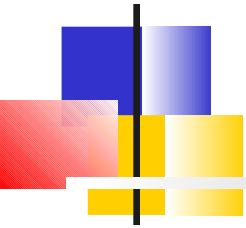
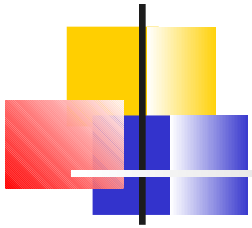


Functions

Chapter 9



Objectives

You will be able to

- Write a function definition.
- Write a program that makes use of functions.
- Design a small program by dividing it into multiple functions.
- Make use of pre-existing functions in libraries.

Functions

- A *function* is a named block of code that can be executed by putting its name into a statement.
- A *very* simple example:

Function Name

Function Definition

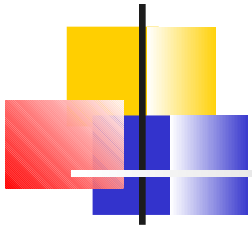
```
void say_hello()
{
    printf ("Hello, world\n");
}

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

Function Header

Function Body

Function "Call"

Whenever the function name is reached in the stream of execution, the function body is executed.

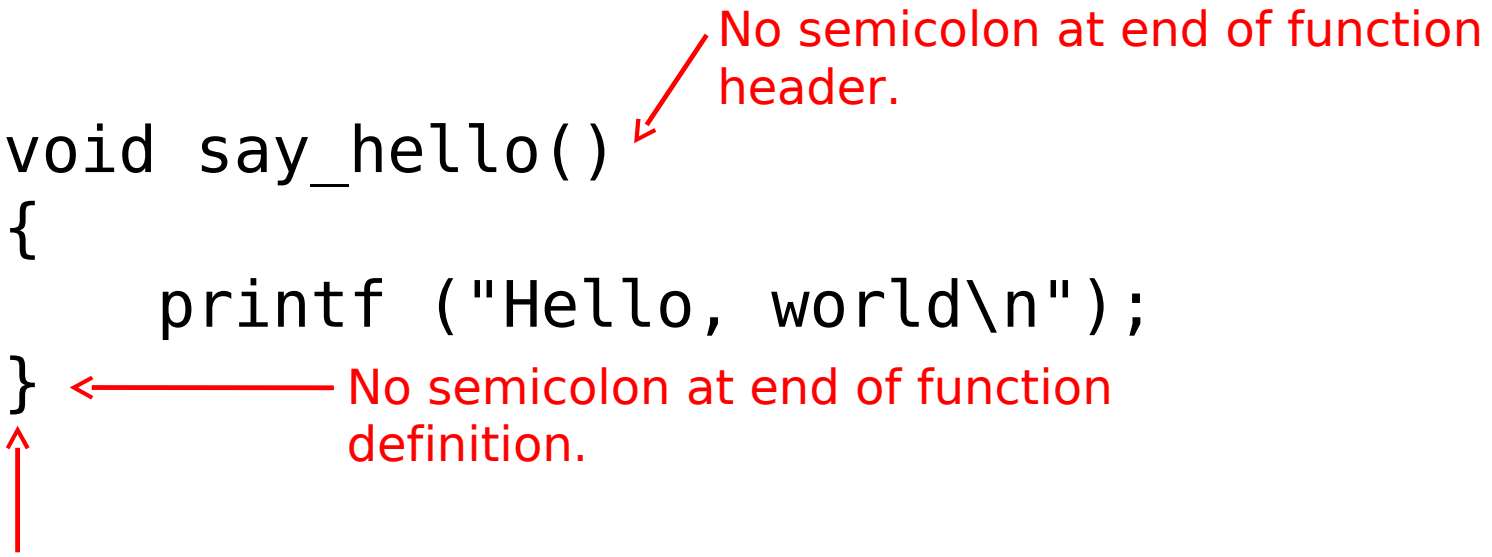


Functions

- Our primary tool for coping with complexity.
- Divide a big program into smaller pieces
  - Each piece performs a well defined action
  - Pieces can be further divided as needed
  - Final result is nothing but small, understandable pieces.
- “Program Design”

Functions

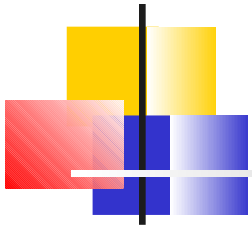
- Some things to notice.



The function body is delimited by curly brackets.

For readability, align the curly brackets with the function header and indent the code within the function body by 3 spaces

(The usual rules for alignment of curly brackets.)



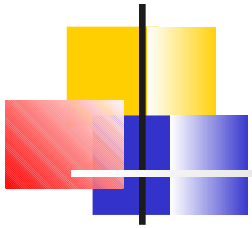
Functions

This is NOT textual substitution like #define

The compiler produces executable code from the function definition.

The compiler generates code for the function call that transfers control to the function body.

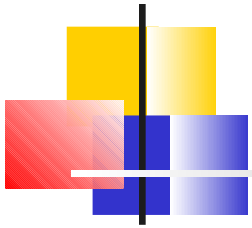
The function body includes executable code to return control back to the next statement after the function call.



A Note about Examples

- Examples used in this lecture will all be *very* simple.
  - Show the mechanisms
  - Avoid obscuring the concepts with details

- In order to be useful, functions in real life are more complex.
  - But not *a lot* more complex.
  - Typically should fit on one page.



Function Parameters

- A function definition can include one or more variables, called ***parameters***, to be supplied by the call.

Parameters

