storing data and functions in a single unit (class) is encapsulation
- Data cannot be accessible to the out side world and only those functions which are stored in the class and can access it

Inheritance

- It is the process by which one object acquires the properties of another object.
- This supports the hierarchical classification.
- Without the use of hierarchies, each object would need to define all its characteristics explicitly.
- However, by use of inheritance, an object need only define those qualities that make it unique within its class.
- It can inherit its general attributes from its parent.
- A new sub-class inherits all of the attributes of all of its ancestors.

Polymorphism

- Polymorphism means the ability to take more than one form.
- An operation may exhibit different behaviours in different instances.
- The behaviour depends on the data types used in the operation.
- It is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation. In general, polymorphism means "one interface, multiple methods", This means that it is possible to design a generic interface to a group of related activities.
- This helps reduce complexity by allowing the same interface to be used to specify a general class of action.
- It is the compiler's job to select the specific action (that is, method) as it applies to each situation.

Generalization

- Generalization describes an is-a relationship which represent a hierarchy between classes of objects.

- Other object oriented programming languages include java , c++ , PHP , perl , objective –c ,swift , and smalltalk .

Specialization

- Specialization means an object can inherit the common state and behaviour of a generic object.
- However, each object needs to define its own special and particular state and behaviour.
- Specialization means to subclass.

Advantage of OOP

- OOP provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has a clearly defined interface.
- OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer.

Exception Handling

CONTENTS

- EXCEPTION
- EXCEPTION HANDLING
- EXCEPTION HANDLING MECHANISM
- THROWING AN EXEPTION
- CATCHING AN EXEPTION
- C++ STANDARD EXEPTIONS
- DEFINE NEW EXEPTIONS

EXCEPTIONS

- Exceptions are run time anomalies or unusual conditions that a program may encounter during execution.
- Conditions such as
  - Division by zero
  - Access to an array outside of its bounds
  - Running out of memory
  - Running out of disk space
- It was not a part of original C++.
- It is a new feature added to ANSI C++.

EXCEPTION HANDLING

- Exceptions are of 2 kinds

Synchronous Exception:

- Out of rage
- Over flow

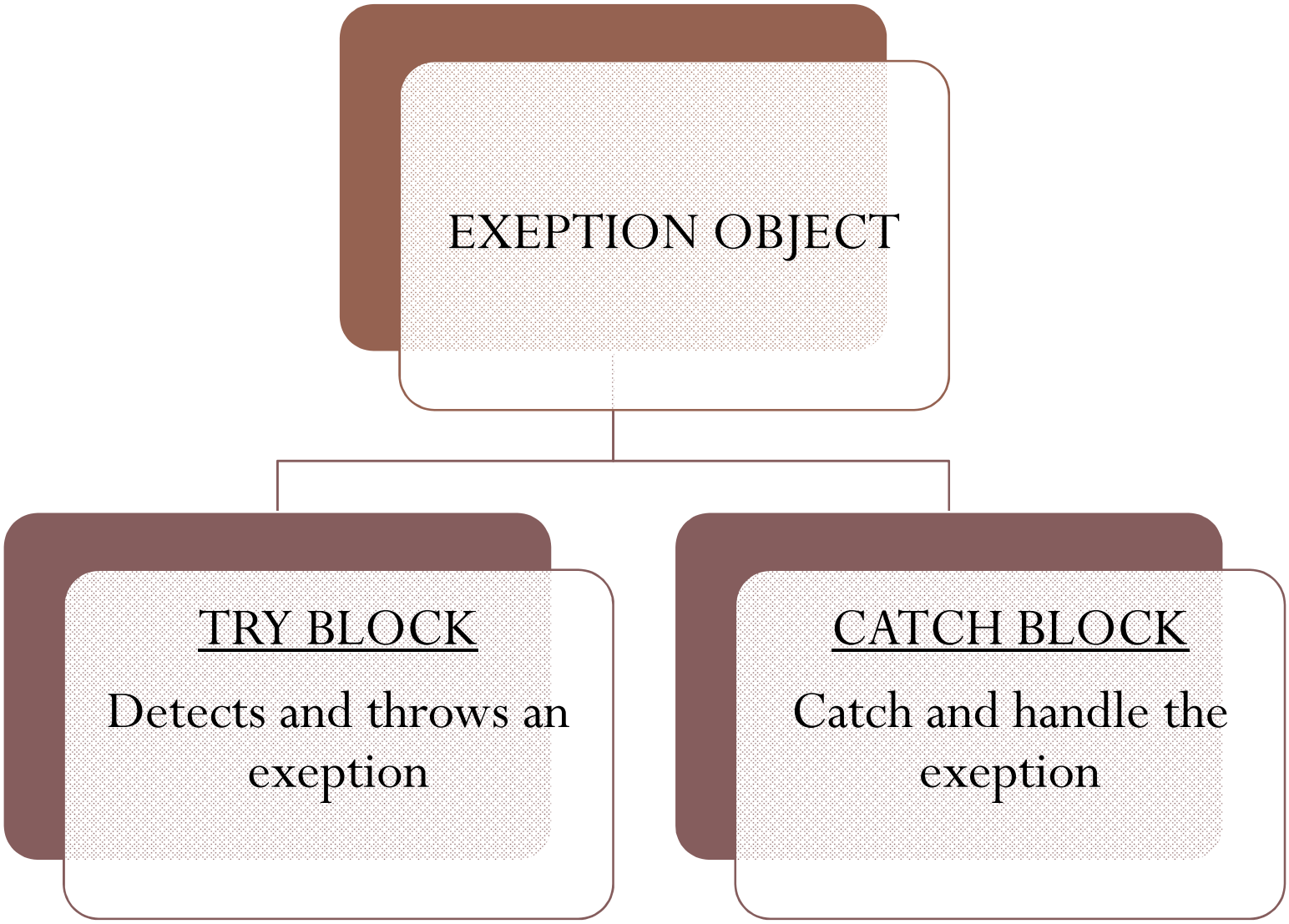
Asynchronous Exception:

