Functions

What is a function?

- A number of statements grouped in a single logical unit is referred to as a function.
- The function `main()` in the program is executed first.
- When are the other functions executed?
  - When they are called directly or indirectly from the `main()` function.
- The function `main()` is user defined except its name, number and type of arguments
  - This is a complete C-program:

```
main() {  }
```

Function inside `main()`

```
#include <stdio.h>
void PrintMesg()
{
  printf("PrintMesg, Welcome to functions.\n");
}

void main()
{
  printf("Main, Wel
  PrintMesg();
}
```

```
Main, Welcome to functions.
PrintMesg, Welcome to functions.
```

A useful function

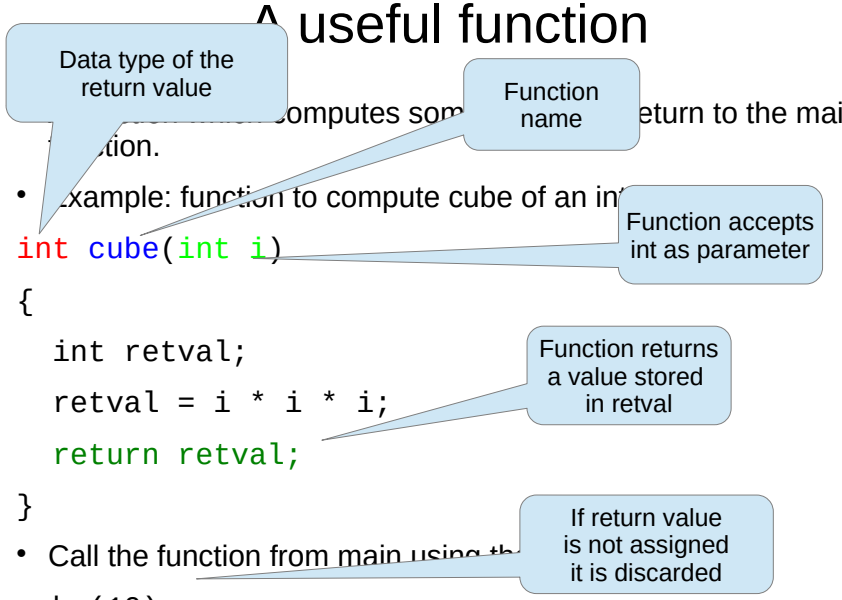
- A function which computes some value and return to the main function.
- Example: function to compute cube of an int

```
int cube(int i)
{
  int retval; /* local function variable */
  retval = i * i * i;
  return retval;
}
```

- Call the function from main using the statement

```
cube(10);
val = cube(5);
```

A useful function



- ...omputes som... ...eturn to the main ...tion.
- ...xample: function to compute cube of an in...

```
int cube(int i)
{
   int retval;
   retval = i * i * i;
   return retval;
}
```

- Call the function from main using th...

```
cube(10);
val = cube(5);
```

```
#include <stdio.h>

int cube(int i)
{
  int retval;
  retval = i * i * i;
  return retval;
}

void main()
{
  int num;
  int numCube;
  /* cube of a constant integer */
  printf("The cube of 10 is %d\n", cube(10));

  /* cube of an input integer */
  printf("Enter an integer: ");
  scanf("%d", &num);
  numCube=cube(num);
  printf("The cube of %d is %d\n", num, numCube);
}
```

```
The cube of 10 is 1000
Enter an integer: 7
The cube of 7 is 343

** another input ***

The cube of 10 is 1000
Enter an integer: 432
The cube of 432 is 80621568
```

