

Functions

Functions

Monolithic Programming

Functions

- Writing the whole program in main function. **(What we are currently doing)**
- What happens when program becomes bigger
 - Code becomes unreadable
 - Code becomes unmaintainable
- Recommended size is at max one page

Modular Programming

The solution to this problem is

**Structured Programming
or
Modular Programming**



**THE DAYS OF MONSTER
CODE IS OVER!!!!**

Functions

Functions

- How to Write a Structured Program?

USE



IDEA Behind Modular Programming

“Divide & Conquer.” - Decomposition Outlines

- Divide the Problem into multiple small independent problems (sub problems) and write **code** separately for each.
- Give that code block a name (Identifier)
- Whenever you see that same sub-problem again, use the function name to say “go to that code now to take care of this problem, and don’t come back until you’re done”

This block of code is known as **Function** in C

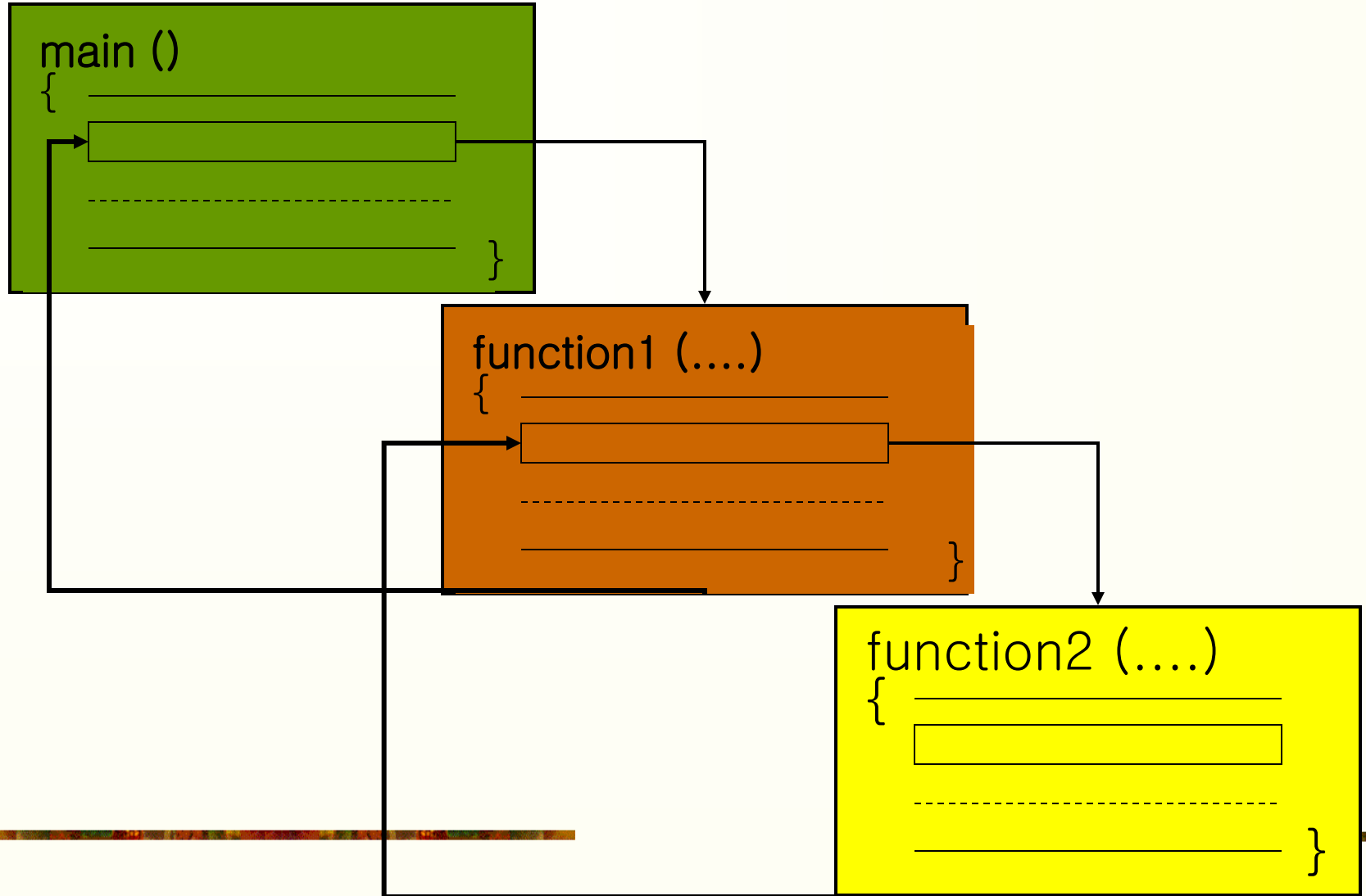
Modular Programming

Advantages

- Readable
- Easy to Maintain
- Reusable Code
- Functions can be called from anywhere in the main program with different input data
- Functions enable the programmers to generate their own libraries of the functions.

