RTTI

Laboratorium Programowania Komputerów

Michał Sosna, Michał Rzepka

Wprowadzenie

– czym jest RTTI?

- RTTI - ang. *Runtime Type Identification*
- Mechanizm służący do dynamicznej identyfikacji typu
- Jednolity sposób sposób określania typu obiektu w czasie wykonywania programu
- Wprowadzony do języka C++ jako środek zaradczy na niezgodne ze sobą rozwiązania problemu implementowane w różnych bibliotekach

Do czego wykorzystuje się RTTI?

1. **Założenie:** dysponujemy hierarchią klas dziedziczących po wspólnej klasie bazowej.
2. Żeby wskazać na dowolny obiekt tej hierarchii, możemy użyć **wskaźnika typu klasy bazowej** do wskazywania na **dowolny obiekt** tej hierarchii. Wskaźnikowi na przodka można przypisać adres potomka.
3. Wywołujemy funkcję, która wybiera jedną z tych klas, tworzy obiekt wybranego typu i zwraca jego adres, który zostaje przekazany do wskaźnika typu klasy bazowej.
4. Skąd wiadomo, na jakiego typu obiekt wskazuje wskaźnik i po co nam ta wiedza?

- Informacja o typie może być przykładowo konieczna, żeby wywołać odpowiednią wersję funkcji
- Mechanizm RTTI wydaje się słusznie zbędny, jeżeli jest to **funkcja wirtualna** obecna we wszystkich klasach hierarchii
- Zalety RTTI są jednak dostrzegalne wówczas, gdy w którejś z klas pochodnych znajduje się **nowa funkcja**, z której mogą korzystać tylko **niektóre obiekty**

Podstawowe funkcjonalności RTTI

Na mechanizm RTTI składają się trzy główne elementy:

- operator **`type_info`** — Klasa przechowująca informacje o danym typie.
- operator **`typeid`** — Zwraca wartość określającą typ danego jako argument obiektu (ściślej - zwraca referencję do obiektu typu **`type_info`**).
- operator **`dynamic_cast`** — Tworzy wskaźnik typu klasy pochodnej ze wskaźnika typu klasy bazowej. Gdy konwersja jest niemożliwa, zwracany jest wskaźnik pusty. Jest to najczęściej używany element RTTI

Podstawowe funkcjonalności RTTI

Na mechanizm RTTI składają się trzy główne elementy:

- operator **`type_info`** — Klasa przechowująca informacje o danym typie.
- operator **`typeid`** — Zwraca wartość określającą typ danego jako argument obiektu (ściślej - zwraca referencję do obiektu typu **`type_info`**).
- operator **`dynamic_cast`** — Tworzy wskaźnik typu klasy pochodnej ze wskaźnika typu klasy bazowej. Gdy konwersja jest niemożliwa, zwracany jest wskaźnik pusty. Jest to najczęściej używany element RTTI

UWAGA:

- Mechanizm RTTI działa tylko dla klas, które są **polimorficzne**, tzn. mają co najmniej jedną metodę wirtualną.
- klasa **`type_info`** jest zdefiniowana w pliku nagłówkowym **`typeinfo`**
- **`dynamic_cast`** jest słowem kluczowym języka C++

Przykład 1 - `typeinfo`

```
1.  #include "pch.h"
2.  #include <iostream>
3.  #include <string>
4.  #include <typeinfo>
5.
6.  class Animal {
7.  public:
8.      virtual void displayAnimal() {}
9.  };
10.
11. class Dog : public Animal {};
12. class Cat : public Animal {};
```

