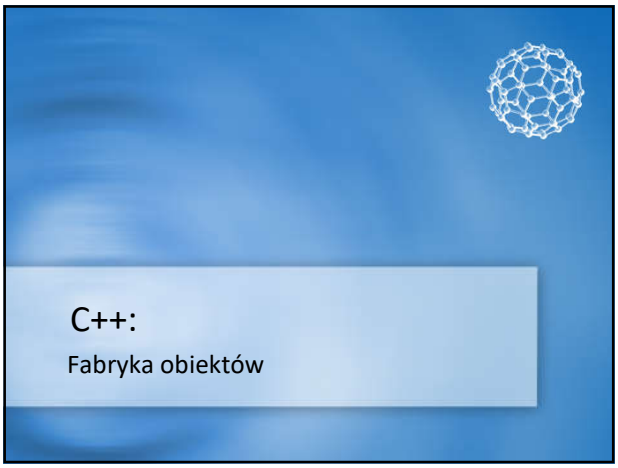


168

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..

180

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.

181

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;
};
```


182

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.

183

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.

184

C++ - szablony

Metaprogramowanie

185

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

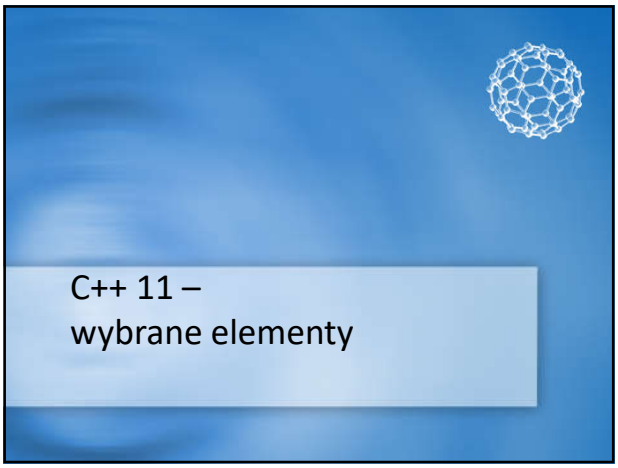
```
template <unsigned int N>
inline double Exp(double x) {
        return Exp<N-1>(x)+Power<N>(x)/Silnia<N>::value;
};
template <> inline double Exp<0>(double x) {
        return 1.0;
};
// …
double d = Exp<5>(a);
```


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;
};
```

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



1

C++11

Lista rozszerzeń C++11 obecnych w VC2013:

1. Referencje do *rvalue*,
2. Jednolite inicjowanie i delegowanie konstruktorów,
3. Konstruktory delegujące
4. Jednolita inicjalizacja
5. Klasa `initializer_list`
6. `static_assert`
7. Domyślne wartości parametrów szablonu dla szablonów funkcji
8. Szablony ze zmienną liczbą parametrów
9. `typedef` i aliasy
10. Szablony zewnętrzne
11. Pętla **`for`** oparta na zakresie,
12. Dedukcja typów danych,
13. Sprytne wskaźniki,
14. **`nullptr`**
15. Kontenery mieszające (kontenery z haszowaniem).

2

C++xx

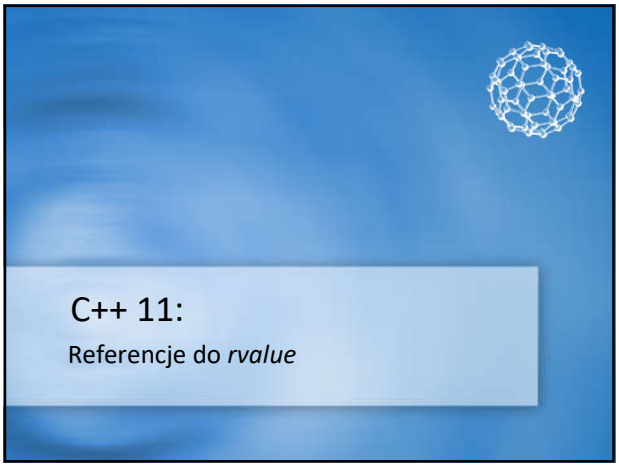
- Do MSVC w kolejnych wersjach środowiska dodawane są kolejne rozszerzenia składni z C++14, C++17, C++20

Więcej informacji na stronie:

Microsoft C++ language conformance table

https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance?view=msvc-160

3



4

C++11: referencje do *rvalue*

lvalue i rvalue

lvalue - wyrażenie które odnosi się do obszaru pamięci (obiektu, który ma nazwę) i pozwala na pozyskanie adresu tego obszaru za pomocą operatora &

rvalue - wyrażenie nie będące *lvalue*

```
int foobar();

int j = 0;
j = 42;                  // ok, 42 jest rvalue
j = foobar();            // ok, foobar() jest rvalue
int* p2 = &foobar();     // błąd, nie można pobrać adresu
```


5

C++11: referencje do *rvalue*

Przykładowy problem

Przyjmijmy, że mamy klasę w której jedno z pól jest wskaźnikiem **`m_pResource`** np. do pewnego kontenera. Przeciążony operator kopiowania dla tej klasy:

```
X& X::operator=(X const & rhs) {
  // [...]
  // zniszcz zasób wskazywany przez m_pResource
  // sklonuj to, na co wskazuje rhs.m_pResource
  // dołącz klon do m_pResource.
  // [...]
}
```

Teraz można wywołać kod:

```
X foo();   // dana jest pewna funkcja foo..

X x;
x = foo(); // wywołanie przeciążonego operatora przypisania
```


6

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Zdecydowanie mniejszy nakład pracy dałby taki przeciążony operator:

```
X& X::operator=(X const & rhs) {
// [...]
// swap ( m_pResource , rhs.m_pResource )
// [...]
```

To jest właśnie **semantyka przenoszenia** (*move semantics*) – *pozwolenie pewnemu obiektowi, pod pewnymi warunkami przejąć kontrolę nad zewnętrznymi zasobami innego obiektu. Dzięki temu unikamy kosztownych kopiowań.*

Żeby `swap` zadziałało, potrzebujemy, aby argument operatora był **referencją**.

Jeżeli argument jest *lvalue* – nie ma problemu. A jeżeli jest *rvalue*?..

.. wtedy przydałby się drugi referencyjny typ danych, który byłby wybierany, kiedy argumentem jest *rvalue*: `&&`

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Rozwiązanie:

```
void foo(X& x) { .. };     // wersja z referencją do lvalue
void foo(X&& x) { .. };    // wersja z referencją do rvalue

X x;
X foobar() { .. ; return ..; };

foo(x);         // argument jest lvalue: wywołuje foo(X&)
foo(foobar()); // argument jest rvalue: wywołuje foo(X&&)
```

Na etapie kompilacji rozstrzyga się, jaki argument zostanie podany w wywołaniu.

C++11: referencje do *rvalue*

Przykładowy problem (*move semantics*)

Rozwiązanie dla przeciążonego operatora przypisania:

```
X& X::operator=(X const & rhs); // klasyczna implementacja

X& X::operator=(X&& rhs) {
  // Move semantics: zamiana zawartości pomiędzy this i rhs
  // ..
  return *this;
}
```

Uwaga: tak, to prawda, że `&&` można stosować w dowolnej funkcji, ale przyjmuje się stosowanie wyłącznie do przeciążonych operatorów przypisania i konstruktorów kopiujących.

C++11: referencje do *rvalue*

Semantyka przenoszenia danych

Czy zmienna referencyjna do *rvalue* jest traktowana jako *rvalue*?

Przykład:

```
void foo(X&& x) {
  X anotherX = x; // który operator przypisania zadziała?
  // ...
}
```

(`x` jest referencją do *rvalue*, więc powinien `operator=(X&& x)`..)

Nie.

Prosta zasada mówi: jeżeli coś ma nazwę, to jest *lvalue*. W przeciwnym razie – jest *rvalue*.

