Functions

Ritankar Mandal

Functions

A program is just a set of definitions of variables and functions.

Communication between the functions is by arguments and values returned by the functions, and through external variables.

```
1 return-type function-name(argument declarations)
2 {
3 declarations and statements
4
5 return expression;
6 }
```

```c
#include<stdio.h>
long int factorial(int n){
    int i, fac = 1;
    for(i=1; i<=n; i++)
        fac = fac*i;
    return fac;
}
int main(){
    int n;
    long int result;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    if(n < 0){
        printf("Enter a non-negative number.\n");
        return 0;
    }
    result = factorial(n);
    printf("Factorial of %d = %ld", n, result);
    return 0;
}
```

External Variables

External variables are defined outside of any function, and are thus potentially available to many functions.

External Variables

External variables are defined outside of any function, and are thus potentially available to many functions.

Internal variables are the arguments and variables defined inside functions.

External Variables

External variables are defined outside of any function, and are thus potentially available to many functions.

Internal variables are the arguments and variables defined inside functions.

Functions themselves are always external, because C does not allow functions to be defined inside other functions.

External Variables

External variables are defined outside of any function, and are thus potentially available to many functions.

Internal variables are the arguments and variables defined inside functions.

Functions themselves are always external, because C does not allow functions to be defined inside other functions.

Automatic variables are internal to a function; they come into existence when the function is entered, and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next.

Scope

The scope of a name is the part of the program within which the name can be used.

Scope

The scope of a name is the part of the program within which the name can be used.

For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared.

Scope

The scope of a name is the part of the program within which the name can be used.

For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared.

Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

Scope

The scope of a name is the part of the program within which the name can be used.

For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared.

Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled.

How to update a local variable

```
1  #include<stdio.h>
2
3  void increase(int);
4  int increase_a(int);
5  int main()
6  {
7      int i;
8      i = 1;
9      printf("value = %d \n", i);
10
11     increase(i);
12     printf("value = %d \n", i);
13
14     i = increase_a(i);
15     printf("value = %d \n", i);
16
17     return 0;
18 }
```

How to update a local variable

```
 1  void increase(int n)
 2  {
 3      n++;
 4      printf("value (inside function) = %d \n", n);
 5  }
 6
 7  int increase_a(int n)
 8  {
 9      n++;
10      printf("value (inside function) = %d \n", n);
11      return n;
12  }
```

How to NOT update a global variable

```
1  #include<stdio.h>
2  void increase(int);
3  int j;
4
5  int main()
6  {
7      j = 1;
8      printf("value = %d \n", j);
9
10     increase(j);
11     printf("value = %d \n", j);
12     return 0;
13 }
14 void increase(int n)
15 {
16     n++;
17     printf("value (inside function) = %d \n", n);
18 }
```

How to update a global variable

```
1  #include<stdio.h>
2  void increase_a();
3  int j;
4
5  int main()
6  {
7      j = 1;
8      printf("value = %d \n", j);
9
10     increase_a();
11     printf("value = %d \n", j);
12     return 0;
13 }
14 void increase_a()
15 {
16     j++;
17     printf("value (inside function) = %d \n", j);
18 }
```

Recursion

C functions may be used recursively; that is, a function may call itself either directly or indirectly.

```c
#include<stdio.h>
long int factorialRec(int n)
{
    if (n>=1)
        return n*factorialRec(n-1);
    else
        return 1;
}
int main()
{
    int n;
    long int result;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    result = factorialRec(n);
    printf("Factorial of %d = %ld", n, result);
    return 0;
}
```

The C Preprocessor: File Inclusion

`#include` is the preferred way to tie the declarations together from different source files for a large program.

Any source line of the form
`#include‘‘filename’’`
or
`#include<filename>`
is replaced by the contents of the file filename.

The C Preprocessor: Macro Substitution

A definition has the form

`#define name replacementText`

substitutes occurrences of the token name will be replaced by the replacementText.

The C Preprocessor: Macro Substitution

A definition has the form
`#define name replacementText`
substitutes occurrences of the token name will be replaced by the replacementText.

`#define` $max(A, B)((A) > (B)?(A) : (B))$
Thus the line

```
x = max(p+q, r+s);
```

will be replaced by the line

```
x = ((p+q) > (r+s) ?  (p+q) :  (r+s));
```

The C Preprocessor: Macro Substitution

A definition has the form
`#define name replacementText`
substitutes occurrences of the token `name` will be replaced by the replacementText.

`#define` $max(A, B)((A) > (B)?(A) : (B))$
Thus the line
```
x = max(p+q, r+s);
```
will be replaced by the line
```
x = ((p+q) > (r+s) ?  (p+q) :  (r+s));
```

Some care also has to be taken with parentheses to make sure the order of evaluation is preserved. Consider what happens when the macro `#define square(x) x * x`
is invoked as `square(z+1)`.