Przykład 1- `typeinfo`

```
Type of 'c' is: char
Type of '*pc' is: char
Type of 'i' is: int
Type of 'l' is: long
Type of 'f' is: float
Type of 'd1' is: double
```

```
15. int main()
16. {
17.     //built-in types:
18.     char c;
19.     char *pc;
20.     int i;
21.     long l;
22.     float f;
23.     double d1;
24.     double d2;
```

```
26.     std::cout << "Type of 'c' is: "       << typeid(c).name() << '\n';
27.     std::cout << "Type of '*pc' is: "   << typeid(*pc).name() << '\n';
28.     std::cout << "Type of 'i' is: "       << typeid(i).name() << '\n';
29.     std::cout << "Type of 'l' is: "       << typeid(l).name() << '\n';
30.     std::cout << "Type of 'f' is: "       << typeid(f).name() << '\n';
31.     std::cout << "Type of 'd1' is: "     << typeid(d1).name() << '\n';
32.     std::cout << '\n';
```

Przykład 1- `typeinfo`

```
1
0
1
```

```
15. int main()
16. {
17.     //built-in types:
18.     char c;
19.     char *pc;
20.     int i;
21.     long l;
22.     float f;
23.     double d1;
24.     double d2;
```

```
34.     //types comparision:
35.     std::cout << (typeid(d1) == typeid(d2))     << '\n';
36.     std::cout << (typeid(c) == typeid(d2))      << '\n';
37.     std::cout << (typeid(double) == typeid(d2)) <<'\n';
38.     std::cout << '\n';
```

Przykład 1- `typeinfo`

```
41.     //polymorphic types:
42.     Animal* ptr;
43.     Dog dog;
44.     Cat cat;
45.
46.     std::cout << "Type of 'ptr' is: " << typeid(ptr).name() << '\n';
47.     std::cout << "Type of 'dog' is: " << typeid(dog).name() << '\n';
48.     std::cout << "Type of 'cat' is: " << typeid(cat).name() << '\n';
49.
50.     ptr = &dog;
51.     std::cout << "Type of '*ptr' when pointing to 'dog' is: " << typeid(*ptr).name() << '\n';
52.
53.     ptr = &cat;
54.     std::cout << "Type of '*ptr' when pointing to 'cat' is: " << typeid(*ptr).name() << '\n';
55.
56.
57.     return 0;
58. }
```

```
Type of 'ptr' is: class Animal *
Type of 'dog' is: class Dog
Type of 'cat' is: class Cat
Type of '*ptr' when pointing to 'dog' is: class Dog
Type of '*ptr' when pointing to 'cat' is: class Cat
```

Przykład 2 - `dynamic_cast`

```
1.  // zakładamy, że dysponujemy następującą hierarchią klas
2.  class A {...}; // posiada co najmniej jedną metodę wirtualną
3.  class B : public A {...};
4.  class C : public B {...};
5.
6.  // w realizowanym programie pojawiają się następujące wskaźniki
7.  A* ptrA = new A;
8.  A* ptrB = new B;
9.  A* ptrC = new C;
10.
11. //czy poniższe rzutowania typu są bezpieczne?
12. C* ptr1 = (C*) ptrC;
13. C* ptr2 = (C*) ptrA;
14. B* ptr3 = (C*) ptrC;
```

Przykład 2 - `dynamic_cast`

```
1.  // zakładamy, że dysponujemy następującą hierarchią klas
2.  class A {...}; // posiada co najmniej jedną metodę wirtualną
3.  class B : public A {...};
4.  class C : public B {...};
5.
6.  // w realizowanym programie pojawiają się następujące wskaźniki
7.  A* ptrA = new A;
8.  A* ptrB = new B;
9.  A* ptrC = new C;
10.
11. //czy poniższe rzutowania typu są bezpieczne?
12. C* ptr1 = (C*) ptrC;
13. C* ptr2 = (C*) ptrA;
14. B* ptr3 = (C*) ptrC;
```

- linia 12 - rzutowanie bezpieczne

Przykład 2 - `dynamic_cast`

```
1.  // zakładamy, że dysponujemy następującą hierarchią klas
2.  class A {...}; // posiada co najmniej jedną metodę wirtualną
3.  class B : public A {...};
4.  class C : public B {...};
5.
6.  // w realizowanym programie pojawiają się następujące wskaźniki
7.  A* ptrA = new A;
8.  A* ptrB = new B;
9.  A* ptrC = new C;
10.
11. //czy poniższe rzutowania typu są bezpieczne?
12. C* ptr1 = (C*) ptrC;
13. C* ptr2 = (C*) ptrA;
14. B* ptr3 = (C*) ptrC;
```