```
double average (double n1, double n2 )
{
    return (n1 + n2) / 2.0;
}
```

  - The function call must provide values for function's parameters.

- A value passed to the function by the call is referred to as an *argument*.
  - Each argument becomes the initial value of a parameter when the function is called.
  - Inside the function body, parameters act like normal

Function Arguments

```
void print_int (int x)
{
   x = 33;     /* Effect of call */
   printf ("The integer is %d\n", x);
}
```

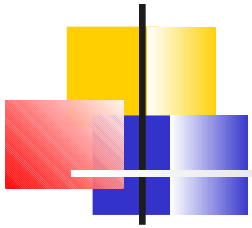
Parameter

Function Definition

```
int main ()
{
    ...
    print_int (33);
}
```

Function Call

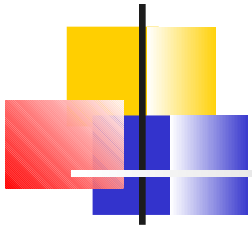
Argument



Returned Value

- A function can perform a computation and return the result.
- The result is the “value” of the function name.
  - Can be used in assignment statement.
  - Can be used in an expression.

Returning a Result



This says that the function returns a value of type int.

```
int square (int x)
{
    return x*x;
}

int main ()
{
    int num;
    int num_squared;
    printf ("Enter number to be squared: ");
    scanf ("%d", & num);
    num_squared = square(num);
    printf ("The square of %d is %d\n", num, num_squared);
    return 0;
}
```

Function Definition

The value of x*x becomes the value of square(num)

The value of num becomes the value of x for the function

num_squared is set to the value returned by function

Returning a Result

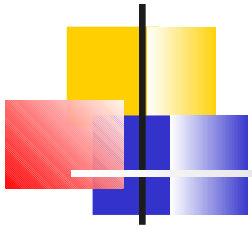
turnerr@login1:~/test

```
[turnerr@login1 test]$ cat  test3.c
#include <stdio.h>

int square (int x)
{
    return x*x;
}


int main ()
{
    int num;
    int num_squared;
    printf ("Enter number to be squared: ");
    scanf ("%d", & num);

    num_squared = square(num);

    printf ("The square of %d is %d\n", num, num_squared);
    return 0;
}

[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall test3.c
[turnerr@login1 test]$ ./a.out
Enter number to be squared: 13
The square of 13 is 169
[turnerr@login1 test]$
```

Using Returned Values in an Expression

```
int main ()
{
    int a = 0;
    int b = 0;
    int c_squared = 0;
    printf ("Enter a: ");
    scanf ("%d", &a);
    printf ("Enter b: ");
    scanf ("%d", &b);

    c_squared = square(a) + square(b);

    return 0;
}
```

A function call can be used in any expression just like a variable.



Returning a Result

- If a function is declared with a type other than void, it *must* return a value of that type.

