4.2 Assignment vs Initialization

Reference: Lippman and Lajoie, sec 4.4

- In both C and C++, initialization and assignment are different concepts (but easily confused).
  - However, in C++, the distinction between them is far more important than in C.
    - ```
      void f()
      {
        int x = 1;      /* Initialization. */
        int y = x;      /* Initialization. */
        y = 2;          /* Assignment. */
        x = y;          /* Assignment. */
        y = g(x);       /* Assignment (y = temp; temp is    */
      }                 /*  a temporary on the stack frame of */
                        /*  f(). See return statement below.  */

      int g( int a)     /* Initialization (int a = x). */
      {
         int t = a+1;   /* Initialization. */
         return 2*t;    /* Initialization (int temp =2* t,  */
      }                 /*  where temp is as above.         */
      ```


- Two special cases of initialization in C and C++:

  i) *When a function is called:* Each formal parameter is *initialized* to the value of the corresponding argument.

  ii) *When a function returns:* A temporary location on the caller's stack frame is *initialized* to the value of the return expression.

- The distinction between initialization and assignment:

Initialization:	A newly created object is first given a value. There is no previous value to overwrite (destroy).

Assignment:	An existing object is given a different value (copied from another object). The previous value of the existing object is overwritten (destroyed).

- Even in C, the distinction between initialization and assignment is present.
  - Variables declared `const`
    - must be initialized, but
    - cannot be assigned to.
    - ```
      const int m = 5;      /* Initialization required. */
      m = 7;                /* Illegal. */
      ```
  - Arrays can be initialized, but not assigned to.
    - ```
      double a[3] = {1,3,2}; /* Initialization optional. */
      double b[3];
      b = a;                 /* Illegal.*/
      ```
- However, a C programmer might get by without really understanding the distinction.
  - In C++, this is not possible.

- In C++,

  i) [*initialization*] When the compiler needs to initialize a newly-created object of type `T` (a class or struct type), it invokes a constructor of `T`.

  - [*copy initialization*] If the newly-created object is to be initialized to a copy of an existing object of type `T`, the compiler invokes the copy constructor of `T`:

    ```
    T( const T &x);      // Initialize *this to a copy of x.
    ```

  ii) [*assignment*] When the compiler needs to assign a new value to an existing object of type `T`, it invokes the assignment operator of `T`.

    ```
    T &operator=( const T &x);     // *this = copy of x
    ```
    ```

- For example,

```
DblStack s, u;
s.push(5);
s.push(2);
u.push(8);
DblStack t = s;      // Compiler invokes DblStack(s)
                     //   with this = &t.
u = t;               //Compiler invokes u.operator=(t).
```

- In general, the programmer needs to write the copy constructor and (overloaded) assignment operator.
  - Typically, they will have much in common, but will not be identical.
  - *The main difference:* The assignment operator must take account of the fact that the object already has a value.
    - This value may need to be destroyed, before the new value can be assigned.

    - In particular, if the old value requires reusable resources (e.g., dynamic memory, files), then these may need to be freed, and reallocated.

- For example, for class `DblStack`:

```
// Copy constructor for class DblStack.  Initializes *this
//  to a copy of s.
DblStack( const DblStack &s) {
   height = s.height;
   allocSize = s.allocSize;
   item = new double[allocSize];
   for ( int i = 0 ; i < height ; ++i )
      item[i] = s.item[i];
}
```

```
// A nearly correct overloaded assignment operator for
//  class DblStack.  Performs the assignment *this = s.
void operator=( const DblStack &s) {
   delete[] item;
   height = s.height;
   allocSize = s.allocSize;
   item = new double[allocSize];
   for ( int i = 0 ; i < height ; ++i )
      item[i] = s.item[i];
}
```


- The only difference:
  - ◇ With initialization, there was <u>no old dynamic memory to free</u>.
    - `*this` was a newly created stack.
    - It had no previous value.
  - ◇ With assignment, the dynamic memory allocated for the old value of `*this` had to be freed, before we could allocate the memory for the new value.
    - Note: If

```
this->allocSize <= s.height
```

then the assignment operator could reuse the old dynamic memory, instead of freeing it and then allocating new memory.

- Actually, there are two problems with our overloaded assignment operator.

a) An assignment of the form

```
x = x;
```

doesn't work, and might even crash the program.

b) A multiple assignment, such as

```
s = t = u;
```

doesn't work.

- Note by right associativity of assignment, this means

```
s = (t = u);
     ‾‾‾‾‾
     void
```

- We can fix problem(a) by checking whether `this == &s`.
  - ◇ If so, the assignment operator does nothing.

- ◇ Note: We must check `this == &s`, not `*this == s`. *Why?*

- ➤ We can fix problem (b) by having the assignment operator return `*this`, rather than `void`.
  - ◇ Actually, the assignment operator will return `*this` *by reference* (discussed later).

- Corrected assignment operator for `DblStack`:

```
// Corrected overloaded assignment operator for
// class DblStack.  Performs the assignment *this = s.
DblStack &operator=( const DblStack &s) {
   if ( this != &s ) {
      delete[] item;
      height = s.height;
      allocSize = s.allocSize;
      item = new double[allocSize];
      for ( int i = 0 ; i < height ; ++i )
         item[i] = s.item[i];
   }
   return *this;
}
```


- ▸ What happens if the programmer doesn't supply a copy constructor (or assignment operator) for a class?
  - The compiler creates one implicitly.
  - The implicitly-created copy constructor for `DblStack` would look like this:

```
DblStack( const DblStack &s) {
   height = s.height;
   allocSize = s.allocSize;
   item = s.item;          // Not correct.
}
```

  - ➤ We have already seen that this can produce invalid stacks.
  - ➤ It provides neither value nor reference semantics.

  - In general, a data member is initialized
    - ➤ by bitwise copy, if it is not of class type,
    - ➤ by the copy constructor of the data member's class, if it is of class type.

Exercise 1: Distinguish initializations from assignments below.

```
int m = 5, n = 8;
double a = 12.7, b = 4.2;
double ave = (a+b) / 2,
       max = (a>b) ? a : b;
double r = (n = ave + 0.5) +
           (m = 1);
double c = b;
a = c;
if ( (r = a-b) > c ) {
   int c = a;
   a = b;
   b = c;
}
for ( int i=1; i<10; ++i )
   printf("%d\n", h(i+1));

.
.
.
int h( int k) { return k*k;}
```

m ___ n ___
a ___ b ___
ave ___
max ___
r ___ n ___ m ___

c ___
a ___
r ___
c ___
a ___
b ___

i ___ *(C99 / C++ only)*
k ___ *(parameter of h())*

temporary on caller's stack frame ___

Exercise 2: Consider the code

```
double g( double x, double y)
{
   return (x + y) / 2;
}
 .
 .
 .
double a=1, b=2, c=3, d=4, e;
e = g( a+b+c, d) + 7;
```

What initializations occur when the last statement (shaded) is executed?

Exercise 3: Why must we check `this == &s`, not `*this == s`, in the overloaded assignment operator of `DblStack`?

Exercise 4: Rewrite the overloaded assignment operator of `DblStack`, so that the old item array is freed and reallocated only when necessary?