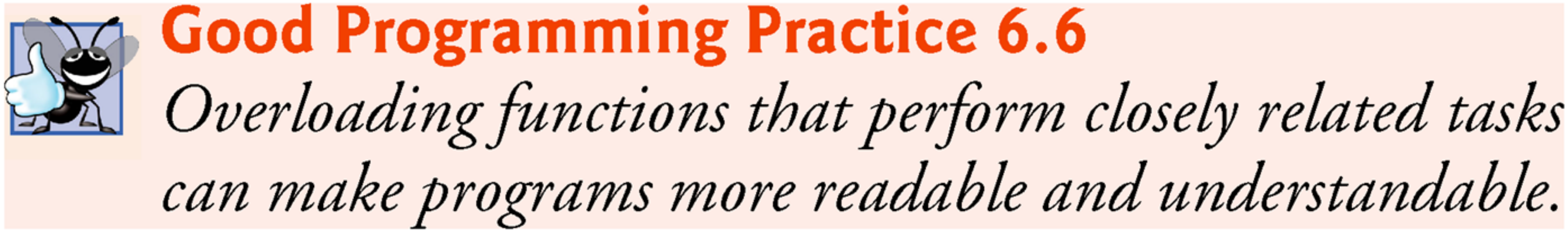
Function Overloading and Template Function

6.16 Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.
- Function overloading is used to create several functions of the *same* name that perform similar tasks, but on different data types.



Good Programming Practice 6.6

Overloading functions that perform closely related tasks can make programs more readable and understandable.

6.16 Function Overloading (cont.)

- Figure 6.20 uses overloaded square functions to calculate the square of an `int` and the square of a `double`.

```cpp
// Fig. 6.20: fig06_20.cpp
// Overloaded square functions.
#include <iostream>
using namespace std;

// function square for int values
int square(int x) {
   cout << "square of integer " << x << " is ";
   return x * x;
}

// function square for double values
double square(double y) {
   cout << "square of double " << y << " is ";
   return y * y;
}

int main() {
   cout << square(7); // calls int version
   cout << endl;
   cout << square(7.5); // calls double version
   cout << endl;
}
```

Fig. 6.20 | Overloaded `square` functions. (Part 1 of 2.)

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 6.20 | Overloaded `square` functions. (Part 2 of 2.)

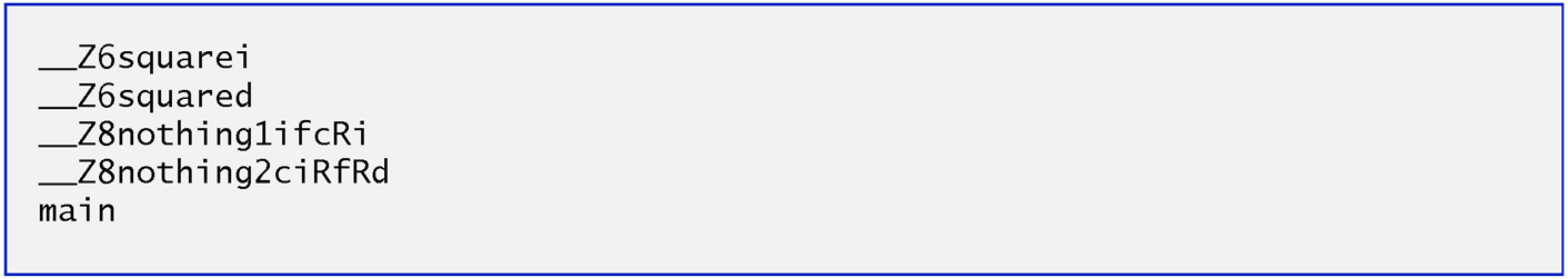
6.16 Function Overloading (cont.)

