Learning Objectives: Variable Scope

- **Differentiate between global and local scope**
- **Explain what the keyword “const” does**

Local Scope

Local Scope

Take a look at the code below. The first function declares the variable `my_var` and then prints it. The second function also prints `my_var`. What do you think the output will be?

```
void MyFunction1() {
  string my_var = "Hello";
  cout << my_var << endl;
}

void MyFunction2() {
  cout << my_var << endl;
}

int main() {
  MyFunction1();
  MyFunction2();
  return 0;
}
```

C++ returns an error such as `error: 'my_var' was not declared in this scope` at the line containing `cout << my_var << endl;` within the second function. This happens because variables declared inside a function have **local** scope. Variables with local scope can **only** be used within that function. Outside of that function, those local variables cannot be accessed. In the image below, the light blue box represents the scope of `my_var`. Since `MyFunction2` (denoted in a light red box) is outside the scope of `my_var`, an error occurs.

```
void MyFunction1() {
    string my_var = "Hello";
    cout << my_var << endl;
}
```

```
void MyFunction2() {
    cout << my_var << endl;
}
```

```
int main() {
    MyFunction1();
    MyFunction2();
    return 0;
}
```

.guides/img/LocalScope

challenge

What happens if you:

- Change `MyFunction2` to look like this:

```
void MyFunction2() {
  string my_var2 = "Hello";
  cout << my_var2 << endl;
}
```

More Local Scope

Each function has its own local scope. That means you can declare two variables with the same name as long as they are in separate functions. The blue `my_var` exists only in the light blue box, and the red `my_var` exists only in the light red box. The boundaries of local scope keep C++ from overwriting the value of the first variable with the contents of the second.

```
void MyFunction1() {
    string my_var = "Hello";
    cout << my_var << endl;
}

void MyFunction2() {
    string my_var = "Bonjour";
    cout << my_var << endl;
}

int main() {
    MyFunction1();
    MyFunction2();
    return 0;
}
```

.guides/img/LocalScope2

```
void MyFunction1() {
  string my_var = "Hello";
  cout << my_var << endl;
}

void MyFunction2() {
  string my_var = "Bonjour";
  cout << my_var << endl;
}

int main() {
  MyFunction1();
  MyFunction2();
  return 0;
}
```

challenge

What happens if you:

- Declare `MyFunction3()` as:

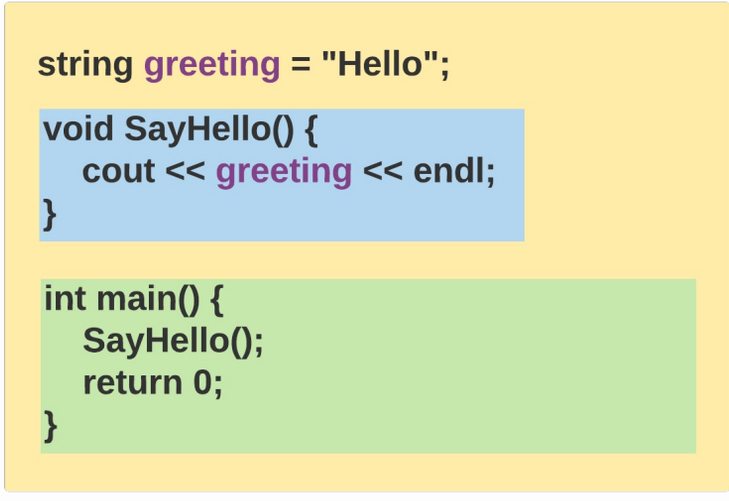
```
void MyFunction3() {
  string my_var = "Hola";
  cout << my_var << endl;
}
```

and call it by including `MyFunction3();` within the `main()` function.

Global Scope

Global Scope - Referencing Variables

When a variable is declared inside a function, it has local scope. When a variable is declared outside of all existing functions, it has global scope. Since global variables are declared outside of functions, they can be referenced inside any function. Look at the image below. The yellow block holds everything within the program including the variable `greeting`. This enables all functions within the program to access that variable since it is considered to be **global**. Copy and paste the code below and then click `TRY IT`.



.guides/img/GlobalScope

```
string greeting = "Hello"; //global variable

void SayHello() {
  cout << greeting << endl; //can access global variable
        greeting
}

int main() {
  SayHello();
  return 0;
}
```

Global Scope - Modifying Variables

Once a global variable becomes available, a function can modify the content of that variable as needed.

```
string greeting = "Hello";

void SayHello() {
  greeting = "Bonjour";
  cout << greeting << endl;
}

int main() {
  SayHello();
  return 0;
}
```

challenge

What happens if you:

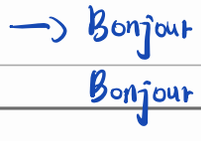
- Change `greeting = "Bonjour";` within `SayHello()` to `greeting = "Hola";`?
- Change the entire program to:

```
string greeting = "Hello";

void SayHello1() {
  greeting = "Bonjour";
  cout << greeting << endl;
}

void SayHello2() {
  cout << greeting << endl;
}

int main() {
  SayHello1();
  SayHello2();
  return 0;
}
```



Notice how in the code above the functions `SayHello1()` and `SayHello2()` end up printing the same output. The result of `greeting` within `SayHello2()` is affected by the modification of `greeting` within `SayHello1()`.

Global vs. Local Scope

Global vs. Local Scope

If a variable is declared and initialized both locally and globally, that variable will retain its content depending on how it is used. In the example below, `my_var` is declared and initialized globally as `global scope` and locally as `local scope`. Since the variable has differing scopes, it retains its value when called or printed.

```
string my_var = "global scope";

void PrintScope() {
  string my_var = "local scope";
  cout << my_var << endl;
}

int main() {
  PrintScope();
  cout << my_var << endl;
}
```

→ local scope
global scope

The exception to this rule is when a function modifies a global variable. In such a case, the content of the global variable is changed.

```
string my_var = "global scope";

void PrintScope() {
  my_var = "local scope";
  cout << my_var <<endl;
}

int main() {
  PrintScope();
  cout << my_var << endl;
}
```

→ local scope
local scope

challenge

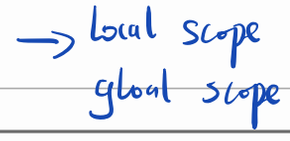
What happens if you:

- Change the code to:

```
string my_var = "global scope";

void PrintScope(string my_var) {
  my_var = "local scope";
  cout << my_var << endl;
}

int main() {
  PrintScope(my_var);
  cout << my_var << endl;
}
```



When a global variable is also a function parameter, it is considered to be the same as if the function declared and initialized its own local variable. In this case, the variable has both a local and global scope and will retain its value depending on its scope.

The “const” Keyword

If you want a global variable to remain unchanged throughout the program, you can declare the variable as `const`. `const` variables are also referred to as **constants**. Constants never change and are “named with a leading ‘k’ followed by mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation.” **Source:** Google

Another common way to label constants is to use names in all uppercase with words separated by an underscore (_). For example: `MY_CONSTANT`.

```
const string kMyConstant = "I NEVER CHANGE";

void PrintScope() {
  kMyConstant = "I CAN'T CHANGE";
  cout << kMyConstant << endl;
}

int main() {
  PrintScope();
  cout << kMyConstant << endl;
}
```

→ Error

challenge

What happens if you:

- Remove the keyword `const` from the code? → I CAN'T CHANGE
  I CAN'T CHANGE