Identyfikacje typu na etapie wykonania (RTTI)

- **(Run Time Type Identification)**

- **Może powstać taka sytuacje, gdy w trakcie kompilacji typ obiektu nie jest znany. C++ implementuje polimorfizm poprzez hierarchie klas, funkcje wirtualne i wskaźniki klas bazowych.**

- **Wskaźniki klas bazowych można wykorzystać do wskazywania obiektów klas bazowych, a również, i dowolnych obiektów pochodnych klasy bazowej.**

- **Nie wykluczone, że w jakimś miejscu kodu powstanie konieczność wyjaśnić, do obiektu którego typu wskazuje wskaźnik klasy bazowej – to się odbywa w trakcie wykonania programu.**

- **Operator typeid (dołożyć nagłówek <typeinfo>) :**

  typeid (*obiekt*) **typeid(*referencje_do_obiektu*)**

  ***obiekt* – ten obiekt, typ którego mamy określić (typ wbudowany lub typ, tworzony przez użytkownika)**

- **Zwraca referencje typu *type_info* do argumentu *obiekt .***

- **W klasie *type_info* zdefiniowane są następujące składowe publiczne:**

  - **bool operator == (const type_info &ob); //służą do porównywania typów**
  - **bool operator != (const type_info &ob); //służą do porównywania typów**
  - **bool before(const type_info &ob); //służy do użycia wewnętrz. przy sortow.**
  - **const char *name(); //zwraca wskaźnik do nazwy typów**

```
class A
{
};

class B
{
};

void main()
{
        int i;
        A a;
        B b;
        cout << typeid(i).name() << endl;
        cout << typeid(a).name() << endl;
        cout << typeid(b).name() << endl;
        system("pause");
}
```

```
int
class A
class B
Для продолжения нажмите любую клавишу . . .
```

Przykład 2

```
class A
{
};

class B
{
};

//template <class T, class K> bool funtypcompare(T &x, K &y)  //typeid(referencje_do_obiektu)
template <class T, class K> bool funtypcompare(T x, K y)
{
		cout << typeid(x).name() << " i " << typeid(y).name() << " ? \n";
		if(typeid(x) == typeid(y))
			return true;
		else
			return false;
}


void main()
{
		int i = 0, j = 0;
		A a;
		B b;
		cout << funtypcompare(a, b) << endl;
		cout << funtypcompare(i, j) << endl;
		system("pause");
}
```

```
class A i class B ?
0
int i int ?
1
Для продолжения нажмите любую клавишу . . .
```

➢ **Przykład 3. Klasy polimorfne. Wskaźnik klasy bazowej może wskazywać obiekt klasy bazowej, a również, klasy pochodnej. Dla wyznaczenia typu obiektu używamy operator wskazania pośredniego *ptr . Jeśli wskaźnik jest zerowy, operator typeid(*ptr) generuje wyjątek bad_typeid**

```
class B
{
public:
        virtual void f() { cout << "Jestem B\n"; }
};

class D1 : public B
{
public:
        void f() { cout << "Jestem D1\n"; }
};
```

```
 void main()
{
		B b, *pb, *pnull = NULL;
		D1 d1;

		pb = &b;
		pb->f();
		cout << "pb wskazuje obiekt typu " << typeid(*pb).name() << endl;

		pb = &d1;
		pb->f();
		cout << "pb wskazuje obiekt typu " << typeid(*pb).name() << endl;

		try
		{
			cout << typeid(*pnull).name() << endl;
		}
		catch(bad_typeid aa)
		{
			cout << "bad pointer\n";
		}
		system("pause");
}
```

```
Jestem B
pb wskazuje obiekt typu class B
Jestem D1
pb wskazuje obiekt typu class D1
bad pointer
Для продолжения нажмите любую клавишу . . .
```

