Reference Parameters

- There are two ways to pass references to functions, *pass-by-value* and *pass-by-reference*.
- When an argument is passed by value, a *copy* of the argument's value is made and passed (on the function-call stack) to the called function.
- Changes to the copy do *not* effect the original variable's value in the caller. This prevents accidental side effects that might otherwise hinder software development.
- One disadvantage of *pass-by-value* is that, if a large data item is being passed, copying that data can take execution time and memory space.
- With *pass-by-reference*, the caller gives the called function the ability to access the caller's data *directly*, and to *modify* that data.
- *Pass-by-reference* is good for performance reasons, because it eliminates the *pass-by-value* overhead of copying large amounts of data. But *pass-by-reference* can weaken security, and the called function can corrupt the caller's data.
- To indicate a function parameter is passed by reference, simply follow the parameter type with an ampersand.

```
squareByReference(int& valueRef);
```

- Warning: as reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original variable are changed by the function.

```
int squareByValue(int number)
{
    return number *= number;  // caller's argument not modified
}

void squareByReference(int& nuberRef)
{
    nuberRef *= nuberRef;
}

int main()
{
    int x = 2;
    int y = 4;

    cout << "x = " << x << " before squareByValue()" << endl;
    cout << "Value returned by squareByValue(): " << squareByValue(x);
    cout << endl;
    cout << "x = " << x << " after squareByValue()" << endl;
    cout << endl;
    cout << "y = " << y << " before squareByReference()" << endl;
    squareByReference(y);
    cout << "y = " << y << " after squareByReference()" << endl;

    return 0;
}
```

const “read-only” References

- There are times when you want the power and speed of passing by reference without the security risk of changing an argument’s content. This is accomplished by using the const qualifier *before* the type name in the parameter’s declaration.
- const declares a variable to be “read only”.
- If you attempt to modify the above previous by making the squareByReference() function ready only, e.g.

```
void squareByReference(const int& nuberRef);
```

- The compiler gives the error “Cannot assign to variable with const qualified type”.
- Functions can return references to local variables, but this can be dangerous. When returning a reference to a local variable—unless that variable is declared static—the reference refers to a variable that’s *discarded* when the function terminates. References to undefined variables are known as “dangling references”.

swap() function using Reference Parameters

- A common dilemma when programming is needing to swap two values.
- A simple `swap()` function can easily be created to do this using reference parameters, e.g.,

```
void swap(int& value1, int& value2)
{
    int temp;
    temp = value1;
    value1 = value2;
    value2 = temp;
}

int main()
{
    int a = 4;
    int b = 7;

    cout << "BEFORE a:" << a << endl;
    cout << "BEFORE b:" << b << endl;

    swap(a, b);

    cout << "AFTER a:" << a << endl;
    cout << "AFTER b:" << b << endl;

    return 0;
}
```

- In `main()`, two integer variables `a` and `b` are defined.
- Those integers are passed to a function `swap()` by reference.
- Compiler can identify this is pass by reference because function definition is `void swap(int& value1, int& value2)` (notice the `&` sign after data type).
- Only the reference (address) of the variables `value1` and `value2` are received in the `swap()` function, and swapping takes place in the original address of the variables.
- In the `swap()` function, `value1` and `value2` are formal arguments which are actually same as variables `v1` and `v2` respectively.
- There is another way of doing this same exact task using pointers, see the Pointers notes for the pointer `swap()` example.
- You'll often see such a `swap()` function used in sorting programs.