OO Programozás

Sablonok, kivételkezelés

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM085

2024. november 7.

Sablonok (template)

- Különböző típusokat kezelő függvények / osztályok csoportjának egyszerű létrehozása
- *Típusparaméter* (fordítási időben eldől) vs. függvény paraméter (futási időben adják át)
- Kulcsszavak:
  - Sablon → `template`
  - Típusparaméterek → `class` vagy `typename` (*majdnem* egyenértékűek)
- *Sablon példányosítás:* adott típusparaméterrel új változat előállítása → egyszerűbb, mint sok felültöltött függvényt kézzel létrehozni

Feladat:

Készítsünk `max` függvényt, ami két paramétere közül visszaadja a nagyobbat!

max1.cpp

```
3  // Overloaded max functions
4  int max(int a, int b) {
5    return a > b ? a : b;
6  }
7
8  double max(double a, double b) {
9    return a > b ? a : b;
10 }
```

max1.cpp

```
12 int main() {
13   const int x = 1;
14   const int y = 2;
15   std::cout << "Of " << x << " and " << y << ", "
16             << max(x, y) << " is greater.\n";
17
18   const double i = 1.5;
19   const double j = 2.5;
20   std::cout << "Of " << i << " and " << j << ", "
21             << max(i, j) << " is greater.\n";
22 }
```

Kimenet

```
Of 1 and 2, 2 is greater.
Of 1.5 and 2.5, 2.5 is greater.
```

max2.cpp

```
3  template<class T> T max(T a, T b) {
4    return a > b ? a : b;
5  }
6
7  int main() {
8    const int x = 1;
9    const int y = 2;
10   std::cout << "Of " << x << " and " << y << ", "
11             << max(x, y) << " is greater.\n";
12
13   const double i = 1.5;
14   const double j = 2.5;
15   std::cout << "Of " << i << " and " << j << ", "
16             << max(i, j) << " is greater.\n";
```

- Egy sablon több típusparamétert is fogadhat, de akkor ezeket mind használni is kell!
- Csak akkor állít elő kódot a fordító, amikor a függvény konkrét hívásával találkozik $\rightarrow$ a példányosításig a hibás sablon nem vált ki hibaüzenetet
- Pontosan olyan változatok jönnek létre, amikre szükség is van
- Típusonként pontosan egy változatot hoz létre, akkor is, ha létezik a típusok között implicit konverzió (pl. `int` $\rightarrow$ `long`)
- Mi történik, ha két *eltérő* típusú paraméterrel hívják a függvényünket?

max2.cpp

```
18    // const int k = 3;
19    // const double l = 4.5;
20    // std::cout << "Of " << k << " and " << l << ", "
21    //           << max(k, l) << " is greater.\n";
22    // error: no matching function for call to
23    // 'max(const int&, const double&)'
24    // note: candidate: template<class T> T max(T, T)
25    // note: template argument deduction/substitution failed:
26    // note: deduced conflicting types for parameter 'T' ('int' and 'double')
27  }
```

Probléma:

A két paraméter típusának mindenképpen egyeznie kell!

Félmegoldás:

Két típusparaméter használata (nem befolyásolható futásidőben, melyikben keletkezzen az eredmény)

max3.cpp

```
3  template<class T1, class T2> T1 max(T1 a, T2 b) {
4    return a > b ? a : b;
5  }
6
7  int main() {
8    const int i = 3;
9    const double d = 4.5;
10   std::cout << "Of " << i << " and " << d << ", "
11             << max(i, d) << " is greater.\n";
12   std::cout << "Of " << d << " and " << i << ", "
13             << max(d, i) << " is greater.\n";
14 }
```

Kimenet

```
Of 3 and 4.5, 4 is greater.
Of 4.5 and 3, 4.5 is greater.
```

Ha létezik függvény a szükséges paraméterekkel, nem jön létre példány a sablonból, és a korábbi hiba orvosolható.

max4.cpp

```
3  template<class T1, class T2> T1 max(T1 a, T2 b) {
4    return a > b ? a : b;
5  }
6
7  double max(int a, double b) {
8    return a > b ? a : b;
9  }
```

Kimenet

```
Of 3 and 4.5, 4.5 is greater.
Of 4.5 and 3, 4.5 is greater.
```

Fontosabb tudnivalók:

- A függvénysablonokhoz hasonlóan *osztálysablonokat* is létrehozhatunk.
- A típusparaméterek mellett *konstansparamétereket* is megadhatunk → bárhova helyettesíthetők, ahol konstans kifejezés állhat.
- Ha egy tagfüggvényt az osztályon kívül definiálunk, meg kell ismételni a `template` kulcsszót a paraméterekkel együtt. A definíciónak kivételesen a fejfájlban a helye, hogy a fordító mindig elérje.
- Osztálysablon példányosítása: a típus után < és > között meg kell adni a konkrét típusokat, konstansokat. Csak ezek együtt azonosítják egyértelműen a sablon alapján előállított osztályt!