C++ 11:

Konstruktory delegujące

C++11: konstruktory delegujące

Delegowanie:

Problem: w kodzie napisanym w C++98 jest dana klasa, która ma kilka konstruktorów. Często część kroków inicjalizujących powtarza się w każdym z nich, np.:

```
class A{
public:
   A(): num1(0), num2(0) {average=(num1+num2)/2;}
   A(int i): num1(i), num2(0) {average=(num1+num2)/2;}
   A(int i, int j): num1(i), num2(j) {average=(num1+num2)/2;}
private:
   int num1;
   int num2;
   int average;
};
```

C++11: konstruktory delegujące

Delegowanie:

W C++11 dostajemy możliwość stworzenia jednego konstruktora docelowego i kilku delegujących:

```
class A{
   public:
      A(): A(0){}              // A() deleguje do A(0)
      A(int i): A(i, 0){}      // A(int i) deleguje do A(i,0)
      A(int i, int j) { num1=i; num2=j; average=(num1+num2)/2; }
   private:
      int num1;
      int num2;
      int average;
   };
```

Uwaga: w konstruktorach delegujących nie wolno w liście inicjalizatorów konstruktora dodawać instrukcji inicjalizacji pól

C++11: konstruktory delegujące

Delegowanie i wyjątki:

Wywołanie docelowego konstruktora może być zamknięte w bloku **try**:

```
class A{
public:
   A();
   A(int i);
   A(int i, int j);
private:
   int num1;
   int num2;
   int average;
};
```

```
A:: A()try: A(0) {
   cout << "A() body"<< endl;
}
catch(...) {
   cout << "A() catch"<< endl;
}

A::A(int i) try: A(i, 0){
   cout << "A(int i) body"<< endl;
}
catch(...) {
   cout << "A(int i) catch"<< endl;
}
```


C++11: konstruktory delegujące

Delegowanie i wyjątki:

Wywołanie docelowego konstruktora może być zamknięte w bloku **try**:

```
class A{
public:
   A();
   A(int i);
   A(int i, int j);
private:
   int num1;
   int num2;
   int average;
};
```

```
A::A(int i, int j) try {
   num1=i;
   num2=j;
   average=(num1+num2)/2;
   cout << "A(int i, int j) body"<< endl;
   throw 1;
}
catch(...) {
   cout << "A(int i, int j) catch"<< endl;
}
```


C++ 11:

Jednolita inicjalizacja

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Możliwa do zastosowania do dowolnej klasy, struktury czy unii, o ile istnieje (zdefiniowany jawnie lub *implicite*) konstruktor domyślny:

```
class class_a {
public:
    class_a() {} // konstruktor domyślny
    class_a(string str) :
       m_string{ str } {}
    class_a(string str, double dbl) :
       m_string{ str },
       m_double{ dbl } {}

double m_double;
string m_string;
};
```

```
int main()
{
    class_a c1{};
    class_a c1_1;
    class_a c2{ "ww" };
    class_a c2_1("xx");
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}
```

Kolejność argumentów jak w konstruktorze, ponieważ ta inicjalizacja de facto wywołuje konstruktor

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Jeżeli nie zdefiniowano żadnego konstruktora, domyślny porządek argumentów taki jak pól w deklaracji klasy/struktury/unii:

```
class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};
```

```
int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // błąd
    class_d d5{ "string", 'c', 2.0 }; // błąd
}
```

C++11: jednolita inicjalizacja

Jednolita inicjalizacja za pomocą nawiasów klamrowych:

Można jej używać w dowolnym miejscu kodu, np.:

```
class_d* cf = new class_d{4.5};
```

albo

```
kr->add_d({ 4.5 });
```

albo

```
return { 4.5 };
```


C++11: initializer_list

Lista obiektów do inicjalizacji:

Klasa reprezentuje listę obiektów jednego, określonego typu, które mogą być używane w konstruktorze i innych inicjalizujących kontekstach

```
#include <initializer_list>
..
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "tak!" << endl; // spodziewamy się "tak!"
```

Standardowe klasy kontenerów z STL, a także np. **string** mają zdefiniowane konstruktory przyjmujące jako argument listę obiektów.

C++11: initializer_list

Lista obiektów do inicjalizacji:

Można używać list w konstruktorach własnych klas:

```
class C1
{
public:
 list<int> L;
 C1(const std::initializer_list<int>& v) : L(v.begin(), v.end()) {}
};

int main() {
 C1 c{ 3, 5, 7, 11 }; // inicjalizujemy listą wartości
 ..
```


C++11: initializer_list

Lista obiektów do inicjalizacji:

Można używać list do inicjalizacji klas kontenerów wyrażeniami z nawiasami klamrowymi

```
vector<int> v1{ 9, 10, 11 };

map<int, string> m1{ {1, "a"}, {2, "b"} };

string s{ 'a', 'b', 'c' };
```