Placement of the function

- Factor to notice: is the order of placement of the functions.
- Previous code, function cube() placed before the function main. This is appropriate as main uses cube and that it should be aware about it.
- However, if you have a large program with many functions, calling each other; it becomes difficult to maintain order of placement to ensure correctness.
- In large programs the function main is kept at the top to give an idea to the user of the program logic. But here when you compile, all functions called from main are not known.
- Therefore add declarations of all functions before main, so that the compiler knows the interface of each function.
  - Function name, return value and parameter number and types.

```c
#include <stdio.h>

/* Declaration of function cube */
int cube( int ); /* semicolon here */

void main()
{
  int num;
  int numCube;
  /* cube of a constant integer */
  printf("The cube of 10 is %d\n", cube(10));
  printf("Enter an integer: ");
  scanf("%d", &num);
  numCube=cube(num);
  printf("The cube of %d is %d\n", num, numCube);
}

/* Definition of function cube */
int cube(int i)
{
  int retval;
  retval = i * i * i;
  return retval;
}
```

Return type

- If function returns a value it must be assigned to be useful
  - `numCube=cube(num);`
- If the function does not return a value then the statement `return;` is optional
- The function declarion must say
  - `void cube( int );`
- Default return type assumed is `int`

```c
#include <stdio.h>

/* Declaration of function cube */
void cube( int );

void main()
{
  int num;
  int numCube;
  /* cube of a constant integer */
  cube(10);
  printf("Enter an integer: ");
  scanf("%d", &num);
  cube(num);
}

/* Definition of function cube */
void cube(int i)
{
  int retval;
  retval = i * i * i;

  printf("The cube of %d is %d\n", i, retval);
}
```

Function parameters

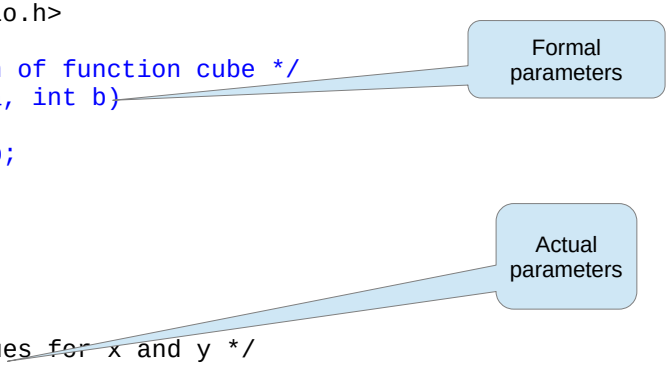
- Function parameters are a means to communicate between calling and the called functions.
- Parameter classification
  - Formal: given in function declaration and definition. Also called parameters, dummy parameters or placeholders.
  - Actual: are specified in the function call. Often known as arguments. These are the actual values sent to the function.
- Conditions:
  - Number of arguments in the function call and its declaration must be the same.
  - The data type of each argument in definition should be the same as the corresponding parameter in declaration.
  - Names of arguments in function call and the names of parameters in definition are unrelated. They can be same of different.

```c
#include <stdio.h>

/* Declaration of function cube */
int sum( int a, int b)
{
  return a + b;
}

void main()
{
  int x, y, z;

  /* Read values for x and y */
  z = sum(x, y);
}
```



Argument passing

- Two mechanisms to pass arguments to a function:
  - Call by value
  - Call by reference

Passing by value

- Values of arguments are sent to called function.
- Contents of arguments in calling function are not changed, even if they are changed in the called function.
- The content of the variable is copied to the formal parameter of the function definition, thus preserving the contents of the argument in the calling function.

```c
#include <stdio.h>

void modifierFunc(int n)
{
  printf("In function, the value of num is : %d \n", n);
  n = 19;
  printf("In function, after changing,
              the value of num is : %d \n", n);
}

void main()
{
  int num;
  num = 100;

  printf("In main, the value of num is : %d \n", num);
  modifierFunc(num);
  printf("After calling function, in main,
              the value of num is : %d \n", num);
}
```

```c
#include <stdio.h>

void modifierFunc(int n)
{
  printf("In function, the value of num is : %d \n", n);
  n = 19;
  printf("In function, after changing, the value of num is : %d \n", n);
}

void main()
{
  int num;
  num = 100;

  printf("In main, the value of num is : %d \n", num);
  modifierFunc( num );
  printf("After calling function, in main, the value of num is : %d \n", num);
}
```

```
In main, the value of num is : 100
In function, the value of num is : 100
In function, after changing, the value of num is : 19
After calling function, in main, the value of num is : 100
```

Passing by reference

- What is the solution to the previous swap program.
- If we send copies of the values they will not reflect in the main function.