Stack1.h

```
6  template<class T, int I> class Stack {
7      T* array;
8      const int size;
9      int used;
10   public:
11     Stack() : size(I) {
12       array = new T[I];
13       used = 0;
14     }
15     void push(T data);
```

Stack1.h

```
16      T pop() {
17        if(used > 0) {
18          return array[--used];
19        } else {
20          std::cerr << "Oops, the stack is empty.\n";
21          return array[0]; // something must be returned
22        }
23      }
24      bool isEmpty() {
25        return used == 0;
26      }
27  };
```

Stack1.h

```
29 // It MUST be in the header!
30 template<class T, int I> void Stack<T, I>::push(T data) {
31   if(used < size) {
32     array[used++] = data;
33   } else {
34     std::cerr << "Stack full :(\n";
35   }
36 }
```

stackMain1.cpp

```
1  #include <iostream>
2  #include "Stack1.h"
3
4  int main() {
5    Stack<int, 3> si;
6    si.push(1); si.push(2); si.push(3);
7    si.push(4); // stack full
8    while(not si.isEmpty()) {
9      std::cout << si.pop() << std::endl;
10   }
```

stackMain1.cpp

```
12    Stack<const char*, 10> scp;
13    scp.push("World!\n"); scp.push("Hello ");
14    std::cout << scp.pop();
15    std::cout << scp.pop();
16    std::cout << scp.pop(); // stack empty
17  }
```

Kivételkezelés (*exception handling*)

- *Kivételes*, azaz ritka események (hibák) kezelésére
- Előidézheti hardveres (pl. elfogy a memória) vagy szoftveres (pl. dinamikus tömb túlindexelése) hiba
- Védett blokk: felkészülés kivételes helyzet bekövetkezésére → `try`
- Kivétel kiváltása → `throw`
- Kivétel elfogása → `catch`

`try` blokkok jellemzői:

- Egymásba ágyazhatók.
- Benne közvetlenül vagy közvetetten (függvényen belül) is használható `throw` utasítás.
- Több `catch` ág is tartozhat egy `try`-hoz, vagy akár egy sem.

`throw` jellemzői:

- Tetszőleges típusú kivétel kiváltható.
- A `try` blokk végrehajtása azonnal félbeszakad a hatására.
- Használható akár konstruktorban, vagy `catch` blokkban is.
- A `throw;` utasítás a korábban kiváltott és már elfogott típusú és tartalmú kivétellel azonosat vált ki újra.

`catch` blokkok jellemzői:

- Az első illeszkedő blokk kezeli a kivételt.
- Nyilvánosan származtatott kivétel osztályok esetén a szülő típusára illeszthető a leszármazott típusú kivétel → speciálisak előre, általánosabbak hátra!
- Nem illeszkednek viszont az implicit típuskonverzióval egyébként egymásba alakítható típusok `catch` blokkjai (pl. `int` típusú kivételt nem fogja el a `float` kezelője).
- A speciális, `...` típusú `catch` minden, korábban nem kezelt kivételt elfog → ez legyen az utolsó!
- A kivétel kezelését követően a program fut tovább az utolsó `catch` blokk után.
- Ha nincs illeszthető `catch` ág, a vezérlés a befoglaló `try-catch` szerkezetre kerül, és ott keres illeszkedő kivételkezelőt. Ha ilyen nincs, a vezérlés a `terminate()` függvényre kerül, és a program leáll.

Kivételkezelési stratégiák:

1. Figyelmeztetés, felhasználói beavatkozás kérése, megpróbálkozás a hiba javításával.
2. Program leállítása (pl. `exit()`, `abort()`) súlyos hibák esetén.
3. Kivétel kezeletlenül hagyása.

A dinamikusan foglalt erőforrásokat szükség esetén fel kell szabadítani!

trycatch1.cpp

```
1  #include <iostream>
2
3  class BaseException {};
4  class DerivedException : public BaseException {};
5  // private inheritance --> exception object's reference
6  //                         cannot be bound to base reference
7
8  int main() {
9    for(int exceptType = 0; exceptType < 6; exceptType++) {
10     try {
11       switch(exceptType) {
```

trycatch1.cpp

```
12          case 0:
13            std::cout << "Throwing␣int...␣";
14            throw 42;
15            break;
16          case 1:
17            std::cout << "Throwing␣double...␣";
18            throw 3.14;
19            break;
20          case 2:
21            std::cout << "Throwing␣const␣char*...␣";
22            throw "Can␣you␣access␣it?";
23            break;
```

trycatch1.cpp

```
24          case 3:
25            std::cout << "Throwing␣BaseException...␣";
26            throw BaseException();
27            break;
28          case 4:
29            std::cout << "Throwing␣DerivedException...␣";
30            throw DerivedException();
31            break;
32          case 5:
33            std::cout << "Throwing␣char...␣";
34            throw 'x';
35            break;
36        }
```