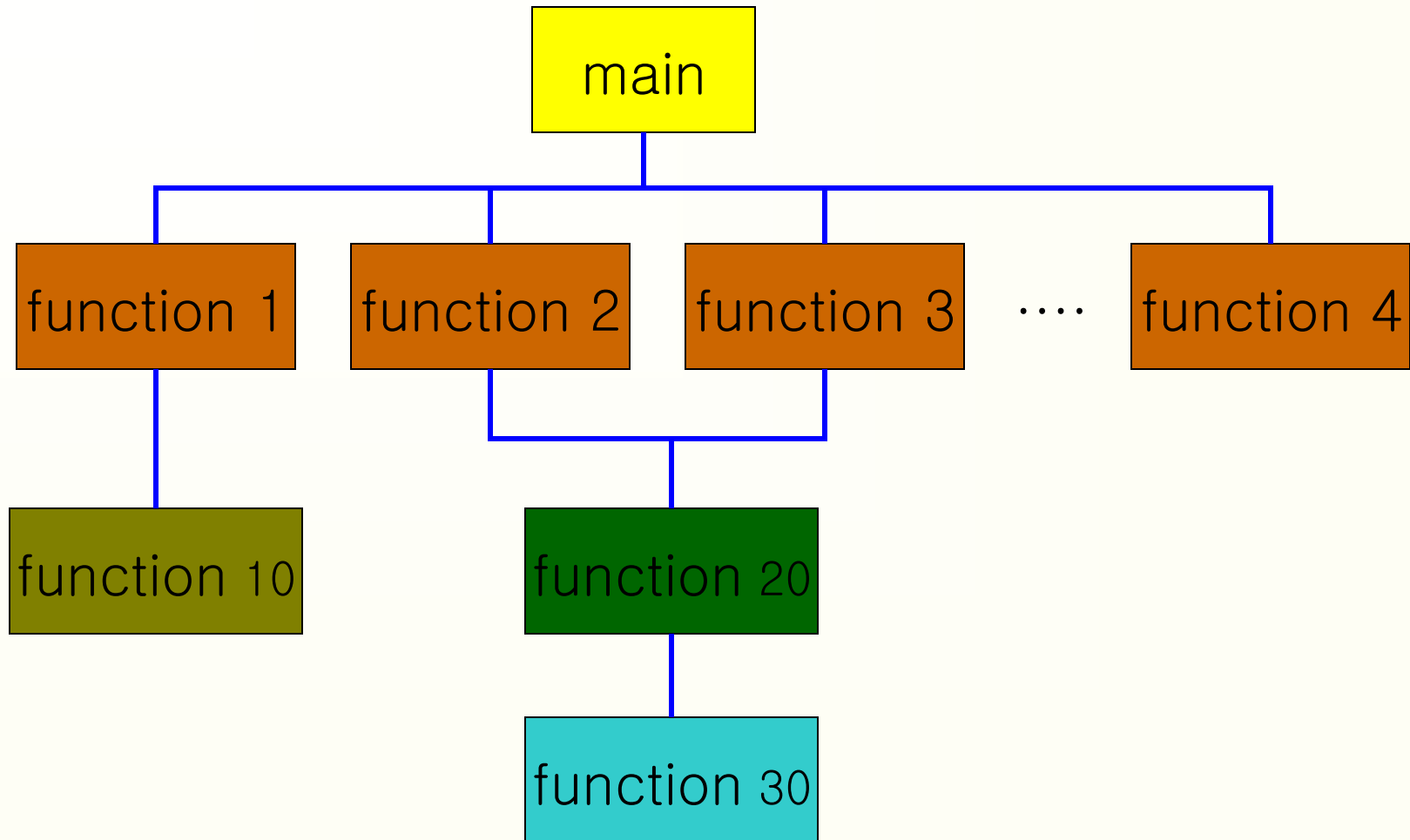
Use of Function

Functions



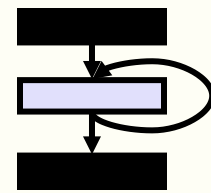
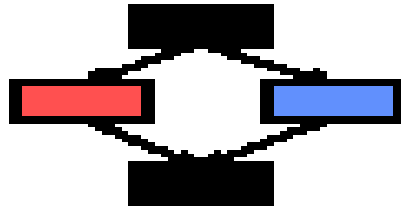
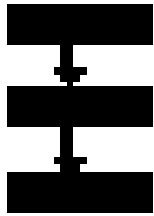
Structured Charts (Modular Charts)

Functions

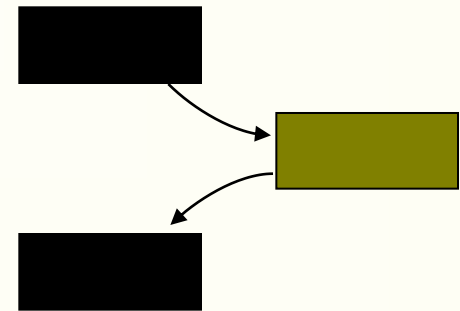


Functions Control Flow

- Control flow is the order in which statements are executed
- We've discussed two forms of Control flow so far: sequential and conditional



- “**Functions**” (or **procedures**” or “**subroutines**”) allow you to “visit” a chunk of code and then come back
- The function maybe elsewhere in your own program, or may be code in another file altogether



Why Use Functions?

- Here's one example:

- Suppose we are writing a program that displays many messages on the screen, and...
- We'd like to display two rows of asterisks ('*'s) to separate sections of output:

```
*****  
*****
```

- Moving towards a **solution**:

- The result we want is this:

```
*****  
*****
```

- And the basic code needed is this:

```
printf("*****\n");  
printf("*****\n");
```

A Full Solution

```
#include <stdio.h>
int main(void)
{ /* produce some output */
  ...
  /* print banner lines */

  printf("*****\n");
  printf("*****\n");

  /* produce more output */
  ...
  /* print banner lines */

  printf("*****\n");
  printf("*****\n");

  /* produce final output */
  ...
  return 0 ; }
```

Anything Wrong With This?

- It's correct C code
