FunctionTemplates

August 8, 2020

1 Function Overloading and Templates

- http://www.cplusplus.com/doc/oldtutorial/templates/

1.1 Table of Contents

- Section **??**
- Section **??**

1.2 Headers and helper functions

- run include headers and helper function cells if Kernel crashes or is restarted

[11]:
```
// headers and namespace required in this notebook demo codes
#include <iostream>
#include <string>
#include <sstream> //stringstream type
#include <typeinfo> //typeid operator

using namespace std;
```

1.3 Function overloading

- a function works only on the types, it is defined to work on!
- we can define the same function name several times to work on different types; called function overloading
- same function name can be redefined as long as the type and number of parameters differ from each other

[2]:
```
int add(int n1, int n2) {
    int sum = n1 + n2;
    return sum;
}
```

[3]:
```
cout << add(10, 100) << endl;
```

```
110
```

[4]:
```
// what about float, and double?
// by default decimal numbers are stored as double;
// so, add f at the end to tell compiler to treat it as a float
cout << add(10.5f, 100.99f) << endl;
cout << add(100.2, 200.8) << endl;
```

```
input_line_12:5:13: warning: implicit conversion from
'float' to 'int' changes value from 10.5 to 10 [-Wliteral-conversion]
cout << add(10.5f, 100.99f) << endl;
        ~~~ ^~~~~
input_line_12:5:20: warning: implicit conversion
from 'float' to 'int' changes value from 100.99 to 100
      [-Wliteral-conversion]
cout << add(10.5f, 100.99f) << endl;
        ~~~        ^~~~~~
input_line_12:6:13: warning: implicit conversion
from 'double' to 'int' changes value from 100.2 to 100
      [-Wliteral-conversion]
cout << add(100.2, 200.8) << endl;
        ~~~ ^~~~~
input_line_12:6:20: warning: implicit conversion
from 'double' to 'int' changes value from 200.8 to 200
      [-Wliteral-conversion]
cout << add(100.2, 200.8) << endl;
        ~~~        ^~~~~


110
300
```

[5]:
```
cout << add(string("hello"), string("world!"));
```

```
input_line_13:2:10: error: no matching function for
call to 'add'
 cout << add(string("hello"), string("world!"));
         ^~~
input_line_10:1:5: note: candidate function not viable:
no known conversion from 'std::__1::string' (aka
      'basic_string<char, char_traits<char>, allocator<char> >') to 'int' for
1st argument
int add(int n1, int n2) {
```

^

Interpreter Error:

[8]:
```
// overload add function to work on float type
float add(float n1, float n2) {
    float s = n1 + n2;
    return s;
}
```

[9]:
```
cout << add(10.5f, 100.99f) << endl;
```

111.49

[11]:
```
// overload add function to work on double type
double add(double n1, double n2) {
    double s = n1 + n2;
    return s;
}
```

[12]:
```
cout << add(100.2, 200.8) << endl;
```

301

[13]:
```
// overload add function to work on string type
string add(string s1, string s2) {
    string s = s1 + s2;
    return s;
}
```

[15]:
```
cout << add(string("hello"), string("world!"));
```

helloworld!

[9]:
```
// overload add function to work on char types
string add(char c1, char c2) {
    string s = string(1, c1) + string(1, c2);
    return s;
}
```

[10]:
```
cout << add('A', 'B') << endl;
```

AB

[14]:
```
// another implementation to overload add function to work on char types
string addChar(char c1, char c2) {
    stringstream ss;
    ss << c1 << c2;
    string s;
    ss >> s;
    return s;
}
```

[15]:
```
cout << addChar('A', 'B') << endl;
```

```
AB
```

1.4 Function templates

- special function that can operate with generic types
- allows us to create a single function that can be adapted to more than one type or class without repeating entire code for each type as in function overloading
- template parameters allow us to pass parameters types to functions
- syntax:

```
template <class T1, class T2, ...>
T1 functionName(T1 para1, T2 para2, ...) {
    // function body
}
```

[2]:
```
template <class T>
T addition(T para1, T para2) {
    T result = para1 + para2;
    return result;
}
```

[3]:
```
// implicit: types are automatically inferred based on the types of arguments
cout << addition(10, 20) << endl;
cout << addition(10.5f, 20.5f) << endl;
cout << addition(10.5, 20.5) << endl;
cout << addition(string("hi "), string("there!"));
```

```
30
31
31
hi there!
```

[4]:
```
// explicit: types are explicitly passed
cout << addition<int>(10, 20) << endl;
cout << addition<float>(10.5, 20.5) << endl; // notice no f at the end of␣
 ↪numbers
cout << addition<double>(100.5, 200.5) << endl;
```

```
cout << addition<string>(string("hi "), string("there!"));
```

```
30
31
301
hi there!
```

[10]:
```
// expected output AB, but get error!
cout << addition<string>('A', 'B') << endl;
```

```
input_line_20:3:9: error: no template named 'addition';

did you mean 'addition1'?
cout << addition<string>('A', 'B') << endl;
        ^~~~~~~~

        addition1

input_line_10:2:9: note: 'addition1' declared here
returnT addition1(T para1, T para2) {
        ^

input_line_20:3:9: error: no matching function for

call to 'addition1'
cout << addition<string>('A', 'B') << endl;
        ^~~~~~~~~~~~~~~~

input_line_10:2:9: note: candidate template ignored:
couldn't infer template argument 'returnT'
returnT addition1(T para1, T para2) {
        ^



        Interpreter Error:
```

[7]:
```
// typeid operator found in typeinfo header comes to rescue!!
//https://en.cppreference.com/w/cpp/language/typeid
// if we can find type of generic type T, we can write more generic template␣
 ↪function
// that works for addition of char type as well!
cout << typeid(char).name() << endl;
cout << typeid(float).name() << endl;
cout << typeid(double).name() << endl;
cout << typeid(string).name() << endl;
cout << typeid(int).name() << endl;
```

```
c
```

```
f
d
NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE
i
```

[2]:
```cpp
template <class T, class returnT>
returnT addition1(T para1, T para2) {
    returnT result;
    stringstream ss;
    if (typeid(T).name() == typeid(char).name()) {
        ss << para1 << para2;
        ss >> result;
    }
    else
        result = para1 + para2;
    return result;
}
```

[4]:
```cpp
// may help in overflow situations! addition of two large ints could overflow!!
cout << addition1<int, long long>(10, 20) << endl;
cout << addition1<float, double>(10.5, 20.5) << endl; // notice no f at the end␣
 ↪of numbers
cout << addition1<double, long double>(100.5, 200.5) << endl;
cout << addition1<string, string>(string("hi "), string("there!"));
```

```
30
31
301
hi there!
```

[9]:
```cpp
cout << addition1<char, string>('A', 'B') << endl;
```

```
AB
```

[ ]:
```
```