Topic 8

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals
9. Reference parameters
10. Recursive functions

Variable Scope

You can only have *one* **`main`** function but you can have as many variables and parameters spread amongst as many functions as you need.

Can different variables have the same name in different functions?

YES! (though it is a bad practice to intentionally do so)

How does the compiler keep track of variables of the same name?

By analyzing their ***SCOPE***

Variable Scope is Limited to the Code Block

A variable or parameter that is defined within a function or other statement is visible from the point at which it is defined until the end of the block.

Ie, from the first appearance of the variable until the next closing curly brace (or the end of the statement).

This area is called the *scope* of the variable.

Variable Scopes Don’t Overlap

The scope of a variable is the part of the program in which it is *visible*.

Because scopes do not overlap, a name in one scope cannot conflict with any name in another scope.

A name in one scope is “invisible” in another scope

Variable Scope Example

```
double cube_volume(double side_len)
{
   double volume = side_len * side_len * side_len;
   return volume; //this volume is local to this function
// and it disappears at the completion of the return
}
int main()
{
   double volume = cube_volume(2);
   cout << volume << endl;
   return 0;
}
```

Each **`volume`** variable is defined in a separate function.

Coincidentally in this code, the 2 variables have the same value (8), but that is not usually the case with variables of the same name in different scopes.

Variable Scope and Blocks

Variables defined inside a block are *local* to that block, and have no meaning outside the {} of the block.

Example:

```
for(int i=0; i<5; i++)
     cout << i << endl;
// "i" exists only until the closing ; of
  the for() statement
```

A function names a block.

Recall that variables and parameters do not exist after the function is over—because they are local to that block.

Variable Scope Errors

It is *not legal* to define two variables or parameters with the same name in the same scope.

For example,

```
int test(double volume)
{
   double volume = cube_volume(2); //ERROR
   double volume = cube_volume(10); //ERROR
// ERROR: cannot define another volume variable
// ERROR: or variable with same name as input
// parameter in the same scope
...
}
```

Variable Scope – Nested Blocks

However, you can define another variable with the same name in a *nested block*.

```
double withdraw(double balance, double amount)
{
   if (...)
   {
      double amount = 10;
      ...
   }
   ...
}
```

a variable named **`amount`** local to the **`if`**'s block

– *and* a parameter variable named **`amount`**.

But this is a confusing construct, and is discouraged.

Global Variables

- A **global variable** is one that is defined outside of any function, and thus is visible to all functions

All the variables we have used so far are called “local”

- Generally, global variables are ***not*** a good idea

- Because they are seen by all functions, determining which functions are interacting by changing them becomes tricky when debugging

- Some situations require global variables, such as a shared time-of-day clock in an embedded control system

Global Variable Example

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}

int main()
{
   withdraw(1000);
   cout << balance << endl;
   return 0;
}
```

Global Variable Pitfalls

In the previous program there is only one function that updates the **`balance`** variable.

But there could be many, many, many functions that might need to update **`balance`** each written by any one of a huge number of programmers in a large company.

Then we would have a problem.

Avoiding the Global Variable

```
//A better way to code the previous banking example,
// eliminating the global variable

// Function returns new balance after a withdrawal
double withdraw(double balance, double amount)
{
   return balance - amount;
    //negative balance will indicate overdraft
}

int main()
{
   int balance = 10000; // local variable
    balance = withdraw(balance, 1000);
   cout << "Balance = " << balance << endl;
   return 0;
}
```

Self-Check: Global Variable Example #2

What does this program print? (Answer: 4 16)

```
#include <iostream>
using namespace std;

int p;
int contribute(int f)
{
   if (f != 0) { p = p * f; }
   return p;
}
int main()
{
   int n = 2;
   p = n;
   n = contribute(n);
   contribute(p);
   cout << n << " " << p << endl;
   return 0;
}
```

As you can see, the global variable p makes it difficult to follow the program flow.