Oddities with local variables

In C++ and many other programming languages, local variables are used to temporarily store data. An identifier is declared to be of a specific type and that identifier can be assigned a value that can later be retrieved.

We have already described how local variables have a scope or lifetime that is restricted to the block of statements in which that local variable was declared. We have also described how the parameters of a function can be considered to be local variables, the scope of which is the entire function body, but where the initial value is that value passed by the argument when the function is called.

It is possible to have the same identifier declared to be a local variable twice if the second declaration is inside of a block within the block in which the outer local variable is declared. The identifier declared inside the inner block is said to shadow the declaration of the outer block, as it is no longer possible to access the outer local variable. There are, however, some potentially confusing issues here, so we will look at:

1. redeclarations of the same identifier in different (possibly nested) blocks,
2. redeclarations of the same identifier within the same block,
3. other oddities, including:
    a. multiple declarations, and
    b. using an identifier in its initialization.

1. Redeclarations in different blocks (what is allowed)

The following program has a parameter and two local variables all with the same identifier. One may believe that this should cause a compile-time error, but this is actually quite reasonable:

```
// Pre-processor include statements
#include <iostream>

// Function declarations
int main();
void f( int n );

// Function definitions
int main() {
    f( 3 );
    return 0;
}

void f( int n ) {
    std::cout << n << std::endl;

    if ( n > 0 ) {
        std::cout << n << std::endl;

        double n{ 3.14 };

        std::cout << n << std::endl;

        if ( n > 0 ) {
            std::cout << n << std::endl;

            bool n{true};

            std::cout << n << std::endl;
        }

        std::cout << n << std::endl;
    }

    std::cout << n << std::endl;
}
```

When this function begins, any time `n` is referred to, it refers to the parameter. It is not until the body of the conditional statement that `n` is redeclared to be of type `double`, but only after that declaration up until the end of that block. Once that block exits, the parameter `n` is no longer shadowed, so the last `std::cout` statement will once again print `3`. Inside the nested conditional statement, again, in the condition and in the first `std::cout` statement, the reference to `n` is the reference to the last declaration of that variable, so in both cases, it is still `3.14`. Next, however, `n` is declared to be a local variable of type `bool` with an initial value of `true` (`1`). Once the Boolean `n` goes out of scope, the next `std::cout` statement prints the `n` that was declared a `double`, as that declared local variable is still in scope until the end of the block in which it was defined.

2. Redeclarations in the same block (what is not allowed)

You cannot declare an identifier to be a local variable twice in the same block of statements. This means that a local variable cannot have the same identifier as that of a parameter within the body of the function, and you cannot declare an identifier to be a local variable twice within the same block of statements.

You can run this code at repl.it.

```
#include <iostream>

int main();
void f( int n );

int main() {
        f( 3 );
        return 0;
}

void f( int n ) {
        // This is the body of the function 'f'

        // The scope of the parameter 'n' is the body
        // of the function unless it is shadowed
        int n{5};
        double PI{3.14};
        double PI{3.1415926535897932};

        if ( n >= 0 ) {
                // This is the body of the conditional statement

                // The first declaration is acceptable
                //  - it goes out of scope at the end of the
                //    body of the conditional statement
                double PI{355.0/113.0};
                int PI{3};

                // This is a block of statements
                //  - local variables within this block go out
                //    of scope at the end of the block
                {
                        // This is just
                        double n{-19.70};
                        int n{20};
                }
        }
}
```

If you were to compile this, you would get error messages such as

```
example.cpp: In function ‘void f(int)’:
example.cpp:16:6: error: declaration of ‘int n’ shadows a parameter
  int n{5};
      ^
example.cpp:18:9: error: redeclaration of ‘double PI’
  double PI{3.1415926535897932};
         ^~
example.cpp:17:9: note: ‘double PI’ previously declared here
  double PI{3.14};
         ^~
example.cpp:27:7: error: conflicting declaration ‘int PI’
   int PI{3};
       ^~
example.cpp:26:10: note: previous declaration as ‘double PI’
   double PI{355.0/113.0};
          ^~
example.cpp:35:8: error: conflicting declaration ‘int n’
    int n{20};
        ^
example.cpp:34:11: note: previous declaration as ‘double n’
    double n{-19.70};
           ^
```

The first error message states that within the outermost body of the function, a local variable cannot have the same identifier as that of a parameter.

