C++ - przeciążanie operatorów

Automatyczna konwersja typów

2 sposoby:

1. może być realizowana za pomocą konstruktora, pobierającego jako jedyny argument obiekt lub referencje do obiektu innego typu, lub
2. może być realizowana przez napisanie specjalnych przeciążonych operatorów

C++ - przeciążanie operatorów

```
class Jeden {
      public:
      Jeden() { }
};
class Dwa {
      public:
      Dwa(const Jeden&) { } // sposób 1:
};                          // specjalny konstruktor
void f(Dwa) { }

int main(int argc, char *argv[]){
    Jeden J;
    f(J); // kompilator to akceptuje, bo w Dwa zadeklarowano
          // konstruktor, którego argumentem jest obiekt typu Jeden
```


C++ - przeciążanie operatorów

Może się zdarzyć, że nie chcemy, aby konstruktor był wywoływany automatycznie, jeżeli zajdą takie okoliczności, w których mógłby być tak wywołany

Aby zablokować automat, używamy słowa **explicit**

C++ - przeciążanie operatorów

```
class Jeden {
      public:
      Jeden() { }
};
class Dwa {
      public:
      explicit Dwa(const Jeden&) { }
};
void f(Dwa) { }

int main(int argc, char *argv[])
{
    Jeden J;
    // f(J) - ta instrukcja nie zadziała, bo zablokowano automatyczną konwersję
    f(Dwa(J));
```


C++ - przeciążanie operatorów

Sposób 2:
Automatyczna konwersja typów za pomocą operatora konwersji

Można utworzyć metodę składową, pobierającą aktualny typ i przekształcającą go na typ docelowy, wykorzystując słowo kluczowe **operator**

W takiej składowej słowo **operator** poprzedza nazwę typu do którego ma zostać dokonana konwersja, zamiast symbolu operatora

C++ - przeciążanie operatorów

```
class Trzy {
      int i;
      public:
      Trzy(int ii): i(ii) { }
};

class Cztery {
      int x;
      public:
      Cztery(int xx): x(xx) { }
   operator Trzy() const
      {return Trzy(x); }
};

void g(Trzy) { }

int main(int argc, char
  *argv[])
{

 Cztery cz(1);
 g(cz); // wywołanie operatora
 g(1);  // wywołanie Trzy(1.0)
```

C++ - przeciążanie operatorów

Automatyczna konwersja typów – podsumowanie:

1. może być realizowana za pomocą konstruktora, pobierającego jako jedyny argument obiekt lub referencje do obiektu innego typu.
   - Utworzenie jednoargumentowego konstruktora zawsze definiuje automatyczną konwersję typów – nawet jeżeli argumentów jest więcej, ale pozostałe posiadają wartości domyślne. Jeżeli chcemy tego uniknąć, deklarujemy ten konstruktor jako explicit.
   - To klasa docelowa musi mieć odpowiedni konstruktor:
     `Dwa::Dwa(const Jeden&) { }`
     przykład: `void f(Dwa) { }`
2. może być realizowana przez napisanie specjalnych przeciążonych operatorów
   - To klasa źródłowa musi mieć odpowiedni operator konwersji:
     `Cztery::operator Trzy() const {return Trzy(x); }`
     przykład: `void g(Trzy) { }`

C++ - przeciążanie operatorów

Automatyczna konwersja typów – podsumowanie:

	konstruktor	operator
KlasaA	KlasaA::KlasaA(const KlasaB&k)	KlasaA::operator KlasaB()
KlasaB	KlasaB::KlasaB(const KlasaA&k)	KlasaB::operator KlasaA()

Poprawne zestawy do konwersji KlasaA ↔ KlasaB:

Niepoprawne zestawy (*zdublowana konwersja tylko w jedną stronę*):

C++ - przeciążanie operatorów

Automatyczna konwersja typów – podsumowanie:

Deklarowanie dwóch sposobów konwersji jednocześnie:

Jeżeli:

klasa X posiada możliwość przekształcenia obiektu w obiekt klasy Y za pomocą operatora Y(),

a jednocześnie:

klasa Y posiada konstruktor, pobierający pojedynczy argument typu X

to:

obydwa mechanizmy reprezentują tę samą konwersje typów i w razie potrzeby kompilator nie wie, której użyć – wtedy zgłasza błąd dwuznaczności.

