Chapter - 9
Variable Scope and Functions

Variable Scope and Class

Variables are defined by two attributes:

Scope — The area where a variable is valid: *Local* or *Global*

Storage Class
Describes the storage allocation of the variable: *Permanent* or *Temporary*

Variable Scope

Global variables are valid everywhere.

Local variables are only valid inside the {} where they are defined.

Scope of **global**

Scope of local

Scope of very_local

```
int global;                  // a global variable
int main() {
    int local;               // a local variable

    global = 1;              // global can be used  here
    local = 2;               // so can local
    {                        // beginning a new block
        int very_local;      // this is local to the block
        very_local = global + local;
    }
    // Block closed
    // very_local can not be used
    return (0);
}
```

Hidden Variables

scope of global count

```
int total; // Total number of entries
int count; // Count of total entries
int main() {
    total = 0;
    count = 0;
    {
        int count; // Local counter
        count = 0;

        while (count < 10) {
            total += count;
            ++count;
        }
    }
    ++count;
    return (0);
}
```

scope of local count

global count is hidden

Storage Class

A variable is either permanent or temporary

Permanent Variables

- Declared either as global variables or with the keyword **static**.
- Initialized only once (when the program begins).
- Destroyed only once (when the program ends).

Temporary Variables

- Must be local variables
- Initialized each time they come into scope
- Destroyed when they go out of scope

Temporary variables are allocated from a memory space called the “stack”. On PC class machines this is very limited (64K max, sometimes only 4K). “Stack Overflow” errors are the result of allocating too many temporary variables.

Permanent vs. Temporary

```
#include <iostream>

int main() {
    int counter;    // loop counter

    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;
        static int permanent = 1;

        std::cout << "Temporary " << temporary <<
                " Permanent " << permanent << '\n';
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3

Types of Declarations

Declared	Scope	Storage Class	Initialized
Outside all blocks	Global	Permanent	Once
static outside all blocks	Global	Permanent	Once
Inside a block	Local	Temporary	Each time block is entered
static inside a block	Local	Permanent	Once

Namespaces

```
namespace io_stuff {
    int output_count; // # chars sent
    int input_count;  // # chars in
};

int main()
{
    ++io_stuff::output_count;
...
```


Special namespaces

`std`	Used for system variables and classes (`std::cout`)
--	Global namespace. (Usage: `::global`)
<blank>	File specific namespace (for variables not used outside the file.)

using Statement

using – don't use

Imports variables from other namespaces into the current code:

```
namespace foo {
    int foo1; // Something
    int foo2;// Something else
};
using foo::foo1;
foo1 = 2;

using namespace foo; // Imports all
foo1 = foo2 = 3;
```


Functions

Comments at the beginning of a function:

- Name -- Name of the function
- Description -- What the function does
- Parameters -- Define each of the parameters to the function
- Return value -- Describe what the function returns

```
/*******************************************
 * triangle -- compute area of a triangle  *
 *                                         *
 * Parameters                              *
 *  width -- width of the triangle         *
 *  height -- height of the triangle       *
 *                                         *
 * Returns                                 *
 *  area of the triangle                   *
 *******************************************/
```

The function itself begins with:

```
float triangle(float width, float height)
```

Triangle Function

```
float triangle(float width, float height){
    float area; // area of the triangle

    area = width * height / 2.0;
    return (area);
}
```

It's use:

```
#include <iostream>
int main(){
    float size;    // Size of our triangle

    size = triangle(1.3, 3.3);

    std::cout << "Area: " << size << "\n";
    return (0);
}
```


Parameter Passing and Return

```
size = triangle(1.3, 8.3)
```

Turns into

```
Triangle's variable width = 1.3
Triangle's height = 8.3
```

```
Begin execution of the first line of the
function triangle.
```

The return statement:

```
return (area);
// ......
size = triangle(1.3, 8.3)
```


Function Prototypes (declaration)

Just like variables functions must be declared before they can be used.

Typical prototype (declaration)

```
float triangle(float width, float height);
```

This function returns a floating point number (the first float) and takes two floating point parameters.

This can also be written as:

```
float triangle(float, float);
```

This version is frowned upon because it doesn't tell the reader what the two parameters are. Also it's easier to make the first form by cutting out the first line of the function and pasting it where you need the prototype using your editor. Don't forget to add the semicolon at the end.

Functions with no parameter

Declaration:

```
int get_value();
```

or

```
int get_value(void);
```

The second form is a holdover from C which uses `(void)` to indicate a function with no parameters and `()` to indicate a function with no parameter checking (i.e. any number of parameters of any type.)

Usage:

```
value = get_value;
```

Functions that return nothing (subroutines)

```
void print_answer(int answer);
```

Usage:

```
print_answer(45);
```


`const` Parameters and Returns Values

Constant parameters can not be changed.