How the Compiler Differentiates Among Overloaded Functions

- Overloaded functions are distinguished by their signatures.
- A signature is a combination of a function's name and its parameter types (in order).
- The compiler encodes each function identifier with the types of its parameters (sometimes referred to as name mangling or name decoration) to enable type-safe linkage.
  - Ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.
- Figure 6.21 was compiled with GNU C++.
- Rather than showing the execution output of the program, we show the mangled function names produced in assembly language by GNU C++.

```cpp
// Fig. 6.21: fig06_21.cpp
// Name mangling to enable type-safe linkage.

// function square for int values
int square(int x) {
   return x * x;
}

// function square for double values
double square(double y) {
   return y * y;
}

// function that receives arguments of types
// int, float, char and int&
void nothing1(int a, float b, char c, int& d) { }

// function that receives arguments of types
// char, int, float& and double&
int nothing2(char a, int b, float& c, double& d) {
   return 0;
}

int main() { }
```

Fig. 6.21 | Name mangling to enable type-safe linkage. (Part 1 of 2.)



```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
main
```

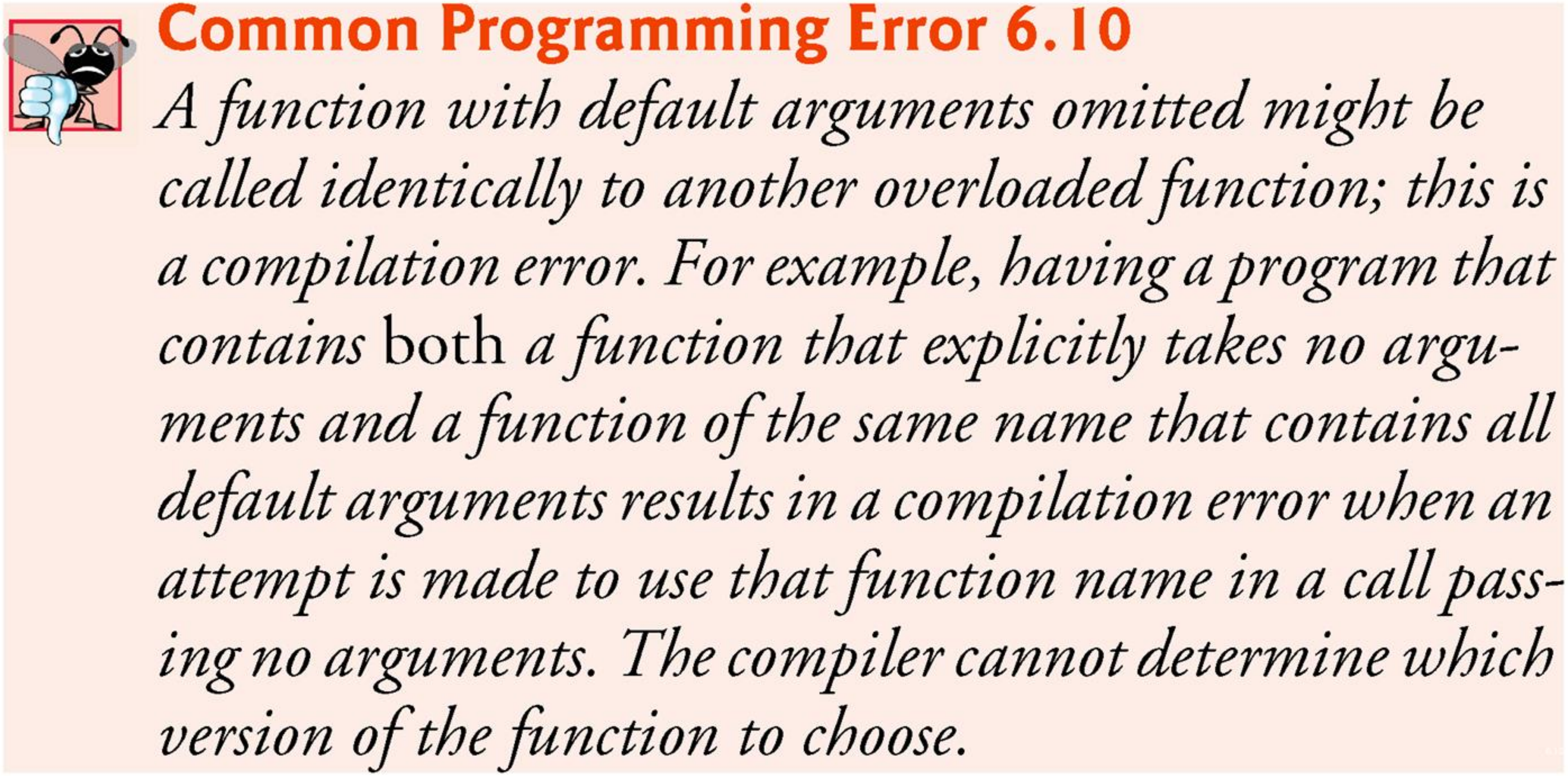
Fig. 6.21 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