C++ - przeciążanie operatorów

Przewaga globalnych przeciążonych operatorów nad operatorami będącymi składowymi klas – podsumowanie:

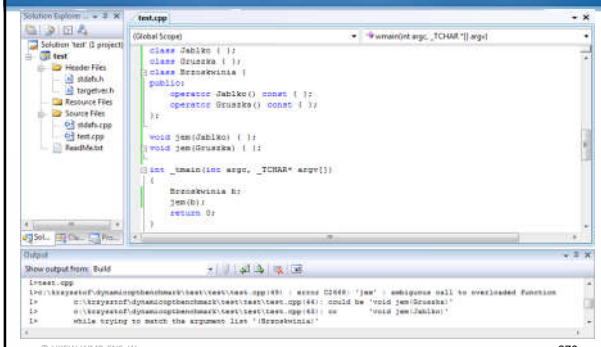
- w przypadku operatorów globalnych konwersja typów może zostać zastosowana do każdego z argumentów
- w przypadku składowej argument po lewej stronie operatora musi być odpowiedniego typu

C++ - przeciążanie operatorów

```
class Integer {
    int i;
public:
    Integer(int ii): i(ii) {}
 const Integer operator+(const Integer& rv) {
          return Integer(i+rv.i);
    }
friend const Integer operator/(const Integer
    &arg_lewy, const Integer &arg_prawy);
}
```

```
const Integer operator/(const Integer
    &arg_lewy, const Integer &arg_prawy) {
      if (arg_prawy.i == 0)
        return Integer(INT_MAX);
      else
        return
    Integer(arg_lewy.i/arg_prawy.i);
}
int main(int argc, char *argv[])
{
   Integer I(1), J(2), K(3);
   I = J + 5;
   //I = 5 + J;          // to się nie uda
   I = K/2;
   I = 10/K;           // tak jest OK
```


C++ - przeciążanie operatorów



279

C++ - szablony

- Szablony, nazywane też wzorcami, pozwalają na abstrakcyjne definiowanie funkcji i klas w terminach parametrów, którymi są typy danych
- Na podstawie szablonu kompilator sam generuje definicje konkretnych funkcji i klas, kiedy będą potrzebne
- Standardowa biblioteka C++ bazuje na szablonach i dlatego jest nazywana STL: Standard Template Library

280

281

C++ - szablony

Szablony funkcji

Zdarza się, że wiele funkcji jest do siebie bardzo podobnych w działaniu – różnią się tylko typem argumentu, który przyjmują

Np. funkcja która z tablicy wybiera element o najmniejszej wartości: algorytm szukania jest wspólny dla większości typów danych.

Jednak jeżeli chcemy znajdować najmniejszą wartość dla tablic z elementami typu: **`int`**, **`double`**, lub **`Integer`** (*zdefiniowana klasa, dla której istnieje przeciążony operator porównania*), to dla każdego z typów należy napisać na nowo całą funkcję

Jeżeli te funkcje będą fragmentem biblioteki, to za każdym razem kiedy będziemy potrzebowali dopisać funkcję dla nowego typu danych w tablicy, będzie trzeba rekompilować bibliotekę i udostępniać jej użytkownikom nowy release

To jest uciążliwe. Lepiej zrobić tę funkcję raz wykorzystując szablony.

282

C++ - szablony

Kompilator tworzy tyle wersji funkcji wg wskazanego szablonu, ile programista zażąda w swoim programie: będą takie same co do działania, ale będą różniły się co do typów argumentów wywołania, typów zmiennych lokalnych czy typu wartości zwracanej

Szablony można tworzyć nawet dla typów, które w chwili pisania szablonu jeszcze w ogóle nie istniały

Aby poinformować kompilator, że kod, który piszemy reprezentuje szablon, należy użyć słowa kluczowego **`template`**
– kod, który znajduje się poniżej tego słowa jest definicją wzorca funkcji lub klasy

283

C++ - szablony

Przykład:

```
template <class T1, class T2>
T2& moja_funkcja(T1 arg1, T2* arg2)
{
  … // Tutaj jakiś kod funkcji
    // wykorzystującej typy T1 i T2
  return *arg2;
}
```

W nawiasach kątowych lista parametrów formalnych wzorca, każdy poprzedzony słowem **`class`**, lub **`typename`** (**`typename`** wprowadzono później)

