Rozszerzenia składni C w środowisku C++

1

Rozszerzenia składni C *vs.* C++

- Nowe symbole komentarzy:
  `// …. tu jest komentarz`
- Możliwość mieszania instrukcji i deklaracji zmiennych w jednym bloku kodu
- Możliwość deklarowania zmiennych np. w instrukcji sterującej pętli **`for`**, np.:
  ```
  for (int i=0; i<100; i++)
  ```
- Nowy typ podstawowy **`bool`**. Zmienne **`bool`** przyjmują dwie wartości **`true`** i **`false`**. Ze względu na wsteczną zgodność jednak pozostawiono domyślną konwersję typu **`bool`** na **`int`**.
  Pisząc w C można jednak było sobie samemu zrobić typ **`bool`** *:*
  ```
  typedef enum {TRUE = 1, FALSE = 0} bool;
  ```
- Deklaracja zmiennych strukturalnych bez słowa **`struct`**, np.:
  ```
  struct Struktura { int a; char c; };
  Struktura X;
  ```


2

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

`void fun(int &k)`

Taki zapis oznacza, że funkcji zostanie przekazane odniesienie do zmiennej. Kopia, która zostanie utworzona, będzie korzystała z podanego adresu zewnętrznej zmiennej.

Przykład deklaracji:

```
void fun(int &k){
  k += 2;
};
```

Jak to działa?

3

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

`void fun(int &k)`

- W czasie wykonania funkcji nazwa parametru ‘k’ będzie tylko inną nazwą zmiennej podanej w argumencie wywołania
- Wywołanie:

```
int m = 1;
fun(m);
```

spowoduje, że zmienna ‘m’ po zakończeniu działania funkcji będzie miała wartość 3.

4

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

Uwaga:

Wywołanie obydwu poniższych funkcji:

```
void fun(int &k)
void fun(int k)
```

wygląda identycznie:

```
fun(m);
```

Natomiast takie wywołanie:

```
fun(m+n);
```

jest dozwolone tylko w przypadku funkcji: **`void fun(int k)`**

5

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentu przez wartość, odniesienie i wskaźnik

```
void fun(int x, int& y, int *z) {
    x = 2*x;
    y = 3*y;
    *z = 4* (*z);
}
```

`x` – przekazywany przez wartość – pracuje na kopii
`y` – przekazywany przez odniesienie – pracuje na oryginale
`z` – przekazywany przez wskaźnik – pracuje na adresie oryginału

Wywołanie:

```
int a=1,b=2,c=3;
fun(a,b,&c);
```


6

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentu przez wartość, odniesienie i wskaźnik

```
void fun(int x, int& y1, int &y2) {
    x = 2*x;
    y1 = 3*y1;
    y1 = y2;
    y1 = 2*y1;
}
```

Raz zainicjalizowana zmienna referencyjna reprezentuje tę samą zmienną aż do końca swego istnienia,

tzn. nie można sprawić, żeby zmienna referencyjna zaczęła od pewnego momentu działania kodu reprezentować inną zmienna.

W szczególności, instrukcja **`y1 = y2`** nie spowoduje, że zmienne referencyjne **`y1`** i **`y2`** zaczną reprezentować tę samą zmienną.

Rozszerzenia składni C *vs.* C++

Przekazywanie tablic w argumencie wywołania funkcji przez odniesienie

```
void funtab(double t[])      // nieznany rozmiar
void funref(double (&t)[6]) // znany rozmiar
```

Wywołanie:

```
double tab[] = { 0 }; /* tablica zer */
funtab(tab);
funref(tab);
```

`funtab` – **`t`** jest typu `double*`; rozmiar tablicy nie jest znany (jest dowolny)

`funref` – **`t`** jest typu *„odniesienie do sześcioelementowej tablicy elementów typu* `double`*”*

Rozszerzenia składni C *vs.* C++

Przekazywanie tablic w argumencie wywołania funkcji przez odniesienie

```
void funref(double (&t)[6])
          /* Nawias: (&t) jest konieczny. */

    double (&t)[6]          double &t[6]
```

Bez nawiasu jest to tablica odniesień do zmiennych typu double – coś takiego nie istnieje w C (!).

