Global Versus Local Environments

In this laboratory session you will:

1. Investigate the scope of variables.
2. Learn the distinction between global and local variables in C++ programs.
3. Learn how to define `const` identifiers.
4. Experiment with conditional compilation techniques.

Your instructor will tell you which of the proposed experiments you are to perform.

In preparation for this laboratory session you should read Sections 5.3, 5.4 and 6.3 of *Computer Science: An Overview*. 
Global Versus Local Variables

When a variable is declared within a function, that declaration is recognized only within the function. The variable remains undefined outside the function. The range over which a variable's declaration is recognized is called the scope of the variable. Thus, the scope of a variable declared within a function is restricted to the body of that function. Such variables are said to be local variables. Each local variable is created each time its function is called and destroyed when the function in which it resides is completed.

An attempt to reference a variable outside of its scope leads to an error. For example, the statement

\[ x = 5; \]

in the following program is invalid since the variable \( x \) is not declared in the function \( \text{fun1} \).

```c
void fun1();
int main()
{
    int x;
    fun();
    return 0;
}
void fun1()
{
    x = 5;
}
```

Different functions can contain local variables of the same name without conflict. For example, the variables named \( x \) in the following program are distinct—one resides within the function \( \text{fun2} \), the other resides within \( \text{main} \). The assignment to \( x \) made within the function \( \text{fun2} \) has no effect on the value of \( x \) in the function \( \text{main} \). Thus, the \( \text{cout} \) statement will print the value 6—not the value 5 that was assigned to the variable \( x \) in the function \( \text{fun2} \). In fact, by the time the \( \text{cout} \) statement is executed, the variable that was assigned the value 5 will have been destroyed since the function in which it resided will have been completed.

```c
#include <iostream.h>
void fun2();
int main()
{
    int x = 6; // Local variable
    fun2();
    cout << x << endl;
    return 0;
}
void fun2()
{
    int x; // Local variable
    x = 5;
}
```

The language C++ allows variables to be declared in such a way that their scopes encompass the entire program. Such variables are called global variables and are created by placing their
declarations at the beginning of the program outside all function declarations. For example, the following program contains a single variable named \( x \) whose scope is the entire program. In turn, the \texttt{cout} statement in the function \texttt{main} prints the value 5 that was assigned to \( x \) in the function \texttt{fun3}.

```cpp
#include <iostream.h>
int x;          // Global variable  
void fun3();
int main()
{  
x = 6;
fun3();
cout << x << endl;
return 0;
}
void fun3()
{
    x = 5;
}
```

**Experiment 7.1**

**Step 1.** Execute the following program (CP07E01A) and record the results.

```cpp
#include <iostream.h>
void potato();
void advertisement();
int main()
{  
    char x;
x = 'Z';
cout << "x = " << x << endl;  
potato();
cout << "x = " << x << endl;
advertisement();
cout << "x = " << x << endl;
return 0;
}
void potato()
{
    char x;
x = 'H';
cout << x << "ey ho, I'm from Idaho!\n";
}
void advertisement()
{
    char x;
x = 'Y';
cout << x << "ou gotta taste it all!\n";
}  ```
Step 2. Explain the results obtained in Step 1 by tracing the program's execution while describing the creation, destruction, and assignment of variables. At each point in the execution at which a cout statement is executed, identify the local variables that have been created but not destroyed. What is the scope of each? What is the value of each?

Step 3. Execute the following modified version of the program from Step 1 (CP07E01B). Identify the variables that have been created but not destroyed when the cout statement in the function advertisement is executed. What is the scope of each? What is the value of each?

```cpp
#include <iostream.h>
void advertisement();
void potato();
int main()
{
    char x;
    x = 'Z';
    cout << "X = " << x << endl;
    potato();
    cout << "X = " << x << endl;
    return 0;
}
```
Step 4. Execute the following version of the program from Step 1 (CP07E01C) and explain the results.

```cpp
#include <iostream.h>
char x;
void potato();
void advertisement();
int main()
{
    x = 'Z';
    cout << "x = " << x << endl;
    potato();
    cout << "x = " << x << endl;
    advertisement();
    cout << "x = " << x << endl;
    return 0;
}

void potato()
{
    x = 'H';
    cout << x << " ey ho, I'm from Idaho!\n";
}

void advertisement()
{
    char x;
    x = 'Y';
    cout << x << " ou gotta taste it all!\n";
}

void potato()
{
    char x;
    x = 'H';
    cout << x << " ey ho, I'm from Idaho!\n";
    advertisement();
}
```
```cpp
void advertisement()
{
    x = 'Y';
    cout << x << " ou gotta taste it all!\n";
}

