Przeładowywanie operatorów, dziedziczenie

Przeładowywane operatorów

- Jest bardzo wygodną metodą, zamiast definiować funkcje typu `add` itp. możemy użyć odpowiednich operatorów
- Przeładowanie operatora dokonuje się definiując własną funkcję o nazwie `operatorX`, gdzie `X` oznacza symbol interesującego nas operatora
  - Może być funkcją składową
  - Może być globalną funkcją – wyjątki!!!
- Lista operatorów, które można przeładowywać
  - `+, -, *, /, %, ^, &, |, ~, ! , =, <, >, +=, -=, *=, /=, %=, ^=, &=, |=, <<, >>, >>=, <<=, ==, !=, <=, >=, &&, ||, ++, --, ‘,’ , ->*, ->, new, delete, () , []`
  - Tylko jako metody: `=`, `()`, `[]`, `->`
- Natomiast następujące operatory nie mogą być przeładowywane
  - `.`, `.*`, `::`, `?:`

Przeładowywane operatorów...

- Nie można wymyślać swoich operatorów
  - Np. **`**`**
- Nie można zmieniać priorytetów operatorów
- Nie można też zmienić argumentowości operatorów, czyli tego czy są jedno- czy dwuargumentowe
- Nie można również zmieniać łączności operatorów
- Dla każdej klasy następujące operatory są generowane automatycznie
  - **`=`**, **`&`**(jednoargumentowy – pobranie adresu), **`new`**, **`,`** , **`delete`**

Funkcja operatorowa jako funkcja składowa

- Jeżeli definiujemy składową funkcje operatorową to przyjmuje ona zawsze o jeden mniej argument niż ta sam funkcja napisana w postaci funkcji globalnej
- Funkcja ta nie może być typu `static`, bo w jej działaniu bierze udział wskaźnik `this`
- Nie mogą istnieć dwie funkcje operatorowe pracujące na tych samych argumentach (zdefiniowane jako funkcja globalna i funkcja składowa)
- W tym wypadku po lewej stronie operatora zawsze musi stać obiekt klasy, dla której ten operator jest zdefiniowany
  - `Obiekt1 + Obiekt2;`
  - `Obiekt1.operator+(Obiekt2);`
- Przykład cpp_6.1

Funkcja operatorowa jako funkcja globalna

- Nie musi być funkcją zaprzyjaźnioną
- Jeżeli wymaga dostępu do zmiennych prywatnych to musi być zaprzyjaźniona
- Dzięki globalnym funkcjom operatorowym można zdefiniować operatory do klas już istniejących np. bibliotecznych
  - W przypadku takich klas muszą one udostępniać odpowiedni interfejs (w szczególności dostęp do składowych)
- Nie ma takiego ograniczenia jak dla funkcji operatorowej zdefiniowanej jako metoda
- Przykład cpp_6.2

Przemienność

- Funkcja operatorowa będąca funkcją składową klasy wymaga, aby obiekt stojący po lewej stronie operatora był obiektem tej klasy
  - `Fraction Fraction::operator*(int i);`
  - `aFraction = bFraction * 2; //OK`
  - `aFraction = 2 * bFraction; //Błąd`
- Zwykła funkcja globalna nie ma tego ograniczenia
  - `Fraction operator*(int i, Fraction K);`
    `//Argumenty mogą być w odwrotnej kolejności`
  - Oczywiście przy wywołaniach z niezgodnością typów muszą być zdefiniowane odpowiednie konwersje

Operatory, które muszą być funkcjami składowymi

- Operator przypisania `=`
  - Generowany automatyczne przez kompilator, tak że przepisuje obiekt składnik po składniku
    - Nie zawsze dobry – wskaźniki
  - Nie jest generowany automatycznie w sytuacjach
    - Jeżeli klasa ma składnik `const`
    - Jeżeli klasa ma składnik będący referencją
    - Jeżeli klasa ma składową klasę, w której operator przypisania jest prywatny
    - Jeżeli klasa ma klasę podstawową z prywatnym operatorem przypisania
  - Nie jest dziedziczony
  - Na ogół zawiera
    - Cześć destruktorową
    - Cześć konstruktorową
- Przykład cpp_6.3
- Test cpp_6.01