- Error that are caused by causes beyond the control of the program
- Keyboard interrupts
- In C++ only synchronous exception can be handled.

EXCEPTION HANDLING MECHANISM

- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error information (Catch the exception)
- Take corrective action (handle the exception)

- It is basically build upon three keywords
  - Try
  - Throw
  - Catch


EXEPTION OBJECT
TRY BLOCK
Detects and throws an exeption
CATCH BLOCK
Catch and handle the exeption

❖ The keyword **try is used to preface a block of** statements which may generate exceptions.

❖ When an exception is detected, it is thrown using a **throw statement in the try block.**

❖ A **catch block defined by the keyword ‘catch 'catches** the exception and handles it appropriately.

❖ The catch block that catches an exception must immediately follow the try block that throws the exception.

SYNTAX

```
try
{
…                    // Block of statements
throw exception;     // which detect and
                     // throws an exception
…
}
catch(type arg)      // catch exception
{                    // Block of statement
…                    // that handles the
…                    // exception
…
}
```

❖Exceptions are objects used to transmit information about a problem.

❖ If the type of the object thrown matches the arg type in the catch statement, the catch block is executed.

❖ If they do not match, the program is aborted using the **abort() function (default).**

❖Often, Exceptions are thrown by functions that are invoked from within the try blocks.

❖ The point at which the throw is executed is called the throw point.

❖ Once an exception is thrown to the catch block, control cannot return to the throw point.

THROWING AN EXCEPTION

❖When an error condition is detected, an exception can be created and control transferred to an exception handler by executing a throw expression.

❖A throw expression consists of the operator throw optionally followed by an operand of some type.

```
if (i != 42) //detect error condition
throw i; //throw an exception
```

