Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Programowanie obiektowe
Polilmorfizm i RTTI
(Run-time Type Information)

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

Politechnika
Warszawska

Czy RTTI jesy potrzebne?

- Języki proceduralne (jak C) obchodzą się bez tego mechanizmu
- Dziedziczenie i polimorfizm w językach obiektowych zmniejszają zapotrzebowanie na RTTI: nie trzeba znać typu obiektu polimorficznego – wystarczy znajomość interfejsu wirtualnego
- Mechanizm obniża efektywność kodu (języki z RTTI, np. Smalltalk, Lisp, Java są mało efektywne)
- W C++ RTTI - jeśli ustawiono odpowiednie opcje kompilatora

Argumenty za RTTI

- Są sytuacje, w których mechanizm jest przydatny - statyczna kontrola typów i funkcje wirtualne nie wystarczają - np. korzystanie z dziedziczenia wielobazowego i wirtualnego
- Presja środowiska programistów:
  Java ma mechanizm refleksji,
  C++ tylko typy statyczne :(

```
// bez mechanizmu refleksji
Foo foo = new Foo();
foo.hello();

// z refleksją
// z pakietem java.lang.reflec
Class cl = Class.forName("Foo");
Method method = cl.getMethod("hello");
method.invoke(cl.newInstance());
```

Polimorfizm

Deklaracja określa typ statyczny

Włączenie mechanizmu polimorficznego

typ rzeczywisty

```
#include<iostream>
using namespace std;

class A {
  public:

  void f()
  { cout << "A.f()" << endl; }
  };

class B : public A {
 public:
  void f()
  { cout << "B.f()" << endl; }
};
```

```
int main()
{

A a, *pa = new A, *pab = new B,
&raa = a, &rab = *pab;

a.f();
pa->f();
pab->f();
raa.f();
rab.f();
system("pause");
return 0;
}
```

	z virtal
A.f()	A.f()
A.f()	A.f()
A.f()	B.f()
A.f()	A.f()
A.f()	B.f()

Składniki RTTI

C++98

- operator **typeid**
- klasa **std::typeinfo**
- operator **dynamic_cast<>**
- wyjątki **bad_typeid**, **bad_cast**

C++11

- klasa **std::type_index**
- właściwości typów (type traits)

Operator typeid

typeid stosuje się (jak sizeof) do wyrażeń lub typów:

typeid(*exp*)
typeid(*typ*)

- Wyrażenie *exp* nie jest obliczane, z wyjątkiem sprawdzenia efektu operatora dereferencji '*'; próba dereferencji wskazania 0 powoduje zgłoszenie wyjątku bad_typeid.
- Jeśli nie ma powodu do zgłoszenia wyjątku, to operator zwraca referencję na **ustalony** obiekt klasy std::type_info reprezentujący typ argumentu
- Kwalifikatory const i volatile są ignorowane.

typeid **jest jedynym** kreatorem obiektów type_info

Klasa std::type_info [C++11]


```
namespace std {
class type_info
{
public:
virtual ~type_info();
bool operator==(const type_info& rhs) const noexcept;
bool operator!=(const type_info& rhs) const noexcept;
bool before(const type_info& rhs) const noexcept;
size_t hash_code() const noexcept;
const char* name() const noexcept;

// bez kopiowania
type_info(const type_info& t) = delete;
type_info& operator=(const type_info& t) = delete;
};
}
```

Zakaz generacji akcesoriów standardowych (tu: konstruktora kopiującego i operatora przypisania).

Klasa std::type_info przykład (1)


```
#define RTTI(x) cout<<"RTTI(" #x "): "\
   <<typeid(x).name()<<endl
struct EE { };
struct EP{ virtual ~EP(){} }; // Klasa polimorficzna
enum Dir { E, N, W, S };

int main()
{
  enum Dir1 { E1, N1, W1, S1 };
  int i = 5;
  const char *s = "Tekst";
  RTTI(2 * i);
  RTTI("Tekst");
  RTTI(s);
  RTTI('A');
  RTTI(EE);
  RTTI(EP);
  RTTI(Dir);
  RTTI(W);
  RTTI(Dir1); RTTI(W1);
}
```