#include <iostream.h>
char x;
void potato();
void advertisement();
int main()
{
    x = 'Z';
    cout << "x = " << x << endl;
    potato();
    cout << "x = " << x << endl;
    advertisement();
    cout << "x = " << x << endl;
    return 0;
}

void potato()
{
    x = 'H';
    cout << x << " ey ho, I'm from Idaho!\n";
}

void advertisement()
{
    char x;
    x = 'Y';
    cout << x << " ou gotta taste it all!\n";
}

Step 5. Execute the following version of the program from Step 1 (CP07E01D) and explain the results. In particular, note that variables named x are declared at various locations. What can you conclude about a local variable residing within the scope of a global variable of the same name?
```
Experiment 7.2

Step 1. The following program (CP07E02) represents an unsuccessful attempt to design a function `swap` that interchanges the values assigned to the variables $x$ and $y$. Try to execute the program and record the errors.

```cpp
// This program contains errors.
#include <iostream.h>
void swap();
int main()
{
    int x = 5, y = 6;
    cout << "x = " << x << " and y = " << y << endl;
    swap();
    cout << "x = " << x << " and y = " << y << endl;
    return 0;
}
void swap()
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```
Step 2. Rewrite the program in Step 1 so that the function swap can have access to the data needed to perform its task. Summarize your modifications below.

__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________

Static Variables
In general, a local variable is created each time its function is called and destroyed when the execution of the function is completed. This means that any value assigned to the variable will be lost. To preserve a local variable until the function is called again, the variable can be declared as static. This is done by preceding its declaration with the term static.

Experiment 7.3
Step 1. Execute the following program (CP07E03) and record the results. What does this program demonstrate about the preservation of local environments between calls to the same function?

```c++
#include <iostream.h>
void fun();
int main()
{
fun();
fun();
return 0;
}
void fun()
{
int y = 5;
cout << "Starting function fun\n";
cout << "The value of y is " << y << endl;
y++;
cout << "The value of y is " << y << endl;
}
```
Step 2. Change the declaration

```c
int y = 5;
```

in the function `fun` in Step 1 to

```c
static int y = 5;
```

and execute the program again. Record the results and explain them.
Experiment 7.4

Step 1. Execute the following program (CP07E04) and record the results.

```cpp
#include <iostream.h>
int x;
void fun();
int main()
{
 x = 10;
 cout << x << endl;
 fun();
 cout << x << endl;
 fun();
 cout << x << endl;
 return 0;
}
void fun()
{
 static int x = 0;
 x++; 
 cout << x << endl;
}
```

Step 2. Does retaining a static variable after its function has terminated make the variable accessible to other parts of the program? Explain your answer.
The const Type

As explained in *Computer Science: An Overview*, it is advantageous to use descriptive terms for variable names within a program because this practice tends to add to the readability of the program. Likewise, descriptive names are helpful when referring to fixed values. For instance, the word *pi* conveys the intention more readily than the numeric value 3.1416, and the term *AirportAlt* is more descriptive than the actual value of 675 feet when referring to an airport’s altitude.

On occasion, you may want to define an identifier whose value can be used by a program but not be modified. In other words, the value of the identifier should remain constant. C++ provides a way to define such identifiers by using a `const` statement of the form

```cpp
const data_type identifier = value;
```

where `data_type` is any C++ data type, `identifier` is the name of the identifier that is being defined and `value` is the value to be associated with the identifier. So

```cpp
const int MYVALUE = 50;
```

declares `MYVALUE` to be an integer and assigns it the value 50. The identifier `MYVALUE` can then be used to reference the value 50 in the program. By convention, most C++ programmers use all uppercase for the names of constant identifiers as a visual cue that this identifier represents a fixed value. Such identifiers obey the same scope rules as variables.

**Experiment 7.5**

**Step 1.** Execute the following program (CP07E5) and then record and explain the results.

```cpp
#include <iostream.h>
void test_const();
int main()
{
    const float PI = 3.1415;
    cout << "The value of PI is " << PI << endl;
    test_const();
    cout << "The value of PI is " << PI << endl;
    return 0;
}
void test_const()
{
    const float PI = 3.1415;
    cout << "The value of PI is " << PI << endl;
}
```
Step 2. Modify the program in Step 1 by adding the statement

\[ \text{PI} = 3.141592; \]

to immediately follow the statement

\[ \text{test\_const();} \]

Explain the results.