Nowy operator=

- Jest to funkcja składowa niestatyczna i nieszablonowa o nazwie operator=
  - `class_name & class_name :: operator= ( class_name && )`
  - Funkcja wywoływana jest kiedy pojawia się po lewej stronie =, a po jego prawe stoi rvalue
  - „Kradnie" zasoby obiektu stojącego po prawej stronie
    - np. dla std::string zostawia po prawej stronie obiekt pusty
- Generowana automatycznie w sytuacji kiedy
  - Nie ma konstruktora kopiującego (niedomyślnego)
  - Nie ma konstruktora przenoszalnego (niedomyślnego)
  - Nie ma kopiującego operatora=
  - Nie ma destruktora
  - Generowany jest wtedy publiczny i inline `T& T::operator=(T&&)`
- Jeżeli jest „trywialny" wykorzystuje do przenoszenia `std::memmove`
  - Trywialny znaczy
    - Generowany automatycznie
    - T nie posiada funkcji wirtualnych i wirtualnych klas bazowych
    - Trywialny jest przenaszalny operator= dla klas bazowych oraz składników
- Przykład 6.3a1

Operatory które muszą być funkcjami składowymi...

- Operator `[]`
  - Przeładowany operator `[]` powinien mieć działanie podobne do działania w stosunku do typów wbudowanych
    - Z tego powodu powinien być zadeklarowany
      `klasa& klasa::operator[](unsigned i);`
      czyli zwracać referencję do pojedynczego elementu tablicy o indeksie `i`
    - Możliwe będzie wtedy wykonanie
      - `a = tab[i];`
      - `tab[i] = a;`
- Przykład cpp_6.3a

Operatory które muszą być funkcjami składowymi...

- Operator `()`
  - Może przyjmować dowolną liczbę parametrów
  - Może posłużyć do indeksowanie wielowymiarowych tablic
    - `tab(1,2,3);`
  - Może też upraszczać zapis, nie musimy wywoływać funkcji tylko wystarczy sam operator `()`
  - Bardzo przydatny operator przy wykorzystaniu funktorów z algorytmami STL
  - Przykład cpp_6.3b i cpp_6.3c
- Operator `->`
  - Rzadko używany
  - Przydaje się gdy piszemy klasę, której obiekty pełnią rolę podobną do wskaźników
  - Wykorzystany między innymi przy tworzeniu klasy `unique_ptr` z STL-a
  - Zrobić przykład samodzielnie !!!

Operatory pre i post ++ --

- Operatory preinkrementacji ++ i -- działają jak zwykłe inne operatory jednoargumentowe
- Problem z operatorami postinkrementacji ++ i --, których w normalny sposób nie da się przeładować
  - Rozwiązano ten problem deklarując te operatory jak dwuargumentowe
  - `Point Point::operator++(int)`
  - Tworzony jest obiekt tymczasowy, o czym należy pamiętać przy optymalizacji
- Przykład cpp_6.4

Operatory << i >>

- Przy przeładowywaniu tych operatorów w stosunku do klasy **`iostream`** możemy je zdefiniować tylko jako globalne funkcje
  - Precyzyjniej w stosunku do klas **`istream`** oraz **`ostream`**
    - Będzie działać wtedy na standardowy WE/WY oraz z plikami
- Funkcja operatorowa musi pracować na zmiennych lub metodach globalnych
- Ewentualnie musi być zaprzyjaźniona z naszą klasą, jeżeli ma pracować na zmiennych prywatnych
- Przykład cpp_6.4

Operator `new` i `delete`

- W stosunku do klas funkcje przeładowujące te operatory są zawsze typu `static`, nawet jeśli tego nie zadeklarujemy
  - Przydają się kiedy chcemy uzyskać jakąś dodatkową funkcjonalność np. statystykę
  - Tworzymy obiekty w predefiniowanej wcześniej pamięci
  - Używamy niestandardowej biblioteki do tworzenia nowych obiektów
- Istnieją również globalne wersje tych operatorów
  - `void* operator new(size_t sz)`
  - `void operator delete(void* m)`
- Przykład cpp_6.3d, cpp_6.3e i cpp_6.3f

Podsumowanie C++03

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment
Nothing	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted
copy constructor	not declared	defaulted	user declared	defaulted
copy assignment	defaulted	defaulted	defaulted	user declared

https://howardhinnant.github.io/

Podsumowanie C++11 + ...