VS2013 .raw_name

VS2013 .hash_code

```
RTTI(2 * i): 2515107422
RTTI("Tekst"): 2439601482
RTTI(s): 782596980
RTTI('A'): 2823553821
RTTI(EE): 1469252910
RTTI(EP): 1284699101
RTTI(Dir): 3812359063
RTTI(W): 3812359063
RTTI(Dir1): 1210433392
RTTI(W1): 1210433392
```


Klasa std::type_info przykład (1)


```
#define P(x) std::cout<< (x) <<endl
// ...
int f(int), g(int);              // Tylko deklaracje
Figura *fs = new Odcinek, *f0 = nullptr;
const Figura *cf = new Odcinek;
typedef Odcinek Segment;
P(typeid(fs).name());       // struct Figura *
P(typeid(f0).name());       // struct Figura *
P(typeid(*fs).name());      // struct Odcinek
P(typeid(*cf).name());      // struct Odcinek
P(typeid(*f0).name());      //==> bad_typeid()
P(typeid(sizeof 0).name());     // unsigned int
P(typeid(f(1) + g(2)).name()); // int
P(typeid(Segment).name()); // struct Odcinek
```

Klasa `std::type_info` - Uwagi

- Nazwa zwracana przez **`name()`** zależy od implementacji (nie ma zobowiązania do przenośności).
- Obowiązuje wewnętrzna spójność nazewnictwa: nazwy dla wyrażeń generujących te same typy muszą być tożsame.
- Operatory '==' i '!=' przenoszą porównywanie typów na poziom porównywania generowanych przez **`typeid`** deskryptorów.

Przykład: rysowanie jednego typu figur


```
void rysujWybrane(vector<Figura *>FP,
        const type_info& selektor)
{
  if (FP.empty())
     return; // Wykaz pusty
  for (auto fp : FP)
     if (typeid(*fp) == selektor) // Te same typy
        fp->rysuj();
}
// ...
vector<Figura *> FP =
        { new LiniaBeziera(), new Odcinek() };
rysujWybrane(FP, typeid(LiniaBeziera));
```

typ wybranej figury

Rzutowanie - rodzaje

- `const_cast` – tzw. rzutowanie przeciwwariancyjne (zniesienie const, volatile)
- `static_cast` – rzutowanie niesprawdzane
- `dynamic_cast` – rzutowanie polimorficzne
- `reinterpret_cast` – rzutowanie zmieniające interpretację reprezentacji
- `(typ)` – rzutowanie w stylu C (wymuszone)

Operator const_cast


```
#include <iostream>
using namespace std;

void print(char * str)
{
  str[0] = 'H'; // zachowanie niezdefiniowane,
                // zwykle błąd dostępu
  cout << str << '\n';
}

int main() {
  const char * c = "hello!";
  print(c); // Błąd kompilacji
  print(const_cast<char *> (c));
  return 0;
}
```

Operator static_cast, reinterpret_cast


```
class A {  };
class B : public A {  };
class O {  };  // inna hierarchia dziedziczenia

main() {
  A  *pX;  // wskaźnik na klasa bazową (ustawiony gdziekolwiek)

  B *pB;
  pB = static_cast<B*>(pX); // OK, jeśli wiemy co robimy
  pB = (B*)(pX);            // To samo co static_cast<>

  O *pO;
  pO = static_cast<O*>(pX); // Błąd kompilacji
                            //  - konwersja niemożliwa

  pO = reinterpret_cast<O*>(pX); // kompilacja OK
                                 // Co programista miał na myśli ?

  pO = (O*)(pX);   // kompilacja OK
                   // To samo co reinterpret_cast<>

}
```


Operator dynamic_cast

dynamic_cast<*Typ*>(*exp*)

- Forma zapisu - jak specjalizacja pewnego szablonu (podobieństwo powierzchowne; mechanizm dotyczy czasu wykonania programu)
- **Typ** i wyrażenie **exp** muszą być odpowiednio wskazaniami albo referencjami na klasy **polimorficzne w tej samej strukturze dziedziczenia**
- Konwersja wskazań / referencji **w górę hierarchii** (od klasy pochodnej do klasy bazowej) nie wymaga stosowania operatora
- Inne nawigacje bez użycia **dynamic_cast** nie są bezpieczne
- Specjalny przypadek konwersji:

```
void* vp = dynamic_cast<void*>(ptr);
```

Konwersja zawsze kończy się sukcesem zwracając wskazanie na pełny obiekt wskazywany przez ptr.

