CSCI 3110
Parameter types, Function return types

1. Reference Types:

An object may be defined to a reference type by using the & operator:

```
 int x=3;
 int & y = x;
```

After the declaration above, y is referencing x, ie x and y are two different names for the same integer variable. If we make the assignment

```
  y=5;
```

we will not only be setting the value of y to 5, but also setting x to 5.

If we write a swap function

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

and we call this function using

```
        swap(c, d);  // c=3, d=6
```

the function call would have the effect of defining the references in the argument like this:

```
        int & a=c;
        int & b=d;
```

Then the function would modify the local arguments a and b. The result would also be to modify their aliases c and d.

2. Parameter Passing

Suppose we have defined a struct employee as follows:

```
        struct Employee
        {
                string  firstName;
                string  lastName;
                int       yearWorked;
                float     salary;
        };
```

Also suppose we want to pass the employee structure to a print routine. One possibility would be to use call-by-value:

```
void PrintInfo( Employee   theEmployee)
```

This would cause a copy of the actual argument to be made into the formal argument for every call to PrintInfo. Because employee can be quite a large structure, e.g., containing a lot of information about the employee, this copying operation is expensive.

An alternative is to pass the parameter by reference

```
void PrintInfo(Employee   &theEmployee)
```

This avoids the overhead of a copy. However, this declaration tells the reader, and also the compiler, that the actual argument might be changed as a result of the call to PrintInfo. When the parameter was passed by value, we were guaranteed that the actual parameter would not be altered.

There is a third form of parameter passing, the constant reference:

```
void PrintInfo(const Employee & theEmployee)
```

The constant reference parameter guarantees that

a) the overhead of a copy is avoided
b) the actual parameter is unchanged by the function call.

On the other hand, if we want the value of the actual parameter to be modified as the result of the function call, we need to use reference parameter, for example:

```
void ReadRecord(ifstream &inputFile, Employee &theEmployee)
```

Selecting the proper parameter passing mechanism is important for efficiency, readability, and program maintenance. The guidelines for the selection:

a) Call by value is appropriate for small objects that should not be altered by the function
b) Call by constant reference is appropriate for large objects that should not be altered by the function
c) Call by reference is appropriate for objects that may be altered by the function.

3. Return Types

There are three return types, namely return by values, by reference, or by constant reference. **Return by value** will call a copy constructor or create a temporary copy of the value to be returned. As is the case of parameters, this can be expensive.

Return by **reference** and return by **constant reference** are the other choices. If return by reference is used, a reference of the return value is returned. The function call can appear as the target of an assignment. For example, (A=B)=C becomes legal (but not meaningful).

When return by reference is used, we say that an l-value is returned. An l-value is an object that can be the target of an assignment operation in C++.

When a function's return value is declared as a reference, an l-value not local to the function must be returned. Consider the function:

```
int & Larger (int & x, int & y)
{
        return (x>y?x:y);
}
```

it returns a reference to its maximum argument allowing the following use:

```
int a=8, b=9;
Larger(a, b) = 15;
cout << a  << "  " << b << endl;

Larger(a, b) -= 10;
cout << a << "  "  << b << endl;

Larger(a, b) ++;
```

If **constant reference** is used as the return type, an r-value is returned. Thus, if the function header above were

```
const int & Larger (int & x, int & y),
```

then the return value could not be the target of an assignment making all three function calls above illegal.

Notes

1. local variable may not be returned by reference – why? Think how stack memory works in function calls
2. value parameter may not be returned by reference – same reason as above
3. when a const variable is returned by reference, it must be returned as a const reference
   ```
   const int & Larger(const int &x, int &y) { … return x;}
   ```
and not just reference return, e.g.,

```
int & Larger(const int &x, int &y)
```

What about cases when object types are involved? (as parameter and as return value)