22. Reference Variables and Reference Parameters

Objectives

- Understand and use reference variables
- Write functions with reference parameters

You have already seen references (or reference variables). This chapter is a quick review and some more information on references.

Problems with Our Functions

What about writing a function to perform “array assignment”, i.e., we want to do the following:

```
int x[] = {1,2,3};
int xSize = 3;
int y[3];
int ySize = 3;
arrayAssign(y, 3, x, 3); // like y = x
```

That's easy: Just write a for-loop in the function to pass the values from one to the other.

This works for arrays of the same size. What if I use a “length” for an array?

What I mean is that suppose I have an array of size 1000. I want to use the array like a container: I want to put values into the array and I want to pull values out of the array. The length of an array measures the number of things I put into the array that has not been taken out. For instance suppose I have the following:

```
const int GOOGLE_STOCK_PRICE_SIZE = 1000;
double googleStockPrice[GOOGLE_STOCK_PRICE_SIZE];
```

which records the stock as the day progresses. `googleStockPrice[0]` is the stock price of Google at 9:00AM when the stock market opens, and `googleStockPrice[1]` is the stock price at 9:01AM, etc. At 9:05AM, I have stock prices of Google for 9:00AM, 9:01AM, 9:02AM, 9:03AM, 9:04AM, 9:05AM, i.e. there are 6 values. Of course the array has a size of 1000. But only `googleStockPrice[i]` for i = 0, 1, 2, 3, 4, 5 are valid at that point in time. I can keep track of how many values (from index 0) are valid using an integer variable, say `googleStockPriceLen` (Len for length).

```
const int GOOGLE_STOCK_PRICE_SIZE = 1000;
double googleStockPrice[GOOGLE_STOCK_PRICE_SIZE];
int googleStockPriceLen = 0;
```

So make sure you see the difference between the two different measures of an array: one measures the **total number of values the array can hold**, and the other measures **the number of values that have been placed into the array that has not been taken out**.

Now suppose we want to write an array assignment function for such a general situation (remember: you want functions to be re-usable, therefore you want to make them general enough), then you want to be

able to do this:

```
const int XSIZE = 10;
int x[XSIZE] = {42, 41, 40};
int xlen = 3;
const int YSIZE = 10;
int y[YSIZE];
int ylen = 0;

arrayAssign(y, ylen, x, xlen); // sorta like y = x
```

After the function call you want `y[0] = 42`, `y[1] = 41`, `y[2] = 40` and `ylen = 3`.

Now note this: A function can change the values of `y` which is an array. But the problem is that `ylen` is an integer variable and hence cannot be changed by a function when it is passed in as a pass-by-value parameter. Right? Here's an example:

```
void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}

int main()
{
    int x = 0;
    int y = 42;
    swap(x, y); // x, y NOT CHANGED!!!!!!!!!!!!!!!!
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

One way to get around that is that you can return a value from a function and use that to change the value of `ylen` in the calling function. So the code using the array assignment function would look like this:

```
const int XSIZE = 10;
int x[XSIZE] = {42, 41, 40};
int xlen = 3;
const int YSIZE = 10;
int y[YSIZE];
int ylen = 0;

ylen = arrayAssign(y, x, xlen); // sorta like y = x
```

That will work. **But** the prototype of this function would look very different from other useful array functions. This is how a header file for all the useful array functions if you have one:

```
// array.h
```

```
bool arrayEqual(int[], int, int[], int);
bool arrayNotEqual(int[], int, int[], int);
void bubbleSort(int[], int);
int binarySearch(int[], int, int target);
int arrayAssign(int[], int[], int);
```

One quality of a well-designed program is one with **least surprises**. In all parameters, you always see an array followed by a length ... EXCEPT for the case of `arrayAssign()`.

There's another problem with using return values to change variables: If you want to change the value of two variables through one single function call you CAN'T. **A function cannot return two values at the same time**. The following **does NOT work**:

```
int,int swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
    return a, b // abominable
}


int main()
{
    int x = 0;
    int y = 42;
    x, y = swap(x, y); // :o
    return 0;
}
```

Pass-by-Reference

Now try this

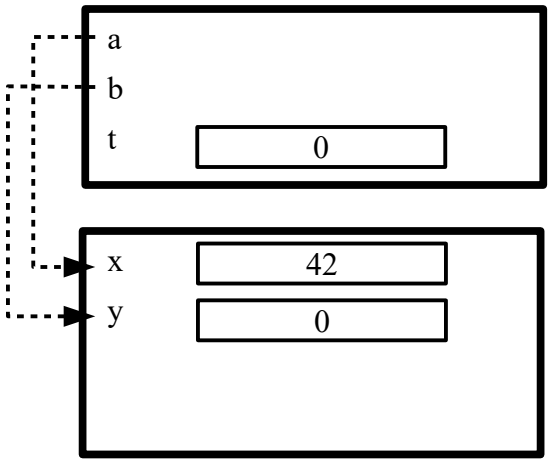
```
void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}

int main()
{
    int x = 0;
    int y = 42;
    swap(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

The variables `a` and `b` are called **reference variable**, in particular they are integer reference variables. They are different from the regular plain-jane variables. They **do not have their own memory for keeping values**. They **refer** to other variables. A reference variable is an **alias** of another variable.