- linia 12 - rzutowanie bezpieczne
- linia 13 - rzutowanie niebezpieczne

Przykład 2 - `dynamic_cast`

```
1.  // zakładamy, że dysponujemy następującą hierarchią klas
2.  class A {...}; // posiada co najmniej jedną metodę wirtualną
3.  class B : public A {...};
4.  class C : public B {...};
5.
6.  // w realizowanym programie pojawiają się następujące wskaźniki
7.  A* ptrA = new A;
8.  A* ptrB = new B;
9.  A* ptrC = new C;
10.
11. //czy poniższe rzutowania typu są bezpieczne?
12. C* ptr1 = (C*) ptrC;
13. C* ptr2 = (C*) ptrA;
14. B* ptr3 = (C*) ptrC;
```

- linia 12 - rzutowanie bezpieczne
- linia 13 - rzutowanie niebezpieczne
- linia 14 - rzutowanie bezpieczne

Przykład 2 - `dynamic_cast`

```
1.  // zakładamy, że dysponujemy następującą hierarchią klas
2.  class A {...}; // posiada co najmniej jedną metodę wirtualną
3.  class B : public A {...};
4.  class C : public B {...};
5.
6.  // w realizowanym programie pojawiają się następujące wskaźniki
7.  A* ptrA = new A;
8.  A* ptrB = new B;
9.  A* ptrC = new C;
10.
11. //czy poniższe rzutowania typu są bezpieczne?
12. C* ptr1 = (C*) ptrC;
13. C* ptr2 = (C*) ptrA;
14. B* ptr3 = (C*) ptrC;
```

- linia 12 - rzutowanie bezpieczne
- linia 13 - rzutowanie niebezpieczne
- linia 14 - rzutowanie bezpieczne

Bezpieczna alternatywa:

```
dynamic_cast <Typ*> (wskaznik)
```

Np.:

```
C* ptr3 = dynamic_cast<C*> (ptrA);
```

- czego wynikiem jest **`nullptr`**

Przykład 2 - `dynamic_cast`

- możliwe jest używanie **`dynamic_cast`** na **referencjach**
- sposób działania różni się nieznacznie od sposobu działania w przypadku użycia wskaźników
- referencje nie mają specjalnej wartości, która mogłaby oznaczać niepowodzenie
- o niepowodzeniu w przypadku nieudanego rzutowania **zgłaszany jest wyjątek `bad_cast`** dziedziczący po klasie **`exception.`** Jest on zdefiniowany w pliku nagłówkowym **`typeinfo`**

Przykład 2 - `dynamic_cast`

- możliwe jest używanie **`dynamic_cast`** na **referencjach**
- sposób działania różni się nieznacznie od sposobu działania w przypadku użycia wskaźników
- referencje nie mają specjalnej wartości, która mogłaby oznaczać niepowodzenie
- o niepowodzeniu w przypadku nieudanego rzutowania **zgłaszany jest wyjątek `bad_cast`** dziedziczący po klasie **`exception;`** jest on zdefiniowany w pliku nagłówkowym **`typeinfo`**

```
1.  #include <typeinfo>
2.  ...
3.
4.  A a;
5.  A & aRef = a;
6.
7.  try{
8.      C & cRef = dynamic_cast<C &> (aRef);
9.      ...
10. }catch(bad_cast &){
11.     ...
12. };
```

Materiały źródłowe...

- Gręboszz Jerzy, “Symfonia C++ standard - tom 2”
- “C++ NotesForProfessionals”
- https://www.linuxtopia.org/online_books/programming_books/c++_practical_programming
- Stephen Prata, “Język C++. Szkoła programowania. Wydanie VI”.

Dziękujemy za uwagę!

Michał Sosna, Michał Rzepka

RTTI