Object Oriented Programming with C++

15. Templates

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

```
int product(int num1, int num2) {
    return num1 * num2;
}

float product(float num1, float num2) {
    return num1 * num2;
}

double product(double num1, double num2) {
    return num1 * num2;
}

int main() {

    cout << product(2, 3) << endl;
    cout << product(1.1f, 2.2f) << endl;
    cout << product(1.1, 2.2) << endl;

    return 0;
}
```

Output

```
6
2.42
2.42
```

- Notice duplication of code in all three variants of product function
- Is there a way to consolidate these different functions? As what they are doing remains same but only thing that differs is the type they handle

```
template<typename T>
T product(T num1, T num2) {
    return num1 * num2;
}

int main() {

    cout << product(2, 3) << endl;
    cout << product(1.1f, 2.2f) << endl;
    cout << product(1.1, 2.2) << endl;

    return 0;
}
```

Output

```
6
2.42
2.42
```

- Compiler generates actual code for function ***product*** (for types with which it has been called in the program). Here compiler will instantiate three overloads of function ***product*** as we are calling it with int, float and double types.

```
template<typename T>
T product(T num1, T num2) {
    return num1 * num2;
}
int main() {
    cout << product(2, 3) << endl;
    cout << product(1.1f, 2.2f) << endl;
    // error: no matching function for call to ‘product(int, double)
    // note: deduced conflicting types for parameter ‘T’ (‘int’ and ‘double’)
    // cout << product(3, 4.5) << endl;
    cout << product<int>(3, 4.5) << endl;
    cout << product<float>(3, 4.5) << endl;
    // string s1 = "Hello ", s2 = "world!\n";
    // error: no match for ‘operator*’
    // cout << product(s1, s2) << endl;

    return 0;
}
```

Output

```
6
2.42
12
13.5
```

- If template arguments are not explicitly specified during function call, then they are deduced by compiler based on function arguments and function template parameters.
  - Compiler will raise an error if there is conflict in deduction
- We may call function with explicit template arguments
  - e.g. product<int>(3, 4.5)
    - It will now consider T as int
- Compiler will instantiate only two overloads of function *product* as we are calling it with int and float types.

```
template<typename T>
T product(T num1, T num2) {
    return num1 * num2;
}
```

Is same as (atleast for our basic use)

```
template<class T>
T product(T num1, T num2) {
    return num1 * num2;
}
```

- Even when you use class keyword you can call this function with fundamental data types as T
- So, prefer use of keyowrd ***typename*** over keyword ***class*** as it is close to its actual behaviour

```
template<typename T>
void swap(T &num1, T &num2) {
    T temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
}
int main() {
    int int1 = 5, int2 = 7;
    cout << "Before swap: int1 = " << int1;
    cout << " and int2 = " << int2 << endl;
    swap(int1, int2);
    cout << "After swap: int1 = " << int1;
    cout << " and int2 = " << int2 << endl;

    double double1 = 5, double2 = 7;
    cout << "Before swap: double1 = " << double1;
    cout << " and double2 = " << double2 << endl;
    swap(double1, double2);
    cout << "After swap: double1 = " << double1;
    cout << " and double2 = " << double2 << endl;

    return 0;
}
```

Output

```
Before swap: int1 = 5 and int2 = 7
After swap: int1 = 7 and int2 = 5
Before swap: double1 = 5 and double2 = 7
After swap: double1 = 7 and double2 = 5
```

- Just another example, Templates can have references or pointers of T as well
- T is generally used as typename, but it could be any valid identifier

```
template<typename _abc_T>
void swap(_abc_T &num1, _abc_T &num2) {
    _abc_T temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
}
int main() {
    int int1 = 5, int2 = 7;
    cout << "Before swap: int1 = " << int1;
    cout << " and int2 = " << int2 << endl;
    swap(int1, int2);
    cout << "After swap: int1 = " << int1;
    cout << " and int2 = " << int2 << endl;

    double double1 = 5, double2 = 7;
    cout << "Before swap: double1 = " << double1;
    cout << " and double2 = " << double2 << endl;
    swap(double1, double2);
    cout << "After swap: double1 = " << double1;
    cout << " and double2 = " << double2 << endl;

    return 0;
}
```

