Lesson #6

Modular Programming and Functions

Modular Programming

- Proper programming is always modular.
- Modular programming is accomplished by using functions.
- **Functions:** Separate program modules corresponding to the individual steps in a problem solution.

Functions

- We know already about the math library functions that are predefined for our use. Now we will learn how to program our own functions.
- Functions are “black boxes” that accept data and return results. Most library functions have 1 input and 1 result (like *sqrt*). Others have 2 inputs and one result (like **pow**).

Arguments

- We call the value sent to a function (the input) an argument.
- Functions can have many arguments but only one result.
- With sqrt(x), ***x*** is the argument. The result is ***sqrt(x)***, which can be assigned to a variable like in y=sqrt(x);

Defining Functions

- A function is like a separate program that can be called by the main program to perform a specific task.
- The function definition is placed on top of the program code just after the preprocessor directives and before the main program (or after the main function with a prototype placed before).
- It is important that the function be declared before the main program. If you don't, the compiler gives you an error as your function call is incompatible with the function definition.

Defining Functions

- A function from the math library is already defined. So, when you use it in a program you actually call the function.
- For a function that is defined by you, you will need first to define the function before using (or calling) it.

Defining Functions

Here is the syntax to define a function:

```
functiontype
functionname (type and names of parameters
   separated by commas)
{
   Local variable declarations;
   . . .
   C statements executed when function is called
   . . .
   return (result); (only if there is one, must match
   function type)
}
```

A *void* Function with No Arguments

- A **void** function is a function that generates no results.

```
/* a function definition */
void
stars (void)
{
    printf ("************************\n");
}
...
```

- Inside the main program, the function would be called like this:

```
stars ();
```

- See the complete program at c.ihypress.ca/05.php.

A void *Function with One Argument*

- The *stars* function only displays the same number of stars every time.
- What if I wanted to display a different number of stars that could be linked to a numerical value that I send to the function?
- All I would need to add is an argument to the function to specify the number of stars. See *stars2* on the next slide. *stars2*, like *stars*, has no result. When a function has no result, we declare its type as **void**.

A void Function with One Argument

/* a void function returns nothing */

```
void
stars2 (int n)
{
   int i;

   /* a loop displaying a star at each iteration */
   for (i=1; i<=n; ++i)
   {
    printf ("*");
   }
   /* change line after each series */
   printf ("\n");
}
```

The function would be called like this: `stars2(x);` See c.ihypress.ca/05.php

A Function with One Result

- A function that has a result will have a type (char, double, or int) and will finish with a return statement.
- The result in the return statement must always match the type declared on top of the function.
- See an example of a function calculating the factorial of a number on the next slide.

A Function with One Result

```
/* this function will return an integer */
int
factorial (int n)
{
  int i, product;

  product = 1; /* initialization */

  /* computes n*n-1... */
  for (i=n; i>1; i=i-1)
  {
   product = product * i;
  }

  /* the value that goes out */
  return (product);
}
```

Things to Know

- A function usually does not display anything on the screen (unless the function contains a printf statement).
- Variables declared inside a function are unknown in the main program.
- Variables of the main program are unknown inside the function.
- The only way to send values from the main program to a function is through an argument.

A Function with Two Arguments

- A function with two arguments will require two parameters in the function header. Parameters are presented in a list separated by commas.

Ex:

```
double bigger (double n1, double n2)
{
    double larger;
    if (n1 > n2)
        larger = n1;
   else
        larger = n2;
   return (larger);
}
```

Remember ***NOT***

- The **N**umber of actual arguments used in a function call must be the same as the number of parameters listed in the function definition.
- The **O**rder of arguments used in a function call must match the order of parameters in the function definition.
- The **T**ypes of arguments used in a function call must match the types of the corresponding parameters in the function definition.

Case Study: Is a Number Prime?

- A prime number is a number only divisible by itself or 1.
- A function that determines if a number is prime or not will accept an integer number as an argument and return a truth value (1 if prime or 0 if not prime).
- The solution is to try to find a divisor for the number. If unsuccessful, the number is prime. To find a divisor, we try all possible numbers to see if it is divisible.