Inna implementacja `rysujWybrane`


```
template<class Selektor>
void rysujWybrane(vector<Figura *> FP) {
    if (FP.empty())
        return; // Wykaz pusty
    for (auto fp : FP)  {
        if (auto sp = dynamic_cast<Selektor*>(fp))
            sp->rysuj();
    }
}
// ...
vector<Figura*> FP =
    { new LiniaBeziera(), new Odcinek() };
rysujWybrane<Odcinek>(FP);
```


Operator dynamic_cast

- Dwie wersje funkcji - szablonowa i z parametrem `const type_info&` nie są równoważne.
- Funkcja szablonowa jest konkretyzowana w czasie kompilacji dla każdego ustalonego Selektora.
- Funkcja z parametrem `type_info` istnieje w jednym egzemplarzu i może obsługiwać dowolne wybory.
- Istotna różnica:
  - wersja sprawdzająca równość identyfikacji typów wybiera tylko obiekty **dokładnie** odpowiadające selektorowi;
  - wersja z `dynamic_cast` wybierze również **obiekty pochodne** względem selektora (w przykładzie Odcinek lub jego pochodne).
- Operator `dynamic_cast` umożliwia kontrolowaną nawigację po całej strukturze dziedziczenia polimorficznego.

Przykład – typowe zastosowanie RTTI


```
class File  { //klasa abstrakcyjna
  public:
    virtual void open() = 0;
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~File() {}// Trzeba zdefiniować
};
class BinFile : public File {
   public:
     void open(){ OS_exec(this); }
     //...inne funkcje
};
class TextFile : public File {
public:
     void open() { NotePad(this); }
     //...inne funkcje
     virtual void print(); // Rozszerzenie interfejsu
};
```


Przykład - założenia

- Funkcje polimorficzne `open()`, `read()`, `write()` są w interfejsie każdej klasy pochodnej względem `File`.
- Stosowanie operacji `open()`, `read()`, `write()` w całej hierarchii `File` **nie potrzebuje** mechanizmu RTTI.
- Funkcja **`print()` rozszerza interfejs** dla plików tekstowych i pochodnych; binarny plik musi być poddany konwersji do pliku tekstowego przed zastosowaniem `print()`.
- Stosowanie operacji `print()` w hierarchii `File` wymaga specjalnych zabiegów, np. wsparcia RTTI
- RTTI ma praktyczne zastosowanie **tylko** *do obiektów polimorficznych (jest uzupełnieniem polimorfizmu)*
- Typy polimorficzne zawierają co najmniej jedną funkcję wirtualną; informacja o typie i wskaźnik na obiekt `std::type_info` rezydują w tabeli funkcji wirtualnych.
- Typy niepolimorficzne podlegają statycznej kontroli; `typeid` zwraca deskryptor typu statycznego (mało użyteczne)

Obsługa plików - przykład


```
// Polimorficzny interfejs użytkownika dla plików
void OnRightClick(File& file, Message m)
{
    switch (m)
    { //...
    case M_OPEN:
        file.open();
        break;
    case M_PRINT:
    // ??? Jaka obsługa tego zdarzenia?
        break;
    }
}
```


Rozwiązanie z użyciem typeid


```
void OnRightClick(File& file, Message m)
{
    if (typeid(file) == typeid(TextFile))
    {  //otrzymano TextFile; druk dopuszczalny
       // ...
       if (m == M_PRINT) f->print();
       // ...
    }
    else {
       /* nie TextFile, druk zakazany */
    }
}
```

Rozszerzenie hierarchii (HTML)


```
struct HTMLFile : public TextFile { // ...
     void open() {
          Opera(this); // Lub inna przeglądarka
     }
     void virtual print();
     // Interpretuj znaczniki i drukuj
};

void OnRightClick(File& file, Message m)
{
     if (typeid(file) == typeid(TextFile)) {
          //otrzymano TxtFile; druk dopuszczalny
     }
     else {
          // dla HTML druk będzie zakazany
     }
}
```


