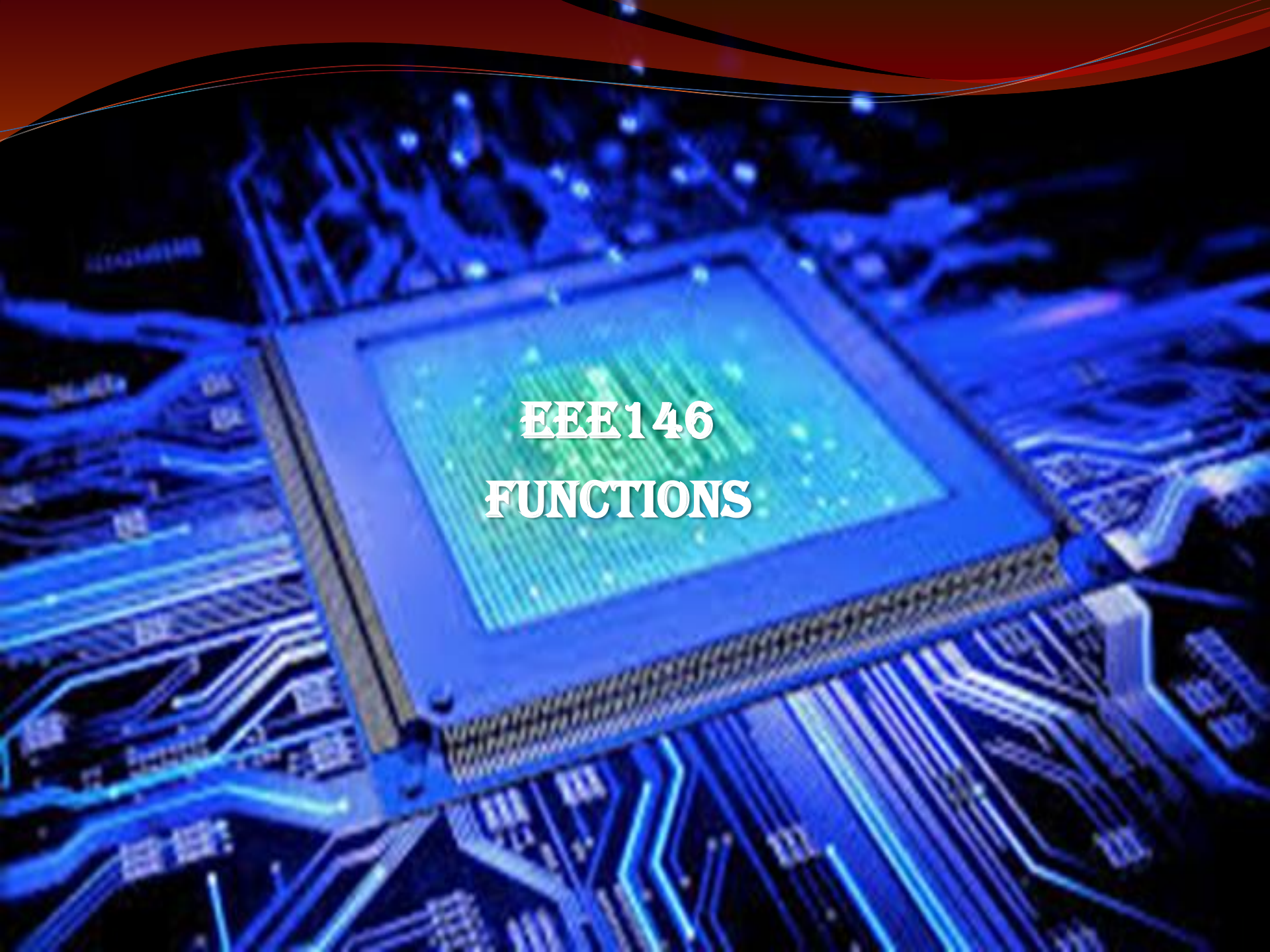

EEE146
FUNCTIONS

Functions

- A function groups a set of related statements under a single title.
- You can "call" the function using its name.
- You may also provide parameters to the function during the call.
- A function performs the actions defined by its statements and returns a result.