- Przy używaniu operatora odniesienia podawanie wymiaru tablicy jest konieczne. *Tablica czteroelementowa to nie to samo co tablica pięcioelementowa.*
- Wywołując **`funtab`** możemy podać tablicę dowolnego rozmiaru. W **`funref`** musi być to tablica dokładnie 6-elementowa.

Rozszerzenia składni C *vs.* C++

Funkcje statyczne

funkcje zadeklarowane jako **`static`**, np.:

```
static int inc(int &x) {
        return ++x;
}
```

Funkcje te są widoczne tylko w obrębie pliku, w którym zostały zadeklarowane, w przeciwieństwie do pozostałych funkcji, które są widoczne we wszystkich plikach należących do projektu (domyślnie: funkcji globalnych).

Rozszerzenia składni C *vs.* C++

Funkcje rozwijane

zadeklarowane z wykorzystaniem słowa kluczowego **`inline`**, np.:

```
inline int pow2(int x) {
        return x*x;
}
```

W miejscach wywołania funkcji kompilator umieszcza kod funkcji zamiast jej wywołania. W ten sposób powstaje szybszy kod (*jeżeli funkcja zawiera niewielki kod, to przyrost objętości programu jest minimalny, za to otrzymujemy zysk w postaci większej szybkości*)

Rozszerzenia składni C *vs.* C++

Przeciążanie funkcji

zadeklarowanie kilku funkcji o tej samej nazwie, ale różnym zestawie argumentów wywołania , np.:

```
int fun(int a);
int fun(double a);
int fun(char a);
```

Aby funkcję były uważane za różne, muszą różnić się **sygnaturą**. W skład sygnatury wchodzą: nazwa funkcji i lista typów argumentów.

Typ zwracany nie należy do sygnatury.

Rozszerzenia składni C *vs.* C++

Przeciążanie funkcji

Różnienie się sygnaturą nie jest warunkiem wystarczającym, np.:

```
void fun(int k);
void fun(int &k);
```

różnią się sygnaturą, ale to nie wystarcza, aby jednoznacznie rozstrzygnąć, którą funkcję wywołać w przypadku:

```
int k = 0;
fun(k);
```


Rozszerzenia składni C *vs.* C++

Dynamiczny przydział pamięci:

Funkcje biblioteczne malloc, calloc, realloc i free zostają zastąpione operatorami C++:
new i delete

Ogólna postać wyrażenia alokującego:

new *typ*;

gdzie *typ* jest typem zmiennej, która ma zostać zaalokowana na stercie, np.:

```
new int;          new char[100];
```

Wyrażenie new zwraca wskaźnik do nowo utworzonego obiektu dokładnie takiego typu, o jakiego przydzielenie został poproszony, np.:

```
char *buffer = new char[100];
```


Rozszerzenia składni C *vs.* C++

Dynamiczny przydział pamięci:

Ogólna postać wyrażenia zwalniającego:

delete *wskaźnik*;

gdzie *wskaźnik* przechowuje adres do zmiennej dynamicznej i jest dokładnie takiego typu, jak ten zaalokowany na stercie, np.:

```
delete x;   delete []buffer;
```

Uwaga: jeżeli była alokowana tablica, to należy ją zwalniać używając składni właściwej dla tablic

Rozszerzenia składni C *vs.* C++

Stałe wyliczeniowe w C:

umożliwiają definiowanie typów, wraz ze zmiennymi do których można przypisać wartości

```
enum dzien {PON, WT, SR, CZW, PT, SOB, NIE};
enum srodekTransportu {SAMOCHOD, TRAMWAJ, AUTOBUS,
ROWER, NOGI};
```