__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________

Conditional Compilation

In C++, the define directive has the form

\[ \#define \text{identifier} \text{symbol\_string} \]

where \text{identifier} is a descriptive term that will be used to represent \text{symbol\_string} in the program. All occurrences of \text{identifier} following the directive are replaced with \text{symbol\_string} before compilation begins. As an example,

\[ \#define \text{TEST} 1 \]

causes all references to \text{TEST} to be replaced by 1.

The define directive can be combined with other preprocessor directives to achieve conditional compilation—that is, a compiling process that recognizes or ignores parts of the source program depending on conditions. These other directives include if, else, elif, and endif, each of which is preceded by the \# symbol. The word if in the if directive is followed by an expression that is considered true if it evaluates to a nonzero value, and false otherwise. If the expression is true, the statements following the if directive will be compiled. Otherwise, the compiler will skip to the following else or elif directive or, if such a directive is not present, to the following endif directive.

As an example, suppose you were developing a program in which you want certain cout statements included for testing purposes, but do not want these statements in the final version of the program. You could remove the cout statements once the testing of the program was completed. A better approach, however, might be to begin the program with the directive
# define TEST 1

and then surrounding each cout statement with if and endif directives of the form

    #if TEST
    cout << ...;
    #endif

Then, each time you compile the program the cout statements will be included. Once the program has been tested, all that is needed is to change the first directive to read

    #define TEST 0

and then recompile the program one last time. This time the compiler will skip the cout statements, producing an object program that does not contain the cout statements. If, however, more testing should become necessary, changing the define directive back to

    #define TEST 1

is all that is required to obtain a source program that will produce an object program containing the cout statements.

As another example, the following program segment will produce a function that computes the area of a rectangle if RECT is defined to be 1, and a triangle if RECT is 0.

    float compute_area(float width, float height)
    {
        #if RECT
        return(width * height);
        #else
        return(0.5 * width * height);
        #endif
    }

**Experiment 7.6**

**Step 1.** Compile and execute the following program (CP07E06). Record the results.

    #include <iostream.h>
    #define SHOW_WORK 1
    int main()
    {
        int i, n, factorial = 1;
        cout << "Enter a positive integer.\n";
        cin >> n;
        for (i = 1; i <= n; i++)
        {
            factorial = factorial * i;
            #if SHOW_WORK
            cout << "Result so far is " << factorial << endl;
            #endif
        }
        cout << "The factorial of " << n << " is " << factorial;
        cout << "\n";
        return 0;
    }
Step 2. Change the define directive in the program above to

#define SHOW_WORK 0

Recompile and execute the modified program, record the results, and explain the difference between these results and those of Step 1.

Experiment 7.7

Compile and execute the following program (CP07E07) using different values in the define directives. Summarize and explain your findings.

```cpp
#include <iostream.h>
#define GERMAN 0
#define PERSONAL 0
int main()
{
    #if GERMAN && PERSONAL
    cout << "Guten Abend. Ich liebe Dich.\n";
    #elif GERMAN
    cout << "Guten Abend.\n";
    #else
    cout << "Good evening.\n";
    #endif
    return 0;
}
```
Post-Laboratory Problems

7.1. Write a program to assist a person estimate the future cost of owning a car. It should include payments, interest, maintenance, operating expenses, and insurance. Use a modular design. What items did you implement as global items and why? What items are passed as parameters and why? In which cases did you design functions so that they return values and why?

7.2. Write a program that computes properties of various geometric shapes. For example, your program might offer the following choices:

1 - compute the area of a triangle
2 - compute the area of a rectangle
3 - compute the volume of a rectangle
4 - compute the surface area of a rectangle
5 - compute the area of a circle
6 - compute the volume of a circular cone

After the user makes his or her choice, your program should ask for the pertinent information. For example, if option number 2 is chosen, the program should ask for the length and the width of the rectangle. Design your program with a modular structure and explain the techniques you use for transferring data between the various functions.

7.3. Experiment with the following program until you understand its features. Summarize your findings.

```cpp
#include <iostream.h>
int main()
{
    int x = 0;
    cout << x << endl;
    { int x;
      x = 5;
      cout << x << endl;
    }
    cout << x << endl;
    return 0;
}
```

7.4. Write a program that simulates the machine described in Appendix C of Computer Science: An Overview—that is, write your own version of the simulation program you used earlier in Session 4.

7.5. Write a program that computes the nutritional value of meals. Your program should query the user to obtain the contents of the meal and serving sizes. Use a modular design. Explain the decisions you make regarding the transfer of data between modules. What items did you implement as global items and why? What items are passed as parameters and why? In which cases did you design functions so that they return values and why?

7.6. Modify a program that you have written earlier so that it communicates with the user in one of two languages depending on the settings of preprocessing directives.