Now back to our program with the `swap()` function:

```
void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}

int main()
{
    int x = 0;
    int y = 42;
    swap(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

The memory model just before the return from `swap()` to `main()`. Note that a and b do not have their own memory for values



when `main()` calls `swap()`, the integer reference variable `a` in `swap()` will **refer** to the memory of `x`; it **does not receive a value** from `x`. As a matter of fact, referring the diagram, you see that `a` does not even have its own memory!

Likewise the integer reference variable `b` in `swap()` does not have have

its own memory but rather it refers to the memory of `y`.

We call this type of parameter passing **pass-by-reference**. We say that the `a` in `swap()` is a pass-by-reference parameter.

To simplify the scenario, let's forget about functions and just look at reference variables alone. Try this:

```
int i = 5;
int & j = i;
std::cout << i << ' ' << j << std::endl;
i = 42;
std::cout << i << ' ' << j << std::endl;
j = 0;
std::cout << i << ' ' << j << std::endl;
```

As you can see `j` has type

```
int &
```

i.e. `j` is an integer reference. It's initialized to `i`. That means that `j` refers to the memory of `i`.

Exercise. Can you declare reference variables of doubles?

```
double i = 3.14;
_________ j = i; // declare j to be a reference to I

std::cout << i << ' ' << j << std::endl;
i = 2.718;
std::cout << i << ' ' << j << std::endl;
j = -1;
std::cout << i << ' ' << j << std::endl;
```

See how a reference variable behaves? `j` refers to the memory of `i`. `j` is just another name for `i`; it's an alias for `i`.

In terms of our program

```
void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}

int main()
{
    int x = 0;
    int y = 42;
    swap(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
```

```
}
```

when `swap()` is called, the reference variable `a` is created and is assigned `x` which makes `a` a reference to `x` – `a` uses the memory of `x`. Whatever you do to `a` in `swap()` therefore also change `x`.

Exercise. Fix this program so that it works:

```
void inc(int x)
{
  x++;
}

int main()
{
  int a = 42;
  inc(a);
  std::cout << a << std::endl; // should see 43
  return 0;
}
```

(Of course this is just for demo. In a real world program, you should use ++ instead of writing a function for it!)

Exercise. Complete this program:

```
// sort x, y, z in ascending order using bubblesort
void sort(int & x, int & y, int & z)
{
  ... CODE ...
}

int main()
{
  int x = 3, y = 5, z = 2;
  sort(x, y, z);
  std::cout << x << ' ' << y << z << ' '
            << std::endl; // should get: 2 3 5
  return 0;
}
```

Advice: In general we only use references only when a function is meant to change the value of a variable that is passed in. We do not want accidental changes in the function to propagate back to the caller. In other words, use references only when you need to. Therefore you should minimize the use of reference variables.

Style

Instead of

```
int & x
```

It's actually more common to write

```
int& x
```

Another style is this:

```
int &x;
```

Gotcha

Warning!!! This code fragment

```
int x = 42;
int & a = x, b = x, c = x;
```

means this:

```
int x = 42;
int & a = x;
int b = x;
int c = x;
```

If you really want `b` and `c` to be references, then you should do this:

```
int x = 42;
int & a = x, & b = x, & c = x;
```

or just declare one thing per line:

```
int x = 42;
int & a = x;
int & b = x;
int & c = x;
```

Array assignment function

Let's get back to our array assignment function.

```
#include <iostream>

void println(int x[], int len)
{
    for (int i = 0; i < len; i++)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;
}


void arrayAssign(int y[], int & ylen,
                 int x[], int xlen)
{
    ylen = xlen;
    for (int i = 0; i < xlen; i++)
    {
        y[i] = x[i];
    }
}


int main()
{
    int x[] = {1, 2, 3};
    int y[100];
    int ylen = 0;
    arrayAssign(y, ylen, x, 3);
    println(y, ylen);
    return 0;
}
```

(WARNING: the code does not check that `y` has enough space for the copying process.)

Now's your turn:

Exercise. Write an `arrayConcat()` function (concatenation of arrays) that appends an array to another. In other words, if `x` is the array `{1, 2, 3}` and `y` is the array `{6, 7, 8, 9}`, after calling this function with `x` and `y`, you want `x` to be `{1, 2, 3, 6, 7, 8, 9}`.

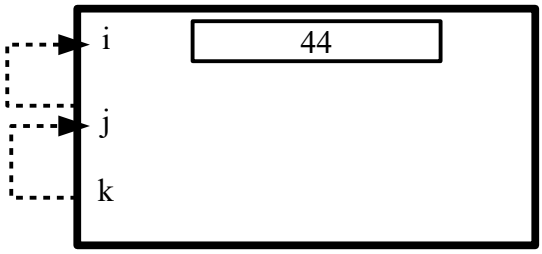
```
#include <iostream>