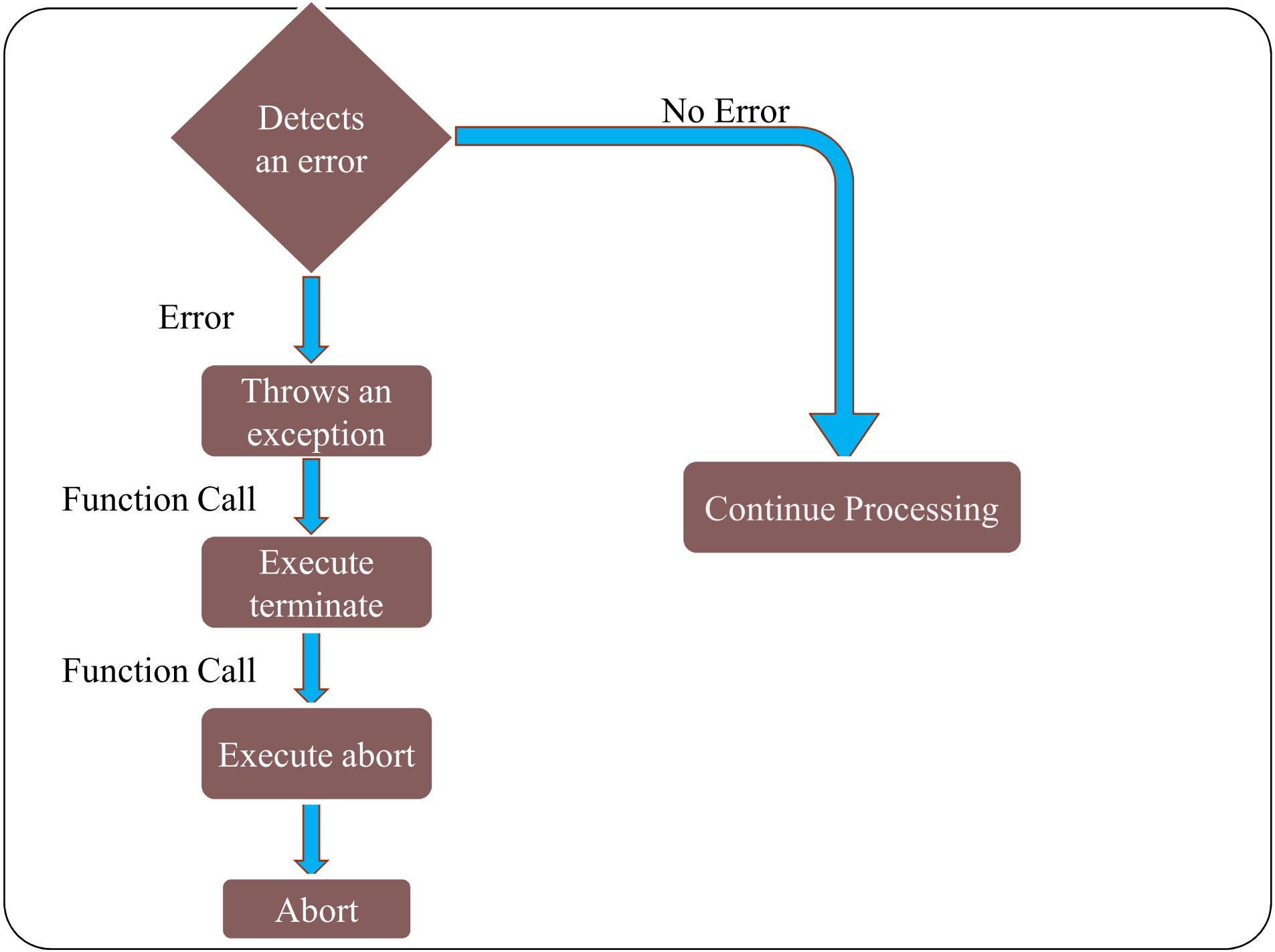
❖This causes the program to abort whenever an exception occurs. The abort occurs because the default in C++ is to abort whenever an exception is thrown that is not explicitly processed by the program. This is probably only useful for the simplest programs.

❖An exception that is thrown and is not directly detected and handled by the program results in a call to a run time library function called terminate. The default behavior for terminate is to call abort.

❖Fortunately, our program can gain control by replacing the call to abort with a call to a function that we provide.

❖We pass the address of our function to the library function set terminate. The address of the previous function is returned and the address we pass is saved in its place.

❖Whenever terminate is called, our function will also be called.


Detects an error
No Error
Error
Throws an exception
Function Call
Execute terminate
Function Call
Execute abort
Abort
Continue Processing

There are some rules that apply to the function that we supply:

1)it must not take any arguments,

2)it must not return data,

3)it must not return; it can only terminate by calling exit or abort, and

4)it is not allowed to throw an exception.