```
return x*x;
```

- If a function is declared as void, it *must not* return a result.
  - Can return using the statement:

```
return;
```

  - Can simply run to completion.

Function Prototypes

- A function definition begins with a *function header*

```
int average (int a, int b)
```

Type of returned value

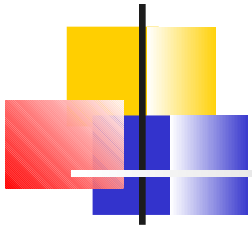
Function name

Parameters and their types.

The compiler needs this information in order to compile the function call.

This information is referred to as the *function prototype*

a.k.a. function *signature*



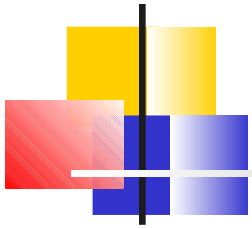
Function Prototypes

- If the function definition appears in the source file before the function call, the compiler automatically knows the function prototype.
- Otherwise, we have to provide it in the form of a function *declaration* somewhere prior to the function call.

```
int average (int a, int b);
```

A function *declaration* must be followed by a semicolon.

In the function *definition*, this information is NOT followed by a semicolon. (It is followed by the function body.)



Function Prototypes

Programming Style Issue:

- Function declarations, if needed, should appear near the beginning of the source file, following any #define lines.

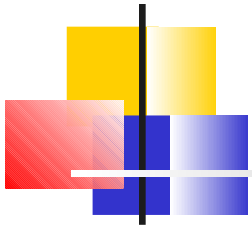
Function Declaration Example

```
#include <stdio.h>

int square (int x);

int main ()
{
   int value = 0;
   int value_squared = 0;
   printf ("Enter value to be squared: ");
   scanf ("%d", &value);
   value_squared = square(value);
   printf ("The square of %d is %d\n", value, value_squared);
   return 0;
}

int square (int x)
{
   return x*x;
}
```

Function declaration

Function call

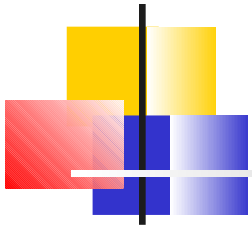
Function definition



Function Prototypes

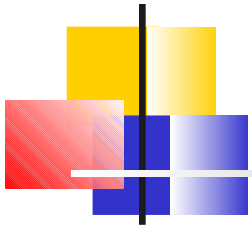
- If we tell the compiler the function prototype in a function declaration prior to any calls, the function *definition* can appear *anywhere*.
  - Even in a different file that is compiled separately.

End of Section



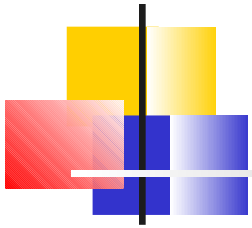
Libraries

- Functions can be collected and compiled separately to form a *library*.
  - Standard Libraries
    - Defined by the C Language standards
    - Included with every C compiler
    - Example: The Standard IO Library
    - Details in Chapter 23.
  - Local Libraries
    - Locally written functions intended for reuse
  - Proprietary Libraries
    - Software products



Standard Libraries

- The standard libraries are essentially like extensions to the C language.
- Standard I/O Library
  - printf(), getchar(), scanf(), others
  - Chapter 22
- The C Mathematics Library
  - sin(), cos(), exp(), log()
  - Complete list starting on page 593



Libraries

- Functions in libraries are compiled separately.
- Separately compiled object code is added to the executable file by the linker.
- Function prototypes for the library functions are collected in a header file for the library.
  - Added to user's source file with #include.

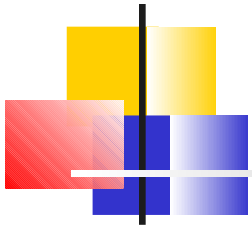
Example

