Funkcje wirtualne

- W C++ polimorfizm jest zrealizowany w dwa sposoby: na etapie kompilacji i na etapie wykonania. Na etapie kompilacji polimorfizm jest zrealizowany poprzez przeciążenie funkcji i operatorów. To jest polimorfizm statyczny. Na etapie wykonania (polimorfizm dynamiczny) polimorfizm jest zrealizowany poprzez funkcje wirtualne.

- Wskaźniki do klas pochodnych są podstawą dla funkcji wirtualnych i polimorfizmu dynamicznego.

- Wskaźnik typu klasy bazowej może występować jako wskaźnik do obiektu dowolnej klasy pochodnej.

- Podany poniżej przykład ilustruje fragment poprawnego kodu. Błąd o niespójności typów się nie generuje, kiedy wskaźnik typu klasy bazowej wskazuje do obiektu klasy pochodnej

```
class  base
{
……..
};

class derived : public base
{
…….
};

void main()
{
        base ob_base;
        derived ob_derived;
        base *ptr = NULL;

        ptr = &ob_base;       //Pointer ptr points to the base class object
        ptr = &ob_derived;    //Pointer ptr points to the derived class object
}
```

- Jeśli wskaźnik typu klasy bazowej wskazuje do obiektu klasy pochodnej, to:

  - przez ten wskaźnik możemy mieć dostęp tylko do tych składowych klasy pochodnej, które dziedziczą się z klasy bazowej – klasa bazowa nic nie wie o składowych, które powstałe w klasie pochodnej.

  - arytmetyka wskaźników jest skojarzona z typem danych, do którego ten wskaźnik pozostał zadeklarowany. Z tego wynika:

```
derived ob_derived[10];
base *ptr = &ob_derived[0];
ptr++;     //does not point to the next object of the derived class!
```

  - Wskaźnik klasy pochodnej nie można wykorzystać dla dostępu do obiektów klasy bazowej.

- Funkcja wirtualna to składowa zadeklarowana w klasie bazowej i zdefiniowana w klasie pochodnej.

- Aby utworzyć funkcję wirtualną, należy jej deklaracje w klasie bazowej poprzedzić kluczowym słowem *virtual*. Klasa pochodna, która dziedziczy po klasie bazowej funkcje wirtualną, definiuje ją zgodnie ze swoimi potrzebami.

- W klasie bazowej funkcje wirtualne definiują postać interfejsu. Każde (ponowne) definiowanie funkcji wirtualnej w klasie pochodnej wyznacza specyficzną realizacje tej funkcji dla podanej klasy pochodnej. W taki sposób powstaje konkretna metoda.

- Przy definiowaniu funkcji wirtualnej w klasie pochodnej kluczowe słowo *virtual* nie jest potrzebne.

- Funkcje wirtualne mogą być wywołane jako zwykłe funkcje – metody klasy. Przecież polimorfizm dynamiczny występuje, jeśli funkcje wirtualne są wywoływane przez wskaźnik do typu klasy bazowej, który wskazuje na obiekt klasy pochodnej.

- Gdy wskaźnik klasy bazowej jest wykorzystywany do wskazania obiektu, zawierającego funkcję wirtualną, C++ decyduje o wyborze wersji funkcji na podstawie typu obiektu wskazywanego przez ten wskaźnik. Wybór wersji odbywa się na etapie wykonywania programu – będzie wywoływana ta realizacja funkcji wirtualnej, która należy obiektowi klasy, wskazanej przez wskaźnik.

Example W34.

- Ponowne definiowanie funkcji wirtualnej w klasie pochodnej różni się od przeciążenia funkcji:

prototyp funkcji (nazwa, ilość i typy argumentów formalnych i typ wartości zwracanej), redefiniowanej w klasie pochodnej, musi dokładnie odpowiadać prototypowi, występującemu w klasie bazowej. (Przy przeciążeni funkcji każda realizacja się różni listą argumentów formalnych). Jeśli prototyp funkcji wirtualnej pozostanie zmieniony w klasie pochodnej, to taka funkcja będzie potraktowana jako funkcja przeciążona, a nie wirtualna.

- funkcje wirtualne muszą być niestatycznymi składowymi klasy, do której należą (funkcji przeciążone – nie).

- Wybór realizacji funkcji wirtualnych jest dokonywany przy wykonani programu (dla funkcji przeciążonych – na etapie kompilacji).