Rozszerzenie hierarchii (HTML) cd(1)


```
// Korekta
void OnRightClick(File& file, Message m)
{
    if (typeid(file) == typeid(TextFile) ||
        typeid(file) == typeid(HTMLFile)) {
        /* druk dopuszczalny */
    }
    else    {
        /* plik binarny, druk zakazany */
    }
    // ...
}
```

Wady

- kłopotliwa konserwacja kodu
- każdy nowy typ pliku może wymagać interwencji w kodzie interfejsu użytkownika `OnRightClick()`

Rozszerzenie hierarchii (HTML) cd(2)


```
// Użycie operatora dynamic_cast<>
void OnRightClick(File& file, Message m)
{
    // Próba konwersji dynamicznej
    TextFile* p = dynamic_cast<TextFile*>(&file);
    if (p) //dynamic_cast<> udany
    { // można używać *p jako TxtFile
        p->print();
    }
    else // p==0, file nie jest plikiem tekstowym
    { /* .... */
    }
}
```

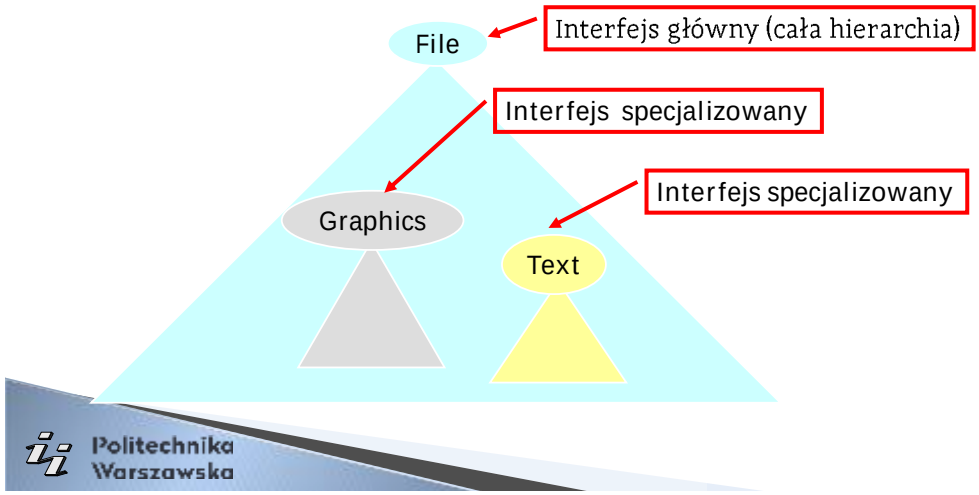
- Można dodawać do hierarchii następne klasy podległe `TextFile` (bezpośrednio lub pośrednio)
- Zastosowana konwersja dynamiczna nazywa się **wstępującą** (up-cast) – prowadzi do klasy ustanawiającej interfejs podhierarchii.

Rozszerzenie hierarchii (HTML) cd(3)


```
// Operator dynamic_cast<> dla referencji
void OnRightClick(File& f, Message m) {
    try {
        TextFile &t = dynamic_cast<TextFile&>(f);
        // Może zakończyć się wyjątkiem bad_cast
      switch (m) {
        case M_OPEN:
            t.open(); //polimorfizm czysty
            break;
        case M_PRINT:
            t.print(); // TextFile:: albo HTMLFile::print()
            break;
        }
    }
    catch (std::bad_cast& bc)
    {
        // traktuj f jako plik binarny, bez print
    }
}
```

Podsumowanie: typowa sytuacja dla RTTI

- Interfejs główny nie wymaga RTTI
- Interfejsy specjalizowane - tak



Inne zastosowania dynamic_cast<>


```
struct A {
  int i;
  virtual ~A(){} // Typ
polimorficzny
};

struct B {
    bool b;
};

struct D : public A, public B
//Dziedziczenie 2-bazowe
{
    int k;
    D() { b = true; i = k = 0; }
};
```

```
int main(int argc, char* argv[])
{
  A *pa = new D;
  B *pb = dynamic_cast<B*>(pa);
// Konwersja "skrośna" – dostęp do drugiej bazy
// w pełni bezpieczny. Jednostka kompilacji z
// tą konwersją może nie wiedzieć o klasie D.

  B *p = static_cast<B*>(pa);
// Błąd: niemożliwe, A i B są niezależne.

  B *q = reinterpret_cast<B*>(pa);// Fatalne
  B *r = (B*)pa;     // Fatalne
  return 0;
}
```