- It fulfils the problem specification, i.e., gives the desired result

What if...

- **What's “wrong” has to do with other issues such as**
 - how hard it would be change the program in the future
 - How much work is it to write the same statements over and over
- **What if? Later on the client wants us to change...**
 - The number of rows of asterisks
 - The number of asterisks per row
 - Use hyphens instead of asterisks
 - Print the date and time with each separator
 - ...
- **How much work is involved?**

Features

Functions

- A function must do only one small thing but do it very efficiently.
- A function must be small. If the printout (hardcopy) of the function exceeds one **or two** pages, then it is too long.
- Every function must hide some details.

Function In C

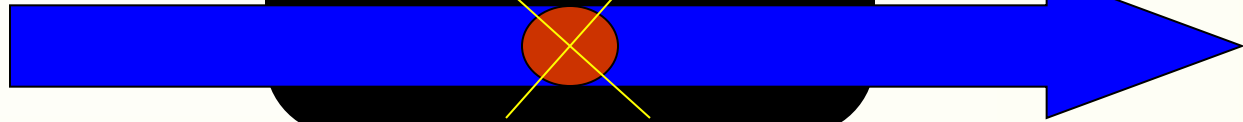
Functions

Input Variables

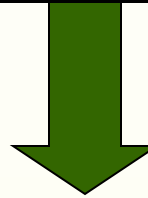


FUNCTIONS

Input/Output
Variables



Return
Variable



Function

A function is **defined** in the following manner in C.

return-type **function-name** (list of formal parameters)
{
 function code
 return return-value;
}

Should be valid
identifier

Return Values

- When one value is returned, the data type of the returned value **must be specified** before the name of the function.
- When the function does not return any value, the returned data type is specified as **void**.
- If **no return type** is specified before a function name, the returned data type is assumed to be **int by default**.