Motivation

1. Programming-in-the-large is feasible using functions: Divide and conquer.
   - Easier to develop
   - Easier to read and understand
   - Easier to maintain
   - Easier to reuse
   - Provides abstraction
2. You can call a function several times (instead of repeating the same group of statements over and over).
3. You may also provide different parameters to the function during each call.

Functions

- Syntax:

```
return_type function_name (parameter_list)
{  local_variable_definitions
   statement(s)
}
```

- Functions operate on their parameters and produce results.
  - The result is returned via the return value.
- The parameters may or may not be altered by the function.
- If return_type is missing, it means int.

Example

Consider the polynomial

$$P(x) = 8x^5 + 5x^4 + 6x^3 + 3x^2 + 4x + 2$$

Read a value for x and display the result.

Example: Solution without functions

```
#include <iostream>
using namespace std;
int main()
{
   float P_x, x, t;
   int i;

   P_x=0;
   cin>>x;
   /* Calculate term x^5 */
   for (t=1, i=0; i<5; i++)
      t *= x;
   P_x += 8*t;
   /* Calculate term x^4 */
   for (t=1, i=0; i<4; i++)
      t *= x;
   P_x += 5*t;
   /* Calculate term x^3 */
   for (t=1, i=0; i<3; i++)
      t *= x;
   P_x += 6*t;
   /* Calculate term x^2 */
   for (t=1, i=0; i<2; i++)
      t *= x;
   P_x += 3*t;
   /* Calculate term x^1 and x^0 */
   P_x += 4*x+2;
   cout<<"Result="<< P_x;
   return 0;
}
```

Example: Solution with functions

```
#include <iostream>
using namespace std;

float power(float a, int b)
{
   float result=1;

   for (; b>0; b--)
      result *= a;
   return result;
}
int main()
{
   float P_x, x;

   cin>>x;
   P_x = 8 * power(x,5) +
         5 * power(x,4) +
         6 * power(x,3) +
         3 * power(x,2) +
         4 * x + 2;
   cout<<"Result="<< P_x;
   return 0;
}
```

We will visit this example again.

Functions

- A function that is being called is named callee.
- The function from which the call is made is named caller.
- In the previous slide, `main()` is the caller, `power()` is the callee.

Functions

- We will cover functions in the following order:
  - Void functions
  - Functions without parameters
  - Functions with parameters
  - Functions that return a value
  - Functions that alter their parameters

Void functions

- A function that is not supposed to return a value has a return type of **`void`**.
  - Note that skipping the return type makes it an integer function, not a void function.
- If callee has nothing to send back to the caller, make callee a void function.

Example: Void functions

```
#include <iostream>
using namespace std;

void isosceles_triangle()
{  int line, i, j;

   cin>>line;
   for (i = 1; i<=line; i++)
   {
      for (j = 0; j<line - i; j++)
         cout<<" ";
      for (j = 0; j<i*2-1; j++)
         cout<<"*";
      cout<<"\n";
   }
}

int main()
{  int i, n;

   cin>> n;
   for (i=0; i<n; i++)
      isosceles_triangle();
   return 0;
}
```

Note that **`isosceles_triangle()`** makes I/O, but it does not take any parameters or return any value.

Functions with parameters

- You can direct how a function operates "within the program" if you use parameters.

Example: power() function without parameters

```
float power()
{
    float result=1, a ; int b;

    cin>>a>>b;
    for (; b>0; b--)
       result *= a;
    return result;
}
```

- **power()** finds **result** based on the values read from the input. You cannot specify **a** and **b** from the **main()** function.

Example: power() function with parameters

```
float power(float a, int b)
{
    float result=1;

    for (; b>0; b--)
        result *= a;
    return result;
}
```

- Now, `power()` finds `result` based on the parameters specified by the caller. So, you can direct it within the program

Changes in parameters are not reflected

- Note that in the previous example, the changes made on the parameters are not reflected to the caller.
- This rule applies as long as you use value parameters.

