CSE / ENGR 142 Programming I

Functions, Part I

Chapter 3

Read All!

- **3.1: Reusing program parts**
- **3.2: Built-in math functions**
- **3.3: Top-Down Design**
- **3.4: Functions with no parameters**
- **3.5: Functions with parameters**

Thought For Today

"A lazy person invented the wheel"

A Problem

- **Suppose we are writing a program that displays messages on the screen.**
- **We'd like to display rows of ********* to separate sections of output.**

Solution

```
#include <stdio.h>
int main(void)
{
    /* produce some output */
    ...
    /* print banner line */
    printf("********************");
    printf("********************\n");

    /* produce more output */
    ...
    /* print banner line */
    printf("********************");
    printf("********************\n");

    /* produce even more output */
    ...
    /* print banner line */
    printf("********************");
    printf("********************\n");

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

Critique

- **Redundant code**
- **What if we want to change the display**
  - **e.g., to print a blank line before and after each line of ***********?**
- **What if we want to print banner lines in some other program?**

The Solution: Functions

- Definition: *A function is a named code sequence.*
- A function can be executed by using its name as a statement or expression.
- The function may have parameters - information that can be different each time the function is executed.
- The function may compute and return a value.

Advantages (1)

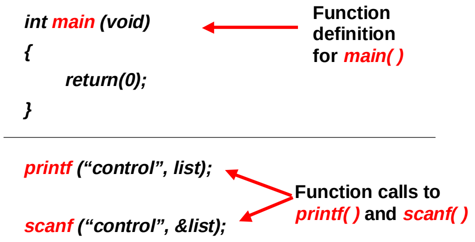
- Able to package a computation we need to perform over and over again as a single, named piece of code.
- Write once, use many times.
- Able to reuse the same operation in other programs.
- If changes are needed, they only have to be done once, in one place.

Advantages (2)

- Many programs are far to large to understand all at once.
- Functions give us a way to break a large program into smaller pieces, each of which can largely be written and understood apart from the rest of the program.

Common Functions

We have already seen and used several functions:



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

Function definition for *main( )*

```
printf ("control", list);

scanf ("control", &list);
```

Function calls to *printf( )* and *scanf( )*

More common functions

C's standard math library functions:

sqrt, pow, log, exp, sin, cos, fabs, …

```
#include <math.h>
...
x = sin( ( 2.0 * PI ) / 17.0 );
z = sqrt( 2.0 * y );
```

Your own functions vs. pre-written functions

- Pre-written functions are commonly packaged in "libraries"
  - Every standard C compiler comes with a set of standard libraries
- Remember #include <stdio.h>?
  - Tells the compiler you will use the "standard I/O library"
  - You may include as many libraries as needed
- You can define your own functions in your programs

New Solution to Problem

Function definition

```
/* print banner line */
void print_banner (void)
{
    printf(“****************”);
    printf(“****************|n”);
}
```

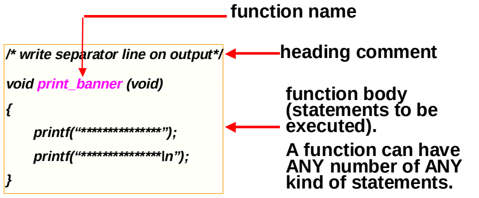
New Solution (cont)

```
int main (void)
{
   /* produce some output */
   …
   print_banner( );

   /* produce more output */
   …
  print_banner( );

  /* produce even more output */
   …
  print_banner( );

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

Empty **( )** is required when a void function is called.

Defining your own functions

- You define a function by giving its name and writing the code that is executed when the function is called.



```
/* write separator line on output*/

void print_banner (void)

{

      printf(“***************”);

      printf(“***************|n”);

}
```

function name

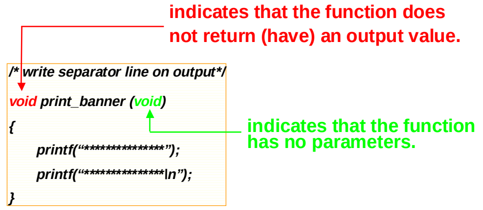
heading comment

function body (statements to be executed).

A function can have ANY number of ANY kind of statements.

void

- The keyword void has two different rolls in this function definition.



indicates that the function does not return (have) an output value.

```
/* write separator line on output*/

void print_banner (void)

{

      printf(“***************”);

      printf(“***************|n”);

}
```

indicates that the function has no parameters.

Calling a Function

- To execute the function, it is called or invoked from within a program or another function:

```
int main (void)
{
  …
  print_banner( );
  ....
  return(0);
}
```

- Note: a function that does not return a value can be called wherever a statement is allowed.

