Advanced Programming Concepts with C++

CSI 2372

Tutorial # 9

Selected exercises from chapters 16

uOttawa

Ahmedou Jreivine

Exercise 16.1: Define instantiation.

- Instantiating a template is the process in which
  1. the compiler uses the argument(s) of the call to deduce the template parameter(s),
  2. the compiler uses the deduced template parameter(s) to create a specific version (new "instance") of the template using the actual argument(s) in place of the corresponding template parameter(s).
- Instantiation of a template is the compiler-generated specific version (with the actual argument) of the template.

Exercise 16.2: Write and test your own versions of the `compare` functions

```
#include <iostream>
using std::cout; using std::endl;
#include <vector>
using std::vector;
#include <cstring>
template <typename T>
int compare(const T &v1, const T&v2) {
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
        return 0;
}
template <unsigned N, unsigned M>
        int compare(const char(&p1)[N], const char(&p2)[M]) {
        return std::strcmp(p1, p2);
}
int main() {
        cout << compare(1, 0) << endl;
        vector<int> vec1{ 1, 2, 3 }, vec2{ 4, 5, 6 };
        cout << compare(vec1, vec2) << endl;
        cout << compare("hello", "world!") << endl;
        return 0;
}
```

Output:

```
 1
-1
-1
```

Exercise 16.3: Call your `compare` function on two `Sales_data` objects to see how your compiler handles errors during instantiation.

```
// based on ex14.45 and ex16.2
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <cstring>
template <typename T>
int compare(const T &v1, const T&v2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
/* Sales_data.h */
class Sales_data;
std::istream &read(std::istream &, Sales_data &);
class Sales_data {
    friend Sales_data operator+(const Sales_data &, const Sales_data &);
    friend std::istream &operator>>(std::istream &, Sales_data &);
    friend std::ostream &operator<<(std::ostream &, const Sales_data &);
```

Answer 16.3 con.

```
public:
Sales_data() : Sales_data("", 0, 0.0) {}
explicit Sales_data(const std::string &no) : Sales_data(no, 0, 0.0) {}
Sales_data(const std::string &no, unsigned us, double price): bookNo(no), units_sold(us),
revenue(price * us) {}
explicit Sales_data(std::istream &is) : Sales_data() { read(is, *this); }
Sales_data &operator+=(const Sales_data &);
Sales_data &operator=(const std::string &);
explicit operator std::string() const { return bookNo; }
explicit operator double() const { return revenue; }
std::string isbn() const { return bookNo; }
private:
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
    Sales_data operator+(const Sales_data &, const Sales_data &);
    std::istream &operator>>(std::istream &, Sales_data &);
    std::ostream &operator<<(std::ostream &, const Sales_data &);
    inline
    double Sales_data::avg_price() const {
        return units_sold ? revenue / units_sold : 0;
    }
```

uOttawa

Answer 16.3 con.

```
/* Sales_data.cpp */
bool compareIsbn(const Sales_data &sd1, const Sales_data &sd2) {return sd1.isbn() <
sd2.isbn();}
Sales_data &Sales_data::operator+=(const Sales_data &rhs) {
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
Sales_data operator+(const Sales_data &lhs, const Sales_data &rhs) {
    Sales_data sum = lhs;
    sum += rhs;
    return sum;
}
std::istream &operator>>(std::istream &is, Sales_data &item) {
    double price;
    is >> item.bookNo >> item.units_sold >> price;
    if (is)
    item.revenue = item.units_sold * price;
    else
    item = Sales_data();
    return is;
}
```

uOttawa

Answer 16.3 con.

```
std::ostream &operator<<(std::ostream &os, const Sales_data &item) {
    os << item.isbn() << " " << item.units_sold << " "
    << item.revenue << " " << item.avg_price();
    return os;
}
Sales_data &Sales_data::operator=(const std::string &s) {
    bookNo = s;
    return *this;
}
/* Test.cpp */
int main(int argc, char **argv) {
    Sales_data s1("book1", 4, 2.5);
    Sales_data s2("book1", 10, 1.5);
    std::cout << compare(s1, s2) << std::endl;
    return 0;
}
```

```
error C2678 : binary '<' : no operator found which
takes a left - hand operand of type 'const
Sales_data' (or there is no acceptable conversion)
```

Exercise 16.5:

- **Write a template version of the `print` function from § 6.2.4 (p. 217) that takes a reference to an array and can handle arrays of any size and any element type.**

```
#include <iostream>
template <typename T, size_t N>
void print(const T(&arr)[N]) {
    for (size_t i = 0; i != N; ++i)
    std::cout << arr[i] << std::endl;
}
struct C {
    int i;
    double d;
};
std::ostream &operator<<(std::ostream &os, const C &c) {return os << c.i << " " << c.d; }
int main() {
    int a[] = { 1, 2, 3, 4 };
    print(a);
    char b[] = "abcde";
    print(b);  // will print the trailing `\0`
    C c[] = { 1, 1.1, 2, 2.2, 3, 3.3 };
    print(c);
    return 0;
}
```