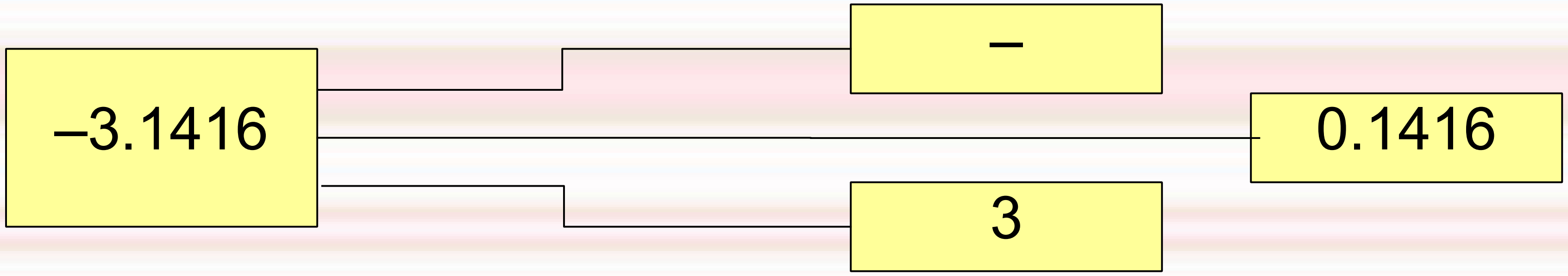
See www.ihypress.net/programming/c/05.php (program #6) for the solution.

Function Results

- A function can have only <u>one</u> result.
- A function can execute only <u>one</u> return statement.
- A return statement <u>ends</u> the function execution and gives the control back to the calling function (usually main).

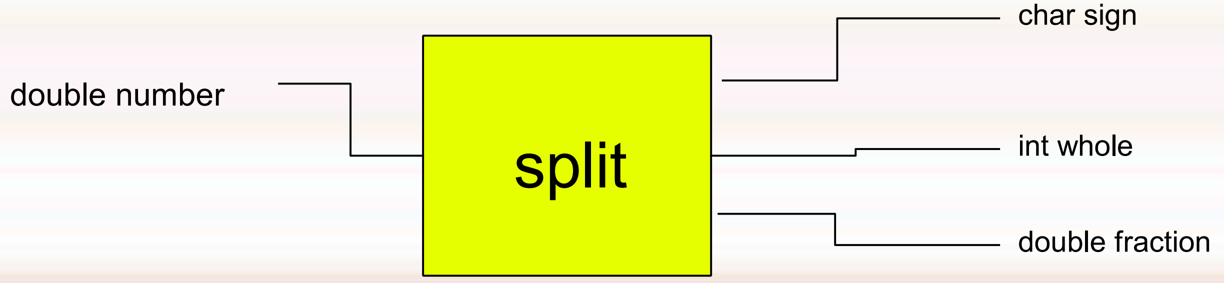
Functions with Multiple Results

- We know that functions can only have one result (one return statement), but we can trick a function into giving multiple results by using pointer parameters.
- Let's have a function that takes in a double number and gives back **3** “results”: a sign, a whole part, and a fraction.



Functions with Multiple Results

- The best thing to do next is to analyze our function by making a diagram of what goes in and of what comes out by labeling everything properly with their types.



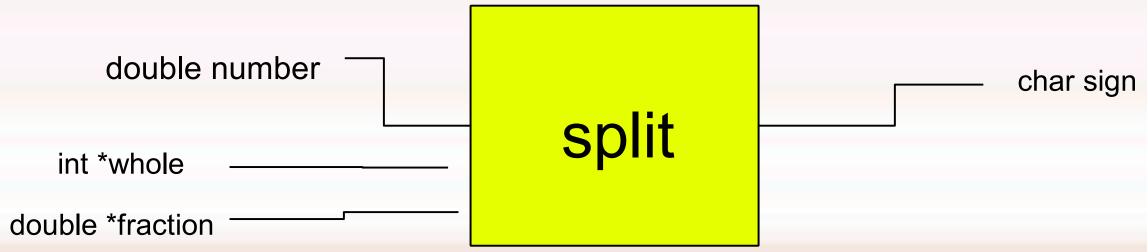
- Notice the 3 arrows on the right side. Since a function can only have 1 true result, we will have to do something about the two extra results.