5) the header file <exception> is needed to use set_terminate.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
if( b == 0 )
{
throw "Division by zero condition!";
}
return (a/b);
}
```

CATCHING EXCEPTIONS:

The **catch block following the try block catches any exception. You can specify what type of** exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
// protected code
}catch( ExceptionNam e e )
{
// code to handle ExceptionNam e exception
}
```

Above code will catch an exception of **ExceptionName type. If you want to specify that a catch** block should handle any type of exception that is thrown in a try block, you must put an ellipsis between the parentheses enclosing the exception declaration as follows:

```
try
{
// protected code
}catch(...)
{
// code to handle any exception  }
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream >
using nam espace std;
double division(int a, int b)
{
if( b == 0 )
{
throw "Division by zero condition!";
}
return (a/b);
}
int m ain ()
{
int x = 50;
int y = 0;
double z = 0;
Try
 {
z = division(x, y);
cout << z << endl;
}catch (const char* m sg) {
cerr << m sg << endl;
}
return 0;
}
```

Because we are raising an exception of type const char*, so while catching this exception,

wehave to use const char* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

C++ STANDARD EXCEPTIONS:

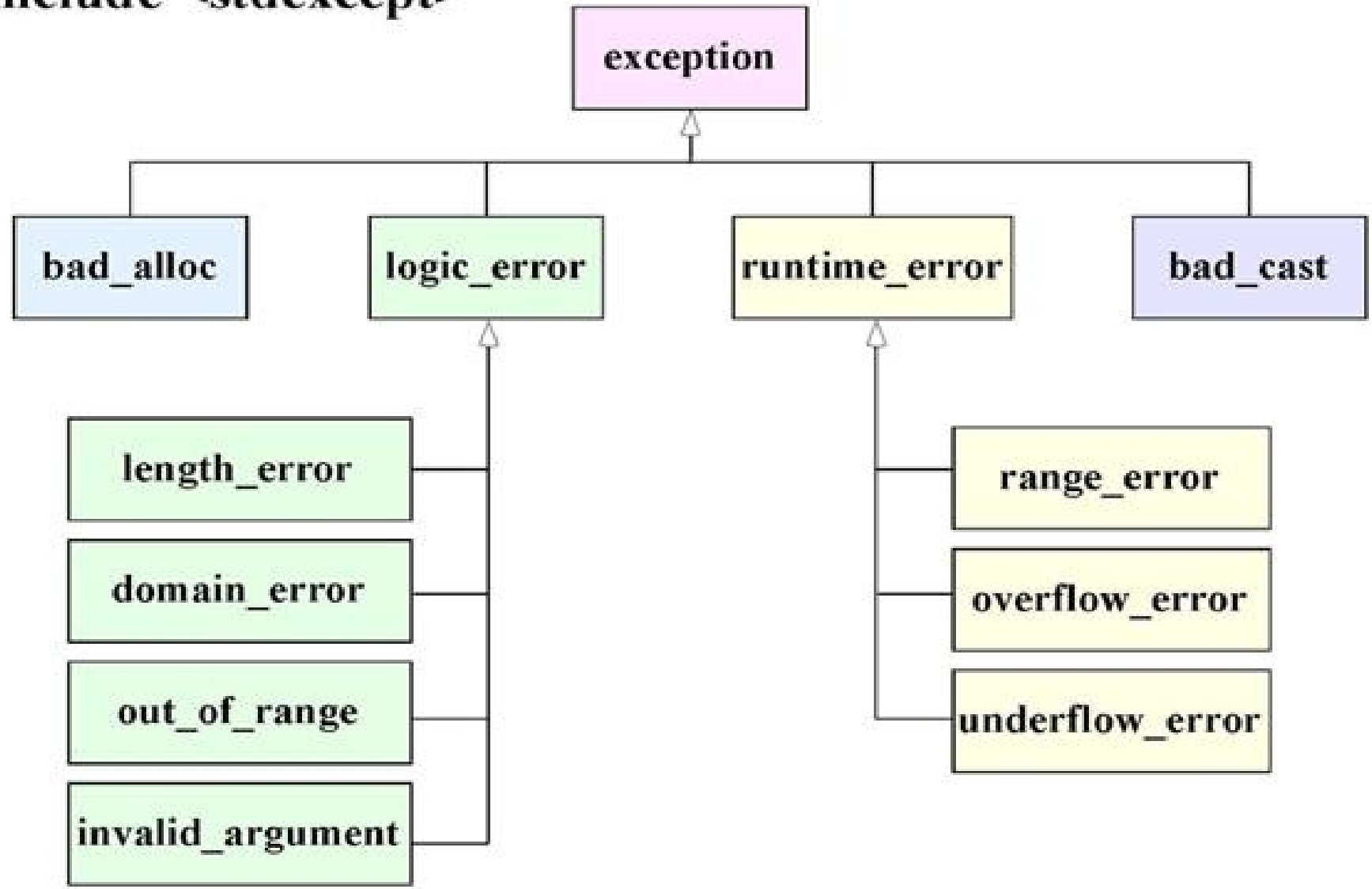
- C++ provides a list of standard exceptions defined in **<exception> which we can use in our** programs.