void println(int x[], int len)
{
```

```
    for (int i = 0; i < len; i++)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;
}


void arrayConcat(int x[], int & xlen,
                 int y[], int ylen)
{
    ... CODE ...
}


int main()
{
    int x[100] = {1, 2, 3};
    int xlen = 3;
    int y[100] = {6, 7, 8, 9};
    int ylen = 4;
    arrayConcat(x, xlen, y, ylen);
    println(x, xlen);
    return 0;
}
```

More examples

Exercise. Can you create a reference that refers to a reference?

```
double i = 3.14;
double & j = i;
double & k = j; // k references j which references i
i = 2.718;
std::cout << i << ' ' << j << ' ' << k << std::endl;
j = 1.414;
std::cout << i << ' ' << j << ' ' << j << std::endl;
k = 0.693;
std::cout << i << ' ' << j << ' ' << j << std::endl;
```



Here's the relevant rule: **A reference can refer to another reference**. In other words, a reference variable can reference a variable or it can reference another reference variable.

Exercise. What is the output?

```
void f(int & a, double c)
{
    a = a + int(c);
    c = 1.0;
}


void g(int & b, double & c)
{
    b *= 2 + int(c);
    c = 0.0;
    f(b, c);
}


int main()
{
    int x = 42;
    double y = 3.14;
    g(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

(Suggestion: Drawing a memory model might help.)

Here's another rule: **References must be initialized. Once you declare a reference, it**

must immediately refer to the memory of another variable. Try this:

```
int i = 5;
int & j;      // create a reference variable ...
j = i;        // ... and *then* make the reference
```

Exercise. Does this work?

```
int i = 5;
int & j = i + 1;
```

Or this?

```
int i = 5;
int k = 6;
int & j = i + k;
```

Why?

Yet another rule: **Variable references must be initialized to variables or another reference (not an expression)** (This one is kind of obvious.) Here's an example:

Exercise. Of course you know that you can do this:

```
double x = 3.14;
int y = x;
```

What about this:

```
double x = 3.14;
int & y = x;
```

Rules, rules, rules ... **The type of a reference must match the type it's referring to**.

Exercise. What is the output? (Or find the errors).

```
void swap(int & b, int & c)
{
    int t = b;
    b = c;
    c = t;
}


int main()
{
    int x = 42;
```

```
    double y = 1.234;
    swap(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

Exercise. What is the output? (Or is there an error?)

```
void g(double & b, double & c)
{
    b *= 2 + int(c);
    c = 0.0;
}


int main()
{
    int x = 42;
    double y = 3.14;
    g(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

(Suggestion: Drawing a memory model might help.)

Exercise. You now know this works:

```
int i = 0;
int & a = i;
```

What about the following?

```
int i = 0;
const int & a = i;
```

```
const int i = 0;
const int & a = i;
```

```
const int i = 0;
int & a = i;
```

Try all of them. Give a reason for those that does not work. For those that work, can you change the value of `i` using `i`?

```
i = 42;
```

Can you change the value of `i` using `a`?

```
a = 42;
```

Why?

Exercise. What is the output? (Or find all the errors).

```
void g(const int & b, const double & c)
{
```

```
    b *= 2 + c;
    c = 0.0;
    f(b, c);
}

int main()
{
    const int x = 42;
    double y = 3.14;
    g(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

Exercise. What is the output? (Or find all the errors).

```
void g(int & b, const double & c)
{
    b *= 2 + c;
    c = 0.0;
    f(b, c);
}

int main()
{
    const int x = 42;
    double y = 3.14;
    g(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

As mentioned before, a variable reference must reference a variable (or another reference). One exception is that **a constant reference can refer to a constant**. Try this:

```
const int & i = 5;
```

On the other hand the following won't work since you already know that a reference must refer to a variable:

```
int & i = 5;
```

A **constant reference** is a reference that refers to a value and treat that value as a constant.

```
int x = 42;
const int & y = x; // y is a const ref to x
const int a = 0;
const int & b = a; // b is a const ref to a
```