Functions with Multiple Results

- Of course, our function can only have **one** true result (one return), so we'll pick one (any one will do). Let's pick the sign as the result. Since the sign is a character, our function will be a **char** function.
- The two other “results” (the whole part and the fraction) will be made accessible to the main program by the pointer parameters. The diagram on the next slide shows the outcome.

Functions with Multiple Results

- To get rid of the extra results, we will pick one as the true result and the other two will be fake results. We then transfer the fake results to the left side but as pointers (we add a star to their names).



- Now our function has only one result, hence perfectly legal.

Functions with Multiple Results

```
char
split (double number, int *whole, double
   *fraction)
{
     char sign;
     if (number < 0)
          sign = '-';
     else
          if (number > 0)
               sign = '+';
          else
               sign = ' ';
     *whole = abs ((int)number);
     *fraction = fabs (number) - *whole;
     return (sign);
}
```

Functions with Multiple Results

```
int
main (void)
{
     double n, f;
     int w;
     char s;
     printf ("Enter a double number:");
     scanf ("%lf", &n);
     s = split (n, &w, &f);
     printf ("The sign is: %c\n", s);
     printf ("The whole part is: %d\n", w);
     printf ("The fraction is: %lf\n", f);
     return (0);
}
```

- *See that when a parameter is a pointer, you must send an address value to it.*

Recursion

- Recursion is simply the calling of a function by itself.
- We know that we can call a function from within a function (remember the prime numbers program?). That function could the function itself.
- Recursion is very powerful and very easy to understand but hard to program and debug.

Recursion

- Recursion, in mathematics and computer science, is a method of defining functions in which the function being defined is applied within its own definition.

- Even if properly defined, a recursive procedure is not easy for humans to perform, as it requires distinguishing the new from the old (partially executed) invocation of the procedure

- When the surfaces of two mirrors are exactly parallel with each other, the nested images that occur are a form of infinite recursion.

Recursion

- Recursive behavior can be defined by two properties:

  >> A simple base case (or cases)
  >> A set of rules which reduce all other cases toward the base case.

- The following is a recursive definition of a person's ancestors:

  >> One's parents are one's ancestors (base case).
  >> The parents of one's ancestors are also one's ancestors (recursion step).

Example of Recursion

- The Fibonacci sequence is a classic example of recursion:
- fib(0) is 0 [base case]
- fib(1) is 1 [base case]
- For all integers n > 1: fib(n) is (fib(n-1) + fib(n-2)) [recursive definition]

Recursion in Programming

- Recursion in programming is a method where the solution to a problem depends on solutions to smaller/simpler instances of the same problem. The approach can be applied to many types of problems, and is one of the central ideas of computer science.
- Most high-level computer programming languages (like C) support recursion by allowing a function to call itself.
- There is always a recursive solution to an iterative algorithm (a loop can be substituted by a recursive approach).

Recursive Algorithms

- A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those problems, and combine the results.
- A recursive function definition has one or more base (trivial) cases, and one or more recursive (complex) cases. For example, the factorial function can be defined recursively by the equations $0! = 1$ and, for all $n > 0$, $n! = n(n - 1)!$. Neither equation by itself constitutes a complete definition; the first is the base case, and the second is the recursive case.

A recursive function

```
int
factorial (int n)
{
   int product;
   if (n == 0)
    product = 1;
   else
    product = n * factorial (n-1);

   return (product);
}
```

Compare this function with the loop-based factorial function seen in the previous lesson!

Another recursive function

```
int
multiply (int m, int n)
{
   int answer;
   if (n == 1)
    answer = m;
   else
    answer = m + multiply (m, n-1);

   return (answer);
}
```

End of Lesson