- Function `scanf()` takes variables as parameters and modifies them. Remember it takes the address of the variables: `&i`
- The program of swap can send address of variables to the function to make the swap effective
- This is called pass by address or reference.
- Concept of pointers will be clear in the later part of the course.

```c
#include <stdio.h>

void swap(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}

void main()
{
  int i, j;
  printf("Input two integers: ");
  scanf("%d %d", &i, &j);
  printf("Before swapping: %d %d\n", i, j);
  swap( i, j );
  printf("After swapping: %d %d\n", i, j);
}
```

```
Input two integers: 3 5
Before swapping: 3 5
After swapping: 3 5
```

```c
#include <stdio.h>

void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void main()
{
  int i, j;
  printf("Input two integers: ");
  scanf("%d %d", &i, &j);
  printf("Before swapping: %d %d\n", i, j);
  swap( &i, &j );
  printf("After swapping: %d %d\n", i, j);
}
```

```
Input two integers: 3 5
Before swapping: 3 5
After swapping: 5 3
```

Return Values

- Functions with return values, require a return statement to send the value to the calling function.
- The return value can be a
  - Constant
  - Variable
  - User-defined data structure
  - General expression
  - Pointer to a function
  - Function call (in which case this call must return a value)
  - Note that return value cannot be an array
    - A pointer to an array can be returned

Example: arguments:no, return:yes

```
#include <stdio.h>

unsigned int sum20( void);

void main( void )
{
  printf("Program to print the sum of first 20 numbers\n");
  printf("The sum of first 20 numbers is : %u\n", sum20());
}

unsigned int sum20( void )
{
  unsigned int sum = 0;
  int i;
  for (i = 0; i<20 ; i++)
  {
    sum += i;
  }
  return sum;
}
```

```
Program to print the sum of first 20 numbers
The sum of first 20 numbers is : 190
```

Ex: returns user-defined data type

- Program to check if input string is a palindrome

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

enum Boolean { false, true };

enum Boolean isPalindrome(char string[])
{
  int left, right, len;
  enum Boolean matched = true;

  len = strlen(string);
  if(len == 0)
    return true;
  left = 0;
  right = len - 1;
/* compare the first and last letter, second and last-but-one...*/
  while (left < right && matched)
  {
    if(string[left] != string[right])
       matched = false;
    else
    {
      left++;
      right--;
    }
  }
  return matched;
}

void main()
{
  char string[40];
  printf("Program to test the given string is a palindrome\n");
  printf("Enter a string: ");
  scanf("%s", string);
  if(isPalindrome(string))
    printf("The given string %s is a palindrome\n", string);
  else
    printf("The given string %s is not a palindrome\n", string);
}
```

```
Program to test the given string is a palindrome
Enter a string: abcdcba
The given string abcdcba is a palindrome
*** another input ***
Program to test the given string is a palindrome
Enter a string: abcddcba
The given string abcddcba is a palindrome
*** another input ***
Program to test the given string is a palindrome
Enter a string: ratsdrowninwordstar
The given string ratsdrowninwordstar is a palindrome
```

```
Program to test the given string is a palindrome
Enter a string: acdbbcda
The given string acdbbcda is not a palindrome
*** another input ***
Program to test the given string is a palindrome
Enter a string: iamapalindrome
The given string iamapalindrome is not a palindrome
*** another input ***
Program to test the given string is a palindrome
Enter a string: aaabbaaaa
The given string aaabbaaaa is not a palindrome
```

Recursion

- Expressing an entity in terms of itself is called recursion.
- A function can call any function that has been defined including itself.
- Recursive functions are those in which atleast one function calls to itself.
  - Direct recursion: fn1 calls fn1
  - Indirect recursion; fn1 calls fn2 which in turn calls fn1. This can be extended to any number of functions.
- This method of problem solving substitutes a given problem with another problem of the same form in such a way that the new problem is simpler than the original.
- Two conditions must be satisfied:
  - (1) Each time a function calls itself it must be closer, in some sense to a solution.
  - (2) There must be a decision criterion for stopping the process or computation.

Recursion

Recursion ...

- In the example of factorial, each recusive call decrements the argument (satisfies condition-1). The stopping criterion is the if statement which checks for the zero argument (satisfies condition-2).
- The number of times the function calls itself is called the depth of recursion.
- Declaring variables as static within the function makes them retain their values.
- Each call of the recursive function returns to the previous instant of the recursive function...this continues till all calls are over before finally returning to the `main()` function.
- During each recursive call, a new environment is created. All local variables and arguments are localized to the current recursive call and are not accessible outside.

Example: iterative factorial