```
#include <stdio.h>

int square (int x);

int main ()
{
    int value = 0;
    int value_squared = 0;
    printf("Enter value to be squared: ");
    ...
}
```

Function prototypes for all functions in the Standard I/O Library are inserted here by the preprocessor.

The compiler knows the function prototype for printf() because or the #include.

The call to function printf() causes the linker to add the precompiled code for function printf() to the executable file.



Example: hypotenuse.c

- A program to compute the length of the hypotenuse of a right triangle, given the lengths of the other two sides, a and b.
  - a and b must be in the range 0 < x < 1000000
- Will use a function to get user inputs for the two sides.
  - Verify that input is valid.
  - Ask user to provide a different value if not

Example: hypotenuse.c

```
/* Compute length of hypotenuse of a right triangle */

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

/* Get user input for length of a side. */
double get_length()
{
    double length = 0;
    while (1)
    {
        scanf ("%lg", &length);
        if ((length > 0) && (length <= 1000000))
        {
            return length;
        }
        printf ("Value must be greater than 0 “);
        printf (“and no more than 1000000\n");
        printf ("Please enter a valid value\n");
    }
}
```

A *local variable*.

Example: hypotenuse.c

```
int main ()
{
    double a = 0;
    double b = 0;
    double c = 0;   /* Hypotenuse */

   printf ("This program computes the length of the hypotenuse\n");
   printf ("of a right triangle, given the lengths of the other\n");
   printf ("two sides, a and b\n");
   printf ("a and b must be between zero and 1000000\n");

    printf ("Enter a: ");
    a = get_length();
    printf ("Enter b: ");
    b = get_length();

    printf (“Sides are %10.4f and %10.4f\n", a, b);

    c = sqrt(a*a + b*b);