Inne zastosowania dynamic_cast<> cd(1)


```
int main(int argc, char* argv[]) {
  A *pa = new D;  // Konwersja gwarantowana
  B *pb = (B*)pa;
  // Fatalne: to tylko zmiana interpretacji wskaźnika;
  // pb wskazuje dalej podobiekt A w obiekcie klasy D,
  // ale kompilator ma uważać, że wskazuje na obiekt B

  bool bb = pb->b; // bb nieokreślone
  cout << "pa=" << pa << " pb=" << pb << endl;  //pa == pb
  pb = dynamic_cast<B*>(pa); // RTTI musi być włączone (opcje
kompilatora)
  bb = pb->b;                //OK, bb == true
  cout << "pa=" << pa << " pb=" << pb << endl;//pa != pb
  return 0;
}
```

VS2013

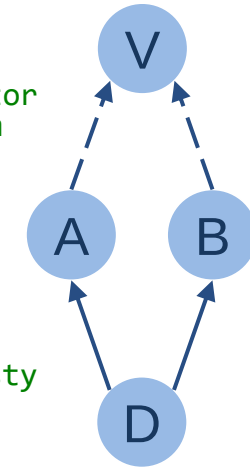
```
pa=00975600 pb=00975600
pa=00975600 pb=00975608
```


Inne zastosowania dynamic_cast<> cd(2)


```
#include <iostream>
#define P(x) std::cout<<#x " = " <<(x) << std::endl

struct V {
  virtual ~V() /*=0*/ // wirt. destruktor
                      // => polimorfizm
};
struct A : virtual V {/* .... */ };
struct B : virtual V {/* .... */ };
struct D : A, B      {/* .... */ };

V v; // Błąd jeśli destruktor ~V() czysty
```



Inne zastosowania dynamic_cast<> cd(3)


```
int main() {
 D d;          P(&d);
 V *pv = &d;   P(pv);
 P(typeid(pv).name());
 P(typeid(*pv).name());
 A* pa = dynamic_cast<A*>(pv); // "down-cast"
 P(pa);
 P(typeid(pa).name());
 P(typeid(*pa).name());
 B* pb = dynamic_cast<B*>(pv); // "down-cast"
 P(pb);
 P(typeid(pb).name());
 P(typeid(*pb).name());

 B* rb = dynamic_cast<B*>(pa); // "cross-cast"
 P(rb);
 //A* qa = pv;                 // Błąd
 //A* ra = static_cast<A*>(pv);// Błąd
}
```

```
&d = 001DFA10
```

```
pv = 001DFA18
typeid(pv).name() = struct V *
typeid(*pv).name() = struct D
```

```
pa = 001DFA10
typeid(pa).name() = struct A *
typeid(*pa).name() = struct D
```

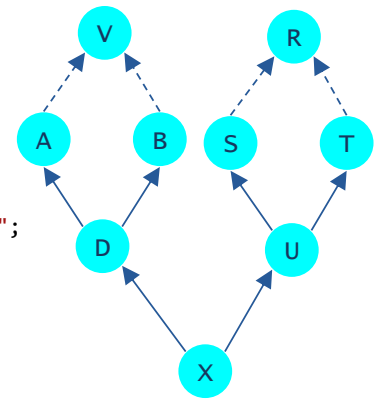
```
pb = 001DFA14
typeid(pb).name() = struct B *
typeid(*pb).name() = struct D
```

```
rb = 001DFA14
```


Inne zastosowania dynamic_cast<> cd(4)


```
void fun(V* vptr) {
     P(typeid(*vptr).name());
     R *rptr = dynamic_cast<R*>(vptr);
     if (rptr)
          rptr->f();
     else
          cout << "Konwersja zakazana\n";
}

int main() {
     D d;
     X x;
     fun(&x);
     fun(&d);
     return 0;
}
```




```
typeid(*vptr).name() = struct X
R::f()
typeid(*vptr).name() = struct D
Konwersja zakazana
```