Parameter List

(type1 var1, type2 var2, . . ., type-n var-n)

EXAMPLE

widget(int n, float x, char p)

Local Variables

- A function can define its own **local variables** (Parameters are also local variables)
- The locals have meaning **only** within the function.

Local variables are **created** when the function is called.

Local variables **cease to exist** when the function returns.

```
/* Compute area of circle with radius r */
```

```
double CircleArea (double r)
```

```
{
```

```
    double x, area1;
```

```
    x = r * r ;
```

```
    area1 = 3.14 * x ;
```

```
    return area1;
```

```
}
```

Formal Parameter

Local Variables

return Statement

- The function **ends** when a return statement is encountered.
- If the return statement is followed by an expression, the value of the expression is **converted** to the return-type and used as the return value.
- If no return statement is found in a function, the function **terminates** at the last statement in the function and the return value is undefined.

Calling a Function

- A function can be **called** by any other program by **name**.
- The calling function **must supply** the arguments.
- These **arguments** must have a one-to-one correspondence with the formal parameters both in type and their place in the list of arguments

Example

If we have defined a function

void foobar(int var1, float var2, char p)

Then, we can call this function as

foobar(arg1, arg2, arg3);

Functions

Developing Simple Functions

```
void sayhello(void)
{
    printf("Hello!\n");
    return;
}
```

function name:	sayhello
return-type :	void (no value is returned)
formal parameters:	none (indicated by void)

Developing Simple Functions

```
n_abs(int n)
{
    if (n < 0)
        return (-n);
    else
        return n;
}
```

function name:	n_abs
return-type :	int (default type)
formal parameters:	int n (one parameter)

Developing Simple Functions

```
float reciproc(float x)
{
    float y;
    y = 1.0/x;
    return y;
}
```

function name:	reciproc
return-type :	float
formal parameters:	float x (one parameter)

Developing Simple Functions

```
char get_yesno(void)
{
    char ans;
    int valid;
    do
    {
        printf("\nPlease enter Y for yes or N for no >> ");
        ans=getchar();
        /* change to upper case*/
        if(ans > 96) ans -= 32;

        valid = ans=='Y' || ans=='N';
        if(!valid)
            printf("\n\aERROR: Illegal input, please try again!");
    }
    while(!valid);
    return ans;
}
```

function name:

get_yesno

return-type :

char

formal parameters:

none (use of void)

Function Prototypes

- **What is a prototype**
 - It is a statement before main to tell the compiler that there is a subprogram that will take certain parameters and return the same values
- To ensure type matching between the arguments in the calling function, we use a **function prototype declaration**

Function Prototypes

Functions

- **return-type** **function-name**(type1 dummy1, type2 dummy2, ... ,type-n dummy-n);

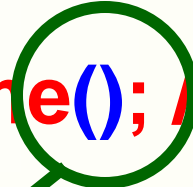
NOTE: semi-colon at end

Function Prototypes

A function with no parameters is declared as
return-type function-name(void);

and not as

return-type function-name(); // INCORRECT !