Changes in parameters are not reflected

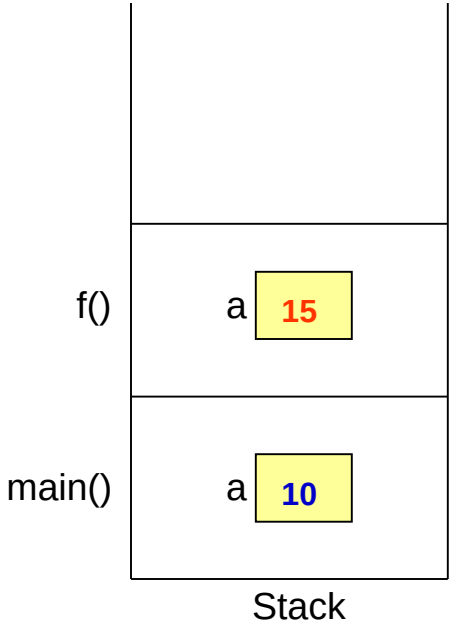
```
#include <iostream>
using namespace std;

void f(int a)
{
  a+=5;
  cout<<"in function f(): a="<< a<<endl;
}

int main()
{  int a=10;
   cout<<"in main(), before calling f(): a="<<a<<endl;
   f(a);
   cout<<"in main(), after calling f(): a="<<a<<endl;
}
```

OUTPUT

```
in main(), before calling f(): a=10
in function f(): a=15
in main(), after calling f(): a=10
```



Return value

- A function performs a task and finds a result.
- The result replaces the function call.
- Eg: For the function call below

```
y = 3*power(2,5)+4;
```

the function `power()` calculates 2^5 and the function call is replaced by `32`. So the statement becomes

```
y = 3*32+4;
```

Return value

- Note that the return value has got nothing to do with input/output.
- Eg:

```
int func()
{   int i=10;
    cout<< i;
    return i/2;
}
```

- `func()` outputs `10`, but returns `5`.

Example: Functions with return type and parameters

```
#include <iostream>
using namespace std;

/* Calculate a^b */
float power(float a, int b)
{
   float result=1;

   for (; b>0; b--)
      result *= a;
   return result;
}

int main()
{
   float P_x, x;

   cin>>x;
   P_x = 8 * power(x,5) +
         5 * power(x,4) +
         6 * power(x,3) +
         3 * power(x,2) +
         4 * x + 2;
   cout<<"Result="<< P_x;
   return 0;
}
```

Example: Functions with return type and parameters

- What is the output of the following program?

```
#include <iostream>
using namespace std;
float half(int a)
{
    return a/2;
}
int main()
{   float r;
    r=half(10)/3;
    cout<<"Result="<< r;
}
```

Global vs. local variables

```
#include <iostream>
using namespace std;
#define PI 3.14

int count;

int func()
{
   int i, j;

   cin>>i>>j;
   count += i+j;
   return 0;
}

int main()
{
   int i;

   func();
   for (i=0; i<count; i++)
      cout<<i;
   return 0;
}
```

Global variable

Local variables of func()

Local variable of main()

Scope of variables

- A local variable can be used only in the function where it is defined
  - i.e., the scope of a local variable is the current function.
- A global variable can be used in all functions below its definition
  - i.e., the scope of a global variable is the range from the variable definition to the end of the file.

Lifetime of variables

- The lifetime of a variable depends on its scope.
- A local variable is alive as long as the function where it is defined is active.
  - When the function terminates, all of the local variables (and their values) are lost.
  - When the function is called again, all variables start from scratch; they don't continue with their values from the previous call (except for local static variables which are not discussed).
- The lifetime of a global variable is equivalent to the lifetime of the program.

