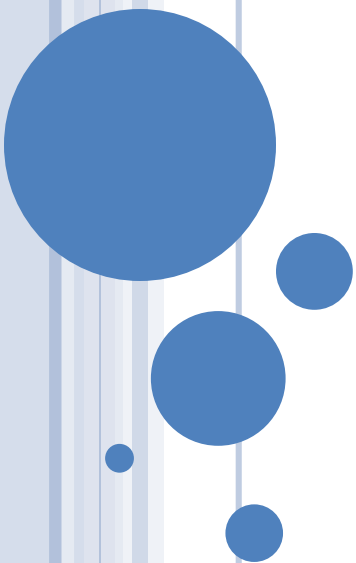




KURS JĘZYKA C++ – WYKŁAD 8 (18.04.2018)

Konwersje

SPIS TREŚCI

- Tradycyjne operatory rzutowania
- Konstruktory konwertujące
- Operatory konwersji
- Rzutowanie `static_cast`
- Rzutowanie `const_cast`
- Rzutowanie `reinterpret_cast`
- Rzutowanie `dynamic_cast`
- RTTI – operator `typeid()`
- Automatyczne określanie typu (`auto`)
- Wydobycie typu wyrażenia (`decltype`)



RZUTOWANIE

- Rzutowanie to zmiana typu danej (powstaje nowy obiekt innego typu) albo zmiana interpretacji danych (obiekt się nie zmienia ale traktujemy go w kategoriach innego typu).
- W C++ w stosunku do C została zastrzona kontrola typów – na przykład, gdy prześlemy funkcji zmienną o innym typie dostaniemy błąd od kompilatora (główna zmiana dotyczy wskaźników rzutowanych na typ `void*` i w drugą stronę).



TRADYCYJNE

OPERATORY RZUTOWANIA

- Tradycyjne operatory rzutowania jawnie przekształcają typ danych.
- Tradycyjne operatory konwersji mogą przyjmować dwie formy:
(typ) wyrażenie
typ(wyrażenie)
Przykłady:
`(int) 3.1415926` // forma rzutowania
`double(7*11+5)` // forma konstruktorowa
- Operacja jawnej konwersji typów jest niebezpieczna i należy ją stosować bardzo ostrożnie (tylko w razie konieczności).
- Zaleca się używać konstruktorowej formy przy rzutowaniu tradycyjnym.



TRADYCYJNE

OPERATORY RZUTOWANIA

- Kompilator umie przekształcać na siebie wszystkie typy podstawowe.
- Operator rzutowania eliminuje ostrzeżenia kompilatora przy przekształcaniu typów podstawowych.
- Kompilator nie będzie generował ostrzeżeń w przypadku konwersji na typach podstawowych, w których mamy do czynienia z promocją (konwersje niejawne).
- Przykłady:

```
const double e = 2.71828182845904523;  
int x = (int)e; // wymagana konwersja  
double y = 2*x+1; // konwersja niejawna
```



KONSTRUKTORY KONWERTUJĄCE

- Konstruktor konwertujący to konstruktor bez deklaratora `explicit`, który można wywołać z jednym parametrem:

```
K::K (typ x) { /*...*/ } // typ!=K
```

- Konstruktorów konwertujących może być wiele w jednej klasie.
- Deklarator `explicit` zabrania używać konstruktora konwertującego niejawnie.



KONSTRUKTORY KONWERTUJĄCE

- Przykład konstruktora konwertującego i jego niejawnego użycia:

```
class zespolona {  
    double re, im;  
public:  
    zespolona (double r=0, double i=0);  
    // ...  
};  
// ...  
zespolona a;  
zespolona b = zespolona(1.2); // jawna konwersja  
zespolona c = 3.4; // niejawna konwersja  
zespolona d = (zespolona)5.6; // rzutowanie  
zespolona e = static_cast<zespolona>(7.8);  
zespolona f(9.0, 0.9);
```



OPERATORY KONWERSJI

- Operator konwersji ma następującą postać:
`operator typ ();`
- Operator konwersji ma pustą listę argumentów i nie ma określonego typu wyniku (typ wyniku jest określony poprzez nazwę tego operatora).
- Operator konwersji musi być funkcją składową w klasie.
- Operator konwersji jest dziedziczony.
- Operator konwersji może być wirtualny.
- Operatorów konwersji może być wiele w jednej klasie.
- Przy operatorach konwersji można użyć słowa kluczowego `explicit` aby uniknąć konwersji niejawnej.



OPERATOR `static_cast`

- Rzutowanie `static_cast` działa tak jak rzutowanie tradycyjne – jeśli jest zdefiniowana operacja rzutowania to zostanie ona wykonana.
- Operator rzutowania `static_cast` ma następującą postać:
`static_cast<typ>(wyrażenie)`
- Rzutowania `static_cast` używa się do:
 - konwersji podstawowych typów liczbowych,
 - wyliczenia do typu całkowitego,
 - konwersji typów pokrewnych (zmiana typu wskaźnikowego czy referencyjnego w tej samej hierarchii klas – rzutowanie do góry albo w dół hierarchii dziedziczenia),
 - konwersji zdefiniowanych przez użytkownika.
- Typ obiektu na który rzutujemy musi być znany w momencie kompilacji.
- Operator rzutowania `static_cast` działa na etapie kompilacji za pomocą dostępnych operatorów konwersji.