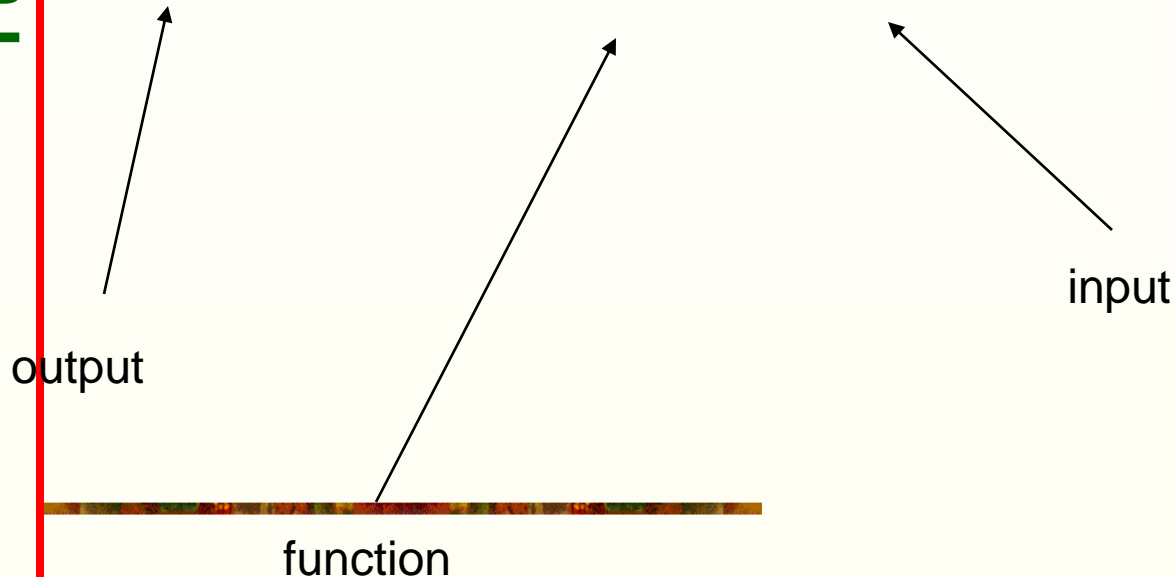
An empty list () signifies an unspecified number of formal parameters in the function.

FtoC

- We'll write a function that converts temperature in Fahrenheit to Celsius.

- `TempC = FtoC(TempF);`

Functions



FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Here is an example of a function in C.

Notice it is a part of your program, between the #includes and the main() program. And after the main () function

FtoC

Functions

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Functions are
“declared”, just like a
variable would be.

Functions can be
“float”, “int”, etc...

FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF)
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

The “arguments” (or “parameters”) of the function need to be declared as well, so that the function knows what kind of data will be the input.

FtoC

Functions

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Inside the function,
code works just like in
the main program.

You can call other
functions inside of this
function.

FtoC

Functions

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Whatever follows the “return” is “returned” to the main program.

Running FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

When the program runs, it will start at the top of the main function.

(Yes, “main” is a FUNCTION!)

Running FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Execution continues normally until you reach the line circled.

Running FtoC

Functions

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

C has no idea what this “FtoC” is, so it looks to see if it is a function, and it is!

The VALUE of tempF is “passed” to FtoC.

Running FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

Inside FtoC, the VALUE passed to FtoC is called “tF”.

This VALUE is used inside FtoC to compute tC.

tC is “returned” to the main program.

Running FtoC

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF );

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
}

float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```

So now tempC is set equal to the returned value from FtoC.

Now we could printf this value, use it in further calculations, etc.

Why Use Functions?

- Functions are “reusable”:
- The same function is used a second time in this program to printf the value of 10°F converted to Celsius.

```
#include <stdio.h>
#include <math.h>
float FtoC (float tF )

main () {
    float tempC, tempF;
    tempF = 98.6;
    tempC=FtoC(tempF);
    printf ("%f\n",FtoC(10.));
}
float FtoC (float tF )
{
    float tC;
    tC = (tF - 32.0) * (5./9.);
    return tC;
}
```


void Functions

- Not all functions return anything—they might just be a calculation or a piece of reusable code.
- Declare these as “void”.