Scope and lifetime of variables

```
#include <iostream>
using namespace std;
#define PI 3.14

int count;

int func()
{
   int i, j;

   cin>>i>>j;
   count += i+j;
   return 0;
}

int main()
{
   int i;

   func();
   for (i=0; i<count; i++)
      cout<<i;
   return 0;
}
```

scope of count

scope of i,j

scope of i

"count" is visible in all functions

Global vs. local variables

Global variables	Local variables
Visible in all functions	Visible only within the function they are defined
Zero by default	Uninitialized
A change made by a function is visible everywhere in the program	Any changes made on the value are lost when the function terminates (since the variable is also removed)
Scope extends till the end of the file	Scope covers only the function
Lifetime spans the lifetime of the program	Lifetime ends with the function

Changing local variables

- Any change made on local variables is lost after the function terminates.

```
void f()
{   int a=10;
    a++;
    cout<<"in f(): a= " << a << endl;
}
int main()
{   int a=5;
    f();
    cout<<"After first call to f():a= "<< a << endl;
    f();
    cout<<"After second call to f():a= "<< a <<endl;
}
```

Changing global variables

- Any change made on global variables remains after the function terminates.

```
int b;
void f()
{
   b++;
   cout<<"in f(): b="<< b << endl;
}
int main()
{
   f();
   cout<<"After first call to f(): b="<< b << endl;
   f();
   cout<<"After second call to f(): b="<< b <<endl;
}
```

This is called side effect. Avoid side effects.

Changing value parameters

- Any change made on value parameters is lost after the function terminates.

```
void f(int c)
{
    c++;
    cout<<"in f(): c= " << c << endl;
}
int main()
{   int c=5;
    f(c);
    cout<<"After f(): c= " << c << endl;
}
```

Local definition veils global definition

- A local variable with the same name as the global variable veils the global definition.

```
int d=10;
void f()
{
   d++;
   cout<<"in f(): d= " << d << endl;
}
int main()
{  int d=30;
   f();
   cout<<"After first call to f(): d= " << d << endl;
   f();
   cout<<"After second call to f(): d= " << d << endl;
}
```

Parameter definition veils global definition

- Similar to local variables, parameter definition with the same name as the global variable also veils the global definition.

```
int e=10;
void f(int e)
{
    e++;
    cout<<"in f(): d= " << e << endl;
}
int main()
{   int g=30;
    f(g);
    cout<<"After first call to f(): g="<< g << endl;
}
```

Example

- Write a function that calculates the factorial of its parameter.

$$n! = n(n-1)(n-2)...1$$

```
long factorial(int n)
{  int i; long res = 1;
   if (n == 0)
      return 1;
   for (i = 1; i <= n; i++)
      res *= i;
   return res;
}
```

Example

- Write a function that calculates the permutation

  P(n, k) = n!/(n-k)!

```
long perm(int n, int k)
{  long result;
   if ((n<0)||(k<0)||(n<k))
      return 0;
   else
   {  result = factorial(n)/factorial(n-k);
      return result;
   }
}
```

Example

- Write a function that calculates the combination

$$C(n, k) = n!/[(n - k)! \cdot k!]$$

```
long comb(int n, int k)
{   long result;
    if (n < 0 || k < 0 || n < k)
        return 0;
    else
        return(factorial(n)/(factorial(n-k)*factorial(k)));
}
```

Macro Substitution

- Remember

```
#define PI 3.14
```

- It is also possible to write macros with parameters.

```
#define square(x) x*x
```

  - The name of the macro is **`square`**.
  - Its parameter is **`x`**.
  - It is replaced with **`x*x`** (with paramater **`x`** substituted.)

Macro Substitution

- Note that when the macro is called as

```
square(a+1)
```

  the substituted form will be

```
a+1*a+1
```

  which is not correct.

- The macro should have been defined as

```
#define square(x)  (x)*(x)
```

  so that its substituted form would be

```
(a+1)*(a+1)
```

Macro Substitution

- A macro is NOT a function.
  - A macro is implemented as a substitution.
    - The code segment that implements the macro substitutes the macro call.
    + Code executes faster.
    - Code becomes larger.
    + It is possible to do things that you cannot do with functions.
    - Limited syntax check (Remember the `"square(a+1)"` example).