6.16 Function Overloading (cont.)

- For GNU C++, each mangled name (other than main) begins with two underscores (__) followed by the letter `Z`, a number and the function name.
  - The number specifies how many characters are in the function's name.
- The compiler distinguishes the two `square` functions by their parameter lists—one specifies `i` for `int` and the other `d` for `double`.
- The return types of the functions are not specified in the mangled names.
- Overloaded functions can have different return types, but if they do, they must also have different parameter lists.
- Function-name mangling is compiler specific.

Common Programming Error 6.9

Creating overloaded functions with identical parameter lists and different return types is a compilation error.



Common Programming Error 6.10

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains both *a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler cannot determine which version of the function to choose.*

6.17 Function Templates

- If the program logic and operations are *identical* for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.

6.17 Function Templates (cont.)

- Figure 6.22 defines a `maximum` function that determines the largest of three values.
- All function template definitions begin with the `template` keyword followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list is preceded by keyword `typename` or keyword `class`.
- The type parameters are placeholders for fundamental types or user-defined types.
    - Used to specify the types of the function's parameters, to specify the function's return type and to declare variables within the body of the function definition.

```
// Fig. 6.22: maximum.h
// Function template maximum header.
template <typename T>  // or template<class T>
T maximum(T value1, T value2, T value3) {
   T maximumValue{value1}; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if (value2 > maximumValue) {
      maximumValue = value2;
   }

   // determine whether value3 is greater than maximumValue
   if (value3 > maximumValue) {
      maximumValue = value3;
   }

   return maximumValue;
}
```

Fig. 6.22 | Function template `maximum` header.

6.19 Function Templates (cont.)

- Figure 6.23 uses the `maximum` function template to determine the largest of three `int` values, three `double` values and three `char` values, respectively.
- Separate functions are created as a result of the calls—expecting three `int` values, three `double` values and three `char` values, respectively.

```cpp
// Fig. 6.23: fig06_23.cpp
// Function template maximum test program.
#include <iostream>
#include "maximum.h" // include definition of function template maximum
using namespace std;

int main() {
   // demonstrate maximum with int values
   cout << "Input three integer values: ";
   int int1, int2, int3;
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
      << maximum(int1, int2, int3);

   // demonstrate maximum with double values
   cout << "\n\nInput three double values: ";
   double double1, double2, double3;
   cin >> double1 >> double2 >> double3;

```

Fig. 6.23 | Function template `maximum` test program. (Part 1 of 2.)

```cpp
   // invoke double version of maximum
   cout << "The maximum double value is: "
      << maximum(double1, double2, double3);

   // demonstrate maximum with char values
   cout << "\n\nInput three characters: ";
   char char1, char2, char3;
   cin >> char1 >> char2 >> char3;

   // invoke char version of maximum
   cout << "The maximum character value is: "
      << maximum(char1, char2, char3) << endl;
}
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 6.23 | Function template `maximum` test program. (Part 2 of 2.)