It's very important to remember that if `x` is a non-constant variable and `y` is a constant reference to `x`, then you cannot change the value of `x` using `y` – because `y` views the value of `x` as though it's constant. However `x` can still change it's value:

```
int x = 42;
const int & y = x;
std::cout << x << ' ' << y << '\n'
x = 0;                // changing x ... OK
std::cout << x << ' ' << y << '\n'
y = -1;               // changing x using y ... BAD!!!
```

Exercise. What is the output? (Or find the error)

```
#include <iostream>

int f(int x[], int & len)
{
    int p = 1;
    for (int i = 0; i < len; i++)
    {
        p *= x[i];
    }
    return p;
}

int main()
{
    int a[] = {1, 2, 3};
    std::cout << f(a, 3) << std::endl;
    return 0;
}
```

Exercise. After declaring a reference can the reference change the variable it's referring to? Does = work?

```
int x = 1;
int y = 2;
int & z = x;        // make z refer to x
z = 0;
std::cout << x << ' ' << y << ' ' << z << std::endl;
z = y;              // make z refer to y ... can we?
z = -1;
std::cout << x << ' ' << y << ' ' << z << std::endl;
```

Here the rule: **a reference variable cannot refer to another variable after it's initialization.**

Return value or pass-by-reference???

Now we have a dilemma ...

The following is a simple `square()` function:

```
double square(double x)
{
    return x * x;
}


int main()
{
    std::cout << square(3.14) << std::endl;
    return 0;
}
```

But you know that you can also do this:

```
void square(double x, double & theSquare)
{
    theSquare = x * x;
}


int main()
{
    double theSquare = 0.0;
    square(3.14, theSquare);
    std::cout << theSquare << std::endl;
    return 0;
}
```

Make sure you run it.

Which should you use??? (Don't you hate making decisions ...)

In general, if your function is like a mathematical function that computes a value, then you should use a return value. (Principle of least surprise.)

For instance look at the above example. Notice how clumsy is the usage of the `square()` function.

If you have a function that computes two values, then you might want to have to have two pass-by-reference variables. Of course you can have two different functions. Here's an example:

Here's functions:

```
double max(int x[], int len)
{
    double m = x[0];
    for (int i = 1; i < len; i++)
    {
        if (m < x[i]) m = x[i];
```

```
    }
    return m;
}


double min(int x[], int len)
{
    double m = x[0];
    for (int i = 1; i < len; i++)
    {
        if (m > x[i]) m = x[i];
    }
    return m;
}
```

You can combine them into one:

```
void max_min(int x[], int len,
             int & max, int & min)
{
    double max = x[0];
    double min = x[0];
    for (int i = 1; i < len; i++)
    {
        if (max < x[i]) max = x[i];
        if (min > x[i]) min = x[i];
    }
}
```

If your program(s) always have to compute both the maximum and minimum, then the second version might be more useful. In fact it's faster than calling both `max()` and `min()` which requires **two** function calls and also **two** scans of the array.

However this means that one fine day, if you really need to compute only the minimum of an array, you have to call this function and you have to create a dummy variable for the max which is not going to be used:

```
double x, y;
max_min(a, 10, x, y); // a is some array of length 10
... only x is used ...
```

Or you have to write a `min()` function.

A basic principle of good software design is to design each function to perform one task and not two.

Now if you look at our `swap()` function

```
void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}
```

It does not perform a numerical computation; there is really no return

value. It performs a transformation on variables. That's why the design of this function is reasonable.

Prototypes for references

How do you write function prototypes for pass-by-reference parameters? Easy! It's the same as your other variables. For instance if you have this function:

```
void f(int a, double & b, char & c, const bool & b)
{
    ... code ...
}
```

The function prototype is just

```
void f(int, double &, char &, const bool &);
```

That's all there is to it. So there's nothing new here.

Exercise. Given this code

```
#include <iostream>

int f(int & a, double & b)
{
    a++;
    b += a;
    return a * b
}

int main()
{
    int x = 42;
    double y = 1;
    int z = f(x, y);
    std::cout << x << ' ' << y << ' ' << z
              << std::endl;
    return 0;
}
```

Rewrite it into this form:

```
#include <iostream>

// prototype of f() here

int main()
{
    int x = 42;
    double y = 1;
    int z = f(x, y);
    std::cout << x << ' ' << y << ' ' << z
              << std::endl;
    return 0;
}

int f(int & a, double & b)
{
    a++;
```

```
    b += a;
    return a * b
}
```

Run your program to make sure that you did it correctly.

Exercise. Here's the `swap()` example again.

```
void swap(int & b, int & c)
{
    int t = b;
    b = c;
    c = t;
}

int main()
{
    int x = 42;
    int y = 1;
    swap(x, y);
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```

Write a header file for the `swap()` function. Implement the body of the `swap()` function in a separate source file. Test your program.

Exercise. One of the parameters in `f()` need not be a reference. Which one?

```
int f(int & a, double & b)
{
    b += a;
    return a * b
}
```