Lecture Two: Functions

Functions are the way we represent actions and computations. Whenever we want to do something that is worthy of a name, we write a function. The simplest way of passing an argument to a function is to give the function a copy of the value you use as an argument. This is called pass-by-value.

```cpp
#include <iostream>
#include <vector>

using namespace std;

void display(vector<int> v) {
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

void fun(vector<int> v) {
    v[v.size()-1] = -1;
    v[0] = 2;
    cout << "After passing it to fun and changing "
         << "its first and last elements: \n   ";
    display(v);
}

int main() {
    vector<int> v(6);
    cout << "This is the vector in the main program: \n   ";
    display(v);
    fun(v);
    cout << "The vector in main after fun finished: \n   ";
    display(v);
}
```

Passing a vector argument by value to a function. Both fun and display get their argument by value. This means each one of them receives a full-blown copy of the vector v. One of the functions (display) only prints the copy, the other one (fun) changes it a bit. But changes are not seen in the caller program. This is obvious even though display always gets a copy to print. To drive this point home even further let’s examine the basics of copying, see below.
Both `a` and `b` are initialized when they are declared. When we assign `a` to `b` we create a copy of the value that `a` has and pass it to `b`, which now points to this copy. Changing `a` and `b` separately is now possible.

What happens when function `fun` receives its argument, in slow motion. But notice the change.

Pass by value is simple, straightforward, and efficient when we pass small values, such as `int`, a `double`, or a `Point`. But what if the value is large, such as an image, a large table of values, or a very long string? Then, copying can be costly. In the code above we changed the way the argument `v` is passed to `display`. The `&` means reference and the `const` is there to stop `display` to modify its argument by accident. Pass by const reference is a useful and popular mechanism. But what if we did want a function to modify its arguments? Let’s consider references from a more technical point of view:

```cpp
#include <iostream>
#include <vector>

using namespace std;

void display(const vector<int>& v) {
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

int main() {
    vector<int> a(6), b;
    b = a;
    a[a.size() - 1] = -1;
    display(a);
    display(b);
}
```
Passing a plain reference to a function eliminates the copying and allows the function to make changes.

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<double> read_inputs() {
    vector<double> result;
    cout << "Please enter values, Q to quit:" << endl;
    bool more = true;
    while (more) {
        double input;
        cin >> input;
        if (cin.fail()) more = false;
        else result.push_back(input);
    }
    return result;
}

void modify(vector<double>& a) {
    for (unsigned int i = 0; i < a.size(); i++)
        a[i] += 2;
}

void display(const vector<double>& v) {
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

int main() {
    vector<double> v = read_inputs();
    display(v);
    modify(v);
    display(v);
}
```

One of the functions works with the standard input and collects floating-point numbers, in a vector. The vector is stored in the caller program. Another function modifies the values in the vector by adding 2 to them. Finally the display function is called twice, before and after modify, to show what’s happening.

Pass by reference is clearly a very powerful mechanism: we can have a function operate directly on any object to which we pass a reference. Here are some thoughts on when to use each of the parameter passing mechanisms we have outlined above:

- use pass by value to pass very small objects
- use pass by const reference to pass large objects that you don’t need to modify
- return a result rather than modifying an object through a reference argument
- use pass by reference only when you have to

These rules lead to the simplest, least error-prone, and most efficient code, says Stroustrup.
Write a function that receives two vectors of ints v1 and v2 (must have the same size) as arguments. The function should rearrange the values in the two vectors as follows: make each element in v1 the larger of the corresponding elements in v1 and v2; similarly, make each element of v2 the smaller.

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

void separate(vector<int>& v1, vector<int>& v2) {
    for (unsigned int i = 0; i < v1.size(); i++)
        if (v1[i] < v2[i]) {
            int temp = v1[i];
            v1[i] = v2[i];
            v2[i] = temp;
        }
}

void display(const vector<int>& v) {
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

int main() {
    vector<int> a(6), b(6);
    for (unsigned int i = 0; i < a.size(); i++) a[i] = i;
    for (unsigned int i = 0; i < b.size(); i++) b[i] = b.size() - i - 1;
    display(a);
    display(b);
    cout << "----------------\n";
    separate(a, b);
    display(a);
    display(b);
}
```

Using pass by reference arguments is the only reasonable choice for such a function.