Default Parameters

The material in this handout is collected from the following reference:

- Section $6.5.1$ of the text book C++ Primer.

What is a parameter? What is an argument?

All the different terms that have to do with parameters and arguments can be confusing. However, if you keep a few simple points in mind, you will be able to easily handle these terms.

- The *parameters* for a function are listed in the function declaration and are used in the body of the function definition. A parameter (of any type) is a placeholder that is filled in with something when the function is called.
- An *argument* is something that is used to fill in a parameter. When you write down a function call, the arguments are listed in parentheses after the function name. When the function call is executed, the arguments are plugged in for the formal parameters.
- The terms *call-by-value* and *call-by-reference* refer to the mechanism that is used in the plugging-in process.
- In the call-by-value method, the argument is an expression that is evaluated and the result of this evaluation is used to initialize the corresponding parameter which is a local or internal variable of the function definition so that any change made to the parameter has nothing to do with the argument.
- In the call-by-reference mechanism, the argument must evaluate to an lvalue, the corresponding parameter must be a lvalue reference type, and is initialized as a lvalue reference to the argument so that any change made to the parameter is actually made to the argument variable.

Introduction to default parameters

Some functions have parameters that are given a particular value in most, but not all, calls. In such cases, the common value(s) can be declared as *default parameter(s)* for the functions. Functions with default parameters can be called with or without the corresponding arguments. The primary use of default parameters is to extend the parameter list of existing functions without requiring any change in the function calls already used in a program.

Let's work our way using a simple example to motivate the discussion on default parameters. Consider the following function to write array contents to standard output:

```cpp
void print_array1(int const a[], int size) {
  for (int i {0}; i < size; i++) {
    std::cout << a[i];
    if (i < size - 1) {
      std::cout << ", ";
    }
  }
  std::cout << "\n";
}
```

A call to this function looks like this:

```cpp
int a[] {4, 5, 3, 9, 5, 2, 7, 6};
print_array1(a, sizeof(a) / sizeof(*a));
```

and will write the following values to standard output:

```
4, 5, 3, 9, 5, 2, 7, 6
```

To occasionally print each value on a newline, a new function `print_array2` is defined:

```cpp
void print_array2(int const a[], int size) {
  for (int i {0}; i < size; i++) {
    std::cout << a[i] << "\n";
  }
}
```

A call to function `print_array2` looks like this:

```cpp
int a[] {4, 5, 3, 9, 5, 2, 7, 6};
print_array2(a, sizeof(a) / sizeof(*a));
```

and will write the following values to standard output:

```
4
5
3
9
5
2
7
6
```

Functions `print_array1` and `print_array2` implement similar algorithms but with minor variations. It would be convenient for programmers if the common algorithm was implemented in a single function that can be called in more than one way. In most cases, the programmer wishes to write values on a single line and will call the function so that this default action is implemented. In a small number of cases, the programmer wishes to write values on multiple lines and will call the function with an additional argument. This is exactly the situation that default parameters were designed for. The previous two functions `print_array1` and `print_array2` are combined into a single function `print_array` with the two parameters augmented with a default parameter:

```cpp
void print_array(int const a[], int size, bool newlines = false) {
  for (int i {0}; i < size; i++) {
    std::cout << a[i];
    if (i < size - 1) {
      std::cout << (newlines ? "\n" : ", ");
    }
  }
  std::cout << "\n";
}
```

Calling function `print_array` in the following code fragment with only two arguments implies that the missing third argument is `false`:

```
int a[] {4, 5, 3, 9, 5, 2, 7, 6};
int size { sizeof(a)/sizeof(a[0]) };
print_array(a, size);
```

with program output looking like this:

```
4, 5, 3, 9, 5, 2, 7, 6
```

The programmer could have explicitly specified the third argument as `false`:

```
print_array(a, size, false);
```

but it would defeat the purpose of declaring functions to have default parameters. The program output would remain the same:

```
4, 5, 3, 9, 5, 2, 7, 6
```