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

https://howardhinnant.github.io/

Dziedziczenie

- Dziedziczenie to technika umożliwiająca zdefiniowanie nowej klasy z wykorzystaniem klasy już istniejącej
- Nowa klasa staje się automatycznie nowym typem danych
- Klasę z której dziedziczymy nazywamy klasą bazową lub podstawową
- Klasa która odziedzicza składniki i metody po innej klasie nazywana jest klasą pochodną

Możliwości klasy pochodnej

- Definiowanie dodatkowych danych składowych
  - Nie da się usunąć składników już istniejących
- Definiowanie nowych metod
  - Nie da się usunąć metod, ale można je uczynić niedostępnymi z poziomu nowej klasy
- Definiowanie (ponowne) metod, które już istnieją w klasie podstawowej
  - W szczególności związany z tym aspektem polimorfizm, a nie zwykłe zasłanianie nazw
- Klasa pochodna tworzy w pewnym sensie zagnieżdżony zakres
  - Powoduje np. zasłonięcie składników lub/i metod

Sposób zapisu dziedziczenia

- W najprostszym przypadku dziedziczenie zapisuje się następująco
  - ```
    class Nowa : public Baza {
    ... //nowe elementy klasy }
    ```
  - ```
    struct Nowa : Baza {
    ... //nowe elementy klasy }
    ```
- Tworzymy klasę **Nowa**, która otrzymuje wszystkie składniki i metody klasy **Baza**
  - Nie zawsze mamy dostęp do wszystkich składników lub/i metod
  - Natomiast niezależnie czy mamy dostęp czy też nie to dziedziczone jest wszystko
- Przykład cpp_6.5

Dostęp do składników klasy podstawowej

- Prywatne składniki klasy podstawowej
  - Do takich składników klasa pochodna nie ma bezpośredniego dostępu
  - Możliwy jest dostęp poprzez odziedziczone funkcje składowe jeżeli oczywiście nie są one prywatne
- Nieprywatne składniki i metody klasy bazowej są dostępne dla klasy pochodnej
  - Tutaj pojawia się dopiero różnica w dostępie **`private`** i **`protected`**
  - Inne klasy mają dostęp tylko do składników i metod publicznych, natomiast klasy pochodne mają również dostęp do danych i funkcji w zakresie **`protected`**
  - Słowo **`protected`** zostało wymyślone na potrzeby dziedziczenia

Dostęp do składników klasy pochodnej

- Dostęp do odziedziczonych składników w klasie pochodnej zależy od sposobu dziedziczenia
  - Przy dziedziczenia **`public`** (**`class Nowa : public Baza`**) odziedziczone składniki **`public`** i **`protected`** pozostają takie niezmienione
  - Przy dziedziczenia **`protected`** (**`class Nowa : protected Baza`**) odziedziczone składniki zarówno **`public`** i **`protected`** stają się **`protected`**
  - Przy dziedziczenia **`private`** (**`class Nowa : private Baza`**) odziedziczone składniki staję się prywatną własnością klasy pochodnej
  - Domyślnie (bez podania sposobu) dziedziczenie jest prywatne dla klas
    - A dla struktur publiczne (tak jak dostęp)

Deklaracje dostępu

- Jeżeli chcemy ukryć większość składników i metod z klasy podstawowej, ale pozostawić kilka widocznych to możemy zastosować deklarację dostępu
  - Należy wtedy w klasie pochodnej wyspecyfikować po etykiecie **public** lub **protected** tylko nazwy interesujących nas składowych i metod
    - ```
      public: //protected:
              Baza::skladowa;
              Baza::metoda;
      ```
  - Nie rozróżniamy wtedy przeładowanych nazw
- Deklaracja dostępu może jedynie powtórzyć dostęp, nie może go zmieniać
- Przykład cpp_6.6
      ```