Output

```
Before swap: int1 = 5 and int2 = 7
After swap: int1 = 7 and int2 = 5
Before swap: double1 = 5 and double2 = 7
After swap: double1 = 7 and double2 = 5
```

- Just another example, Templates can have references or pointers of T as well
- T is generally used as typename, but it could be any valid identifier

```
template<typename T>
T product(T num1, T num2) {
    cout << "Template function called\n";
    return num1 * num2;
}
float product(float num1, float num2) {
    cout << "Non-template function called\n";
    return num1 * num2;
}
int product(int num1, int num2, int num3) {
    cout << "Non-template function with three argumets called\n";
    return num1 * num2 * num3;
}
int main() {
    cout << product(2, 3) << endl;
    cout << product(1.1f, 2.2f) << endl;
    cout << product<float>(1.1f, 2.2f) << endl;
    cout << product<>(1.1f, 2.2f) << endl;
    cout << product(1.1, 2.2) << endl;
    // Not an exact match
    cout << product(1.1, 2.2, 3.3) << endl;

    return 0;
}
```

Output

```
Template function called
6
Non-template function called
2.42
Template function called
2.42
Template function called
2.42
Template function called
2.42
Non-template function with three argumets called
6
```

- Function template and non-template function can have same name
- Overload resolution for function template and non-template function is as follows (when called without explicit template arguments)
  - Call non-template function that has exact match
  - Instantiate function from function template (if possible with exact match)
  - Try overload resolution with non-template overloads of a function

```
template<typename T>
T product(T num1, int num2, T num3) {
    cout << "Template function called\n";
    return num1 * num2 * num3;
}

int main() {




    cout << product(2.5, 3, 2.5) << endl;

    cout << product(2.5, 3.5, 2.5) << endl;

    // error: no matching function for call to ‘product(double, double, int)’
    // note: deduced conflicting types for parameter ‘T’ (‘double’ and ‘int’)
    // cout << product(2.5, 3.5, 2) << endl;

    return 0;
}
```

Output

```
Template function called
18.75
Template function called
18.75
```

- Function arguments can be mixed
- Implicit conversion can take place for fixed type arguments of a function

```
template<typename T1, typename T2>
T2 product(T2 num1, T2 num2, T1 num3) {
    cout << "Template function called\n";
    return num1 * num2 * num3;
}

int main() {

    // error: no matching function for call to ‘product(double, int, double)’
    // note: deduced conflicting types for parameter ‘T2’ (‘double’ and ‘int’)
    // cout << product(2.5, 3, 2.5) << endl;

    cout << product(2.5, 3.5, 2.5) << endl;



    cout << product(2.5, 3.5, 2) << endl;

    return 0;
}
```

Output

```
Template function called
21.875
Template function called
17.5
```

- Function templates can have multiple template parameters (e.g. T1 and T2 in this case)
- There is no correlation between sequence of template parameters and sequence of function parameters.

```
template<typename T1, typename T2>
T1 product(T2 num1, T2 num2, T1 num3) {
    cout << "Template function called\n";
    T1 result;
    result = num1 * num2 * num3;
    return result;
}

int main() {

    cout << product<double,int>(2.5, 3, 2.5) << endl;

    cout << product(2.5, 3.5, 2.5) << endl;

    cout << product(2.5, 3.5, 2) << endl;

    return 0;
}
```

Output

```
Template function called
15
Template function called
21.875
Template function called
17
```