Only when the programmer occasionally wishes to print each value on a newline, does she specify the third argument explicitly as `true`:

```
print_array(a, size, true);
```

In this case, the program output would be:

```
4
5
3
9
5
2
7
6

```

Default function declaration

Normally, functions are declared in header files and defined in source files. In that case, default parameters must only be specified in the function declaration but not in the function definition. So, the function `print_array` from the previous section is declared in a header file `misc.h` like this:

```
void print_array(int const a[], int size, bool newlines = false);
```

The following declaration is equivalent except that we're not specifying parameter names:

```
void print_array(int const [], int, bool = false);
```

The definition of function `print_array` in a source file `misc.cpp` must not specify the default parameter value and would look like this:

```cpp
void print_array(int const a[], int size, bool newlines) {
  for (int i {0}; i < size; i++) {
    std::cout << a[i];
    if (i < size - 1) {
      std::cout << (newlines ? "\n" : ", ");
    }
  }
  std::cout << "\n";
}
```

> ***Normally, functions are declared in header files and defined in source files. In that case, default parameters must only be specified in the function declaration but not in the function definition.***

Second example

Consider the following function `incr` that adds parameter `amount` to a variable referenced by reference parameter `value` and returns parameter `value` by reference:

```cpp
int& incr(int &value, int amount) {
  value += amount;
  return value;
}
```

The following code fragment illustrates calls to `incr`:

```cpp
int main() {
  int i {10};
  std::cout << incr(i, 1) << "\n";
  std::cout << incr(i, 1) << "\n";
  std::cout << incr(i, 2) << "\n";
  std::cout << incr(i, 4) << "\n";
  std::cout << incr(i, 5) << "\n";
}
```

with program output looking like this:

```
11
12
14
18
23

```

Since `incr` is called often with a second argument `1`, we rewrite function `incr` with parameter `amount` having default value `1`:

```
int& incr(int &value, int amount = 1) {
  value += amount;
  return value;
}
```

The following code fragment illustrates calls to `incr`:

```
int main() {
  int i {10};
  std::cout << incr(i) << "\n";
  std::cout << incr(i) << "\n";
  std::cout << incr(i, 2) << "\n";
  std::cout << incr(i, 4) << "\n";
  std::cout << incr(i, 5) << "\n";
}
```

with program output looking like this:

```
11
12
14
18
23

```

Multiple default parameters

You can have multiple default parameters and they must *default right to left*. The following declarations are legal:

```
void foo(int a, int b, int c = 10);
void bar(int a, int b = 8, int c = 10);
void baz(int a = 5, int b = 8, int c = 10);
```

Calls to these functions are shown below with arguments involved in the function calls listed in the comments:

```
foo(1, 2);    // foo(1, 2, 10)
foo(1, 2, 9); // foo(1, 2, 9)

bar(1);       // bar(1, 8, 10)
bar(1, 2);    // bar(1, 2, 10)
bar(1, 2, 9); // bar(1, 2, 9)

baz();        // baz(5, 8, 10)
baz(1);       // baz(1, 8, 10)
baz(1, 2);    // baz(1, 2, 10)
baz(1, 2, 9); // baz(1, 2, 9)
```

The following two function declarations are illegal because default parameters are not ordered from right to left:

```cpp
void qux(int a, int b = 8, int c);
void quux(int a = 5, int b, int c = 10);
```

Consider the following function declaration with seven parameters of which four have default values:

```cpp
void foo(int x, int y, double t, char z = 'A',
         int u = 67, char v = 'G', double w = 78.34);
```

Parameters `z`, `u`, `v`, and `w` of function `foo` have default values. If no values are specified for `z`, `u`, `v`, and `w` in a call to function `foo`, their default values are used. Consider the following definitions:

```cpp
int    a {111}, b {222};
char   ch {'m'};
double d {123.45};
```

The following function calls are legal:

1.
   ```cpp
   foo(a, b, d);
   ```
   Here, the default values of parameters `z`, `u`, `v`, and `w` are used.
2.
   ```cpp
   foo(a, 15, 34.6, 'B', 87, ch);
   ```
   Here, the default value of `z` is replaced by `'B'`, the default value of `u` is replaced by `87`, the default value of `v` is replaced by the value of `ch`, and the default value of `w` is used.
3.
   ```cpp
   foo(b, a, 14.56, 'D');
   ```
   The default value of `z` is replaced by `'D'`, and the default values of `u`, `v`, and `w` are used.