```
#include <stdio.h>
long calc_fact( unsigned int m )
{
  long fact = 1;
  unsigned int i;
  for(i = m ; i > 1 ; i--)
  {
    fact = fact * i;
  }
  return fact;
}

void main()
{
  unsigned int n;
  printf("Enter an integer: ");
  scanf("%d", &n);
  printf("The factorial of %d is %li\n", n, calc_fact(n));
}
```

```
Enter an integer: 0
The factorial of 0 is 1
***
Enter an integer: 3
The factorial of 3 is 6
***
Enter an integer: 7
The factorial of 7 is 5040
```

Example: recursive factorial

```
#include <stdio.h>
long calc_fact( unsigned int m)
{
  if(m == 0)
    return 1;
  else
    return calc_fact( m-1 ) * m;
}

void main()
{
  unsigned int n;
  printf("Enter an integer: ");
  scanf("%d", &n);
  printf("The factorial of %d is %li\n", n, calc_fact(n));
}
```

```
Enter an integer: 0
The factorial of 0 is 1
***
Enter an integer: 3
The factorial of 3 is 6
***
Enter an integer: 7
The factorial of 7 is 5040
```

Recursive factorial

- calc_fact(0)
  - Function returns 1
- calc_fact(5)
  - calc_fact(4) * 5
  - calc_fact(3) * 4 * 5
  - calc_fact(2) * 3 * 4 * 5
  - calc_fact(1) * 2 * 3 * 4 * 5
  - calc_fact(0) * 1 * 2 * 3 * 4 * 5
  - 1 * 1 * 2 * 3 * 4 * 5

Example: recursion depth

```
#include <stdio.h>

void countNumber( int n )
{
  static int depth = 1;

  printf("In the function, value of n is : %d\n", n);
  printf("\tThe depth of the call is : %d\n", depth);
  depth++;

  if( n > 1 )
    countNumber(n-1);
  printf("\n\t\tAfter recursive call,
                value of depth is : %d\n", depth);
}

void main()
{
  int num=3;
  countNumber(num);
}
```

Example: recursion depth

```
#include <stdio.h>
void countNumber( int n ) {
  static int depth = 1;

  printf("In the function, value of n is : %d\n", n);
  printf("\tThe depth of the call is : %d\n", depth);
  depth++;

  if( n > 1 )
    countNumber(n-1);
  printf("\n\t\tAfter r
}

void main()
{
  int num=3;
  countNumber(num);
}
```

```
countnumber(3)
    n = 3 , initial: depth = 1, later: depth=2

countnumber(2)
    n = 2 , initial: depth = 2, later: depth=3

countnumber(1)
    n = 1 ,  initial: depth = 3, later: depth=4

    n==1 so return

    after recursion depth = 4
  after recursion depth = 4
 after recursion depth = 4
```

Example: recursion depth

```
#include <stdio.h>
void countNumber( int n ) {
  static int depth = 1;

  printf("In the function, value of n is : %d\n", n);
  printf("\tThe depth of the call is : %d\n", depth);
  depth++;

  if( n > 1 )
    countNumber(n-1);
  printf("\n\t\tAfter recursive call, value of depth is : %d\n", depth);
}

void main()
{
  int num=3;
  countNumber(num);
}
```

```
In the function, value of n is : 3
      The depth of the call is : 1
In the function, value of n is : 2
      The depth of the call is : 2
In the function, value of n is : 1
      The depth of the call is : 3

