Lesson 6 - Scope and Extent
The Lifetime of Variables

Summary

Understand about the different scopes in C++ Programs and how this affects the life-time (extent) of variables.

New Concepts

Scope, Extent.

Scope and Extent

C++ programs incorporate the notion of "scope" and "extent". Scope creates hierarchical divisions between different sections of any C++ program, limiting the extent of named variables within that scope. One form of scope we have already seen exists with functions. The extent (or existence) of local-variables is limited to the scope of a function which we create. Curly brackets `{ }` at the beginning and at the end of any function define the scope parameters.

On line 5 we define `some_function` thus beginning a new scope. Stack variables `a` and `letter` are defined within `some_function` and therefore only exist for the duration of that function's execution (see lines 6 and 7). On line 8 we close the definition of `some_function` with a `}` thus ending its scope.

We can declare "global" variables in C++ whose scope extends to the entire `cpp` file. Global variables exist for the duration of the program's execution. We define a global variable `global_variable` on line 10. Notice that this variable is not defined within the enclosing brackets of any function, hence it has global scope.

```
1  #include <iostream>
2
3  using namespace std;
4
5  void some_function() {
6     int a = 400;
7     char letter = 'b';
8  }
9
10 int global_variable = 50;
```

scope_extent.cpp

On line 12 we enter the scope of the obligatory `main` function. In the scope of main we still have access to `global_variable`. Global scope encompasses the scope of main, which means we can (usually) access variables from an encompassing scope, as displayed by the `cout` on line 13.

On lines 15 and 16 we have the declaration of an integer variable `a` and a pointer to an integer `b` which are local to the scope of the main function. We will use these to demonstrate the effects of scope and extent.

```
12 int main () {
13    cout << "global variable contains the value " << global_variable << "\n";
14
15    int a = 100;
16    int* b;
```

scope_extent.cpp

We can create a scoped block of code by inserting curly brackets within a function. As with functions, we initiate a scoped-block with the open-bracket `{` and close it with the close-bracket `}`. Following the open-bracket on line 17 have progressed from the scope of the main function to the scope created by the opening-bracket. Note that scope is hierarchical such that an "inner" scope must exist solely within the brackets of an "outer" scope.

Notice now the "extent" of stack-variables declared within a refined scope. On line 18 we declare another variable called `a` whose extent is limited to this local-scope (within the curly brackets). The naming of variable `a` does not cause a conflict with the previous variable `a` declared outside this scope from line 15. When we output `a`'s value we see that it is 200.

On line 22 we allocate memory on the heap. We must be sure to release it before we leave this scope (here we are using a "constructor" to initialise the heap memory to the value 300; more on constructors later). Alternatively, we can use another pointer which will exist beyond the current scope to ensure the memory is not lost, the pointer `b` for instance will continue to exist beyond this scope.

On line 24 we reassign `b` to point to the memory allocated on the heap, as `b` was not previously pointing to anything, it is safe to reassign it, but this is not always the case. Be careful when reassigning pointers, you might create a memory leak!

```
17    {
18    int a = 200;
19
20    cout << "the value in a is " << a << "\n";
21
22    int* local_ptr = new int(300);
23
24    b = local_ptr;
25    }
26
27    cout << "the value in a is " << a << "\n";
28
29    cout << "the value pointed to by 'b' is " << *b << "\n";
30    delete b;
31    b = NULL;
32
33    std::cout << "output some data" << "\n";
34
35    return 0;
36 }
```

scope_extent.cpp

On line 25 we close the scope using `}` returning to the scope of main function. Any variables created in the inner-scope no longer exist. We can show this by printing the value of variable `a` (see line 27). Note that the output will show 100 again. The variable `local_ptr` has ceased to exist, but because we stored the memory address in `b` we can still access (and release) the heap memory allocated, which we do on lines 29 and 30.

In C++ we also have something called the "scope operator" denoted by a double colon `::` which comes into use with namespaces (see line 33). We have already seen the command to tell the compiler which namespace(s) we wish to use: `using namespace std;`. This allows us to write `cout` without the need to provide the namespace `std` where the `cout` function is defined. However, we could also have used the fully qualified expression.

Exercises

1. When are global variables created and destroyed? How does this differ to variables created in a function?

2. What is the scope of `x` in the following lines of code...

```
37 int y = 2;
38 for(int x = 0; x < y; ++x) {
39     cout << arr[x] << ", ";
40 }
```

scope_extent.cpp

3. Amend the code to create two integer variables in the main function with the same name, use what you've learned about scope and extent to prevent a naming conflict.

4. Alter the program such that the `some_function` function belongs to a newly created namespace called `some_namespace`. Now create another `some_function` function as a duplicate of the original, within another namespace called `some_other_namespace`. Finally, call both functions within the main function using the namespace syntax (i.e. `namespace::function`).