- Type specified in template parameters can also be used in function body
- Template arguments can be specified during function call even when template has multiple parameters

```
template<typename T1, typename T2>
T2 average(T1 arr[], int n) {
    T2 result;
    for(int i = 0; i < n; i++) {
        result += arr[i];
    }
    result /= n;
    return result;
}

int main() {

    float arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
    // error: no matching function for call to ‘average(float [5], int)’
    // couldn't deduce template parameter ‘T2’
    cout << average(arr, 5);

    return 0;
}
```

- Here compiler is not able to deduce type of T2
- Here are two different ways to solve it (there may be more)
  - Call average() function with template arguments
  - Remove T2 from list of template parameters and make result and return type to be of fixed type

```
template<typename T1, typename T2>
T2 average(T1 arr[], int n) {
    T2 result;
    for(int i = 0; i < n; i++) {
        result += arr[i];
    }
    result /= n;
    return result;
}

int main() {

    float arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};


    cout << average<float, double>(arr, 5);

    return 0;
}
```

Output

3.3

- Calling average() function with template arguments

```
template<typename T1>
double average(T1 arr[], int n) {
    double result;
    for(int i = 0; i < n; i++) {
        result += arr[i];
    }
    result /= n;
    return result;
}

int main() {

    float arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};


    cout << average(arr, 5);

    return 0;
}
```

Output

3.3

- Removed T2 from list of template parameters and made result and return type to be of double type

```
template<typename T1, typename T2>
T1 sum(T1 arg1, T2 arg2) {
    return arg1 + arg2;
}

class Complex {
    double re, im;
public:
    Complex(double re, double im) {
        this->re = re;
        this->im = im;
    }
    Complex operator+(const Complex &c) {
        return Complex(re + c.re, im + c.im);
    }
    friend std::ostream &operator<<(std::ostream &, const Complex &);
};

std::ostream &operator<<(std::ostream &strm, const Complex &c) {
    strm << "(" << c.re << ", " << c.im << ")\n";
    return strm;
}
```

```
int main() {

    cout << sum(1.2, 2) << endl;
    cout << sum("Hello ", 2) << endl;
    // error: no match for ‘operator+’
    // cout << sum(string("Hello "), 2) << endl;
    cout << sum(string("Hello "), string("world!")) << endl;

    Complex c1(1.1, 2.2), c2(3.3, 4.4);
    cout << sum(c1, c2) << endl;

    return 0;
}
```

Output

```
3.2
llo
Hello world!
(4.4, 6.6)
```

- Two different parameters of a template (***T1*** and ***T2*** here), can be deduced/specified to be of same type
- We can pass user defined type (e.g. ***Complex*** in this example) as argument to template parameters

```
template<typename T1, typename T2>
T1 sum(T1 arg1, T2 arg2) {
    return arg1 + arg2;
}

int main() {

    cout << sum<int>(1.2, 2) << endl;
    cout << sum(string("Hello "), string("world!")) << endl;

    return 0;
}
```

Output

3

Hello world!

- If only one type is passed as template argument then other types of remaining template parameters will be deduced by compiler.

```
template<typename T1, typename T2>
T1 sum(T1 arg1, T2 arg2 = 5) {
    return arg1 + arg2;
}

int main() {
    // error: no matching function for call to ‘sum(double)’
    // note:   couldn't deduce template parameter ‘T2’
    // cout << sum(1.2) << endl;
    cout << sum<double, int>(1.2) << endl;
    cout << sum(string("Hello "), string("world!")) << endl;
    // could not convert ‘5’ from ‘int’ to ‘std::string’
    // cout << sum<string, string>(string("Hello ")) << endl;

    return 0;
}
```

Output

6.2

Hello world!