Przykład:

```
enum KOLOR { czerwony,zielony,niebieski};
enum KOLOR ulubionyKolor;
ulubionyKolor = czerwony;
```


Rozszerzenia składni C *vs.* C++

Stałe wyliczeniowe w C++:

Nie wymagane już jest pisanie słowa enum, np.:

```
enum KOLOR ulubionyKolor = czerwony;  // OK. w C i C++
KOLOR nielubianyKolor = seledynowy;   // OK. tylko w C++
```

enum już nie jest tożsamy z typem int (brak konwersji), np.:

```
ulubionyKolor  = 2;                   // OK. tylko w C
```


Różne właściwości

Funkcje tak samo jak zmienne mają swoje miejsce w pamięci, gdzie są zapisane. Można więc uzyskać ich adres.

Podobnie jak adres tablicy jest zwracany przez jej nazwę, podaną bez nawiasu kwadratowego, adres funkcji uzyskuje się za pomocą nazwy funkcji pozbawionej listy argumentów.

Można tworzyć tablice wskaźników do funkcji i wywoływać funkcje nie po nazwie, a po numerze w tablicy

Różne właściwości

Wskaźniki do funkcji - przykład:

```
int suma (int a, int b) {
  return a+b;
}

int main () {
  int (*wsk_suma)(int, int);      // zmienna wskaźnikowa
  wsk_suma = suma;          // przypisanie adresu funkcji
  printf("4+5=%d\n", (*wsk_suma)(4,5));
  return 0;
}
```

gdzie:

nazwa zmiennej: wsk_suma

typ wskazywany: int … (int, int)

19

Różne właściwości

Tablice wskaźników do funkcji - przykład:

```
void ErrMsg() { printf("Blad\n"); };
void OKMsg()  { printf("OK\n");   };
void YesMsg() { printf("Tak\n");  };
void NoMsg()  { printf("Nie\n");  };
void (*func_table[])() = {ErrMsg, OKMsg, YesMsg, NoMsg };

int main () {
  srand( (unsigned)time( NULL ) );
  double r = rand();
  int i = floor(4*r/RAND_MAX);
  (*func_table[i])();
  return 0;
}          void fun(int i, void (*func_table[])() );
```


20

OBIEKTOWE METODY INŻYNIERII OPROGRAMOWANIA

21

Obiektowe metody inżynierii oprogramowania

1. Analiza obiektowa (OOA – Object Oriented Analysis)
2. Projektowanie obiektowe (OOD – Object Oriented Design)
3. Programowanie obiektowe (OOP – Object Oriented Programming)

Peter Coad, Edward Yourdon

22

ANALIZA OBIEKTOWA

23

Analiza obiektowa

Analiza obiektowa
służy zbudowaniu *modelu rzeczywistości* wg pewnych zasad.

Trzymanie się tych zasad:

- daje spójną reprezentację stanowiącą podstawę analizy (co budować) i projektowania (jak budować),
- pozwala na identyfikację wspólnych cech atrybutów i usług,
- pozwala na budowę specyfikacji poddających się zmianom,
- pozwala na powtórne wykorzystanie wyników analizy dla rodzin systemów,
- pozwala na lepsze wzajemne zrozumienie analityka i eksperta w danej dziedzinie zastosowania.

24

Analiza obiektowa

Zarządzanie złożonością w analizie:

- **Przyjęcie powszechnie stosowanych metod organizacji**
  – tworzenie klas obiektów i rozróżnianie ich; rozróżnienie między obiektem a jego atrybutami; rozróżnienie między całym obiektem a jego składowymi.
  Abstrakcja proceduralna
  – pomijanie niektórych szczegółów rzeczy lub procesów wybierając tylko pewną istotną część na danym poziomie uogólnienia; definiujemy atrybuty oraz usługi, które mają wyłączność na manipulowanie tymi atrybutami.