The following function calls are not illegal but may not behave as intended by the programmer:

1.
   ```cpp
   void foo(int x, int y, double t, char z = 'A',
            int u = 67, char v = 'G', double w = 78.34);
   foo(a, 15, 34.6, 46.7);
   ```
   The programmer wishes to provide default values to parameters `z`, `u`, and `v` with parameter `w` initialized with value `46.7`. However, the compiler strictly matches arguments to parameters from left-to-right. Therefore, parameter `x` is initialized with expression `a`, parameter `y` is initialized with `15`, parameter `t` is initialized with `34.6`, parameter `z` is initialized by implicitly converting argument `46.7` to type `char`, with the remaining parameters `u`, `v`, and `w` initialized with default values.
2.
   ```cpp
   foo(b, 25, 48.76, 'D', 4567, 99.99);
   ```
   The programmer intended to provide explicit values to all the parameters except `v` and for parameter `w` to be initialized with value `99.99` . Since expression `99.99` can be implicitly converted to a `char` value, the compiler will initializer parameter `v` with value `static_cast<char>(99.99)` and then initialize parameter `w` with default value `78.34`.

The following function declarations are illegal because default parameters are not ordered from right to left:

```cpp
void bar(int x, double z = 23.45, char ch, int u = 45);
int baz(int length = 1, int width, int height = 1);
void qux(int x, int& y = 16, double z = 34);
```

Because second parameter `z` in function declaration `boo` is a default parameter, all other parameters after `z` *must be* default parameters. In the declaration of function `baz`, because the first parameter is a default parameter, all parameters must be default parameters. In declaration of `qux`, a constant value cannot be assigned to `y` because `y` is a reference parameter.

Extending parameter list

The primary use of default parameters is to extend the parameter list of existing functions without requiring any change in the function calls already used in a program. Let's assume we've a function that draws circles declared in `circle.h`:

```cpp
void draw_circle(int x, int y, float radius);
```

The function definition in `circle.cpp` render these circles all black:

```cpp
void draw_circle(int x, int y, float radius) {
  // render black circle
}
```

Our clients might use this function in their applications, say in `client.cpp`:

```cpp
draw_circle(100, 100, 30.5);
```

Later, we add a color by modifying the declaration in `circle.h`:

```cpp
void draw_circle(int x, int y, float radius, color c = black);
```

Thanks to the default parameter, clients don't need to refactor their application since the calls to `draw_circle` with three arguments still work by rendering their circles black. They could draw new circles with other colors, if they so wish:

```cpp
// in client.cpp

draw_circle(100, 100, 30.5);     // the old circle rendered in black
draw_circle(200, 200, 50, red); // new circle rendered in red
```

Summary

When a function is called, the number of arguments in the function call must match the number of parameters in the function declaration. C++ relaxes this condition for functions with default parameters. You specify the value of a default parameter when the function name appears for the first time, such as in the function declaration. In general, four rules must be followed when using default arguments.

1. Default values should be assigned in the function prototype.

2. If any parameter is given a default value in the function declaration, all parameters following it must also be supplied with default values.
3. If one argument is omitted in the function call, all arguments to its right must also be omitted. The second and third rules make it clear to the C++ compiler which arguments are being omitted and permit the compiler to supply correct default parameter values for the missing arguments, starting with the rightmost argument and working in toward the left.
4. The default value used in the function declaration can be an expression consisting of both constants, global variables, or function calls. The default value assignment must pass the compiler's check, even though the expression's actual value is evaluated and assigned at runtime. For example, the compiler will flag an error if an rvalue is specified as the default value to a reference parameter.