Elementy niedziedziczone

- Należy pamiętać, że nie zostają odziedziczone w klasie pochodnej
  - Konstruktory
    - Umożliwia inicjalizowanie dodatkowych składników, które umieszczono w klasie pochodnej
  - Operator przypisania (=)
    - Należy pamiętać, że jeśli nie zdefiniujmy tego operatora w klasie pochodnej to w razie potrzebny zostanie wygenerowany automatycznie (będzie kopiował składnik po składniku)
      - Ale w częściowo inteligentny sposób, tzn. jeśli w klasie podstawowej jest ten operator zdefiniowany to zostanie użyty
  - Destruktor
    - Często unieważnia działanie konstruktorów, przez co jeśli konstrukotry nie są dziedziczone to destruktor też nie

Dziedziczenie wielopokoleniowe

- Nie ma ograniczeń w tworzeniu kolejnych klas pochodnych
  - ```
    class A {
    ...};
    class B: public A {
    ...};
    class C: public B {
    ...};
    ```
- W tym momencie ujawnia się znaczenie rodzaju dziedziczenia (`public`, `protected`, `private`)
- Przykład cpp_6.7
    ```

Kolejność wywoływania konstruktorów

- W klasie pochodnej w pewnym sensie tkwi klasa podstawowa
- Tworzenie klasy pochodnej to dobudowywanie elementów do klasy podstawowej
  - Dlatego pracują dwa konstruktory (klasy podstawowej i pochodnej)
- Do pracy najpierw rusza konstruktor klasy podstawowej, a dopiero potem klasy pochodnej
- Przykład cpp_6.8

Konstruktor klasy pochodnej

- Konstruktory klasy pochodnej tworzy się w znany już sposób
- Należy pamiętać, że na liście inicjalizacyjnej konstruktora klasy pochodnej trzeba umieść konstruktor klasy podstawowej chyba, że
  - Klasa podstawowa nie ma żadnego konstruktora
  - Ma konstruktory, a wśród nich jest konstruktor domyślny
  - ```
    class B : public A {
      public:
            B() : A(param) {}
    }
    ```
  - Na liście inicjalizacyjnej umieszcza się tylko konstruktory klas podstawowych bezpośrednich, czyli znajdujących się o jeden poziom wyżej w hierarchii
- Przykład cpp_6.9
    ```

Przepisywanie składnik po składniku (operator =)

- Jeżeli klasa podstawowa ma operator przypisania (zdefiniowany i nieprywatny) to wygenerowany operator przypisania dla klasy pochodnej skorzysta z niego
- Jeżeli klasa zawiera jakiś składnik `const` lub będący referencją to operator przypisania nie jest generowany automatycznie
  - Wtedy musimy stworzyć taki operator samemu
- Tak jak przy zwykłej klasie nie ma sensu definiować operatora przypisania jeżeli zostanie on wygenerowany przez kompilator i będzie działał dobrze

Konstruktor kopiujący

- Jeżeli nie zdefiniujemy konstruktora kopiującego to klasa pochodna wygeneruje go sobie sama
- Konstruktor kopiujący nie zostanie stworzony gdy
  - Klasa zawiera jako składniki inne klasy, które mają niedostępny konstruktor kopiujący
  - Podobnie jest z klasą podstawową
  - W takich sytuacja należy samemu go zdefiniować
- Kompilator wygeneruje konstruktor kopiujący dla obiektów `const` tylko wtedy, gdy wszystkie klasy podstawowe i składniki tej klasy zagwarantują argumentowi nietykalność
  - Czyli wszędzie konstruktor kopiujący powinien być zadeklarowany wg. schematu `Klasa::Klasa(const Klasa& K)`
  - Przykład cpp_6.10
- Jeżeli nie ma faktycznej potrzeby to nie definiujemy tego konstruktora, pozwalamy kompilatorowi na automatyczną jego generację