Output:

```
1
2
3
4
a
b
c
d
e

1 1.1
2 2.2
3 3.3
```

uOttawa

Exercise 16.6:

- **How do you think the library `begin` and `end` functions that take an array argument work? Define your own versions of these functions.**
- **Answer:**

```
#include <iostream>
template <typename T, size_t N>
    T *begin_arr(T(&arr)[N]) {
    return arr;
}
template <typename T, size_t N>
    T *end_arr(T(&arr)[N]) {
    return arr + N;
}
int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    for (auto it = begin_arr(arr);
    it != end_arr(arr); ++it)
    std::cout << *it << std::endl;
    system("pause");
    return 0;
}
```

uOttawa

Exercise 16.7:

- Write a `constexpr` template that returns the size of a given array.
- **Answer:**

```
#include <iostream>
template <typename T, size_t N>
    constexpr size_t size_arr(T(&arr)[N]) {
    return N;
}

int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    constexpr size_t sz = size_arr(arr);
    std::cout << sz << " " << size_arr(arr) << std::endl;
    return 0;
}
```

uOttawa

Exercise 16.8:

- In the "Key Concept" box on page 108, we noted that as a matter of habit C++ programmers prefer using `!=` to using `<`. Explain the rationale for this habit.

- **Answer:**
  - The operator!= is defined by iterators for all library containers, but the operator< is not.
  - By using operator!= inside template definition, the template can be used by all library containers.

Exercise 16.9:

- What is a function template? What is a class template?

- **Answer:**
  - A function template is a blueprint used by compiler for generating type - specific fuctions.The compiler ordinarily deduces the template parameter(s) using the arguments of the call.

  - A class template is a blueprint used by compiler for generating type - specific classes.The compiler cannot deduce the template parameter(s) for a class template.

Exercise 16.10:

- What happens when a class template is instantiated?

- **Answer:**
  - When a class template is instantiated, the compiler uses explicitly provided template argument(s) generating type-specific version of the class template.

Exercise 16.11:

- The following definition of `List` is incorrect. How would you fix it?

```
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};
```

Answer to Exercise 16.11:

```
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    //void insert(ListItem *ptr, elemType value);  // error: should specify type
    void insert(ListItem<elemType> *ptr, elemType value);
private:
    //ListItem *front, *end;  // error: should specify type
    ListItem<elemType> *front, *end;
};
int main() {
    List<int> l;
    return 0;
}
```

Exercise 16.14-15:

- **16.14:** Write a Screen class template that uses nontype parameters to define the height and width of the Screen.
- **16.15:** Implement input and output operators for your Screen template. Which, if any, friends are necessary in class Screen to make the input and output operators work? Explain why each friend declaration, if any, was needed.

Answer: 16.14-15

```
#include <string>
#include <iostream>
template<unsigned H, unsigned W>
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // needed because Screen has another constructor
                        // cursor initialized to 0 by its in-class initializer
    Screen(char c) :contents(H * W, c) { }
    char get() const{// get the character at the cursor
        return contents[cursor];
    }      // implicitly inline
    Screen &move(pos r, pos c);     // can be made inline later
    friend std::ostream & operator<< (std::ostream &os, const Screen<H, W> & c){
        unsigned int i, j;
        for (i = 0; i<c.height; i++){
            os << c.contents.substr(0, W) << std::endl;
        }
        return os;
    }
```

Answer: 16.14-15

```
    friend std::istream & operator>> (std::istream &is, Screen &  c){
    char a;
    is >> a;
    std::string temp(H*W, a);
    c.contents = temp;
    return is;
    }
private:
    pos cursor = 0;
    pos height = H, width = W;
    std::string contents;
};
template<unsigned H, unsigned W>
inline Screen<H, W>& Screen<H, W>::move(pos r, pos c){
    pos row = r * width;
    cursor = row + c;
    return *this;
}
```

Answer: 16.14-15

Test

```
#include "Screen.h"
#include <iostream>
int main(){
    Screen<5, 5> scr('c');
    Screen<5, 5> scr2;
    // output src to the screen
    std::cout << scr;
    // input connet  to the  src
    std::cin >> scr2;
    // test input
    std::cout << scr2;
    return 0;
}
```

Exercise 16.18:

- Explain each of the following function template declarations and identify whether any are illegal. Correct each error that you find.
  - (a) template <typename T, U, typename V> void f1(T, U, V);
  - (b) template <typename T> T f2(int &T);
  - (c) inline template <typename T> T foo(T, unsigned int*);
  - (d) template <typename T> f4(T, T);
  - (e) typedef char Ctype;
  - template <typename Ctype> Ctype f5(Ctype a);

Answer:

```
// (a) every type parameter must be preceded by `typename` or `class`
//template <typename T, U, typename V> void f1(T, U, V);
template <typename T, typename U, typename V> void f1(T, U, V);
// (b) the function parameter name cannot be same as template parameter name
//template <typename T> T f2(int &T);
template <typename T> T f2(int &t);
// (c) the template parameter list should be first
//inline template <typename T> T foo(T, unsigned int*);
template <typename T> inline T foo(T, unsigned int*);
// (d) the function must have return type
//template <typename T> f4(T, T);
template <typename T> void f4(T, T);
// (e) no error, but the `typedef` will be hidden by template parameter name
typedef char Ctype;
template <typename Ctype> Ctype f5(Ctype a);

int main() {
    return 0;
}
```

Exercise 16.25:

- Explain the meaning of these declarations:
  - extern template class vector<string>;
    - **Answer:**
      - This is a template instantiation declaration. It means that there will be a non *extern* use of this instantiation elsewhere in the program.
  - template class vector<Sales_data>;
    - **Answer:**
      - This is a template instantiation definition. It will instantiate all members of template *vector*.