- **Klasy nie polimorfne: kod jest taki samy, jak poprzednio, tylko kluczowe słowo virtual pozostało zakomentowane**

```
class B
{
public:
        /*virtual*/ void f() { cout << "Jestem B\n"; }
};

class D1 : public B
{
public:
        void f() { cout << "Jestem D1\n"; }
};
```

```
void main()
{
        B b, *pb, *pnull = NULL;
        D1 d1;

        pb = &b;
        pb->f();
        cout << "pb wskazuje obiekt typu " << typeid(*pb).name() << endl;

        pb = &d1;
        pb->f();
        cout << "pb wskazuje obiekt typu " << typeid(*pb).name() << endl;
}
```

```
Jestem B
pb wskazuje obiekt typu class B
Jestem B
pb wskazuje obiekt typu class B
Для продолжения нажмите любую клавишу . . .
```

- **Dla wskaźników niepolimorficznej hierarchii klas otrzymujemy wskaźnik typu bazowego – rzeczywisty typ obiektu wskazywanego nie pozostaje określony.**

- **Druga forma operatora typeid:**

  typeid(*nazwa_typu*)

- **Jest używana dla porównywania typów (czy ma obiekt typ podany?)**

```
class B
{
public:
        virtual void f() { cout << "Jestem B\n"; }
};

class D1 : public B
{
public:
        void f() { cout << "Jestem D1\n"; }
};

void main()
{
        B b, *pb, *pnull = NULL;
        D1 d1;

        pb = &d1;
        if(typeid(*pb) == typeid(D1))
                cout << "pb wskazuyje na obiekt typu D1\n";

        system("pause");
}
```

```
pb wskazuyje na obiekt typu D1
Для продолжения нажмите любую клавишу . . .
```

- **Operator typeid można stosować do szablonów klas: przykład W15_1**

Operatory rzutowania

- ***dynamic_cast*** **– służy do rzutowania wskaźnika na wskaźnik innego typu lub referencji na referencję innego typu w trakcie wykonywania programu i weryfikacji poprawności rzutowania.**

dynamic_cast <*typ_docelowy*> (*wyrażenie*)

- ***typ_docelowy*** **– typ docelowy rzutowania,** ***wyrażenie*** **– wyrażenie, które jest rzutowane na nowy typ.**
- ***wyrażenie*** **– to jest wskaźnik lub referencje,** ***wyrażenie*** **– musi być rozwijane do wskaźnika lub referencji.**
- **Wynik: powodzenie –** ***dynamic_cast*** **zwraca wskaźnik lub referencje poprawnę; niepowodzenie –** ***dynamic_cast*** **zwraca NULL, jeśli wyrażenie jest wskaźnikiem, lub generuje wyjątek** ***bad_cast*****, jeśli wyrażenie jest referencją.**

- **Zadaniem** ***dynamic_cast*** **jest przeprowadzenie rzutowania dla typów polimorficznych.**

- **Mamy dwie klasy polimorficzne B – klasa bazowa, D – klasa pochodna.**

- **Zawsze można rzutować wskaźnik D * na wskaźnik B * (wskaźnik klasy bazowej zawsze może wskazywać na obiekt klasy pochodnej)**

- **Rzutować wskaźnik B * na wskaźnik D * można tylko wtedy, jeśli wskazywany obiekt jest rzeczywiście obiektem klasy D**

- **Uogólnienie: wykorzystanie dynamic_cast zakończy się sukcesem, jeśli rzutowany wskaźnik (lub referencje) ma typ docelowy lub wskazuje (referuje) do obiektu klasy pochodnej typu docelowego. W przeciwnym przypadku – niepowodzenie.**

```
B b, *pb;
D1 d1, *pd1;

pd1 = &d1;
pb = dynamic_cast<B *> (pd1);   //OK, wskaznik klasy bazowej zawsze
                                //moze wskazywac na obiekt klasy pochodnej
pb = &d1;
pd1 = dynamic_cast <D1 *> (pb); //OK, wskaznik klasy bazowej
                                //wskazuje na obiekt klasy pochodnej

pb = &b;
pd1 = dynamic_cast<D1 *> (pb);  //!OK, wskaznik klasy bazowej
                                //nie wskazuje na obiekt typu D1
```