RZUTOWANIE `CONST_CAST`

- Rzutowanie to pozwala dodać albo zlikwidować deklaratorem `const` lub `volatile` w typie wyrażenia (ale nie pozwala zmienić typu głównego).
- Operator rzutowania `const_cast` ma następującą postać:
`const_cast<typ>(wyrażenie)`
przy czym *typ* powinno być wskaźnikiem, referencją lub wskaźnikiem do składowej.
- Operator rzutowania `const_cast` działa na etapie kompilacji.



RZUTOWANIE `reinterpret_cast`

- Operator rzutowania `reinterpret_cast` ma następującą postać:

`reinterpret_cast<typ> (wyrażenie)`

przy czym `typ` powinno być wskaźnikiem, referencją lub typem porządkowym (znaki, liczby całkowite, typ boolowski, wykliczenia).

- Rzutowanie to ma zmienić interpretację typu wyrażenia (kompilator nie sprawdza sensu tego rzutowania).
- Operator rzutowania `reinterpret_cast` tworzy wartość nowego typu, który ma ten sam wzorzec bitowy co podane wyrażenie.
- Rzutowanie to nie gwarantuje przenośności.
- Operator rzutowania `reinterpret_cast` działa na etapie kompilacji.

RZUTOWANIE `DYNAMIC_CAST`

- Operator rzutowania `dynamic_cast` ma następującą postać:
`dynamic_cast<typ>(wyrażenie)`
przy czym *wyrażenie* powinno być wskaźnikiem lub referencją do typu polimorficznego.
- Rzutowanie to wykonuje się w trakcie działania programu.
- `dynamic_cast<T*>(p)` zwraca wskaźnik typu T^* gdy obiekt wskazywany przez p jest typu T lub ma unikatową klasę bazową typu T (w przeciwnym przypadku zwraca `nullptr`).
- `dynamic_cast<T&>(r)` zwraca referencję typu $T\&$ gdy obiekt wskazywany przez r jest typu T lub ma unikatową klasę bazową typu T (w przeciwnym przypadku rzuca wyjątek `bad_cast`).

RTTI

- Operator `typeid()` zwraca referencję do obiektu opisującego typ wyrażenia w nawiasach (można też podać nazwę typu).
- Klasa `type_info` zdefiniowana w `<typeinfo>` służy do opisu typu danych lub wyrażeń.
- W klasie `type_info` są zdefiniowane operatory `==` i `!=` do porównywania informacji o typie.
- W klasie `type_info` jest zdefiniowana metoda `name()` dostarczająca nazwę typu w postaci `const char *`.



AUTOMATYCZNE OKREŚLANIE TYPU

- W definicji zmiennej z jawnym inicjowaniem można użyć słowa kluczowego `auto` – można w ten sposób utworzyć zmienną o typie takim, jak typ inicjującego wyrażenia.
- Przykład 1:
`auto jakasZmienna = L"To jest tekst";`
Typ `jakasZmienna` jest programiście łatwiej napisać słowo `auto` niż `const wchar_t *` (taki jak dla literału tekstowego).
- Przykład 2:
`auto innaZmienna = boost::bind(&Funkcja, _2, _1, Obiekt);`
Typem `innaZmienna` może być cokolwiek zwracanego przez pewną funkcję szablonową pod `boost::bind` dla danych argumentów, typ ten jest łatwy do określenia przez kompilator, natomiast dla użytkownika jest to trudne.



AUTOMATYCZNE OKREŚLANIE TYPU

- Prztykład 3:

Typ `auto` jest przydatny przy ograniczaniu rozwlekłości kodu.

Zamiast pisać:

```
for (vector<int>::const_iterator itr =  
myvec.begin(); itr != myvec.end(); ++itr) ...
```

Programista może użyć krótszego zapisu:

```
for (auto itr = myvec.begin(); itr !=  
myvec.end(); ++itr) ...
```



WYDOBYCIE TYPU WYRAŻENIA

- Operator `decltype` pozwala na uzyskanie typu wyrażenia.
- Jego głównym przeznaczeniem tego operatora jest programowanie uogólnione, w którym często trudno, jeśli w ogóle jest to możliwe, określić typy zależne od parametrów szablonu.
- Typ określony za pomocą operatora `decltype` zgadza się z typem obiektu lub funkcji zadeklarowanym w kodzie źródłowym.
- Podobnie jak w przypadku operatora `sizeof`, operand `decltype` nie jest wykonywany.



WYDOBYCIE TYPU WYRAŻENIA

- Przykłady:

```
const int& foo();  
int i;  
struct A { double x; };  
const A *a = new A();  
decltype(i) x2; // typ to int  
decltype(foo()) x1 = i; // typ to const int&  
decltype(a->x) x3; // typ to double  
decltype((a->x)) x4; // typ to const double&
```

Wyrażenie w nawiasie `(a->x)` nie jest ani id-wyrażeniem ani dostępem do członków klasy, a stąd nie oznacza nazwanego obiektu. Ponieważ to wyrażenie jest l-wartością, jego wydedukowany typ jest referencją do typu wyrażenia, czyli `const double&`.