Exercise 16.27:

- For each labeled statement explain what, if any, instantiations happen. If a template is instantiated, explain why; if not, explain why not.

```
template <typename T> class Stack { };
void f1(Stack<char>);                          // (a)
class Exercise {
    Stack<double> &rsd;                        // (b)
    Stack<int> si;                             // (c)
};
int main() {
    Stack<char> *sc;                           // (d)
    f1(*sc);                                   // (e)
    int iObj = sizeof(Stack< string >);        // (f)
}
```

Answer

```
#include <string>
using std::string;
template <typename T> class Stack {
    typedef typename T::NotExist StaticAssert;     // used to test if instantiated
};
void f1(Stack<char>);                  // (a) not instantiate
class Exercise {
    Stack<double> &rsd;                // (b) not instantiate
    //Stack<int> si;                   // (c) instantiate `Stack<int>`
};
int main() {
Stack<char> *sc;                       // (d) not instantiate
//f1(*sc);                             // (e) instantiate `Stack<char>`
//int iObj = sizeof(Stack<string>);          // (f) instantiate `Stack<string>`
}
```

Exercise 16.32:

- What happens during template argument deduction?

- **Answer:**
  - During template argument deduction, the compiler uses types of the arguments in the call to find the template arguments that generate a version of the function that best matches the given call.

Exercise 16.33:

- Name two type conversions allowed on function arguments involved in template argument deduction.

- **Answer:**
  - *const* conversions
  - *Array* - or *function - to - pointer* conversions

Exercise 16.34

- Given only the following code, explain whether each of these calls is legal. If so, what is the type of T ? If not, why not?

  template <class T> int compare(const T&, const T&);

  - (a) compare("hi", "world");
  - (b) compare("bye", "dad");

- **Answer:**
  - (a) : illegal, as two types are different, the first type being const char[3], second const char[6]
  - (b) : legal, the type is const char(&)[4].

Exercise 16.35

Which, if any, of the following calls are errors ? If the call is legal, what is the type of T ? If the call is not legal, what is the problem ?

```
template <typename T> T calc(T, int);
template <typename T> T fcn(T, T);
double d; float f; char c;
(a)calc(c, 'c');
(b)calc(d, f);
(c)fcn(c, 'c');
(d)fcn(d, f);
```

Answer:

(a)legal, type is char.

(b)legal, type is double.

(c)legal, type is char.

(d)illegal, d is double, but f is float, they are totally different type.

Exercise 16.36

What happens in the following calls :

```
template <typename T> f1(T, T);
template <typename T1, typename T2) f2(T1, T2);
int i = 0, j = 42, *p1 = &i, *p2 = &j;
const int *cp1 = &i, *cp2 = &j;
```

- (a) `f1(p1, p2);`
- (b) `f2(p1, p2);`
- (c) `f1(cp1, cp2);`
- (d) `f2(cp1, cp2);`
- (e) `f1(p1, cp1);`
- (f) `f2(p1, cp1);`

At first, there are some error in function declarations.

- **f1, f2 should have return type, maybe void ?**
- **In f2, typename T2) should be typename T2>.**

Then, the answers :

- (a) T is int*
- (b) T1 and T2 are both int*
- (c) T is const int*
- (d) T1 and T2 are both const int*
- (e) error, p1 is int*, cp1 is const int*, they are different type
- (f) T1 is int*, T2 is const int*

Exercise 16.37

- The library max function has two function parameters and returns the larger of its arguments. This function has one template type parameter. Could you call max passing it an int and a double ? If so, how ? If not, why not?

- **Answer:**
- No, I could not. Because the arguments to max must have the same type.

Exercise 16.39:

Use an explicit template argument to make it sensible to pass two string literals to the original version of compare from § 16.1.1 (p. 652).

- **Answer:**

```
#include <iostream>
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int main(){
    std::cout << compare<std::string>("sss", "aaa") << "\n";
    system("pause");
}
```

Refereces

Accreditation:

- This presentation is prepared/extracted from the following resources:
  - C++ Primer, Fifth Edition.
    Stanley B. Lippman Josée Lajoie Barbara E. Moo
  - https://github.com/jaege/Cpp-Primer-5th-Exercises
  - https://github.com/Mooophy/Cpp-Primer
  - https://github.com/pezy/CppPrimer