- Function parameters of function template can take default values
- But if you call scuh function without passing actual argument (for default parameter), compiler will not deduce its type based on default argument for function parameter. You need to explicitly specify template argument during call
- Here T2 can even be string type as far as you provide value for arg2 during function call

```
template<typename T1, typename T2 = int>
T1 sum(T1 arg1, T2 arg2 = 5.2) {
    return arg1 + arg2;
}

int main() {

    cout << sum(1.2) << endl;
    cout << sum<double, double>(1.2) << endl;
    cout << sum(1.2, 2.2) << endl;


    return 0;
}
```

Output

```
6.2
6.4
3.4
```

- You can specify default type for the template parameter
- If compiler can deduce the type for template parameter from fuction argument then it will ignore default type
- Default value of function argument would not be used for deducing type for template parameter

```
class StackOverflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackOverflowException\n";
    }
};
class StackUnderflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackUnderflowException\n";
    }
};
template <typename T>
class Stack
{
    T *arr;
    int top;
    int capacity;
public:
    Stack(int size) {
        arr = new T[size];
        capacity = size;
        top = -1;
    }
    void push(T element) {
        if(top == capacity - 1)
            throw *(new StackOverflowException);
        arr[++top] = element;
    }
    T pop() {
        if(top == -1)
            throw *(new StackUnderflowException);
        return arr[top--];
    }
    ~Stack(){
        delete [] arr;
    }
};
int main()
{
    Stack<int> int_stack(3);
    try {
        int_stack.push(5);
        int_stack.push(7);
        int_stack.push(3);
        cout << int_stack.pop() << endl;
        cout << int_stack.pop() << endl;
        int_stack.push(9);
        cout << int_stack.pop() << endl;
    }
    catch(std::exception &e) {
        cout << e.what();
    }
    return 0;
}
```

Output

```
3
7
9
```

- An example of class template. ***template*** keyoword and parameters are needed before class (like function template)
- Template keyword and parameter list is not needed for inline methods
- Template parameters can be used by all methods inside class
- Should explicitly specify template arguments while creating object (Deduction possible since C++17 based on constructor parameters and arguments while creating object).

```
class StackOverflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackOverflowException\n";
    }
};
class StackUnderflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackUnderflowException\n";
    }
};
template <typename T>
class Stack
{
    T *arr;
    int top;
    int capacity;
public:
    Stack(int size) {
        arr = new T[size];
        capacity = size;
        top = -1;
    }
    void push(T element) {
        if(top == capacity - 1)
            throw *(new StackOverflowException);
        arr[++top] = element;
    }
    T pop() {
        if(top == -1)
            throw *(new StackUnderflowException);
        return arr[top--];
    }
    ~Stack(){
        delete [] arr;
    }
};
int main()
{
    Stack<string> str_stack(3);
    try {
        str_stack.push(string("ABC"));
        str_stack.push(string("PQR"));
        str_stack.push(string("XYZ"));
        cout << str_stack.pop() << endl;
        cout << str_stack.pop() << endl;
        str_stack.push(string("LMN"));
        cout << str_stack.pop() << endl;
    }
    catch(std::exception &e) {
        cout << e.what();
    }
    return 0;
}
```

Output

XYZ
PQR
LMN

- Creating stack of strings using the same class template as in previous slide

```
class StackOverflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackOverflowException\n";
    }
};
class StackUnderflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackUnderflowException\n";
    }
};
template <typename T>
class Stack
{
    T *arr;
    int top;
    int capacity;
public:
    Stack(int size) {
        arr = new T[size];
        capacity = size;
        top = -1;
    }
    void push(T element) {
        if(top == capacity - 1)
            throw *(new StackOverflowException);
        arr[++top] = element;
    }
    T pop() {
        if(top == -1)
            throw *(new StackUnderflowException);
        return arr[top--];
    }
    ~Stack(){
        delete [] arr;
    }
};
int main()
{
    Stack<string> str_stack(3);
    try {
        str_stack.push(string("ABC"));
        str_stack.push(string("PQR"));
        str_stack.push(string("XYZ"));
        str_stack.push(string("LMN"));
        cout << str_stack.pop() << endl;
        cout << str_stack.pop() << endl;
        cout << str_stack.pop() << endl;
    }
    catch(std::exception &e) {
        cout << e.what();
    }
    return 0;
}
```