25

Analiza obiektowa

Zarządzanie złożonością w analizie:

- **Hermetyzacja** – zasada używana przy budowie całościowej struktury programu:
  - każda składowa programu powinna zamykać w sobie jedną decyzję projektową;
  - styk z każdym modułem powinien być zdefiniowany tak, aby odkrywać możliwie mało ze swej wewnętrznej struktury.
- **Dziedziczenie**
  - uproszczenie definicji klas podobnych do już definiowanych przez wykorzystanie tych zdefiniowanych w definicji nowych;
  - opisuje generalizację i specjalizację czyniąc wspólne atrybuty i usługi jawnymi wewnątrz hierarchii klas.

26

Analiza obiektowa

Zarządzanie złożonością w analizie:

- **Skojarzenie**
  – wiązanie ze sobą idei podobnych lub rzeczy które zdarzają się w tym samym czasie lub w podobnych okolicznościach.
- **Komunikaty**
  – porozumiewanie się między obiektami.
- **Skala**
  – pokazanie relacji między obiektami (części do całości) dla wyobrażenia sobie np. dużego lub bardzo małego obiektu.

27

Analiza obiektowa

Zarządzanie złożonością w analizie:

- **Kategorie zachowania**
  - reakcja na zdarzenie,
  - zachowanie ze względu na podobieństwo ewolucji (zmiany w czasie),
  - zachowanie ze względu na podobieństwo funkcji.

28

Analiza obiektowa

Metody analizy:

1. inne niż obiektowe
   - Rozkład funkcjonalny
   - Metoda przepływu danych (*Data Flow Diagrams*)
   - Modelowanie informacji (*Entity Relationship Diagrams*)
2. podejście obiektowe
   … (o tym będzie dalej)

29

Analiza obiektowa

Rozkład funkcjonalny (*nieobiektowy*)

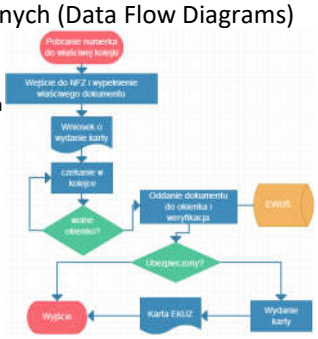
Dekompozycja algorytmiczna –
podział algorytmu na odrębne czynności: każdy projektowany moduł programowy systemu określa znaczący krok w procesie przetwarzania.

30

Analiza obiektowa

Metoda przepływu danych (Data Flow Diagrams) (*nieobiektowy*)

Graficzna prezentacja, tzw. diagram przepływu danych zawiera następujące rodzaje elementów:
- funkcje (procesy),
- magazyny danych,
- źródła i odbiorcy,
- przepływy (znaki pokazujące kierunek przesyłu danych)



31

Analiza obiektowa

Modelowanie informacji (Entity Relationship Diagrams) (*nieobiektowy*)

Graficzna prezentacja logicznej struktury danych w bazie danych.

Występujące pojęcia: **encje** (grupy lub kategorie danych) oraz **atrybuty** (podgrupy wewnątrz encji)).

Relacje wewnątrz modelu:

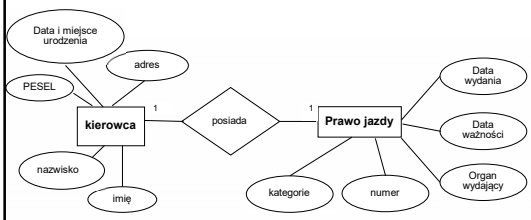
1. Obowiązkowe
2. Opcjonalne
3. Wielu-do-wielu
4. Jeden-do-wielu
5. Jeden-do-jeden
6. Rekurencyjne

32

Analiza obiektowa

Modelowanie informacji (Entity Relationship Diagrams) (*nieobiektowy*)

Graficzna prezentacja logicznej struktury danych w bazie danych:



33

Analiza obiektowa

Podejście obiektowe

Rodzaj dekompozycji, w której zamiast wyszczególniać czynności wyodrębnia się obiekty, na których dokonywane mogą być operacje.