After recursive call, value of depth is : 4

After recursive call, value of depth is : 4

After recursive call, value of depth is : 4
```

Example: add

- Program to find sum of n natural numbers

```
#include <stdio.h>

int sum(int n);

void main()

{

  int num, total;

  printf("Enter a positive number: ");

  scanf("%d", &num);

  total = sum(num);

  printf("Summation of %d natural numbers is: %d\n", num, total);

}

int sum(int n)

{

  printf("Inside function with n = %d\n", n);

  if ( n == 0)

    return 0;

  else

    return n + sum(n-1);

}
```

```c
#include <stdio.h>
int sum(int n);
void main()
{
  int num, total;
  printf("Enter a positive number: ");
  scanf("%d", &num);
  total = sum(num);
  printf("Summation of %d natural numbers is: %d\n", num, total);
}
int sum(int n)
{
  printf("Inside function with n = %d\n"
  if ( n == 0)
    return 0;
  else
    return n + sum(n-1);
}
```

```
Enter a positive number: 5
Inside function with n = 5
Inside function with n = 4
Inside function with n = 3
Inside function with n = 2
Inside function with n = 1
Inside function with n = 0
Summation of 5 natural numbers is: 15
```

Exercise examples

- Fibonacci sequence
- GCD of two numbers
- Tower of Hanoi
- Binary search

Sending parameters to `main()`

- We can send parameters to the `main()` function using the command line
  - `a.out` /* no arguments sent */
  - `a.out param1 param2` /* two arguments sent */
- Inside the program number of arguments sent to main is given in variable `argc`
- The list of parameters is available in an array `char **argv`

Example

```c
#include <stdio.h>

void main(int argc, char ** argv)
{
  int i;
  printf("Program to print command line arguments\n");
  printf("The number of command line
            arguments are: %d\n", argc);

  for( i = 0 ; i < argc ; i++ )
  {
    printf("argv[%d]: %s\n", i, argv[i]);
  }
}
```

Example

```
#include <stdio.h>

void main(int argc, char ** argv)
{
  int i;
  printf("Program to print command line arguments\n");
  printf("The number of command line
            arguments a

  for( i = 0 ; i < argc
  {
    printf("argv[%d]: %
  }
}
```

```
$> ./a.out
Program to print command line arguments
The number of command line arguments are: 1
argv[0]: ./a.out

$> ./a.out one 2 three 4 5
Program to print command line arguments
The number of command line arguments are: 6
argv[0]: ./a.out
argv[1]: one
argv[2]: 2
argv[3]: three
argv[4]: 4
argv[5]: 5
```

Scope and Extent

Scope and Extent

- Every variable in a program has some memory associated with it.
- Memory for variables is allocated and released at different points in the program.
- For example, for automatic variables declared in functions, memory is allocated when the function starts executing, and released when the function returns.
- Variables declared outside all functions are called global variables.

- The region of source code over which the declaration of an identifier is visible is called the scope of the identifier.

Example: Scope

```
#include <stdio.h>
void func() {
  int j = 20;
  printf("j is a variable is func.\n");
  printf("Value of j is %d\n", j);
}
void main() {
  int i = 10;
  printf("i is a variable in main.\n");
  printf("Value of i is %d\n", i);
  printf("Calling func...\n");
  func();
  /* this stmt gives compilation error */
  j=30;
}
```

Example: Scope

```
#include <stdio.h>
void func() {
  int j = 20;
  printf("j is a variable is func.\n");
  printf("Value of j is %d\n", j);
}
void main() {
  int i = 10;
  printf("i is a variable
  printf("Value of i is %
  printf("Calling func...
  func();
  /* this stmt gives co      ation error */
  j=30;
}
```

scope1.c: In function ‘main’:
scope1.c:18:3: error: ‘j’ undeclared (first use in this function)
scope1.c:18:3: note: each undeclared identifier is reported only once for each function it appears in

Example: Scope

```
#include <stdio.h>
void func() {
  int j = 20;
  printf("j is a variable is func.\n");
  printf("Value of j is %d\n", j);
}
void main() {
  int i = 10;
  printf("i is a variable in main.\n");
  printf("Value of i is %d\n", i);
  printf("Calling func...\n");
  func();
}
```

i and j are local variables, since their visibility is restrcited to the function in which they are declared.