Terminology

```
int main (void)
{
   /* produce some output */
   …
   print_banner( );
   /* produce more output */
   …
  print_banner( );
  /* produce even more output */
   …
  print_banner( );
  /* produce final output */
   ....
  return(0);
}
```

"*main( )* is the call<u>er</u>."

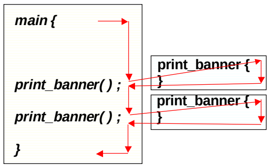
”*print_banner( )”* is the call<u>ed</u>."

- "*main( )* calls *print_banner( )* (3 times)." / "*main( )* invokes *print_banner( )* (3 times)"
- ”*print_banner( )* is called by *main( )*." / ”*print_banner( )* is called from *main( )*."

Function Control Flow

```
/* print banner line */
void print_banner (void)
{
        printf("**************");
        printf("**************\n");
}
```

```
int main ( void )
{
        ...
        print_banner( ) ;
        ...
        print_banner( ) ;
        ...
       return(0) ;
}
```



Marching Orders: Control Flow

All C programs:

1. Start at main( )   /*no matter where main is! */

2. Continue in top-to-bottom order, statement by statement, *unless* the order is changed by:

function call

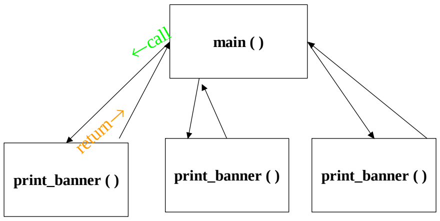
function return

if

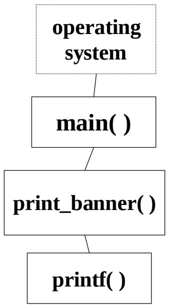
loops

will see soon

Picturing the Call and Return



Simplify and Complete the Picture: "Static Call Graph"



Function Type and Value

- A function can return a **value**.
- Like all values in C, a function return value has a **type**.
- The function is said to have the type of its returned value.

```
/* ask user for input number and */
/* return next number entered. */
int prompt (void)
{
   int k;
   printf("please enter a number: ");
   scanf("%d", &k);
   return (k);
}
```

function type (type of returned value). We say "*prompt( )* is a function of type *int*" or "*prompt( )* returns an *int*."

local variable – exists only while function is executing

return statement

returned value

Calling a Function

- A value-returning function is called by including it in an expression.

```
int main (void)
{
   int k, j;
   j = prompt( );
   k = prompt( );
   printf("the value of %d + %d is %d.",
               j, k, j+k);
   return(0);
}
```

- Note: a value-returning function can be used anywhere an expression of the same type can be used

More on *return*

- In a value-returning function (result type is not void), *return* does two distinct things:
  - 1. specify the value returned by that execution of the function
  - 2. terminate that execution of the function.
- In a void function:
  - *return* is optional at the end of the function body.
  - *return* may also be used to terminate execution of the function explicitly.
  - No return value should appear following *return*

return in *void* functions

```
/* print banner line */
void print_banner (void)
{
      printf(“***************”);
      printf(“***************|n”);
      return;
}
```

optional

```
/* do something */
void example (void)
{
      int no_reason_to_continue;
      ...
      if (no_reason_to_continue)
            return;
      ...
}
```

terminate function execution
before reaching the end

Function Parameters

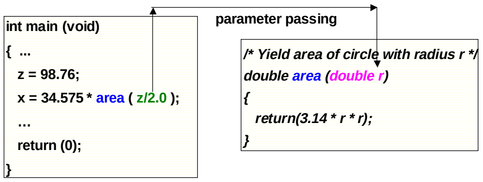
- It is very often useful if a function can operate on different data values each time it is called. Such values are function (input) parameters
  - "input" here is not I/O as we defined it earlier
- The function specifies its inputs as parameters in the function declaration.

```
/* Yield area of circle with radius r */
double area (double r)
{
   return (3.14 * r * r);
}
```

parameter

Arguments

- The function call must include a matching argument for each parameter.
- When the function is executed, the value of the argument becomes the initial value of the parameter.



parameter passing

```
int main (void)
{  ...
   z = 98.76;
   x = 34.575 * area ( z/2.0 );
   …
   return (0);
}
```

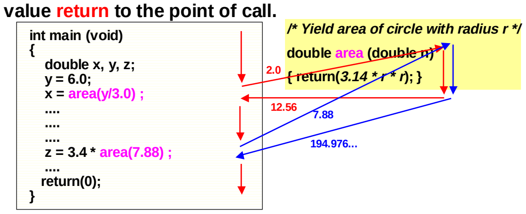
```
/* Yield area of circle with radius r */
double area (double r)
{
   return(3.14 * r * r);
}
```

Terminology

- Many people (including the textbook authors) use the term *formal parameter* instead of *parameter* and *actual parameter* instead of *argument.* We will try to stick to *parameter* and *argument* for simplicity, but the other terminology will probably slip in from time to time.
- People often refer to replacing a parameter with the argument in a function call as “passing the argument to the function”.