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

void printvalue (float x )
{
    printf ("The value is %f.\n",x);
}

main () {
    float value1, value2;
    value1 = 3.14159;
    value2 = 286.1;
    printvalue(value1);
    printvalue(value2);
}
```

void Functions

- There is no “return” statement in a void function.

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

```
#include <math.h>
```

```
void printvalue (float x )
```

```
{
```

```
    printf (“The value is %f.\n”,x);
```

```
}
```

```
main () {
```

```
    float value1, value2;
```

```
    value1 = 3.14159;
```

```
    value2 = 286.1;
```

```
    printvalue(value1);
```

```
    printvalue(value2);
```

```
}
```

void Functions

- Notice that nothing in the main program “equals” these functions!

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

void printvalue (float x )
{
    printf (“The value is %f.\n”,x);
}

main () {
    float value1, value2;
    value1 = 3.14159;
    value2 = 286.1;
    printvalue(value1);
    printvalue(value2);
}
```

Multiple Inputs

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

float area (float width, float height )
{
    float A;
    A = width * height;
    return A;
}

main () {
    float x,y;
    x = 10.;
    y = 15.;
    printf ("The area of a %f m x %f m
    rectangle is %f m^2.\n",x,y,area(x,y));
}
```

Functions can have multiple sources of input, such as in this example which computes the area of a rectangle.

Multiple Inputs

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

float area (float width, float height )
{
    float A;
    A = width * height;
    return A;
}

main () {
    float x,y;
    x = 10.;
    y = 15.;
    printf ("The area of a %f m x %f m
    rectangle is %f m^2.\n",x,y,area(x,y));
}
```

Notice that both of the inputs to the function have to be declared separately!

Multiple Inputs

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

float area (float width, float height )
{
    float A;
    A = width * height;
    return A;
}

main () {
    float x,y;
    x = 10.;
    y = 15.;
    printf ("The area of a %f m x %f m
    rectangle is %f m^2.\n",x,y,area(x,y));
}
```

In this case, notice that there is still just one output (A) which is returned to the main program (and, in this case, printed).

Point To Note

- Prototype declarations are **optional** in C but they are compulsory in C++. Since, it is a reasonable assumption that every C programmer is a future C++ programmer, get into the habit of declaring prototypes for every function!

Points to Remember

- Another advantage of using function prototype declarations is that the functions can be defined in any order in the source file. Otherwise, some compilers place a restriction that a function should be used (i.e., a function call statement is placed) only after it has been defined.

CALBYVAL.C

```
#include <stdio.h>
void foobar(int );/* function declaration for foobar */
int main()
{
    int n=6;
    printf("\n Original value of n = %d",n);
    foobar(n);
    printf("\nValue of n after call to foobar = %d",n);
    return 0;
}
/* function definition for foobar */
void foobar(int n)
{
    printf("\n IN FOOBAR Input value of n = %d",n);
    n += 94;
    printf("\n IN FOOBAR Modified value of n = %d",n);
    return;
}
```

Call by Value

- When a function is called with some arguments, a copy of the values of the arguments is made available to the function. For example, defined a function

foobar(int var1, float var2, char var3)

and it is being called as

foobar(arg1, arg2, arg3);



Pointers as Function Parameters

- To allow a function to make changes to variables in the calling program, we need to pass the address of this variable to the function and then use the indirection operator to change its value.
- The previous function can be written as
`void foobar(int *x)`

Example: Pointers as Function Parameters

```
#include <stdio.h>
void foobar(int * );/* function declaration for foobar */
int main()
{
    int n=6;
    printf("\n Original value of n = %d",n);
    foobar(&n);
    printf("\nValue of n after call to foobar = %d",n);
    return 0;
}
void foobar (int *n)
{
    printf("\n IN FOOBAR Input value of n = %d",*n);
    *n += 94;
    printf("\n IN FOOBAR Modified value of n = %d",*n);
    return; }
```

NOPARAMS.C

*/*Using a function declaration with an undefined parameter list.*/*

```
#include <stdio.h>
#include <stdlib.h>
int ff();
int main()
{
    ff();
    ff(2);
    ff(2,3,4);
    return 0;
}
int ff(int i, int j, int k)
{
    printf("\nHello world %d %d %d\n",i,j,k);
    return 3;
}
```

Points to Remember

Functions

- Functions are the building blocks of a C program.
- A function prototype declares the name of a function, its return type and its list of parameters.
- Parameters are passed by value in C, i.e., every function receives a copy of the parameter being passed.

THANK YOU