All other error messages should be read in pairs, where it first states that an identifier is being redeclared, and the second indicates on which line the original declaration occurred. You will note that:

1. the error message never cares whether or not they types match or not (even if they are given the exact same initial value), and
2. the error message never refers to any identifier declared either as a parameter or local variable in an outer block.

If you were to remove the duplicate declarations:

```
void f( int n ) {
        double PI{3.14};

        // The 'n' in the condition refers back to the parameter
        if ( n >= 0 ) {
                std::cout << n << ", " << PI << std::endl;
                // This prints 3, 3.14

                double PI{355.0/113.0};
                std::cout << n << ", " << PI << std::endl;
                // This prints 3, 3.14159

                {
                        // This is just
                        double n{-19.70};
                        std::cout << n << ", " << PI << std::endl;
                        // This prints -19.7, 3.14159
                } // 'n' goes out of scope at the end of this block
        } 'PI' goes out of scope at the end of this block

        std::cout << n << ", " << PI << std::endl;
        // This prints 3, 3.14
        //  - recall '3' was the argument of the function call to 'f'
} 'n' and 'PI' go out of scope at the end of this block
```

3a. Other oddities: multiple declarations in one statement

Consider the following program:

```
// Pre-processor include directives
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
        int a{3}, b{a + 5}, c{a + b + 2};
        int f{d + e + 2}, e{d + 5}, d{3};

        std::cout << a << ", " << b << ", " << c << std::endl;
        std::cout << d << ", " << e << ", " << f << std::endl;

        return 0;
}
```

In the first declaration, the initial value of `b` depends on `a`, and the initial value of `c` depends on both `a` and `b`. This first line actually compiles, and if you were to compile only this line and the corresponding `std::cout` statement, the output will be as you expect:

```
3, 8, 13
```

Unfortunately, the next line is the problem: declarations are processed by the compiler in order, so f is declared and initialized before e is declared, so this results in a compile-time error.

```
example.cpp: In function ‘int main()’:
example.cpp:10:8: error: ‘d’ was not declared in this scope
  int f{d + e + 2}, e{d + 5}, d{3};
        ^
example.cpp:10:12: error: ‘e’ was not declared in this scope
  int f{d + e + 2}, e{d + 5}, d{3};
            ^
```

3b. Other oddities: using the local variable in its initialization

Consider the following program:

```
// Pre-processor include directives
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
        int n{42};
        std::cout << n << std::endl;

        {
                int n{n + 1};
                std::cout << n << std::endl;
        }

        return 0;
}
```

The first `std::cout` statement clearly should print `42`, which is does; however, in the following block of statements a second `n` is declared, and this identifier shadows the outside identifier with the same name. Your initial thought may be that because `n` is only now being declared, that the second `n` should be initialized to `43`, as it might use the shadowed local variable. This, especially in light of the previous example, may be thought to be reasonable behavior; however, the initialization occurs after the local variable is declared, so instead, the `n` in `{n + 1}` is the uninitialized `n` declared in that block of statements.

Consequently, the value is not `43`. When this program is executed on `ecelinux.uwaterloo.ca`, the uninitialized `n` happens to have the value of `0`, so after initialization, it now has the value `1`. On the other hand, when this code is executed in repl.it, it can be deduced during one execution that the uninitialized value of `n` is `32764` as the initialized value is `32765`; however, executing this program multiple times gave different uninitialized values of `n`.