STL: obiekty funkcyjne

Adaptory wiążące (*binders*)

zmieniają funktor dwuargumentowy w jednoargumentowy

`bind1st(fun,val)` - wiąże pierwszy argument z wartością **`val`**
`bind2nd(fun,val)` - wiąże drugi argument z wartością **`val`**

(*Deprecated in C++11 and removed in C++17*)

Przykłady:

#1: skalowanie współrzędnych punktu **P** i zapisanie ich w **R**: **R = P*s**:

```
transform(P.begin(), P.end(), R.begin(),
                    bind2nd(multiplies<double>(), s));
```

#2: aby znaleźć w kolekcji elementy o wartości **v<7** :

```
bind2nd(less<int>(),7)
```


STL: obiekty funkcyjne

Adaptory negujące

stosowane do predykatów, tworzą predykat obliczający negację pierwotnego

`not1(fun)` - neguje wynik predykatu jednoargumentowego
`not2(fun)` - neguje wynik predykatu dwuargumentowego

Przykład:

Policzenie liczb parzystych w kolekcji **v**:

```
liczba = count_if(v.begin(),v.end(),
                  not1(bind2nd(modulus<int>(),2)));
```

Uwaga: **`modulus`** nie jest predykatem, ponieważ zwraca wynik operatora modulo, ale liczbę typu **`int`** można traktować jak wartość logiczną, więc nadal działa

STL: obiekty funkcyjne

Adaptory zwykłych funkcji i metod

tworzą funktor na podstawie:

`ptr_fun(fun)` - zwykłej funkcji lub metody statycznej
`mem_fun_ref(fun)` - metody (niestatycznej) obiektu
`mem_fun(fun)` - metody obiektu dostępnego przez wskaźnik

(*Deprecated in C++11 and removed in C++17*)

- istnieją jedynie adaptory metod i funkcji jednoargumentowych i dwuargumentowych ponieważ tylko takie są używane w bibliotece STL,
- jawne zamknięcie metod w funktorach jest niezbędne ponieważ:
  - wywołanie metody jest składniowo inne niż zwykłej funkcji (czyli inne niż funktora),
  - aby możliwe było zastosowanie adaptorów wiążących i negujących.

STL: obiekty funkcyjne

Przykład:

Mamy funkcję, która dla elementu zwraca informacje, czy należy on do pewnej klasy, czy nie (wartość **`true`** lub **`false`**). Należy policzyć elementy, które nie należą do tej klasy.

```
bool parzysta(int a) { return !(a&1); }

void main()
{
  int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
  vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
  vector<int>::iterator it;
  //int liczba = count_if(v.begin(),v.end(),not1(parzysta)); // źle!
  int liczba = count_if(v.begin(),v.end(),not1(ptr_fun(parzysta)));
};
```


STL: obiekty funkcyjne

Przykład:

Policzyć długości słów przechowywanych w wektorze **`vector<string>`**.
Dla typu **`string`** można wywołać metodę **`length`**

```
vector<string> numbers;         // wektor słów

numbers.push_back("raz");
numbers.push_back("dwa");
numbers.push_back("trzy");
numbers.push_back("cztery");

vector <int> lengths (numbers.size());   // wektor długości słów

transform (numbers.begin(), numbers.end(), lengths.begin(),
                                    mem_fun_ref(&string::length));
```


STL: obiekty funkcyjne

Przykład:

Policzyć długości słów przechowywanych w wektorze **`vector<string*>`**.
Dla typu **`string`** można wywołać metodę **`length`**

```
vector<string*> numbers;        // wektor słów

numbers.push_back("raz");
numbers.push_back("dwa");
numbers.push_back("trzy");
numbers.push_back("cztery");

vector <int> lengths (numbers.size());   // wektor długości słów

transform (numbers.begin(), numbers.end(), lengths.begin(),
                                    mem_fun(&string::length));
```