- These are arranged in a parent-child class hierarchy shown below:xc

Exceptions

C++ Exception Classes

#include <stdexcept>



Define New Exceptions:

You can define your own exceptions by inheriting and overriding **exception class functionality.**

Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

```
#include <iostream >
#include <exception>
using nam espace std;
struct MyException : public exception
{
const char * what () const throw ()
{
return "C++ Exception";
}
};
int main()
{
try{
throw MyException();}
catch(MyException& e)
{
std::cout << "MyException caught" << std::endl;
std::cout << e.what() << std::endl;
}
catch(std::exception& e)
{
//Other errors}}
```

```
int main()
{
try
{
throw MyException();
}
catch(MyException& e)
{
std::cout << "MyException caught" << std::endl;
std::cout << e.what() << std::endl;
}
catch(std::exception& e)
{
//Other errors
}
}
```

This would produce the following result:

My Exception caught

C++ Exception

Here, **what is a public method provided by exception class and it has been overridden by all the**

child exception classes. This returns the cause of an exception.

Learning Objectives

- C++ I/O streams.
- Reading and writing sequential files.
- Reading and writing random access files.

C++ Files and Streams

- C++ views each files as a sequence of bytes.
- Each file ends with an ***end-of-file*** marker.
- When a file is ***opened***, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- <fstream.> includes <ifstream> and <ofstream>

Creating a sequential file

```
// Fig. 14.4: fig14_04.cpp  D&D p.708
// Create a sequential file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
   // ofstream constructor opens file
   ofstream outClientFile( "clients.dat", ios::out );

   if ( !outClientFile ) {  // overloaded ! operator
      cerr << "File could not be opened" << endl;
      exit( 1 );    // prototype in stdlib.h
   }
```

Sequential file

```
cout << "Enter the account, name, and balance.\n"
        << "Enter end-of-file to end input.\n? ";
    int account;
   char name[ 30 ];
   float balance;

   while ( cin >> account >> name >> balance ) {
      outClientFile << account << ' ' << name
                    << ' ' << balance << '\n';
      cout << "? ";
   }

   return 0;  // ofstream destructor closes file
}
```

How to open a file in C++ ?

- Ofstream outClientFile(“clients.dat”, ios:out)

OR

- Ofstream outClientFile;
- outClientFile.open(“clients.dat”, ios:out)

File Open Modes

- ios:: app - (append) write all output to the end of file
- ios:: ate - data can be written anywhere in the file
- ios:: binary - read/write data in binary format
- ios:: in - (input) open a file for input
- ios::out - (output) open afile for output
- ios: trunc -(truncate) discard the files' contents if it exists

- ios:nocreate - if the file does **NOT** exists, the open operation fails
- ios:noreplace - if the file exists, the open operation fails
- The file is closed implicitly when a destructor for the corresponding object is called

  OR

- by using member function close:

  outClientFile.close();

- Instant access is possible with random access files.
- Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

UNIT-V

C and C++ PROGRAMS

DNA and RNA SEQUENCE -Using C

```
#include<stdio.h>
int main()
{
char s[50];
int i=0,n;
printf("Convert the DNA to RNA\n");
printf("\nEnter the DNA:\n");
scanf("%s",&s);


for(i=0;i<50;i++)
{
if(s[i]=='T')
{
s[i]='U';
}
}
printf("THE RNA IS:%s\n",s);
}
```

OUTPUT

Convert the DNA to RNA

Enter the DNA:
AGCTAGTT
THE RNA IS:AGCUAGUU

DNA USING LOOP

```
const int dnaNum = DNA number;
const int maxdnaLength = N;
 char dna[dnaNum][maxdnaLength] = {
{ '\0' }, //base number 0 doesn't exist, so this
row wouldn't be used
{ "acgt" },  //base number 1 is in subscript
array row 1
{ "aaacagatcacccgctgagcgggttatctgtt" },
//base number 2 in array row 2
//etc.
};
```

```
const int dnaNum = DNA number;
const int maxdnaLength = N;

char dna[dnaNum][maxdnaLength] = {
{ '\0' }, //base number 0 doesn't exist, so this
row wouldn't be used
{ "acgt" },  //base number 1 is in subscript
array row 1
{ "aaacagatcacccgctgagcgggttatctgtt" }, //base
number 2 in array row 2  //etc.
};
```