Różnice:

- **Dekompozycja algorytmiczna** – opisuje kolejność czynności
- **dekompozycja obiektowa** – opisuje uczestniczące w czynnościach obiekty oraz wzajemne oddziaływanie obiektów na siebie.

Dekompozycja algorytmiczna i obiektowa są sobie przeciwstawne – nie można ich przeprowadzić jednocześnie.

34

Analiza obiektowa

Podejście obiektowe – składowe modelu:

- Pojęcia
- Obiekty
- Związki miedzy pojęciami
- Atrybuty obiektów
- Usługi obiektów
- Komunikaty między obiektami

35

Analiza obiektowa

Pojęcia – są środkiem służącym do rozpoznawania:

- Materialne – osoba, ołówek, samochód
- Niematerialne – czas, jakość, firma
- Role – doktor, pacjent, właściciel
- Opinie – wydajna praca, wysoka płaca, dobry przykład
- Relacyjne – małżeństwo, posiadanie
- Zdarzenia – sprzedaż, zakup, załamanie rynku
- Inne – zestaw, liczba, ikona, obraz, sygnał, proces

Pojęcia pozwalają nadać znaczenie obiektom w naszym świecie.

36

Analiza obiektowa

Trójka pojęciowa:

1. Symboliczna reprezentacja, np. : nazwa, ikonka, znaczek

2. Intensja – treść pojęcia, pełna definicja pojęcia

3. Ekstensja – zakres pojęcia, zbiór wszystkich rzeczy i wyobrażeń abstrakcyjnych, do których stosuje się dane pojęcie.

Analiza obiektowa

Typ obiektowy
– opis obiektu z jednolitym zbiorem atrybutów i usług, zawierający opis tworzenia nowych obiektów w klasie

Obiekty
– coś, do czego da się zastosować jakieś pojęcie. Obiekt jest egzemplarzem pojęcia (instancją).

Struktura i zachowanie obiektu są określone przez pojęcia, które się do niego odnoszą:

Pojęcie ↔ typ obiektowy

Obiekt to kapsułka ze zdefiniowanymi wartościami atrybutów i wyłącznie na nich działającymi usługami. Większość obiektów istnieje tylko przez pewien okres.

Analiza obiektowa

Struktury - związki między typami obiektowymi

Odwzorowania:

np. „zatrudnia” przypisuje obiekt typu „organizacja” związanym z nim obiektom typu „zatrudniony”. Odwzorowania mogą być jedno- lub wielowartościowe.

1. A jest zawsze związane z jednym B
2. A jest zawsze związane z jednym lub wieloma B
3. A jest zawsze związane z żadnym lub tylko jednym B
4. A jest zawsze związane z żadnym, jednym lub wieloma B

Analiza obiektowa

Struktury - związki między typami obiektowymi

Relacje:

Mogą obejmować kilka typów obiektów,

np. „umowa o prace” jest typem związków, którego *krotki* są niezmiennymi parami obiektów typu „osoba” i „organizacja”

Analiza obiektowa

Podtypy i nadtypy obiektowe:

Nadtyp - typ obiektowy, którego zbiór instancji zawiera wszystkie instancje jednego lub więcej typów, oraz definicja jest ogólniejsza niż definicja innych typów

Podtyp – typ obiektowy, którego zbiór wszystkich instancji zawiera się w większym zbiorze, oraz definicja jest bardziej wyspecjalizowana niż definicja innego typu

Przykład:

organizm
|
zwierzę
|
ssak
|
pies

Analiza obiektowa

Atrybuty – dane (stan systemu), dla których każdy obiekt ma swoją własną wartość

Każdy typ obiektowy jest opisany przez atrybuty.

Atrybuty są szczegółowo opisane w specyfikacji obiektu.