trycatch1.cpp

```
37      } catch(char c) {
38        std::cout << "caught " << c << ", re-throw exception"
39                  << std::endl;
40        throw;
41      // no implicite type conversions, no conversion ctor calls
42      } catch(double d) {
43        std::cout << "caught " << d << std::endl;
44      } catch(const char*) { // parameter without name
45        std::cout << "caught an unknown message" << std::endl;
```

trycatch1.cpp

```
46      // warning: exception of type 'DerivedException' will be
47      //          caught by earlier handler for 'BaseException'
48      } catch(BaseException& se) {
49        std::cout << "caught BaseException object" << std::endl;
50      } catch(DerivedException& de) {
51        std::cout << "caught DerivedException object" << std::endl;
52      } catch(...) {
53        std::cout << "caught an UNKNOWN exception" << std::endl;
54      }
55    }
56  }
```

Kimenet

```
Throwing int... caught an UNKNOWN exception
Throwing double... caught 3.14
Throwing const char*... caught an unknown message
Throwing BaseException... caught BaseException object
Throwing DerivedException... caught BaseException object
Throwing char... caught x, re-throw exception
terminate called after throwing an instance of 'char'
Aborted (core dumped)
```

trycatch2.cpp

```
3  void fn() {
4    try {
5      std::cout << "Do you want to throw an exception of type int? ";
6      std::string answer;
7      std::cin >> answer;
8      if(answer == "y") throw 1;
9      throw 1.2;
10   } catch(double) {
11     std::cout << "Caught a double.\n";
12   }
13 }
```

trycatch2.cpp

```
15 int main() {
16   try {
17     fn();
18     try {
19       std::cout << "Do you want to throw an exception of type char? ";
20       std::string answer;
21       std::cin >> answer;
22       if(answer == "y") throw 'a';
23       throw "message";
24     } catch(char) {
25       std::cout << "Caught a char.\n";
26     }
27   } catch(int) {
28     std::cout << "Caught an int.\n";
29   } catch(const char*) {
30     std::cout << "Caught a pointer to char.\n";
31   }
32 }
```

Kimenet1

```
Do you want to throw an exception of type int? y
Caught an int.
```

Kimenet2

```
Do you want to throw an exception of type int? n
Caught a double.
Do you want to throw an exception of type char? y
Caught a char.
```

Kimenet3

```
Do you want to throw an exception of type int? n
Caught a double.
Do you want to throw an exception of type char? n
Caught a pointer to char.
```

Alakítsuk át a verem megvalósítását úgy, hogy kivételkezelést használjon! Használjuk fel a függvénykönyvtár `exception` osztályát!

Stack2.h

```
4  #include <iostream>
5  #include <exception>
6
7  class StackException : public std::exception {
8    public:
9      enum ErrorCode { UNDEFINED, EMPTY, FULL };
10   private:
11     ErrorCode code;
12     std::string message;
```

Stack2.h

```
17    public:
18      StackException() : code(ErrorCode::UNDEFINED), message("") {}
19      StackException(ErrorCode code, const std::string& message) :
20        code(code), message(message) {}
21      ErrorCode getCode() const {
22        return code;
23      }
24      // we have to promise not to throw exceptions in what()
25      virtual const char* what() const noexcept {
26        return message.c_str();
27      }
28  };
```

Stack2.h

```
26  template<class T, int I> class Stack {

30    public:

35      void push(T data);
36      T pop() {
37        if(used > 0) {
38          return array[--used];
39        } else {
40          throw StackException(
41            StackException::ErrorCode::EMPTY,
42            "Oops, the stack is empty."
43          );
44        }
45      }
```

Stack2.h

```
51 template<class T, int I> void Stack<T, I>::push(T data) {
52   if(used < size) {
53     array[used++] = data;
54   } else {
55     throw StackException(
56       StackException::ErrorCode::FULL,
57       "Sorry, stack full."
58     );
59   }
60 }
```

stackMain2.cpp

```
4  int main() {
5    Stack<int, 3> si;
6    try {
7      si.push(1); si.push(2); si.push(3);
8      si.push(4);
9    } catch(const StackException& se) {
10     std::cerr << "Error code: " << se.getCode()
11               << ", " << se.what()
12               << std::endl;
13   }
14   while(not si.isEmpty()) {
15     std::cout << si.pop() << std::endl;
16   }
```

stackMain2.cpp

```
18   Stack<const char*, 10> scp;
19   scp.push("World!\n"); scp.push("Hello ");
20   try {
21     std::cout << scp.pop();
22     std::cout << scp.pop();
23     std::cout << scp.pop();
24   } catch(const StackException& se) {
25     std::cerr << "Error code: " << se.getCode()
26               << ", " << se.what()
27               << std::endl;
28   }
29 }
```

Kimenet

```
Error code: 2, Sorry, stack full.
3
2
1
Hello World!
Error code: 1, Oops, the stack is empty.
```