Functions With References and Pointers
Reference Parameters

- By default, you **copy** the values from argument to parameter
- You can change that
  - If you declare the type of the parameter to be a reference, then the arg and the param refer to the same value
  - A change to the function parameter changes the invoker’s argument
Example 7.1: Swap with references

```c
void swap (int & first, int & second) {
    // a reference is an alias
    int temp = first;
    first = second;
    second = temp;
}
```

Change the reference parameters and you change the corresponding invoker arguments
Compare defs

void swap (int & first, int & second) {
    int temp = first;
    first = second;
    second = temp;
}

void swap (int * first, int * second) {
    int temp = *first;
    *first = *second;
    *second = temp;
}
Example 7.2: Pointer parameters

- You can do the same thing by passing pointers to the original argument
  - Through the pointer you can change the argument
  - You can set them as const as well (pass a kind of copy, no changes)
Which of the following parameters copy their argument?

- a
- b
- c
- I don't know
Process

void function (int& param1, long& param2)

Invocation (arg1, arg2)

arg and param refer to the same value

If the return type is void, then return is illegal

Do stuff

... return result
Invocations

```c
int main() {
    // call with refs
    long one = 100, two = 200;
    swap(one, two);
}

int main() {
    // call with ptrs
    long one = 100, two = 200;
    swap(&one, &two);
}
```
Best of both worlds

- If you want to pass args-to-params by reference (to avoid copying) but do not want to allow the function to change such parameters, make them `const`.
  - You can add `const` to a ref parameter and in so doing make that “gate” a constant.
  - Cannot change the underlying value through it.
Example 7.3
Overloaded Functions
Default Args

- Example 7.4
Setting Defaults

- You may set the default values for a parameter
  - If the parameter is not provided, the default is used
  - If the parameter is provided, the provided value is used
Order Dependency

- There is an order dependency here.
  - You must have all the required parameters (those without defaults) before any default argument parameters
  - You cannot mix and match
  - You cannot call out by name (in the invoker) which parameter you set
    - This is different from Python
  - Everything must be done in order
Overloaded Functions

- Function = Name + Param Types
An overloaded function is a function that
- Has one name
- Represents different operations depending on its parameter types
- C++ supports function overloading
- We’ve seen this before
Name Mangling

- Real process, how the compiler creates a unique name based on the function name and its associated types
  - Mangled name allows for look up of the correct function
  - [http://demangler.com](http://demangler.com)
Function signature

- Function signature consists of
  - Function name
  - The types, and their order, of the parameters
- Names of the parameters do not matter!
- Uniquely identifies (or should) a function
Two different functions!

```cpp
void swap(double & d1, double & d2) {
    cout << "This must be the double swap" << endl;
    double temp = d1;
    d1 = d2;
    d1 = d2;
    d2 = temp;
}

void swap(int & i1, int & i2) {
    cout << "This must be the int swap" << endl;
    int temp = i1;
    i1 = i2;
    i1 = i2;
    i2 = temp;
}
```
Resolving can be complicated

- Section 6.6 of the book goes through the “rules” for deciding, which, if any, function is appropriate for a set of arguments
- The problem is basically conversion. What happens if a conversion is available that might convert one type to another?
Example 7.6
Easier to have happen then you think

- This seems like a bad place to end up, but because code can be written in pieces by different people, conversion might creep in that allow for this kind of problem.

- Beware!
A word on const

- Trying to differentiate parameter types based on top-level const does not work. These are the same functions!

```cpp
int my_fun(const int p1) {
    cout << "const fn" << endl;
}

int my_fun(int p1) {
    cout << "reg fn" << endl;
}

int main() {
    const int c_int = 1;
    int my_int = 2;
    my_fun(c_int);
    my_fun(my_int);
}
```
Overloading, double-edged sword

- Nice to be able to overload a function based on types
- Can be a pain when some function (very general) requires that I rewrite it for every type, especially for any new one I create
Templates

- Making a pattern of a function for multiple types
- Example 7.7
The way to get around the problem of making a function for each type is called a **template**.

A template is a **pattern** that can be used to **create a function** with whatever types we want.

**A template is not a function**, it is how to create a function with some type information set.
Basis of everything in the STL

- While pointers are a basis for a lot of how C (the underlying language) works, templates are the basis for C++/STL and how it really solves many problems of generality with types
template<typename my_type>
void swap(my_type& first, my_type& second) {
    my_type temp = first;
    first = second;
    second = temp;
}

2) Substitute int for my_type to create the function

1) Look for swap with two ints
   int i=1, j=2;
   swap(i, j);

3) Call new function
   void swap(int& first, int& second) {
       int temp = first;
       first = second;
       second = temp;
   }
Generic Function

- By writing the function as a template, we can write a **generic function**
  - A function which, even in C++ (which is type crazy) is generic **for all types**
- Remember: a template is a pattern to make a function. It is not a function
Force the type

- Typically the compiler deduces the type for substitution in the template from the provided arguments.
- You can force the type used (though you must be careful), but it has to work with the args and the created function.
Example 7.8, force the template Type

- Invocation

```cpp
double result;
long i = 1, j = 2;
result = swap<double>(i, j);
```

Template type directly indicated

- We will see this again and again. We specify in the invocation the type we want used in the template.
Trailing return type and auto

- If you want to use an auto for a return type, especially in a template, you use a trailing return type

  ```cpp
  auto my_fun(int x, int y) -> decltype(x + y) if my_fun's return type should be the type returned by adding x and y
  ```