- Konstruktory nie mogą być funkcjami wirtualnymi, a desktruktory – mogą.

Example W35 (destructor virtualny).

➢ Dla tego, żeby podkreślić różnice pomiędzy funkcjami wirtualnymi i funkcjami przeciążonymi, redefiniowanie funkcji wirtualnych jest określone jako przesłanianie (*overriding*).

- Funkcje wirtualne mogą być wywołane przez użycie referencji do klasy bazowej.

Example W36.

- Funkcja wirtualna zachowuje atrybut *virtual* przy dziedziczeni. Oznacza to, że przy kolejnym dziedziczeni w kolejnej klasie pochodnej funkcja wirtualna może być także przesłoniętą.

- Jeśli funkcja wirtualna nie pozostała przesłonięta w klasie pochodnej, wtedy obiekty tej klasy, odwołujące się do takiej funkcji, będą wykorzystywali funkcję, zdefiniowaną w klasie bazowej.

Funkcje abstrakcyjne (czysto wirtualne funkcji – pure virtual functions)

- Często się okazuje, że w klasie bazowej nie da się stworzyć przydatną definicje funkcji wirtualnej – po prostu klasa bazowa nie dysponuje dostatnią informacją. Wtedy klasa bazowa zawiera tylko deklaracje (prototyp) funkcji wirtualnej bez jej definicji – wyznacza interfejs:

virtual return_type function_name (list_of_arguments) = 0;

- Składnia … = 0 informuje kompilator o to, że definicji funkcji nie istnieje w klasie bazowej.

- Każda klasa pochodna powinna mieć swoją własną definicje każdej funkcji abstrakcyjnej – wyznacza swoją własną realizacje.

- Klasę, która zawiera chociażby jedną funkcje abstrakcyjną, nazywamy klasą abstrakcyjną (abstract class). Niewolno tworzyć obiektów klas abstrakcyjnych – takie klasy nie są pełne. Przecież można tworzyć wskaźniki do klas abstrakcyjnych i referencji.

- Ważnym stosowaniem klas abstrakcyjnych i funkcji wirtualnych jest użycie ich w bibliotekach klas. Na przykład, klasa abstrakcyjna *Figura* zawiera wspólne dane dla różnych płaskich obiektów geometrycznych, ale nic nie wie o szczegółach realizacji – obliczeni pola, zapisu danych na dysk, odczyt z dysku, wyświetlani danych na monitorze. Ale jest możliwe rozszerzyć funkcjonalność tej klasy – stworzyć klasę pochodną, i w niej zdefiniować te realizacje metod wirtualnych, które są potrzebne dla każdego podanego obiektu.

- Wczesne wiązanie odnosi się do wydarzenia, które ma miejsce przy kompilacji (obiekt i wywołanie funkcji są powiązane na etapie kompilacji). Przykłady: wywołanie funkcji zwyczajnej, wywołanie funkcji i operatorów przeciążonych. Główną zaletą jest wysoka wydajność (dla tego że wszystkie informacje do wywołania funkcji są określane na etapie kompilacji). Wada – niewielka elastyczność.

- Późnię wiązanie dotyczy funkcji, wywołanie których jest determinowane przy wykonani programu. Przykład – funkcje wirtualne, wywołanie których odbywa się przez wskaźnik lub referencje do klasy bazowej, a decyzje o to, która realizacje pozostanie wywołana, będzie podjęta przy wykonani programu na podstawie typu wskazywanego obiektu. Główna zaleta – elastyczność, która nadaje możliwość reagować na różne zdarzenia zachodzące dopiero w trakcie wykonania programu. Może wydłużyć czas wykonania (dla tego że wywołanie funkcji nie jest zdeterminowane aż do momentu jej wykonania).

Identyfikacja typu na etapie wykonania (RTTI)

- C++ implementuje polimorfizm poprzez hierarchie klas, funkcje wirtualne i wskaźniki klas bazowych. Wskaźniki klasy bazowej mogą służyć do wskazywania obiektów klasy bazowej, a także wszelkich obiektów klas pochodnych.

- 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. Dla rozwiązania tego problemu C++ przedstawia nam operatory *typeid* oraz *dynamic_cast*.