- **Przykład W15_0**

- **Operator dynamic_cast<>() może być wykorzystany zamiast operatora typeid(). Przypuszczamy, że mamy polimorficzne klasy B, D (base, derived):**

```
B *pb;
D *pder;
//……….
if(typeid(*pb) == typeid(D))  //sprawdzamy: wskazuje pb na obiekt klasy pochodnej?
        pder = (D *)pb;   //tak, rzutujemy wskaźnik pb na typ D
else
        pder = NULL;
```

- **Dokładnie to samo można zrobić w jednej linii kodu:**

```
pder = dynamic_cast<D *> (pb);
```

- **A to – błąd:**

```
pder = (D *)pb;
```

Kompilator nic nie zgłosi, ale jeśli pb nie wskazuje na obiekt klasy D, działanie programu jest nie przewidywane. Takie błędy zwykłe bardzo ciężko wykryć w mnie-więcej rozbudowanym kodzie, dla tego że mogą występować nie za każdym uruchomieniem programu.

- **Przykład W15_2.**

- **Jeśli wskaźnik klasy bazowej wskazuje na obiekt klasy pochodnej typu Typ, to wtedy i tylko wtedy rzutowanie** ***p_derived =*** **dynamic_cast<*Typ* *> *(p_base)* skończy się powodzeniem. Więc warunek**

```
if(p_derived = dynamic_cast<Typ *> (p_base) )
{
        //tu jest gwarantowane, że wskaźnik p_base wskazuje na obiekt klasy
        //pochodnej polimorficznej typu Typ
}
else
        //p_base nie wskazuje na obiekt klasy pochodnej polimorficznej typu Typ
```

- **Wykorzystanie dynamic_cast do pracy z szablonami klas – przykład W15_1.**

- **Operator *const_cast***

  const_cast<*typ*> (*wyrażenie*)

  - **zastępuje podczas rzutowania *const* i/lub *volatile* . Typ docelowy musi być takim samym, jako typ źródłowy. Najpopularniejsze użycie – usuwanie wpływu atrybutu *const*.**

  - ***typ* – typ docelowy**
  - ***wyrażenie* – wyrażenie, rzutowane na nowy typ**

```
 void fun(const double *ptr)
{
        double *p;
        p = const_cast<double *> (ptr);  //usuwamy modyfikator const, inaczej kompilator zgłosi
                                         //niespójność typów (double i const double
        *p += 10.0;
        *ptr += 10.0;                    //błąd: error C3892: 'ptr' : you cannot assign to a variable that is
                                         //const
        cout << *p << endl;
}

void main()
{
        double a[2] = {1.0, 2.0};
        fun(a);
        system("pause");
}

11
 Aby kontynuowac, nacisnij dowolny klawisz . . .
```

- **Operator static_cast<> ():**

static_cast<t*yp*> (*wyrażenie*)

- **Wykonuje rzutowanie niepolimorficzne. Na etapie rzutowania nie są przeprowadzane żadne sprawdzenia. Zastępuje oryginalny operator rzutowania.**

- ***typ* – typ docelowy**
- ***wyrażenie* – wyrażenie, rzutowane na nowy typ**

```
int i = 10;
double  a;
a = static_cast<double> (i);
// to samo:  a = (double)i;
```

- **Operator reinterpret_cast<> ():**

reinterpret_cast<*typ*> (*wyrażenie*);

- **służy do konwersji wskazanego typu na typ całkowicie odmienny**

```
void main()
{
	char *p = "to jest wiersz tekstowy";
	int iii = reinterpret_cast<int> (p);
	cout << iii << endl;
	system("pause");
}
```

Przy kompilacji powstaje ostrzeżenie:
warning C4311: 'reinterpret_cast' : pointer truncation from 'char *' to 'int'

Wynik:
4290308
Для продолжения нажмите любую клавишу . . .