C++ Functions

Python	C++

def function_name(*var1, var2, ...*)
   statement
   statement
   …more statements…
	type function_name(*type var1, type var2, ...*)
	{
	   statement
	   statement
	   …more statements…
	}

Comparison:

- C++ forces the programmer to declare the *data type* of each parameter to a function, along with the data type of the return value.
    - If the function takes no arguments, the parentheses can be left empty (like Python).
    - If the function does not return anything, you must use the return type of **`void`**.
- The **`return`** keyword in C++ works just like in Python. When returning a value from a C++ function, you must make sure the value being returned has the same data type as the declared return type.

Function prototypes

The C++ compiler will check that every time you call a function in your code, the function being called has been defined correctly. However, because the compiler reads your source code from top to bottom, if a function call earlier in your code references a function that is defined later, the compiler will give you an error message about an undefined function. To fix this problem, use a function prototype:

```
int f(int a, int b);      // function prototype line

int main() {
  f(3, 6);                // without the above prototype, the compiler would flag this
                          // line as an error, saying the function f is not defined.
}

int f(int a, int b) {
  cout << “This is function f.  The sum of my arguments is ” << a + b << endl;
}
```

A function prototype gives C++ enough information about the function --- its name, parameter types, and return type --- for the C++ compiler to do its type-checking as it is compiling your program.

Function overloading

Because C++ forces you to declare the types of your parameters to the functions you write, C++ allows you to define multiple functions with the same name, but different numbers of parameters or different parameter types. C++ will figure out which one to call based on the arguments that are passed to the function when it is called. These three functions can coexist just fine:

```
void g(int x) {
  cout << “In g, x is an integer:” << x << endl;
}

void g(string x) {
  cout << “In g, x is an string:” << x << endl;
}

void g() {
  cout << “In g, no arguments!” << endl;
}
```