```
const float triangle(const float width,
                     const float height)
{
    float area; // area of the triangle

    area = width * height / 2.0;
    return (area);
}
```

It's illegal to change the value of a constant parameter:

```
   width = 0.5;    // Illegal
```

Constant return values can not be changed. They can be assigned to another variable and changed. As it stands now, constant return values are not useful. A little later we'll see where **const** makes a difference.

Call by value

Ordinary parameters are passed by "call by value." Values go in, but they don't come out. In the following program we try to change the value of count in `inc_counter`, but it doesn't work.

```
#include <iostream>
// This function won't work
void inc_counter(int counter)
{
    ++counter;
}


main(){
   int a_count = 0;     // Random counter

   inc_counter(a_count);
   std::cout << a_count << '\n';
   return (0);
}
```

Prints: 0

Changes made to simple parameters are not passed back to the caller.

References Revisited

Two things occur when we declare a reference parameter such as:

```
int simple;              // A simple variable
int &ref = simple;       // A reference parameter
```

Part one is a reference declaration:

```
int &ref= simple;       // A reference parameter
```

This creates a reference declaration.

The second part *binds* the reference to the variable (in this case simple).

```
int &ref = simple;       // A reference parameter
```

For simple reference declarations, declaration and binding always occur in one statement.

Reference Parameters

```
#include <iostream>
// Works
void inc_counter(int &counter)
{
    ++counter;
}

int main(){
   int a_count = 0;     // Random counter

   inc_counter(a_count);
   std::cout << a_count << '\n';
   return (0);
}
```

In this case the declaration and the binding occur at two different places. Changes to `counter` are reflected in `a_count` because `counter` is a reference. In other words `counter` is the same as `a_count`.

Reference Return Values

```
int &biggest(int array[], int n_elements) {
    int index;  // Current index
    int biggest; // Index of the biggest element

    // Assume the first is the biggest
    biggest = 0;
    for (index = 1; index < n_elements; ++index) {
        if (array[biggest] < array[index])
            biggest = index;
    }

    return (array[biggest]);
}
```

Usage:

```
int item_array[5] = {1, 2, 5000, 3, 4}; // An array

std::cout << "The biggest element is " <<
         biggest(item_array, 5) << '\n';
```


Reference Return Values (cont.)

Remember a reference is treated exactly the same as the real thing.

```
biggest(item_array, 5)
```

is the same as:

```
item_array[2]
```

So:

```
// Zero the largest element
biggest(item_array, 5) = 0;
```

is the same as:

```
// Zero the largest element
item_array[2] = 0;
```


Constant Reference Return Values

Legal:

```
int &biggest(int array[], int n_elements);
// .......
biggest(item_array, 5) = 0;// Zero the biggest elem.
```

const return type prevents changing the returned reference.

Illegal:

```
const int &biggest(int array[], int n_elements);
// .......
biggest(item_array, 5) = 0;// Generates an error
```

Dangling References

```
#include <iostream>

const int &min(const int &i1, const int &i2)
{
    if (i1 < i2)
     return (i1);
    return (i2);
}

main()
{
   int &i = min(1+2, 3+4);

   return (0);
}
```

4) Bind i1 to return (main's i)

1) tmp1 = 1+2

2) tmp2 = 3+4

3) bind tmp1 to i1,
tmp2 to i2,
call min

5) Destroy tmp1, tmp2

6) i is bound to tmp1, which is destroyed

Array Parameters

Array parameters are automatically passed by *call by address*. Changes made to an element of an array *will* be passed back to the caller.

```
int sum(int array[]);
```

For single dimension arrays you don't need to specify the array size. For multi-dimensional arrays, all dimensions except the last must be specified:

```
int sum_matrix(int matrix1[10][10]);   // Legal
int sum_matrix(int matrix1[10][]);     // Legal
int sum_matrix(int matrix1[][]);       // Illegal
```

Note: Array parameters are automatically turned to *pass by reference*.

Question: Why are All Strings of Length 0 no Matter How Long They Really Are?

```
/*************************************************
 * length -- compute the length of a string      *
 *                                               *
 * Parameters                                    *
 *      string -- the string whose length we want*
 *                                               *
 * Returns                                       *
 *      the length of the string                 *
 *************************************************/
int  length(char string[]){
    int index;      // index into the string


    /*
     * Loop until we reach the end of string char.
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* do nothing */
    return (index);
}
```

Function Overloading

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

float square(float value) {
    return (value * value);
}
```

This is allowed in C++. The language is smart enough to tell the difference between the two versions. (Other languages such as FORTRAN or PASCAL or not.)