STL: obiekty funkcyjne

Adaptory – podsumowanie:

```
1. bind1st(fun,val)(arg)          fun(val,arg)
2. bind2nd(fun,val)(arg)          fun(arg,val)

3. not1(fun)(arg)                 !fun(arg)
4. not2(fun)(arg1,arg2)           !fun(arg1,arg2)

5. ptr_fun(fun)(arg)              fun(arg)
6. ptr_fun(fun)(arg1,arg2)        fun(arg1,arg2)
7. mem_fun(fun)(arg)              arg->fun()
8. mem_fun(fun)(arg1,arg2)        arg1->fun(arg2)
9. mem_fun_ref(fun)(arg)          arg.fun()
10.mem_fun_ref(fun)(arg1,arg2)    arg1.fun(arg2)
```


STL: obiekty funkcyjne

Adaptory – podsumowanie

- funktory standardowe są szablonami klas, dlatego przy ich użyciu musimy podać parametry szablonu (dla adaptorów jako szablonów funkcji nie musimy)
- adaptory funktorów są szablonami funkcji, których argumentami są inne funkcje lub funktory
- adaptory funktorów rzadko wywołujemy jawnie, zwykle przekazujemy jako parametr do algorytmów, dlatego wywołania z dwoma parami nawiasów praktycznie się nie zdarzają (poza samym kodem biblioteki STL)

STL: obiekty funkcyjne

Tworzenie własnych funktorów

definiowane jako publiczne klasy (struktury) pochodne od struktur

1. **`unary_function`** – jednoargumentowe
2. **`binary_function`** – dwuargumentowe

```
#include <functional>
```

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};

template <class Arg1, class Arg2,
                          class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 1

jednoargumentowy, dziedziczy po **`unary_function`**

```
template<class TYPE>
struct modulo2 : public unary_function<TYPE, void> {
    void operator() (TYPE& x) {
        x = x%2;
    }
};
```


STL: obiekty funkcyjne

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
template<class TYPE> struct modulo2 : public unary_function<TYPE, void> {
    void operator() (TYPE& x) { x = x%2; }
};
int main(int argc, char *argv[])
{
  int myints[] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector<int> v(myints,myints+8);
  cout << "before: ";
  copy( v.begin(), v.end(), ostream_iterator<int>( cout, " " ) );
  cout << endl;
  for_each( v.begin(), v.end(), modulo2<int>() );
  cout << "after : ";
  copy( v.begin(), v.end(), ostream_iterator<int>( cout, " " ) );
  cout << endl;
  return 0;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 2

- jednoargumentowy, dziedziczy po **`unary_function`**,
- przyjmuje w argumentach szablonu informacje potrzebne do poprawnego działania,
- przyjmijmy, że jest zapisany w pliku **`Przedzial.h`**

```
template <class T,T dol, T gora>
struct Przedzial: unary_function<T,bool> {
  bool operator()(T x) const {
    return x>=dol && x<=gora;
  }
};
```

STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 2

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "przedzial.h"
using namespace std;
void main()
{
int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
int c;
c=count_if(v.begin(),v.end(),Przedzial<int,5,9>());
cout << c << endl ;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 3

- jednoargumentowy, dziedziczy po **`unary_function`**,
- Przyjmuje w konstruktorze i przechowuje w polach informacje potrzebne do poprawnego działania,
- przyjmijmy, że jest zapisany w pliku **`Przedzial.h`**

```
template <class T>
class Przedzial: public unary_function<T,bool>
{
    T dol, gora;
  public:
    Przedzial(T d, T g) : dol(d), gora(g) {}
    bool operator()(T x) const { return x>=dol && x<=gora; }
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów – przykład 3

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "przedzial.h"
using namespace std;
void main()
{
int tab[] = { 2, 5, 4, 9, 12, 3, 8 };
vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
int c;
c=count_if(v.begin(),v.end(),Przedzial<int>(5,9));
cout << c << endl ;
}
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów

- dziedziczenie funktorów po **`unary_function`** i **`binary_function`** nie dodaje im żadnej nowej funkcjonalności,
- jedynym zadaniem dziedziczenia jest nadanie uniwersalnych etykiet (za pomocą **`typedef`**) typom danych, jakie są przekazywane do funktora; informacja ta jest wykorzystywana w kodzie biblioteki **`functional`**.

STL: obiekty funkcyjne

Tworzenie własnych funktorów

Przyjmijmy, że mamy funkcję i nie chcemy z niej korzystać poprzez **`ptr_fun`**, ale zrobić z niej funktor. Zasada tworzenia funktora z funkcji jest identyczna dla **`unary_function`** i **`binary_function`** :

Krok 1 (początek: Zamieniamy dwuargumentową funkcję na funktor)**:**

Przyjmijmy, że mamy funkcję dwuargumentową, która przyjmuje dwie wartości typu **`string`** i **`int`** i zwraca **`bool`**.