Atrybuty opisują wartości trzymane w obiekcie, aby wyłącznie usługi tego obiektu mogły nimi manipulować. Atrybuty i specjalne usługi na nich działające traktujemy jako nierozłączną całość.

Jeżeli inny obiekt chce odczytać wartości w obiekcie lub działać na nich w inny sposób, musi zrobić to poprzez specyfikację powiązania odpowiadającego komunikatowi, który spowoduje realizację usługi zdefiniowanej dla tego projektu.

Analiza obiektowa

Definiowanie usług

Usługa – zdefiniowane zachowanie obiektu, które jest on zobowiązany przejawić.

Usługi stosują się do atrybutów obiektu.

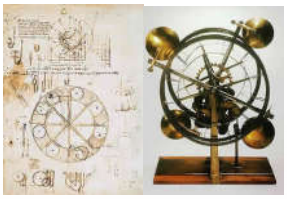
Definiując nowe usługi wyróżniamy następujące działania:

- Identyfikacja stanów obiektu
- Identyfikacja wymaganych usług
- Identyfikacja powiązań odpowiadających komunikatom
- Specyfikacja usług
- Zebranie w jedną całość dokumentacji analizy

43

Analiza obiektowa

- Zbiór powiązanych ze sobą obiektów o pewnej funkcjonalności można nazywać systemem.
- Raz uruchomiony system działa – obiekty oddziałują na siebie, zmieniając swój stan i tworząc nowe obiekty powiązane z tym systemem, a także usuwając już istniejące.

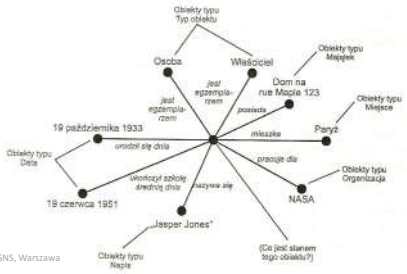


Perpetuum mobile –
projekt: Leonardo da Vinci ,
Muzeum w Monachium, Niemcy
http://www.leifiphysik.de/themenbereiche/arbeit-energie-und-leistung/geschichte

44

Analiza obiektowa

- Stan obiektu – kolekcja powiązań obiektu z innymi obiektami i typami obiektowymi w pewnym okresie czasu



45

Analiza obiektowa

„życie" obiektu:

- W systemie obiekty powstają, istniejąc zmieniają wielokrotnie swój stan, oraz są usuwane
- Zmiana stanu jest zmianą powiązań (atrybutów i/lub związków) obiektu
- Istotna zmiana stanu obiektu nazywana jest **zdarzeniem**
- Rodzaje zdarzeń: tworzenie, kończenie, łączenie/rozłączanie, inne
- Analityk nie potrzebuje wiedzy o każdym możliwym zdarzeniu: klasyfikuje je tworząc typy zdarzeń, np. zamiast definiowania zdarzenia „wczoraj mój pies ugryzł nowego listonosza w lewą nogę" tworzy typ: „pies ugryzł osobę"
- Zdarzenia są historią obiektów

46

Analiza obiektowa

„życie" systemu:

- Zmienność systemu wyraża się przez procesy
- Procesy odczytują i zmieniają stany obiektów
- **Operacje** to jednostki przetwarzania, z których składają się **procesy**
- Każda operacja wymaga obiektów, na których może operować (jeden lub więcej)
- Operacje mogą zwraca nowe obiekty
- Operacje są też związane z typami zdarzeniowymi – efektem operacji mogą być zdarzenia
- Ograniczenia, przy spełnieniu których operacja wykona się poprawnie, to warunki wstępne
- Ograniczenia jakie musza zachodzić w wyniku zakończenia operacji to warunki końcowe
- Operacje zegarowe: emitują zdarzenia - tyknięcia zegara
- Specyfikacja sposobu wykonania operacji to **metoda**

47