Macro Substitution

- A function is implemented by performing a jump to a code segment.
  - Stack operations are performed for function call.
  - - Code executes slower.
  - - Code is smaller.
  - + More structured programming.
  - + Better syntax check.

Example: Macro substitution

- Define a macro for finding the maximum of two values.

```
#define max(A,B) (((A)>(B))?(A):(B))
```

Example: Macro substitution

- Define a macro for finding the absolute value.

```
#define abs(A)  ((A)>=0)?(A):-(A))
```

Variable Parameters

Value parameters

- We used "value parameters" up to now.
- We did not pass the variable in the argument; we passed the **value** of the expression in the argument.
  - That is why the changes made on the parameter were not reflected to the variable in the function call.

Variable parameters

- Sometimes, the caller may want to see the changes made on the parameters by the callee.

It is possible to return a single value using the return type; so it is not practical enough. Also, you should use the same variable both as an argument and for the return value.

Variable parameters

- Converting a value parameter to a variable parameter is easy.

```
void func(int *num)
{   *num = 5;
}
int main()
{   int count=10;
    func(&count);
}
```

Define the parameter as pointer

Use '*' before the parameter name so that you access the value at the mentioned address

Send the address of the argument

Call by reference

- The idea is very simple actually:
  - When a function terminates everything inside the stack entry for that function is erased.
  - Therefore, if you want the changes to remain **after** the function call, you should make the change outside the function's stack entry.
  - Here is what you do:
    - At the caller instead of sending the value, send the reference (address), i.e. a pointer.
    - At the callee, receive the address (i.e., define the parameter as a pointer).
    - Inside the callee, use a de-referencer ('*') with the parameter name since the parameter contains the address, not the value.

Call by reference

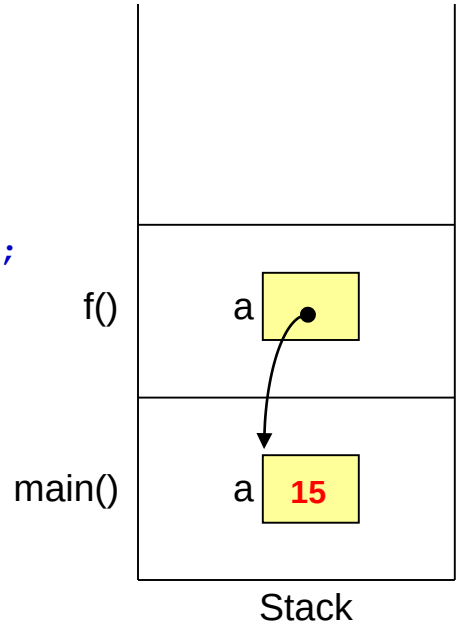
- This is why:
  - "using value parameters" is also called "call by value"
  - "using variable parameters" is also called "call by reference"

Variable parameters

```
#include <iostrem>
using namespace std;

void f(int *a)
{
  *a+=5;
  cout<<"in function f(): a= " << *a << endl;
}

int main()
{  int a=10;
   cout<<"in main(), before calling f(): a= " << a << endl;
   f(&a);
   cout<<"in main(), after calling f(): a= " << a << endl;
}
```



OUTPUT

in main(), before calling f(): a=10

in function f(): a=15

in main(), after calling f(): a=15

Example

- Write a function that exchanges its parameters.
- Solution 1:

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

WRONG!

Example

- Solution 2:

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

STILL WRONG!