```
class MyOtherFunction {
public:
  bool operator() (const string& str, int val) const {
  /* Do something, return a bool. */
  }
};
```


STL: obiekty funkcyjne

Tworzenie własnych funktorów

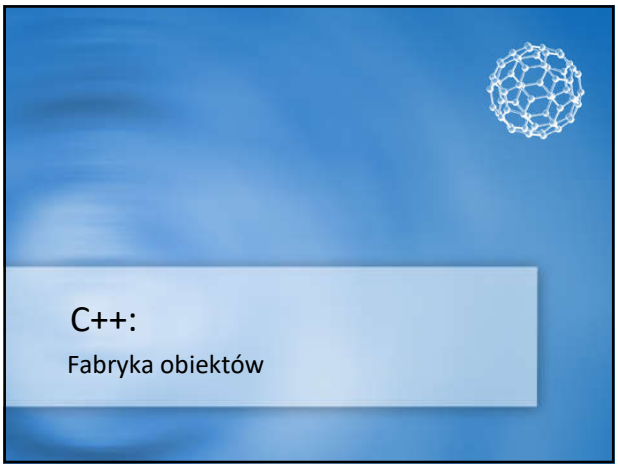
Ponieważ funkcja jest dwuargumentowa, sięgamy do szablonu **`binary_function`**, którego nagłówek wygląda następująco:

```
template <typename Param1Type, typename Param2Type, typename ResultType>
class binary_function;
```

Krok 2 (ostatni)**:**

dodajemy dziedziczenie po **`binary_function`**

```
class MyOtherFunction: public binary_function<string, int, bool> {
public:
  bool operator() (const string& str, int val) const {
    /* Do something, return a bool. */
  }
};
```



168

STL: kontenery

Fabryka obiektów

- Klasa, której obiekty pośrednicza przy tworzeniu innych obiektów.
- Pomagają tworzyć obiekty, jeżeli
  informacja o typie odnosi się do konkretnego typu, znanego w momencie kompilacji,
  ale..
  forma jest nieodpowiednia dla kompilatora.
- Fabryka ukrywa przed użytkownikiem mechanizm zamiany identyfikatora na literał dostarczany do operatora **new**, upraszczając tworzenie obiektów.

169

STL: kontenery

Fabryka obiektów – przykład:

Dana jest aplikacja do rysowania schematów blokowych.

Rysunek zapisywany jest w pliku w postaci listy parametrów opisujących poszczególne figury obecne na diagramie. Jest kilka rodzajów figur, które mogą wystąpić na diagramie.

Klasa bazowa **Figure** dostarcza identyfikatorów dla klas konkretnych:

```
class Figure {
public:
        enum Type {SQUARE, CIRLCE, TRIANGLE };
        virtual void write(ostream& os) const = 0;
        virtual void read(istream& is) = 0;
};
```


170

STL: kontenery

Fabryka obiektów – przykład:

Klasa pochodna nadpisuje metody wirtualne służące do zapisania obiektu do pliku oraz odczytania z pliku:

```
class Square : public Figure {
public:
        void write(ostream& os) const {
                os << SQUARE;
                /* zapis pól */
        };
        void read(istream& is) {
                /* odczyt pól */
        };
};
```


171

STL: kontenery

Fabryka obiektów – przykład:

- Podczas odczytu danych z pliku musimy polegać na informacji dostarczanej w trakcie działania, tj. na identyfikatorze obiektu zapisanym w pliku (*zapis do pliku zaczyna się od podania identyfikatora, dopiero potem są zapisywane zawartości pól obiektu*)
- Taki identyfikator to nie jest argument dla **new**. ☹
- Potrzebny jest mechanizm, który tworzy obiekty na podstawie identyfikatora..

172

STL: kontenery

Fabryka obiektów – przykład:

```
Figure* createObj(Figure::Type type) {
        switch (type) {
        case Figure::SQUARE:
                return new Square();
        case Figure::CIRCLE:
                return new Circle();
        case Figure::TRIANGLE:
                return new Triangle();
        default:
                return NULL;
        };
};
```


173

STL: kontenery

Fabryka obiektów – przykład:

- Za pomocą tej funkcji oraz metody **`read`** można dostarczyć funkcję, która pozwala odczytywać obiekty ze strumienia:

```
Figure* create(ifstream& is) {
  Figure::Type type;
  unsigned int t = 0;
  if(is >> t) {                          // odczyt wartości typu int
    type = static_cast<Figure::Type>(t); // rzutowanie t na Type
    Figure* obj = createObj(type);
    obj->read(is);
    return obj;
  }else return NULL;
};
```


..i to już, wszystko?

STL: kontenery

Fabryka obiektów – przykład:

- To rozwiązanie ma wady:
  - kiedy dodamy nowy typ danych, musimy też zmodyfikować funkcję **`createObj`** ☹
  - Brakuje kontroli poprawności wiązania identyfikatora z typem.
  - Zestaw identyfikatorów jest zasobem wspólnym dla całej hierarchii, przy modyfikacji zbioru klas musi też podlegać modyfikacji.

Potrzeba czegoś więcej..

STL: kontenery

Fabryka skalowalna obiektów – przykład:

Fabryka najpierw rejestruje pary: identyfikatory typu i przypisane im funkcje tworzące. Następnie jest gotowa do tworzenia obiektów na podstawie identyfikatora typu.

```
class FigFactory {
public:
  typedef Figure* (*CreateFig)(); // nowy typ: wskaźnik na funkcję tworzącą
  void registerFig(int id, CreateFig fun);  // rejestruje nowy typ
  Figure* create(int id); // tworzy obiekt na podstawie identyfikatora
private:
  typedef map<int, CreateFig> Creators; // nowy typ: kontener-mapa par
  Creators creators_;             // pole: kontener do przechowywania par
};
```


STL: kontenery

Fabryka skalowalna obiektów – przykład:

```
void FigFactory::registerFig(int id, CreateFig fun) {
    creators_.insert(map<int,CreateFig>::value_type(id, fun));
};                    // typedef pair<const Key, Type> value_type;


Figure* FigFactory::create(int id) {     //tworzy obiekt danego typu
    Creators::const_iterator i = creators_.find(id);
    if(i != creators_.end() )       //jeżeli znalazł odpowiedni wpis
        return (i->second)();           //wywołuje metodę fabryczną
    return 0L;   //zwraca pusty wskaźnik, gdy nieznany identyfikator
};
```


STL: kontenery

Fabryka skalowalna obiektów – przykład:

Autor klasy konkretnej musi też dostarczyć funkcję tworzącą obiekt danej klasy. Funkcja ta ma sygnaturę zdefiniowaną w fabryce i jest dostarczana do fabryki podczas rejestracji typu.