Contd…

```
for(letr = 'A'; letr < 'U'; letr++)  {
   //skip non base letters
   if(letr == 'H')
      letr = 'T';

   if(letr != 'A' && letr != 'C' && letr != 'G && letr != 'T")
      continue;

   for(letr2 = 'A'; letr2 < 'U'; letr2++)  {
      //skip non base letters
      if(letr == 'H')
            letr = 'T';
      if(letr != 'A' && letr != 'C' && letr != 'G && letr != 'T")
         continue;

      printf("%c%c  ", letr, letr2);
   }
}
```

Contd…

```
for(letr = 'A'; letr < 'U'; letr++)  {
   //skip non base letters
   if(letr == 'H')
      letr = 'T';

   if(letr != 'A' && letr != 'C' && letr != 'G && letr != 'T")
      continue;

   for(letr2 = 'A'; letr2 < 'U'; letr2++)  {
      //skip non base letters
      if(letr == 'H')
            letr = 'T';
      if(letr != 'A' && letr != 'C' && letr != 'G && letr != 'T")
         continue;

      printf("%c%c  ", letr, letr2);
   }
}
```

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

// test generate base 2
int main() {

  char base[16][3];
  char temp[3];
  char choice[] = "acgt";

  int i, j, k = 0;

  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      sprintf(temp, "%c%c\0\n", choice[i], choice[j]);    //save the string to temp
      strcpy(base[k],temp);           // save each occurrence
      k++;
    }
  }

  for (i = 0; i < 16; i++)
    printf("%s \n", base[i]);

  return 0;
}
```

OUTPUT

```
aa
ac
ag
at
ca
cc
cg
ct
ga
gc
gg
gt
ta
tc
tg
tt
```

Count the amino acids in a protein sequence

```
#include<stdio.h>
int main()
{
char s1[50];
int
j=0,a=0,v=0,i=0,l=0,m=0,f=0,y=0,w=0,k=0,r=0,h=0,
d=0,s=0,t=0,e=0,n=0,q=0,c=0,g=0,p=0;
printf("\n Enter the string :");
scanf("%s",&s1);
while(s1[j]!='\0')
{
if(s1[j]=='A')
{
a=a+1;
}
else if(s1[j]=='V')
{
v=v+1;
}
else if(s1[j]=='I')
{
i=i+1;
}
else if(s1[j]=='L')
{
l=l+1;
}
else if(s1[j]=='M')
{
m=m+1;
}
else if(s1[j]=='F')
{
f=f+1;
}
}
else if(s1[ j ]=='Y')
{
y=y+1;
}
else if(s1[j]=='W')
{
w=w+1;
}
else if(s1[j]=='K')
{
k=k+1;
```

Contd…

```
  }
else if(s1[j]=='r')
{
r=r+1;
}
else if(s1[j]=='H')
{
h=h+1;
}
else if(s1[j]=='D')
{
d=d+1;
}
else if(s1['S'])
{
s=s+1;
}
else if(s1[j]=='T')
{
t=t+1;
}
else if(s1[j]=='E')
{
e=e+1;
}
else if(s1[j]=='N')
{
n=n+1;
}
else if(s1[j]=='Q')
{
q=q+1;
}
else if(s1[j]=='C')
{
c=c+1;
}
else if(s1[j]=='G')
{
g=g+1;
}
else if(s1[j]=='P')
{
p=p+1;
}
```

Contd…

```
j=j+1;
}
if(a!=0)
{
printf("\nAlanine:%d",a);
}
if(v!=0)
{
printf("\nValine:%d",v);
}
if(i!=0)
{
printf("\nIsoleucine:%d",i);
}
if(l!=0)
{
printf("\nLeucine:%d",l);
}
if(m!=0)
{
printf("\nMethionine:%d",
m);
}
if(f!=0)
{
```

```
printf("\nPhenylanine:%d",f);
}
if(y!=0)
{
printf("\nTyrosine:%d",y);
}
if(w!=0)
{
printf("\nTryptophan:%d",w);
}
if(k!=0)
{
printf("\nLysine:%d",k); if(r!=0)
{
printf("\nArginine:%d",r);
}
if(h!=0)
{
printf("\nHistidine:%d",h);
}
if(d!=0)
{
printf("\nAspartate:%d",d);
```

Contd ...