Output

StackOverflowException

```
class StackOverflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackOverflowException\n";
    }
};
class StackUnderflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackUnderflowException\n";
    }
};
template <typename T>
class Stack
{
    T *arr;
    int top;
    int capacity;
public:
    Stack(int size);
    void push(T element);
    T pop();
    ~Stack(){
        delete [] arr;
    }
};
template <typename T>
Stack<T>::Stack(int size) {
    arr = new T[size];
    capacity = size;
    top = -1;
}
template <typename T>
void Stack<T>::push(T element) {
    if(top == capacity - 1)
        throw *(new StackOverflowException);
    arr[++top] = element;
}
template <typename T>
T Stack<T>::pop() {
    if(top == -1)
        throw *(new StackUnderflowException);
    return arr[top--];
}
int main()
{
    Stack<string> str_stack(3);
    try {
        str_stack.push(string("ABC"));
        str_stack.push(string("PQR"));
        str_stack.push(string("XYZ"));
        cout << str_stack.pop() << endl;
        str_stack.push(string("LMN"));
        cout << str_stack.pop() << endl;
        cout << str_stack.pop() << endl;
    }
    catch(std::exception &e) {
        cout << e.what();
    }
    return 0;
}
```

Output

XYZ
LMN
PQR

- Methods of a class template can also be defined outside the class definition.
- When methods aredefined outside class definition then they need to have **template parameters** and specification that it is a method of template class (e.g. Stack**<T>**)

```
class StackOverflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackOverflowException\n";
    }
};
class StackUnderflowException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "StackUnderflowException\n";
    }
};
template <typename T, int size>
class Stack
{
    T *arr;
    int top;
    int capacity;
public:
    Stack() {
        arr = new T[size];
        capacity = size;
        top = -1;
    }
    void push(T element) {
        if(top == capacity - 1)
            throw *(new StackOverflowException);
        arr[++top] = element;
    }
    T pop() {
        if(top == -1)
            throw *(new StackUnderflowException);
        return arr[top--];
    }
    ~Stack(){
        delete [] arr;
    }
};
int main()
{
    // int size;
    // cin >> size;
    // error: the value of 'size' is not usable in a constant expression
    // in this case, size must be known to the compiler at compile time
    // Stack<string, size> str_stack;
    Stack<string, 3> str_stack;
    try {
        str_stack.push(string("ABC"));
        str_stack.push(string("PQR"));
        str_stack.push(string("XYZ"));
        cout << str_stack.pop() << endl;
        cout << str_stack.pop() << endl;
        str_stack.push(string("LMN"));
        cout << str_stack.pop() << endl;
    }
    catch(std::exception &e) {
        cout << e.what();
    }
    return 0;
}
```

Output

XYZ
PQR
LMN

- Class template can have non-type parameters (e.g. int size in this case). Size is not a type (typename or class). Its int type. But its value should be known at compile time.

- Function template and class template can have more than one **non-type parameters**, **type parameters** or **template parameters**
- Type deduction is possible for class template since C++17, based on types of arguments passed while creating object and types of constructor parameters
  - https://en.cppreference.com/w/cpp/language/class_template_argument_deduction
- You may overload a function template either by a non-template function or by another function template.
  - https://stackoverflow.com/questions/2174300/function-template-overloading

Interesting reads

- Function Template
  - https://en.cppreference.com/w/cpp/language/function_template
- Class Template
  - https://en.cppreference.com/w/cpp/language/class_template
- Class template argument deduction (CTAD) (since C++17)
  - https://en.cppreference.com/w/cpp/language/class_template_argument_deduction
- You may overload a function template either by a non-template function or by another function template.
  - https://stackoverflow.com/questions/2174300/function-template-overloading