Function must have different parameter.

```
int get_number(void);
float get_number(void);  // Illegal.
```


Style and Function Overloading

Functions that are overloaded should perform roughly the same job. For example, all functions named `square` should square a number.

The following is syntactically correct, but a style disaster:

```
// Square an integer
int square(int value);

// Draw a square on the screen
void square(int top, int bottom,
   int left, int right);
```


Default Parameters

```
void draw(const rectangle &rectangle;
   double scale = 1.0)
```

Tells C++, “If scale is not specified, make it 1.0.”

The following are equivalent:

```
draw(big_rectangle, 1.0);    // Explicitly specify scale
draw(big_rectangle);       // Let it default to 1.0
```

Unused Parameters

The following generates a warning: “button not used”

```
void exit_button(Widget &button) {
    std::cout << "Shutting down\n";
    exit (0);
}
```

We can tell C++ that we have one parameter, a widget that we don’t use, by not including a parameter name.

```
void exit_button(Widget &) {
```

Good style, however, dictates that we put the parameter name, even as a comment:

```
void exit_button(Widget &/* button */) {
```


inline functions

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

int main() {
    // .....
    x = square(x);
```


Generated Code

Generates the following assembly code on a 68000 machine (paraphrased)

```
label "int square(int value)"
        link a6,#0              // Set up local vars.


        // The next two lines do the work
        movel a6@(8),d1         // d1 = value
        mulsl a6@(8),d1         // d1 = value * d1


        movel d1,d0             // Put ret. value in d0
        unlk a6                 // Restore stack
        rts                     // return(d0)


label "main"


//      x = square(x)
        movel a6@(-4),sp@-      // Put x on the stack
        jbsr "void square(int value)" // Call square


        addqw #4,sp            // Restore the stack
        movel d0,a6@(-4)       // Store return value in X
// ...
```

Notice that we use 8 instructions to call 2 instruction.

inline square

The **inline** keyword causes the function to be expanded inline eliminating any calling overhead:

```
inline int square(int value) {
    return (value * value);
}
```

Generated code:

```
label "main"
// ...
//      x = square(x)
//
        movel d1,a6@(-4)        // d1 = x
        movel a6@(-4),d0        // d0 = x
        mulsl d0,d0             // d0 = (x * x)

        movel d0,a6@(-4)        // Store result
```


inline notes

The keyword **inline** is a suggestion. If the function can not be generated inline, then C++ will generate an ordinary function call automatically. (At least that's what it supposed to do. Some older compilers have problems.)

Use **inline** for very short functions.

Parameter Type Summary

Type	Declaration
Call by value	`function(int var)`
	Value is passed into the function, and can be changed inside the function, but the changes are not passed to the caller.
Constant call by value	`function(const int var)`
	Value is passed into the function and cannot be changed.
Reference	`function(int &var)`
	Reference is passed to the function. Any changes made to the parameter are reflected in the caller.
Constant Reference	`function(const int &var)`
	Value cannot be changed in the function. This form of a parameter is more efficient then “constant call by value” for complex data types.
array	`function(int array[])`
	Value is passed in and may be modified. C++ automatically turns arrays into reference parameters.
Call by address	`function(int *var)`
	Passes a pointer to an item. Pointers will be covered later.

Structured Programming

How to write a term paper using structured programming techniques:

- Start with an outline
- Replace each step in the outline with more detailed sentences.
- Replace each sentence with more detailed sentences.
- Repeat until you've got enough details

Structured programming techniques.

- Write a simple version of your program leaving most of the work up to func-tions that you haven't written yet.
- Fill in some of the details by writing the functions you left out of the first cut.
- Keep writing functions until there are no more to write.

Bottom Up Programing

- Write an overall design of your program.
- Write small functions that perform the basic functions.
- Make sure they are debugged
- Write small functions that build on the basic functions.
- Continue building until the program is written.

Recursion

Recursion occurs when a function calls itself either directly or indirectly.

A recursive function must follow two basic rules:

1. It must have an ending point.
2. It must simplify the problem.

Factorial function:

fact(0) = 1
fact(n) = n * fact(n-1)

In C++ this is:

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* else */
    return (number * fact(number-1));
}
```


Summing an array recursively

```
int sum(int first, int last, int array[]) {
    if (first == last)
        return (array[first]);
    /* else */
        return (array[first] + sum(first+1, last, array));
}
```

Example:

```
Sum(1 8 3 2) =
   1 + Sum(8 3 2) =
      8 + Sum(3 2) =
         3 + Sum (2) =
            2
         3 + 2 = 5
      8 + 5 = 13
   1 + 13 = 14
Answer = 14
```