```
Figure* CreateSquare() {
        return new Square();
};
Figure* CreateCircle() {
        return new Circle();
};
Figure* CreateTriangle() {
        return new Triangle();
};
```


STL: kontenery

Fabryka obiektów – przykład:

- Jeszcze raz tworzymy funkcję, która pozwala odczytywać obiekty ze strumienia:

```
Figure* create(ifstream& is, FigFactory& factory) {
        unsigned int t = 0;
        if(is >> t) {
                Figure* obj = factory.create(t);
                obj->read(is);
                return obj;
        }else
                return NULL;
};
```


..i to już, wszystko?

STL: kontenery

Fabryka obiektów – przykład:

- Trzeba jeszcze pamiętać, aby na początku programu stworzyć fabrykę i zarejestrować odpowiednie typy

```
int main() {
        FigFactory factory;
        factory.registerFig(Figure::SQUARE, CreateSquare);
        factory.registerFig(Figure::CIRCLE, CreateCircle);
        factory.registerFig(Figure::TRIANGLE, CreateTriangle);

        /* .. */
};
```

To wszystko.

Chociaż, może nie..

STL: kontenery

Fabryka obiektów i wzorzec prototypu:

- wzorzec prototypu – pozwala na tworzenie kopii obiektu, jeżeli mamy dostępny wskaźnik lub referencję do klasy bazowej.
- Wykorzystuje mechanizm metod wirtualnych – przenosi odpowiedzialność za tworzenie obiektów do klas konkretnych.
- Klasa bazowa definiuje tylko czysto wirtualną metodę.

STL: kontenery

Fabryka obiektów i wzorzec prototypu:

- Dodajemy czysto wirtualną metodę **clone** do klasy bazowej:

```
class Figure {
public:
        enum Type {SQUARE, CIRCLE, TRIANGLE };
        virtual void write(ostream& os) const = 0;
        virtual void read(istream& is) = 0;
        virtual Figure* clone() const = 0;
};
```


STL: kontenery

Fabryka obiektów i wzorzec prototypu:

- Implementujemy metodę **clone** w klasach konkretnych:

```
class Square : public Figure {
public:
        void write(ostream& os) const {
                os << SQUARE;
                /* zapis pól */
        };
        void read(istream& is) {
                /* odczyt pól */
        };
        Figure* clone() const {
                return new Square(*this);
        };
};
```

Teraz zadaniem fabryki jest przechowywać obiekty wzorcowe, a nie funkcje fabryczne. Można w niej umieścić np. po kilka obiektów tego samego typu, ale w różnym stanie.

STL: kontenery

Fabryka obiektów i wzorzec prototypu:

```
class FigFactory2 {
        typedef map<int,Figure*> Prototypes;
        Prototypes prototypes_;
public:
    void registerFig(int id, Figure* f){
        prototypes_.insert(map<int,Figure*>::value_type(id, f));
    };
    Figure* create(int id){
        Prototypes::const_iterator i = prototypes_.find(id);
        if(i != prototypes_.end() ) //jeżeli znalazł odpowiedni wpis
            return (i->second)->clone(); //woła metodę clone
        return 0L; //zwraca pusty wskaźnik, gdy nieznany identyfikator
    };
};
```

Teraz to już naprawdę wszystko o fabrykach obiektów.

C++ - szablony

Metaprogramowanie

Metaprogramowanie

Tworzenie programów, które w wyniku ich przetwarzania do postaci wykonywalnej dostarczają kodów źródłowych (tzn. innych programów).

Narzędziem jest preprocesor oraz mechanizm szablonów: dostarczają instrukcji pozwalających na manipulowanie kodem źródłowym.

Algorytm budujący nowy kod jest wykonywany w momencie wywołania polecenia kompilacji.

186

Metaprogramowanie

Przykład 1:

```
template <unsigned n>
        struct Silnia {
        static const unsigned value = n*Silnia<n-1>::value;
};
template<> struct Silnia<0> { // specjalizacja jawna
        static const unsigned value = 1;
};

int main() {
        int i = Silnia<5>::value;
        cout << i;
        return 0;
};
```


187

Metaprogramowanie

Tworzenie metaprogramów może dawać *mało czytelny* kod (czasami).

Używanie metaprogramów daje z reguły *bardziej czytelny i elastyczny* kod.

Metaprogramy wykorzystują jedynie byty dostępne w czasie kompilacji (stałe całkowite, typy, itd.), dlatego rekurencja konkretyzacji szablonów jest powszechna w tych programach (nie można tworzyć zmiennych ani pętli).

Tworzenie mechanizmu wyjścia z rekurencji odbywa się przez specjalizację szablonu dla konkretnej wartości granicznej.

188

Metaprogramowanie

Przykład 2:

```
template <unsigned n>
inline double pow(double x) {
        return x* pow<n-1>(x);
};
template<> inline double pow<0>(double x) {
        return 1.0;
};

int main() {
        double b = 3.14;
        double a = pow<3>(b);
        cout << a;
        return 0;
};
```


189

Metaprogramowanie

Szablon **pow** : skraca zapis, zwiększa czytelność kodu oraz jego szybkość. Ale..

Jeżeli wykładnik jest dużą liczbą całkowitą, to utworzona przez szablon **pow** funkcja jest długa, wielkość kodu wynikowego wzrasta.

Ulepszenie: w szablonach można implementować złożenia funkcji. Np. x^n dla $n=2^m$ można wykonać jako m-krotne podnoszenie do kwadratu.

Dla n=1024 to daje (tylko) 10 operacji podnoszenia do kwadratu.

190

Metaprogramowanie

Potęga dowolnej liczby całkowitej dodatniej - pomysł:

1. Zapisać wykładnik n binarnie, np.: n=19, to: B = [0001 0011] gdzie $n = b_m 2^m + b_{m-1} 2^{m-1} + b_{m-2} 2^{m-2} + .. b_0 2^0$
2. Stąd dla przykładowej wartości 19: $n = b_4 2^4 + b_1 2^1 + b_0 2^0$
3. Wtedy x^n jest iloczynem składników $x^{(2m)}$, np.: $x^{19} = x^{16+2+1}$ tj.: $(((x^2)^2)^2)^2 * x^2 * x^1$
4. .. a tego właśnie potrzebowaliśmy.

191

Metaprogramowanie

```
template <unsigned n> double Power(double x) {
        return Power<2>(Power<n/2>(x)) *
                        Power<n%2>(x);
};
template<> double Power<2>(double x) {
        return x*x;
};
template<> double Power<1>(double x) {
        return x;
};
template<> double Power<0>(double x) {
        return 1;
};
/// …
double c = Power<19>(b);
```

n=19: P<2>(P<9>(x)) P<1>(x)
n=9: P<2>(P<4>(x)) P<1>(x)
n=4: P<2>(P<2>(x)) P<0>(x)
$P<2>(x) = x^2$
P<1>(x) = x

Stąd:
$(((x^2)^2)^2 \cdot x)^2 \cdot x$

$X^{19} = x^{16}x^2x^1$

Metaprogramowanie

Metaprogramy mogą dostarczać przybliżenia wartości funkcji matematycznych z założoną dokładnością.

Przykład 3: funkcja wykładnicza e^x

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \qquad -\infty < x < \infty$$

tzw. szereg Taylora.

Korzystając z szablonów **`Power`** i **`Silnia`** możemy dostarczyć szablon, który będzie generował wyrażenie zawierające początkowe wyrazy tego szeregu.

Metaprogramowanie

Przykład 3: funkcja wykładnicza z zadaną dokładnością

```
template <unsigned int N>
inline double Exp(double x) {
        return Exp<N-1>(x)+Power<N>(x)/Silnia<N>::value;
};
template <> inline double Exp<0>(double x) {
        return 1.0;
};
// …
double d = Exp<5>(a);
```


Metaprogramowanie

Przykład 3: funkcja wykładnicza z zadaną dokładnością

```
template <unsigned int N>
inline double Exp(double x) {
        return Exp<N-1>(x)+Power<N>(x)/Silnia<N>::value;
};
template <> inline double Exp<0>(double x) {
        return 1.0;
};
```

- $e^x < 3 > = e^x < 2 > + \frac{x^3}{3!}$,
- $e^x < 3 > = e^x < 1 > + \frac{x^2}{2!} + \frac{x^3}{3!}$,
- $e^x < 3 >= e^x < 0 > + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$,