Control and Data Flow

- When a function is called: (1) control transfers to the function body; (2) argument values are copied; (3) the function executes; (4) control and return value return to the point of call.



```
int main (void)
{
    double x, y, z;
    y = 6.0;
    x = area(y/3.0) ;
    ....
    ....
    ....
    z = 3.4 * area(7.88) ;
    ....
   return(0);
}
```

```
/* Yield area of circle with radius r */
double area (double r)
{ return(3.14 * r * r); }
```

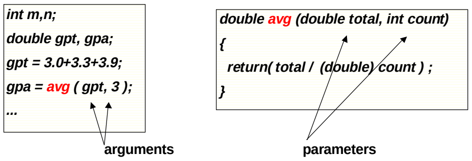
Style Points

- The comment above a function must give a complete specification of what the function does, including the significance of all parameters.
- Someone wishing to use the function should be able to cover the function body and find everything they need in the function heading and comment.

```
/* Yield area of circle with radius r */
double area (double r)
{
   return (3.14 * r * r);
}
```

Multiple Parameters

- a function may have more than one parameter
- arguments must match parameters in <u>number</u>, <u>order</u>, and <u>type</u>



```
int m,n;
double gpt, gpa;
gpt = 3.0+3.3+3.9;
gpa = avg ( gpt, 3 );
...
```

```
double avg (double total, int count)
{
  return( total /  (double) count ) ;
}
```

arguments

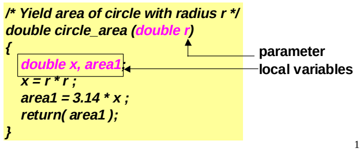
parameters

Rules for Using Functions

- Arguments must match parameters:
  - in number
  - in order
  - in type
- A function can only return one value.
  - but it might contain more than one *return* statement
- In a function with return type T, the return expression must be of type T.
- A function with return type T can be used anywhere an expression of type T can be used.

Local Variables

- A function can define its own local variables.
- The locals have meaning only within the function.
  - Each execution of the function uses a new set of locals
  - Local variables cease to exist when the function returns
- Parameters are also local.



```
/* Yield area of circle with radius r */
double circle_area (double r)
{
   double x, area1;
   x = r * r ;
   area1 = 3.14 * x ;
   return( area1 );
}
```

parameter

local variables

Declaring vs Using

Review: In general in C, identifiers (names of things) must be declared before they are used.

- Variables:

```
int  turnip_trucks;
...
turnip_trucks = total_weight / weight_per_truck;
```

- #define constants:

```
#define  TAX_RATE  0.07
...
tax_owed = TAX_RATE * income;
```

Order for Functions

Function names are identifiers, so… they too must be declared <u>before</u> they are used:

```
#include <stdio.h>
void fun2  (void)  { ... }
void fun1  (void)  { ...; fun2(); ... }
int    main (void)  { ...; fun1(); ... return 0; }
```

fun1 calls *fun2*, so *fun2* is defined before *fun1*, etc.

Alternative: Instead of writing the complete function use function <u>prototypes</u> to declare a function so it can be used.

Function Prototypes

- Looks same as start of a function definition, but ; instead of {…}

  double calculate_tax
  (double income, double rate);
- Write a function prototype near the top of the program
  - Can use the function anywhere thereafter
- Fully define the function wherever convenient
- *Highly recommended to aid program organization*

Why Have Functions?

- Reuse of program text
  - code it once but use it many times
  - saves space and improves correctness
- Centralize changes
  - changes or bug fixes made in one place
- Better program organization
  - easier to test, understand, and debug
- Modularization for team projects
  - each person can work independently

Why Have Functions (II)?

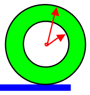
Functions raise the level of discourse

- rise above the “a+b*c” level
- see the forest, not the trees
- reshape a program into meaningful units
  - “hypotenuse”, not sqrt(a*a+b*b)
  - “volume”, not 1.04719*r*r*h

Why Have Functions (III)?