```
}
if(s!=0)
{
printf("\nSerine:%d",s);
}
if(t!=0)
{
printf("\nThreonine:%d",t);
}
if(e!=0)
{
printf("\nGlutamate:%d",e);
}

} if(n!=0)
{
printf("\nAsparagine:%d",n);
}
if(q!=0)
{
printf("\nGlutamine:%d",q);
}
if(c!=0)
{
printf("\nCysteine:%d",c);
}
if(g!=0)
{
printf("\nGlycine:%d",g);
}
if(p!=0)
{
printf("\nProline:%d",p);
}
}
```

OUTPUT

Enter the string :AHCG

Alanine:1
Histidine:1
Cysteine:1
Glycine:1

Mismatches between two sequences

```
#include <stdio.h>
#include <string.h>
int main ()
 {
  int flag;
  char s1[1000], s2[1000];

  printf("Input first string\n");
  scanf("%s",&s1);

  printf("Input second string\n");
  scanf("%s",&s2);

  /** Passing smaller length string
first */
 if (strlen(s1) < strlen(s2))
            flag = check_subsequence(s1, s2);
  else
            flag = check_subsequence(s2, s1);

  if (flag)
            printf("YES\n");
  else
            printf("NO\n");

  return 0;
}

int check_subsequence (char a[], char b[]) {
  int c, d;

  c = d = 0;

  while (a[c] != '\0') {
            while ((a[c] != b[d]) && b[d] != '\0') {
            d++;
```

```
if (strlen(s1) < strlen(s2))
        flag = check_subsequence(s1,
s2);
  else
        flag = check_subsequence(s2,
s1);

  if (flag)
        printf("YES\n");
  else
        printf("NO\n");

  return 0;
}

int check_subsequence (char a[], char b[])
{
  int c, d;

  c = d = 0;

  while (a[c] != '\0') {
        while ((a[c] != b[d]) && b[d] !=
'\0') {
```

Contd…

```
d++;
        }
        if (b[d] == '\0')
        break;
        d++;
        c++;
}
  if (a[c] == '\0')
        return 1;
  else
        return 0;
}
```

OUTPUT

Input first string
AGCTTAG
Input second string
ACCTAGG
NO

Standard genetic code table

```
#include<stdio.h>

int main()

{

char *text[20]={

        "UUU","UCU","UAU","UGU","U","UUC","UCC",

        "UAC","UGC","C","UUA","UCA","UAA","UGA","A",

        "UUG","UCG","UAG","UGG","G"};



char *text1[20]={

        "CUU","CCU","CAU","CGU","U","CUC","CCC",

        "CAC","CGC","C","CUA","CCA","CAA","CGG","A",

        "CUG","CCG","CAG","CGG","G"};
```

Contd…

```
CUU","CCU","CAU","CGU","U","CUC","CCC",
            "CAC","CGC","C","CUA","CCA","CAA","CGG","A",
            "CUG","CCG","CAG","CGG","G"};



}
char *text2[20]={

            "AUU","ACU","AAU","AGU","U","AUC","ACC",
            "AAC","AGC","C","AUA","ACA","AAA","AGA","A",
            "AUG","ACG","AAG","AGG","G"};


char *text3[20]={

            "GUU","GCU","GAU","GGU","U","GUC","GCC",
            "GAC","GGC","C","GUA","GCA","GAA","GGA","A",
            "GUG","GCG","GAG","GGG","G"};
```

Contd…

```
int i,j=0;
/*now,display them*/
printf("\t STANDARD GENETIC CODE TABLE\n");
printf("\t ********************** \n");
printf("\tU\tC\tA\tG\n");
printf("\nU");
while(j<19)
{
for(i=0;i<5;i++)
        {
        printf("\t%s",text[j]);
        j++;
        }
        printf("\n");
        }
```

OUTPUT

STANDARD GENETIC CODE TABLE

****

U C A G

UUU UCU UAU UGU U

UUC UCC UAC UGC C

UUA UCA UAA UGA A

UUG UCG UAG UGG G

OUTPUTS

DISCUSSIONS- 5 UNITS

ASSINGNMENT-I

ASSINGNMENT-II

SEMINAR

INTERNAL EXAM-I

INTERNAL EXAM-II

QUIZ EXAM

MODEL QUESTION PAPERS DISCUSSION