Example

- Solution 3:

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

Example

- Write a program that finds the real roots of a given quadratic equation.

```
#include <iostream>
#include <cmath>
#define small 0.000001
#define equal(a,b) ((a)-(b)<=small)&&((b)-(a)<=small)
using namespace std;

float delta (float a, float b, float c)
{
  return (b * b - 4 * a * c);
}

int solve(float a, float b, float c, float *root1,
          float *root2)
{ float d;
  if (equal(a,0.0))
    return -1;
  d = delta(a,b,c);
  if (equal(d,0.0))
  { *root1 = -b / (2 * a);
    return 1;
  }
  if (d < 0)
    return 0;
  *root1 = (-b + sqrt(d)) / (2 * a);
  *root2 = (-b - sqrt(d)) / (2 * a);
  return 2;
}
```

```
int main()
{
  float r1, r2;
  float a, b, c;

  cout<<"Enter equation coefficients: ";
  cin>>a>>b>>c;
  switch (solve(a,b,c,&r1,&r2))
  { case -1: cout<<"Not quadratic equation"<<endl;
             break;
    case  0: cout<<"No real roots " <<endl;
             break;
    case  1: cout<<"One real root " <<endl;
             cout<<"Root1 = Root2 = " <<r1<<endl;
             break;
    case  2: cout<<"Two real roots\n");
             cout<<"Root1 = " <<r1<<endl;
             cout<<"Root2 = " <<r2<<endl;
             break;
  }
  return 0;
}
```

Example

- What is the output?

```
#include <iostream>
using namespace std;
int i=10, j=20, k=30, m=40;

float func(int i, int *j)
{
   int k=25;
   i++;
   (*j)++;
   k++;
   m++;
   cout<< "in func: i=" << i << "  j=" << *j << "  k=" << k << "  m=" << m << endl;
   return k/5;
}

int main()
{
   float n;

   n=func(j,&m)/3;
   cout<< "in main: i=" << i << "  j=" << j << "  k=" << k << "  m=" << m << "  n="
   << n <<endl;
   return 0;
}
```

Example

- What is the output of this one?

```
#include <iostream>
using namespace std;

int f(int x, int y)
{
    cout<<"in f(): x= " << x << "   y=" << y << endl;
    return x+y;
}
int g(int x, int y)
{
    cout<<"in g(): x= " << x << "   y=" << y << endl;
    return y-x;
}
int h(int x, int y)
{
    cout<<"in h(): x= " << x << "   y=" << y << endl;
    return x/y;
}

int main()
{
    int a=10, b=20, c=30, d=40;
    cout<<"in main(): " << h(f(a,b),g(c,d));
    return 0;
}
```

Recursive functions

- Just a simple example of recursive functions.

```
#include <iostream>
using namespace std;
long factorial(int n)
{
   if (n>0)
     return n*factorial(n-1);
   else if (n==0)
     return 1;
   return 0;
}

int main()
{
   int n;
   cout<<"Enter an integer: ";
   cin>> n;
   cout<< n <<"! =" << factorial(n)<< endl;
   return 0;
}
```

- As we said before, avoid recursive functions.

Recursive functions

```
#include <iostream>
using namespace std;
void recurse(int i)
{
  if (i<7)
   recurse(i+1);
  cout<< i <<"  ";
}

int main()
{
   recurse(0);
   return 0;
}
```

Recursive functions

```
#include <iostream>
using namespace std;
float fibonacci(int n)
{
  if (n<=2)
    return (1.0);
  else
    return(fibonacci(n-1)+fibonacci(n-2));
}
int main()
{
   float result;
   int n=0;
   while(n<=0){
     cout<<"Enter n";
     cin>>n;}
   result=fibonacci(n);
   cout<< n <<" th Fibonacci number:"<< result <<endl;
   return 0;
}
```

Homework

- Write your own **pow(x,y)** by using **log()** and **exp()** functions.

$$a^b = e^{b \log a}$$

Homework

- The greatest common divisor of integers x and y is the largest integer that evenly divides both x and y. Write a recursive function gcd that returns the greatest common divisor of x and y, defined recursively as follows: If y is equal to 0, than gcd(x,y) is x; otherwise gcd(x,y) is gcd(y,x%y),where % is modulus operator.
- (Note: For this algorithm x must be larger than y.)

Euclid Algorithm

1. Input i, j
2. If i>j interchange i and j
3. If i≤0 go to 6
4. Decrement j by i
5. Go to 2
6. Display j as greatest common divisor
7. Stop