Podsumowanie RTTI

- Typy objęte (użytecznie) przez RTTI muszą być polimorficzne (co najmniej 1 funkcja wirtualna)
- Mechanizm jest zwykle włączany odpowiednimi ustawieniami kompilatora
- Stosując operator `dynamic_cast<X*>` z docelowym wskazaniem, zawsze sprawdzać zwróconą wartość.
- Stosując operator `dynamic_cast<X&>` z docelowym typem referencyjnym zawsze używać bloku `try` z reakcją na wyjątek `std::bad_cast`
- Dereferencja pustego wskazania p w wyrażeniu `typeid(*p)`, powoduje wyjątek `std::bad_typeid`.

Interfejsy polimorficzne niewirtualne


```
#include <iostream>
#include <string>
using namespace std;

class Figura { // Klasa abstrakcyjna
  virtual string nazwa() = 0; // private
protected:
  virtual string path() { return "Figura"; }
public:
  virtual ~Figura() = 0 {}  // Destruktor wirtualny

  void show()  // Interfejs niewirtualny
  {
   cout << "Obiekt typu ";
   cout << nazwa() << "; Path = " << path() << endl;
 }
};
```

Interfejsy polimorficzne niewirtualne cd(1)


```
class Polygon : public Figura {
  string nazwa()      { return "Polygon";   }
 protected:
  string path() { return "Polygon:" + Figura::path(); }
};

class Triangle : public Polygon {
  string nazwa()      {return "Triangle";   }
 protected:
  string path() { return "Triangle:" + Polygon::path();     }
};

class Histogram : public Figura {
  string nazwa()      {     return "Histogram";   }
 protected:
  string path() {  return "Histogram:" + Figura::path();    }
};

class Bezier : public Figura{
  string nazwa() {return "Bezier";}
 protected:
  string path() {return "Bezier:" + Figura::path();   }
};
```


Interfejsy polimorficzne niewirtualne cd(3)


```
void show(vector<Figura *> fcoll) { // Jak dla interfejsu wirtualnego
  for (auto fp : fcoll)
          fp->show();
  cout << "Liczba figur: " << fcoll.size() << endl;
}

//Figura f; // Błąd - klasa abstrakcyjna

int main() {
     Histogram h;
     Polygon p;
     Bezier b;
     Triangle t;

     vector<Figura *> fcoll = { &p, &b, &h, &t };
     show(fcoll);
     return 0;
}
```

```
Obiekt typu Polygon; Path = Polygon:Figura
Obiekt typu Bezier; Path = Bezier:Figura
Obiekt typu Histogram; Path = Histogram:Figura
Obiekt typu Triangle; Path = Triangle:Polygon:Figura
Liczba figur: 4
```

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Dziedziczenie czy szablony?

Politechnika Warszawska

Type traits (C++11)

- Interfejs oparty na szablonach do przekazywania informacji o typach (plik nagłówkowy `type_traits`)
- Podstawowe (class template)
  - `is_void`
  - `is_floating_point`
  - `is_enum`
- Własności typow
  - `is_const`
  - `is_volatile`
  - `is_abstract`
  - `is_signed`
  - `is_unsigned`
  - …

```
#include <iostream>
//#include <type_traits>
template< typename T >
struct is_void {
  static const bool value = false;
};
template<>
struct is_void< void > {
  static const bool value = true;
};

int main() {
 std::cout<< std::boolalpha<< "is_void?"<< std::endl;
 std::cout<< "int: " << is_void<int>::value <<std::endl;
 std::cout<< "void: "<< is_void<void>::value<<std::endl;
return 0; }
```

is_void?
int: false
void: true

Politechnika Warszawska

Concepts (C++20)

- Kontrola parametrów szablonów - zestaw wymagań dotyczących parametrów szablonu (sprawdzany podczas kompilacji)
- Lepsza informacja o błędach kompilacji

```
template <typename T> requires CONDITION void DoSomething(T param) {}
template <typename T> void DoSomething(T param) requires CONDITION {}

template <typename T>
concept numeric = std::is_integral_v<T> || std::is_floating_point_v<T>;

template <typename T>
requires numeric<T>
 constexpr double Average(vector<T> const& vec) {
   const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
   return sum / vec.size();
 }
```