```
i is a variable in main.
Value of i is 10
Calling func...
j is a variable is func.
Value of j is 20
```

Example: global variable

```
#include <stdio.h>
/* g is a global variable */
int g;
void func()
{
  printf("In func, g is visible
   here, since it is global.\n");
  printf("Value of g in func
          is %d\n", g);
  printf("incrementing g in
         func...\n");
  g++;
}
```

```
void main() {
  printf("In main, g is visible here,
          since it is global.\n");
  printf("Assigning 20 to g in
          main...\n");
  g = 20;
  printf("Value of g in main
           is %d\n", g);
  printf("Calling func...\n");
  func();
  printf("func returned\n");
  printf("In main again, value of g
          is %d\n", g);
}
```

Example: global variable

```
In main, g is visible here, since it is global.
Assigning 20 to g in main...
Value of g in main is 20
Calling func...
In func, g is visible here, since it is global.
Value of g in func is 20
incrementing g in func...
func returned
In main again, value of g is 21
```

Scope in nested blocks

```
#include <stdio.h>
void main() {
  int i =144, j=132;
  printf("i = %d\n", i);
  {
    /* nested block */
    int k = 12;
    printf("Enter a value for k: ");
    scanf("%d", &k);
    i = i%k;
  }
  if( i==0 )
    printf("i is a divisor of %d\n", k);
}
```

```
nestblk1.c: In function ‘main’:
nestblk1.c:14:38: error: ‘k’ undeclared
            (first use in this function)
nestblk1.c:14:38: note: each undeclared
          identifier is reported only once
          for each function it appears in
```

Scope in nested block

```
#include <stdio.h>
void main() {
  int iNum=5, jNum=30;
  printf("Before the block, iNum=%d, jNum=%d\n", iNum, jNum);
  {
    int iNum = 10;
    printf("Inside the block,  iNum=%d, jNum=%d\n", iNum, jNum);
  }
  printf("Outside the block,  iNum=%d, jNum=%d\n", iNum, jNum);
}
```

Scope in nested block

```
#include <stdio.h>
void main() {
  int iNum=5, jNum=30;
  printf("Before the block, iNum=%d, jNum=%d\n", iNum, jNum);
  {
    int iNum = 10;
    printf("Inside the block,  iNum=%d, jNum=%d\n", iNum, jNum);
  }
  printf("Outside the block,  iNum=%d, jNum=%d\n", iNum, jNum);
}
```

```
Before the block, iNum=5, jNum=30
Inside the block,  iNum=10, jNum=30
Outside the block,  iNum=5, jNum=30
```

File scope

- Variables declared outside all functions are called global variables
- Global variables are of two types:
  - Static or file static: scope limited to the file in which it is declared
  - Global: which can be accessed across files which are linked together

Example: File scope

```
#include <stdio.h>
int iGlobal = 35, Number = 47;
static int iVal = 99;
void main()
{
  int Number = 1000;
  printf("The value of local variable Number is %d\n", Number);
  printf("The value of global variable iGlobal is %d\n", iGlobal);
  printf("The value of static global variable iVal is %d\n", iVal);
}
```

```
The value of local variable Number is 1000
The value of global variable iGlobal is 35
The value of static global variable iVal is 99
```

Extent

- The period of time during which memory is associated with a variable is called the extent of the variable
- Storage classes define the extent of a variable:
  - Auto: default. Can be accessed within the function or nested block in which it is declared
  - Register: prefix register. Are stored in registers of the microprocessor. Usage: `register int iIndex;` This will keep loop index variable which is accessed more often, in the register. Helps to improve performance.
  - Static:
    - Delcared within a function are static to the function
    - Declared in a file are static to the file. Used when multiple files are linked to generate one executable code.
  - Extern: global variable is accessible in all functions inside the file. If you want this variable to be accessible across different files, use the keyword `extern`. For extern variables, only one file must have the variable defined.

Example: function static