- *That’s how modern programming is done!*
- API: Application programming interface
  - Library of functions for a particular purpose
    - graphics, sound, video, windowing, statistics, etc. etc.
- Modern programming relies heavily on libraries and APIs

Example: Washer Area

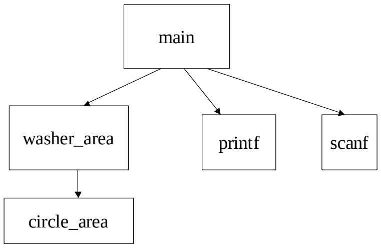


```
/* Yield area of washer with given */
/* inner and outer radius.  */
double washer_area (double inner, double outer)
{
    double area1, area2, washer ;

    area1 = circle_area (inner) ;
    area2 = circle_area (outer) ;
    washer = area2 - area1 ;
    return (washer) ;
}
```

Local Variables: putting it all together

```
#include <stdio.h>
#define PI 3.0
double circle_area (double r)
 {
     double x, area1;

     x = r * r ;
     area1 = PI * x ;
     return (area1) ;
}
double washer_area (double inner, double outer)
{
     double area1, area2, washer ;

     area1 = circle_ area (inner) ;
     area2 = circle_ area (outer) ;
     washer = area2 - area1 ;
     return (washer) ;
}

int main(void)
{
     double inner, outer, y ;

     printf (“Input inner radius and outer diameter: ”) ;
     scanf (“ %lf %lf ”, &inner, &outer) ;
     y = washer_area (inner, outer/2.0) ;

     printf (“ %f ”, y) ;

     return (0) ;
}
```

Showing How Functions are Related



A "static call graph" shows who calls who

Local Variables of main

main		
inner	outer	y

Parameters and local variables of washer_area

washer_area				
inner	outer	area1	area2	washer

Parameters and local variables of circle_area

circle_area		
r	x	area1

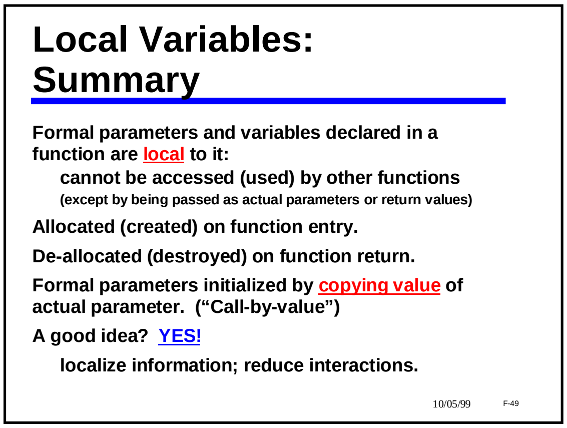
Parameters and local variables of circle_area

circle_area		
r	x	area1

circle_area		
r	x	area1

washer_area				
inner	outer	area1	area2	washer

main		
inner	outer	y



Local Variables: Summary

Formal parameters and variables declared in a function are local to it:

cannot be accessed (used) by other functions

(except by being passed as actual parameters or return values)

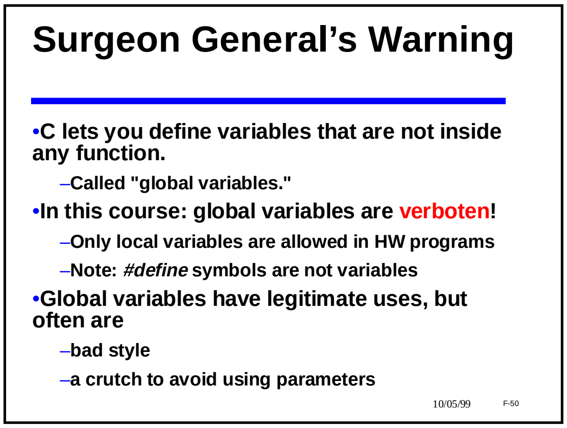
Allocated (created) on function entry.

De-allocated (destroyed) on function return.

Formal parameters initialized by copying value of actual parameter. (“Call-by-value”)

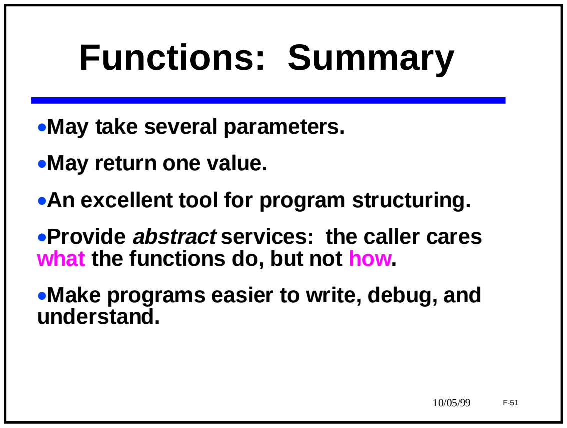
A good idea? YES!

localize information; reduce interactions.



Surgeon General’s Warning

- C lets you define variables that are not inside any function.
  - Called "global variables."
- In this course: global variables are verboten!
  - Only local variables are allowed in HW programs
  - Note: *#define* symbols are not variables
- Global variables have legitimate uses, but often are
  - bad style
  - a crutch to avoid using parameters



Functions: Summary

- May take several parameters.
- May return one value.
- An excellent tool for program structuring.
- Provide *abstract* services: the caller cares what the functions do, but not how.
- Make programs easier to write, debug, and understand.