    printf ("Length of hypotenuse is %10.4f\n", c);
    return 0;
}
```

Local variables of function main()

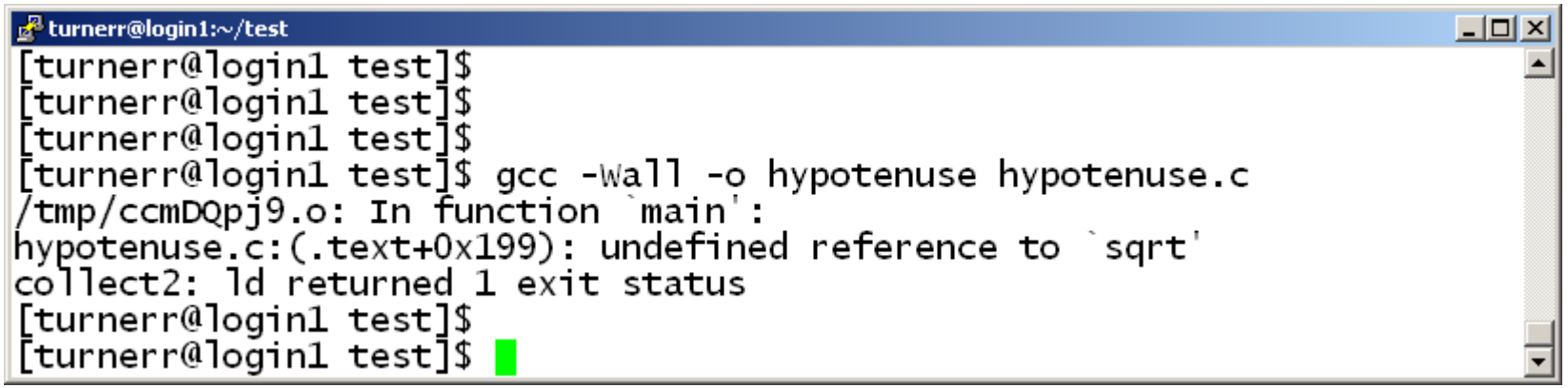
Get value of “a” from user.

Get value of “b” from user.

Function in the math library

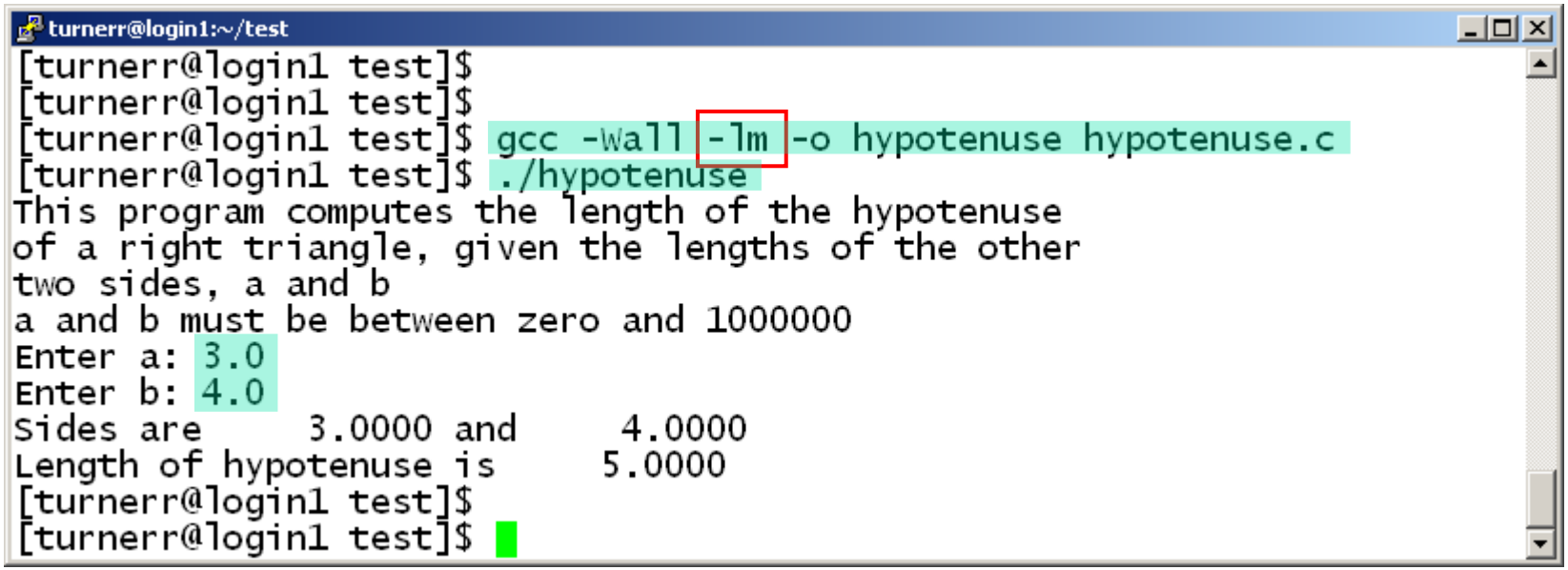
Example: hypotenuse.c

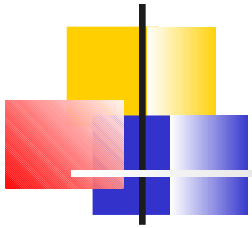
- When we compile, we have to tell gcc to link with the math library.
- Otherwise we get an “Undefined reference” error:



Example: hypotenuse.c

To tell the linker to link with the math library, include **-lm** in the command.





Assignment

- Read Chapter 9
  - Through page 205

End of Presentation