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

typeid(*object*) typeid(*reference_to_object*)

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

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

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

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

Przykład 1.

```
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() << " and " << typeid(y).name() << " ? \n";
	return (typeid(x) == typeid(y));
}

void main()
{
		int i = 0, j = 10;
		A a;
		B b;
		cout << funtypcompare(a, b) << endl;
		cout << funtypcompare(i, j) << endl;
		system("pause");
}
```

```
class A and class B ?
0
int and int ?
1
Press any key to return  . . .
```

➢ Przykład 3. Klasy polimorficzne. 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*. Dla klas polimorficznych operator *typeid*(**ptr*) zwraca automatycznie rzeczywisty typ, do którego wskazuje wskaźnik *ptr*.

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

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

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

	pb = &b;
	pb->f();
	cout << "pb points to the object of type " << typeid(*pb).name() << endl;

	pb = &d1;
	pb->f();
	cout << "pb points to the object of type " << typeid(*pb).name() << endl;

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

```
I am  B
pb points to the object of type class B
I am D1
pb points to the object of type class D1
bad pointer
Press any key to continue...
```

➢ Klasy nie polimorficzne: kod jest taki samy, jak poprzednio, tylko kluczowe słowo *virtual* zostało zakomentowane.

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

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

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

    pb = &b;
    pb->f();
    cout << " pb points to the object of type " << typeid(*pb).name() << endl;

    pb = &d1;
    pb->f();
    cout << " pb points to the object of type " << typeid(*pb).name() << endl;
}
```

```
I am B
pb points to the object of type class B
I am B
pb points to the object of type class B
Press any key to continue...
```

➢ Druga forma operatora *typeid* jest używana do porównania typów (czy ma obiekt typ podany?) :

typeid(*type_name*)

```
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;
        D1 d1;

        pb = &d1;
        if(typeid(*pb) == typeid(D1))
                cout << " pb points to the object of type D1\n";

        system("pause");
}
```

```
pb points to the object of type D1
Press any key to continue...
```

- Operator *typeid* można stosować do szablonów klas.

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 < *target_type* > (*expression*)

- *target_type* - typ docelowy rzutowania, *expression* - wyrażenie, które jest rzutowane do nowego typu.

- *expression* - musi być rozwijane do wskaźnika lub referencji.

- Wynik: powodzenie – *dynamic_cast* zwraca poprawny wskaźnik lub referencje; 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.

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

2. 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 może wskazywać do obiektu 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, wskaźnik klasy bazowej nie wskazuje do obiektu typu D1
```

Example W42.

- 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;           //jeśli tak, rzutujemy wskaźnik pb do typu 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 obiekt klasy D, działanie programu jest nie przewidywane. Takie błędy zwykłe bardzo ciężko wykryć w mniej-więcej rozbudowanym kodzie, dla tego że mogą występować nie za każdym uruchomieniem programu.

Example W43.

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

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

Operator *const_cast*

const_cast<typ> (*expression*)

- 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,
  - *expression* – wyrażenie, rzutowane na nowy typ.

```
void fun(const double *ptr)
{
    double *p = const_cast<double *> (ptr);  //usuwamy modyfikator const, inaczej kompilator zgłosi
                                              // niespójność typów (double i const double)

    //(*ptr)++;                    //błąd! const chroni tablicę ptr od zmian.
    (*p)++;                        //OK
    cout << *p << endl;
}

void main()
{
    double a[2] = {1.0, 4.0};
    fun(a);
    system("pause");
}

2
Press any key to continue . . .
```

Operator static_cast<> ()

static_cast<*typ*> (*expression*)

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

  - *typ* – typ docelowy
  - *expression* – 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*> (*expression*);

- służy do konwersji wskazanego typu na typ całkowicie odmienny.

```
void main()
{
    FILE *pf = fopen("myfile.dat", "w+");  //open file for read and write
    if(!pf) { //error handling }
    double arr[] = {1.0, -2.0, 3.141592};
    const char *str = reinterpret_cast<const char *>(arr);
    for(size_t it=0; it<sizeof(arr); ++it)
    {
        if(fputc(str[it], pf) == EOF) { // error handling }  //byte-by-byte writing to file
    }
    fseek(pf, 0, SEEK_SET); //move the file pointer to the beginning of file
    double brr[3];
    char *strs = reinterpret_cast<char *>(brr);
    char ch = 'a';    size_t it = 0;
    while(ch != EOF)
    {
        if((ch = fgetc(pf))  != EOF) {  // reading from file byte-by-byte until we meet EOF
                strs[it++] = ch;
        }
    }
    fclose(pf);  pf = NULL;  //close file
    remove("myfile.dat");    //remove file
}
```