```
#include<iostream.h>

int i = 3;     // global variable

void myfunction();

int main()
 {
  int j,k;     // variables local to the main function
               // j and k are not accessible outside of main
  j = 2;
  k = i + j;
  cout << "j = " << j << " and k = " << k << '\n';
  cout << "i = " << i << " before the call to myfunction.\n";
  myfunction(); // call to myfunction
  cout << "i = " << i << " after the call to myfunction.\n";
  return 0;
 }

void myfunction()
 {
  int l;       // local variable
  l = ++i;     // the variable i is accessible because i is global
               // the variable i is changed globally
  cout << "l = " << l << '\n';
  cout << "The variable l is lost as soon as myfunction exits.\n";
 }
```

FIGURE 9-5 This simple program illustrates the difference between local and global variables.

memory is released when the function terminates. If a variable is needed only within a particular function, you save memory by creating and disposing of the variable within the function.

Using local variables could limit the number of errors that occur in a program. If all variables were global, an error made in a variable and used by various functions could cause multiple errors. However, if you use local variables, any errors are limited to the function in which the variable is declared.

Use local variables whenever possible. Even a large program should have very few global variables. Using local variables keeps a tighter control over your program's data, resulting in fewer bugs and programs which are easier to maintain.

You may be wondering how data can get to other functions if everything is local. As you will learn next, when a function is created, you can choose what data you want to send to the function.

GETTING DATA TO AND FROM FUNCTIONS

You have learned that the parentheses following a function's name lets the compiler know that it is a function. The parentheses can serve another purpose as well. That is, parentheses can be used to pass data to a function and in some cases to return data from a function.

When a function is called, the data in the parentheses (called the ***argument***) is ***passed*** into the receiving function. There are three ways to pass data to functions: passing by value, passing by reference, and passing by address.

PASSING BY VALUE

When you pass a variable to a function by value, a copy of the value in the variable is given to the function for it to use. If the variable is changed within the function, the original copy of the variable in the calling function remains the same. Shown below is an example of a function that passes data to a function using the *passing by value* technique.

```
void print_true_or_false(int True_False)
 {
  if True_False          // If True_False is not equal to zero,
   {                     // display the word TRUE.
    cout << "TRUE\n";
   }
  else                   // If the True_False is equal to zero,
   {                     // display the word FALSE.
    cout << "FALSE\n";
   }
 }
```

A value comes into the function through the parentheses and the copy of the value will be placed in the variable `True_False`. The variable `True_False` is called a *parameter.*

Arguments and Parameters

Many people use the terms argument *and* parameter *interchangeably, but there is a difference. An argument is a value or expression passed to a function through the parentheses when a function is called. A parameter is the variable that receives the value or any other identifier in the parentheses of the function declaration. In other words, an argument is passed to a function, but once in the function, the argument is a parameter.*

When you write a call to a function, you can put any variable or literal in the parentheses to be passed to the function as long as the data types do not conflict. For example, the statements below are all legal calls to the `print_true_or_false` function.

```
print_true_or_false(complete);          // passes a variable
print_true_or_false(1);                 // passes a literal
print_true_or_false(j == 3 && k == 2);  // passes the result of an
                                        // expression
```

The program in Figure 9-6 illustrates how a value passed to the function named `print_value` does not pass back to the main function. Notice that the `print_value` function uses a variable named `j`, even though the main function passes a variable named `i`. The data types must match, but the names are often different.