Parametry zwyczajowo nazywa się z wykorzystaniem T, np. T1, T2

Następnie występuje definicja wzorca wykorzystująca T1 i T2 tam, gdzie powinna występować nazwa konkretnego typu

284

C++ - szablony

Na podstawie typów argumentów kompilator ma zgadnąć, który szablon i jak należy wykorzystać:

```
template <class T1, class T2>
T2& moja_funkcja(T1 arg1, T2* arg2) {
    return *arg2;
}
int main(int argc, char *argv[])
{
    int a = 9;
    double b = 10.2, c = 3.14;
    moja_funkcja(a, &b);  // konkretyzacja wzorca na
                          // double& moja_funkcja(int arg1, double* arg2)
    moja_funkcja(a, &c);  // tu nie ma konkretyzacji - funkcja już jest.
    moja_funkcja(b, &a);  // konkretyzacja wzorca na
                          //int& moja_funkcja(double arg1, int* arg2)
```


285

C++ - szablony

Słowo kluczowe **class** w linii:

```
template <class T1, class T2>
```

nie oznacza, że T1 i T2 mogą być tylko klasami (jak widać na przykładzie); mogą to być dowolne typy (wbudowane lub definiowane przez użytkownika)

Dla czytelności w późniejszych wersjach kompilatorów zastąpiono to słowo słowem **typename**

```
template < typename T1, typename T2>
```

286

C++ - szablony

Przeciążanie szablonów funkcji:

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
                  return k1 < k2 ? k2 : k1 ;
}
double wiekszy(const double& d1, const double& d2)
                  return d1 < d2 ? d2 : d1 ;
}
int main(int argc, char *argv[])
{
    int a, b=0, c=1;
    a = wiekszy(b,c);  // która wersja funkcji zostanie użyta?
                // Istnieje standardowa konwersja int na double, ale
                // istnieje też szablon wg którego można utworzyć
                // właściwą funkcję
    double x, y=0.3, z=2.14;
    x = wiekszy(y,z);  // która wersja funkcji zostanie użyta?
```


287

C++ - szablony

Wskazywanie wartości parametru szablonu

```
class Integer {
    int i;
public:
    Integer(int ii): i(ii) {}
    bool operator<(
          const Integer& rv) const
    {
          return i < rv.i;
    }
    bool operator>(
          const Integer& rv) const
    {
          return i > rv.i;
    }
    …
};
```

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
    return k1 < k2 ? k2 : k1 ;
}

int main(int argc, char *argv[])
{
    Integer I1(0), I2(1), I3(2);
    I1 = wiekszy<Integer>(I2, I3);
```

Jeżeli nie ma jednoznacznej interpretacji, można za nazwą szablonu w nawiasach kątowych podać listę typów, które należy w tym miejscu zastosować

288

C++ - szablony

Informacja o typie zmiennej

Testując szablony czasem niezbędne jest sprawdzenie, jakiego typu są zmienne, których szablon używa (czy np. kompilator sam podjął decyzję o konwersji zmiennej na podobny typ, etc)

Do sprawdzenia służy operator **`typeid`**

Operator zwraca obiekt typu **`const std::type_info`**

Klasa ta ma pożyteczną metodę **`name()`**

289

C++ - szablony

Chciałoby się napisać:

```
std::type_info typ = typeid(a);
printf("%s\n",typ.name());
```

Jednak autorzy biblioteki zastrzegli konstruktor kopiujący oraz operator przypisania jako **`private`** uniemożliwiając de facto tworzenie w programie własnych obiektów tego typu z obiektów zwracanych przez operator `typeid`:

```
 private:
    /// Assigning type_info is not supported.  Made private.
    type_info& operator=(const type_info&);
    type_info(const type_info&);
```

Dlatego aby wywołać metodę **name** można wyłącznie napisać tak:

```
printf("%s\n",typeid(k1).name());
```


290

C++ - szablony

Porada programistyczna

Na etapie testowania programu można w szablonie funkcji i w funkcji dodać instrukcje ujawniające typ danych, które są przetwarzane:

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
                printf("Szablon dla: %s\n",typeid(k1).name());
                return k1 < k2 ? k2 : k1 ;
}
double wiekszy(const double& d1, const double& d2) {
                printf("Dla double: %s\n",typeid(d1).name());
                return d1 < d2 ? d2 : d1 ;
}
```

Visual Studio

część 2:

SZABLONY KLAS

C++ - szablony

Szablony klas

Na tej samej zasadzie co szablony funkcji można tworzyć szablony klas

Składnia jest analogiczna:

```
template <typename T, typename M>
class Klasa {
  // tu używamy typów T i M
  …
};
```

Aby utworzyć obiekt nie możemy napisać:

```
Klasa K;
```

bo nie wiadomo, jakie typy przypisać parametrom M i T, a kompilator nie ma szansy domyślić się. Wskazanie wartości parametrów jest konieczne.

C++ - szablony

Szablony klas

Deklaracja obiektów:

```
template <typename T, typename M>
class Klasa {
  …        // tu używamy typów T i M
};

Klasa<double, int> K;
Klasa<int, Integer> I;
```


C++ - szablony

Szablony klas

Jeżeli chcemy, żeby metody tej klasy zostały definiowane poza szablonem klasy, to musimy tam prawidłowo napisać nazwę szablonu:

```
Klasa::Klasa();              // konstruktor domyślny – Źle!

template <typename T, typename M>
Klasa<T, M>::Klasa();        // kontruktor domyślny – Dobrze!
```


C++ - szablony

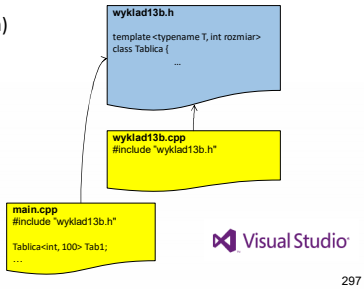
Parametry szablonu klasy mogą być też wartościami określonego typu, np. :

```
template <typename T, int rozmiar >
class Klasa {
  // tu używamy typu T i zmiennej rozmiar
  …
};
Klasa<int, 100> Tab;
Klasa<double, 2000> *Tab2;
Klasa<double, 3000> *Tab3;
```

`Tab`, `Tab2` i `Tab3` są obiektami różnych typów, dlatego kompilator nie zaakceptuje takich instrukcji jak przepisanie wartości między wskaźnikami, itp.

C++ - szablony

Projekt składający się z pliku main.cpp i biblioteki (para plików: .cpp i .h)



297

C++ - szablony

Kompilacja szablonów

Zawsze kiedy ukonkretniany jest szablon klasy, kod definicji takiej klasy dla danej specjalizacji jest generowany wraz z metodami składowymi wywoływanymi w programie – i tylko z tymi *rzeczywiście* wywoływanymi metodami składowymi.

Jest to unikanie nadmiarowego kodu.

To daje możliwość elastycznego projektowania szablonów, wykorzystującego różne właściwości różnych konkretnych typów.

298

C++ - szablony

```
class Pierwsza {
public:
  void f() {}
};

class Druga {
public:
  void g() {}
};
```

```
template<typename T>
class Trzecia {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};

int main() {

  Trzecia<Pierwsza> TP;
  TP.a(); // nie tworzy Trzecia<Pierwsza>::b()

  Trzecia<Druga> TD;
  TD.b(); // nie tworzy Trzecia<Druga>::a()
}
```


299

C++ - szablony

Szablony vs. pliki nagłówkowe

Zasada: deklaracje i kompletne definicje szablonów funkcji i klas umieszczamy w pliku nagłówkowym (i tylko tam).

Uzasadnienie:

Skoro szablony są wykorzystywane do generowania kodu tylko gdy w kodzie wystąpi wykorzystanie szablonu, to:

1. Umieszczenie kodu metod i funkcji w pliku cpp sprawi, że będzie on kompilowany oddzielnie – nie będzie w tym pliku instrukcji wykorzystujących szablon, które należą do innych funkcji i metod, a więc kompilator nie będzie wiedział jakich konkretyzacji dokonać
2. Tam, gdzie będą próby wykorzystania szablonu, kompilator będzie miał same nagłówki klas i funkcji, ale bez definicji, więc nie będzie potrafił skonkretyzować żądanych wersji szablonu

Efekt uboczny: kod szablonów jest jawny dla każdego użytkownika naszej biblioteki (pliki nagłówkowe są dostarczane zawsze w postaci źródłowej) ☹

300