```
#include <stdio.h>
void printCount() {
  static int count = 1;
  /* count is initialised only at the first call */
  printf("Count = %d\n", count);
  count = count + 1;
  /* the incremented value of count is retained */
}
void main() { /* count is not accessible in main function */
  printCount();
  printCount();
  printCount();
}
```

```
Count = 1
Count = 2
Count = 3
```

Example: fibonnaci series

```
#include <stdio.h>
void printNextFibo()
{
  static unsigned int uF1=0,
                      uF2=1;
  unsigned int uFpre;
  uFpre = uF1 + uF2;
  printf("%u,\t", uFpre);
  uF1=uF2;
  uF2=uFpre;
}
```

```
void main()
{
  int num=0;
  int index;
  printf("Fibonacci sequence using
          static variables.\n");
  printf("Enter how many numbers are to
         be generated: ");
  scanf("%d", &num);
  printf("The first %d fibonacci numbers
            are: \n", num);
  for(index=0 ; index < num ; index++)
    printNextFibo();
  printf("\n");
}
```

```
Fibonacci sequence using static variables.
Enter how many numbers are to be generated: 4
The first 4 fibonacci numbers are:
1,   2,   3,   5,
```

```
Fibonacci sequence using static variables.
Enter how many numbers are to be generated: 8
The first 8 fibonacci numbers are:
1,   2,   3,   5,   8,   13,  21,  34,
```

```
Fibonacci sequence using static variables.
Enter how many numbers are to be generated: 12
The first 12 fibonacci numbers are:
1,   2,   3,   5,   8,   13,  21,  34,  55,  89,  144, 233,
```

Example: global static

- 3 programs:
  - `add.c`
  - `multiply.c`
  - `prog1.c` – has main() which calls functions in other two files.
- Compile all 3 files:
  - `$> gcc prog1.c add.c multiply.c`
- Global variables in each file are now accessible in all 3 files. If we have same variable name in multiple files, we get compile-time error.
- To use same global variable name in different files, define them as static
- This is file static variables or global file static variables.

add.c and multiply.c

```c
#include <stdio.h>
int done=0;
void add(int a, int b)
{
  printf("Addition not yet over\
   n");
  printf("Done = %d\n", done);
printf("%d + %d = %d\n", a, b,
   (a+b));
  done++;
  printf("Addition is over\n");
  printf("Done = %d\n", done);
}
```

```c
#include <stdio.h>
int done=4;
void multiply(int a, int b)
{
  printf("Multiply not yet over\
   n");
  printf("Done = %d\n", done);
  printf("%d * %d = %d\n", a,b,
   (a*b));
  done++;
  printf("Multiply is over\n");
  printf("Done = %d\n", done);
}
```

prog1.c

```c
#include <stdio.h>
void add (int, int);
void multiply (int, int);
void main()
{
  int num1, num2;
  printf("Enter two numbers: ");
  scanf("%d %d", &num1, &num2);
  printf("In main: calling function in add.c\n");
  add(num1, num2);
  printf("In main: calling function in multiply.c\n");
  multiply(num1, num2);
}
```

```
$>] gcc prog1.c add.c multiply.c
/tmp/ccxEc7Uh.o:(.bss+0x0): multiple definition of `done'
/tmp/ccrwonrP.o:(.bss+0x0): first defined here
collect2: ld returned 1 exit status
```

Use static int in add.c, multiply.c

```c
#include <stdio.h>
static int done=0;
void add(int a, int b)
{
  printf("Addition not yet over\
   n");
  printf("Done = %d\n", done);
printf("%d + %d = %d\n", a, b,
   (a+b));
  done++;
  printf("Addition is over\n");
  printf("Done = %d\n", done);
}
```

```c
#include <stdio.h>
static int done=4;
void multiply(int a, int b)
{
  printf("Multiply not yet over\
   n");
  printf("Done = %d\n", done);
  printf("%d * %d = %d\n", a,b,
   (a*b));
  done++;
  printf("Multiply is over\n");
  printf("Done = %d\n", done);
}
```

```
$> gcc prog1.c add.c multiply.c
$> ./a.out
Enter two numbers: 2 3
In main: calling function in add.c
Addition not yet over
Done = 0
2 + 3 = 5
Addition is over
Done = 1
In main: calling function in multiply.c
Multiply not yet over
Done = 4
2 * 3 = 6
Multiply is over
Done = 5
```

`extern` variables

- Global variables are global to the file in which they are defined.
- They can be used when we need to use the same variable across files.
- Use of global variables is not recommended, as function independence is the main idea behind modular programming. Avoid using global variables.
- If you need to access a variables across different files, use the keyword `extern` to declare it in all files, except in one file. Linker wants that only one file must have definition of the variable.

Example

- Two files:
  - subfile.c: has function incrGlobal()
  - Mainfile.c: has main() and uses function incrGlobal
- Both files declare global variable as static.
- Linker does not give any error.
- But the global variable is different in each file !
- Each file modifies its own global variable.

subfile.c and mainfile.c

```c
#include <stdio.h>

static int globalVar = 0;

void incrGlobal()

{

  printf("In subfile.c:
  globalVar is %d\n",
  globalVar);

  globalVar++;

  printf("In subfile.c:
  afterincrementing
  globalVar is %d\n",
  globalVar);

}
```

```c
#include <stdio.h>

static int globalVar=10;

/* function prototype */

void incrGlobal();

void main()

{

  printf("In mainfile.c: before calling
   function, globalVar is %d\n",
   globalVar);

  printf("Calling the function
   incrGlobal()\n");

  incrGlobal();

  printf("In mainfile.c: after calling
   function, globalVar is %d\n",
   globalVar);

}
```

```
$> gcc subfile1.c mainfile1.c
$> ./a.out
In mainfile.c: before calling function, globalVar is 10
Calling the function incrGlobal()
In subfile.c: globalVar is 0
In subfile.c: after incrementing globalVar is 1
In mainfile.c: after calling function, globalVar is 10
```

Using extern

- We need some mechanism for the following:
  - When we compile file subfile.c, we should inform the compiler that a variable globalVar exists somewhere; and the exact location will be taken care of by the **linker**.
  - When we compile mainfile.c the proper definition of globalVar must be used.
- The keyword `extern` does the first requirement, i.e. Informs compiler that this globalVar will be available somewhere.

subfile.c and mainfile.c

```c
#include <stdio.h>

extern int globalVar;

void incrGlobal()

{

  printf("In subfile.c:
  globalVar is %d\n",
  globalVar);

  globalVar++;

  printf("In subfile.c:
  after incrementing
  globalVar is %d\n",
  globalVar);

}
```

```c
#include <stdio.h>

int globalVar=10;

/* function prototype */

void incrGlobal();

void main()

{

  printf("In mainfile.c: before
   calling function, globalVar is
   %d\n", globalVar);

  printf("Calling the function
   incrGlobal()\n");

  incrGlobal();

  printf("In mainfile.c: after
   calling function, globalVar is
   %d\n", globalVar);

}
```

Compiler and Linker

```
$> gcc subfile1.c mainfile1.c
$> ./a.out
In mainfile.c: before calling function, globalVar is 10
Calling the function incrGlobal()
In subfile.c: globalVar is 10
In subfile.c: after incrementing globalVar is 11
In mainfile.c: after calling function, globalVar is 11
```

- In `mainfile.c` we just say `int globalVar`. As it is global, any other files linked with `mainfile.c` can use `globalVar`.
- Compiler produces object file for `subfile.c` and does not know the exact location for globalVar. It just notes the places where it is used.
- For `mainfile.c` the compiler keeps information about the variable globalVar, that this variable can be used by other files.
- When linker links the two object files, it knows the location of `globalVar` from the object file of mainfile.c. It uses this information to complete the encoded information about the locations where globalVar is accessed in subfile.c.