Лекция 10. Динамическая идентификация и приведение типа

**Динамическая идентификация и приведение типа**

**Динамическая идентификация типа данных** (run-time typename identification, RTTI) - это механизм, позволяющий определить тип данных переменной или объекта во время выполнения программы. Для получения типа объекта во время выполнения программы используется оператор const type\_info& typeid(объект) - библиотека <typeinfo>.

**Некоторые методы класса typeinfo**

1) Перегруженные операторы сравнения:

bool operator==(consr tipeinfo &ob) const;

bool operator!=(consr tipeinfo &ob) const;

2) Метод, возвращающий указатель на имя типа:

const char \*name() const;

Пример.

#include <iostream>

#include <typeinfo>

using namespace std;

class BCls{

int x;

virtual void f() {};

};

class Cls01:public BCls{

int y;

};

class Cls02:public BCls{

int z;

};

int main()

{

int k;

cout<<"Type of k - "<<typeid(k).name()<<endl;//Выводит Type of k - int

cout<<"Type of 7.56 - "<<typeid(7.56).name()<<endl;//Выводит Type of 7.56 - double

char s[10];

cout<<"Type of s - "<<typeid(s).name()<<endl;//Выводит Type of s - char[10]

BCls a;

cout<<"Type of a - "<<typeid(a).name()<<endl;//Выводит Type of a - class BCls

Cls01 b;

cout<<"Type of b - "<<typeid(b).name()<<endl;//Выводит Type of b - class Cls01

Cls02 c, d;

cout<<"Type of c - "<<typeid(c).name()<<endl;//Выводит Type of c - class Cls02

BCls &m=c;

cout<<"Type of m - "<<typeid(m).name()<<endl;//Выводит Type of m - class Cls02

if(typeid(a)==typeid(b)) cout<<"The objects a and b have the same type"<<endl;

else cout<<"The objects a and b have a different data type"<<endl;

if(typeid(d)==typeid(m)) cout<<"The objects d and m have the same type"<<endl;

else cout<<"The objects d and m have a different data type"<<endl;

return 0;

}

**Преобразование типов**

Недостатки явного приведения типов языка С:

1) Возможность использования неконтролируемых компилятором преобразований типов, которые могут породить ошибку на этапе выполнения программы.

2) Указатель на базовый класс не может быть явно преобразован в указатель на производный класс, в котором базовый класс объявлен виртуально.

Данные недостатки в С++ устраняются использованием следующих технологий:

1) Однозначно интерпретируемым синтексисом операций.

2) Осуществлением контроля за возникающими ошибками как на этапе компиляции, так и на этапе выполнения программы.

3) Разрешением преобразования указателя на базовый виртуальный класс в указатель на производный класс.

**1. Константное преобразование типов**

тип const\_cast<тип>(выражение)

Правила языка С++ запрещают передачу константного указателя на место формального параметра, не имеющего модификатор const. Данная операция удаляет модификатор const, и получаемый тип соответствует типу выражения, за исключением модификатора const.

Пример.

void fun (char\*s){

cout<<s<<endl

}

int main()

{

const char str[]="func";

//fun(str);//Ошибка: аргумент типа const char\* несовместим с параметром типа char\*

fun(const\_cast<char\*>(str));

return 0;}

Пример. Увеличить значение поля объекта класса в 10 раз. Функция увеличения не может изменять значения полей своего класса.

class Cls{

int a;

public:

Cls(int g):a(g){};

void fun() const{

cout<<"a='<<a<<endl;

const\_cast<Cls\*>(this)->a\*=10;

cout<<"a="<<a<<endl;

}};

int main(){

Cls x(5);

x.fun();

}

**2. Динамическое приведение типов**

тип dynamic\_cast<тип>(выражение)

Динамическое приведение типа используется для преобразования указателей родственных классов иерархии с контролем корректности преобразований. Выражение должно быть указателем или ссылкой на класс, подлежащий преобразованию. Тип (тоже указатель или ссылка) должен быть на базовый или производный тип базового класса. Допустимо использование void\*. В случае допустимости и успешности преобразования операция возвращает указатель или ссылку на заданный тип, иначе NULL для указателя или bad\_cast для ссылки. Преобразование из базового класса в производный класс называется **понижающим**, а из производного в базовый - **повышающим**. Приведение типов между производными классами одного базового или между базовыми классами одного производного называется **перекрёстным преобразованием**.

Превышающее преобразование равносильно простому присваиванию.

При понижающем преобразовании компилятор не может проверить правильность приведения типов. Так как производные классы могут иметь собственные методы, то при вызове их через указатель на базовый класс необходимо проверить, действительно ли ссылается данный указатель на объект производного класса. Такая проверка выполняется в момент использования операции динамического приведения типов. Для выполнения проверки допустимости преобразования аргумент операции dynamic\_cast должен быть виртуальным, т. к. операция использует информацию, хранящуюся в таблице виртуальных методов. Результат выполнения операции требуется проверять явным образом.

Пример. Понижающее преобразование.

#include <iostream>

//#include <typeinfo>

//#include <eh.h>

using namespace std;

class BCls{

public:

virtual ~BCls() {}

};

class Cls01:public BCls{};

void flink(BCls \*uk){

try{

cout<<"Link:";

Cls01&rb=dynamic\_cast<Cls01&>(\*uk);

cout<<"Object is created"<<endl;

}

catch(bad\_cast &x){

cout<<x.what()<<endl;

}

}

void fpointer(BCls \*uk){

try{

cout<<"Pointer: ";

Cls01\*pb=dynamic\_cast<Cls01\*>(uk);

if(!pb) throw "Error";

cout<<"Object is created"<<endl;

}

catch(char \*st){

cout<<st<<endl;

}

}

int main(){

BCls\* uk=new BCls;

flink(uk);

fpointer(uk);

delete uk;

uk=new Cls01;

flink(uk);

fpointer(uk);

delete uk;

return 0;}

Перекрёстное преобразование позволяет безопасно связывать производные классы, созданные в разные моменты времени.

Пример. Перекрёстное преобразование.

#include <iostream>

using namespace std;

class BCls01{

public:

virtual void output(char \*s){

cout<<endl<<s;

}

};

class BCls02{

protected:

int x, y;

public:

BCls02(int a, int b):x(a), y(b){}

virtual ~BCls02() {}

friend ostream& operator<<(ostream &out, BCls02 &p);

};

ostream& operator<<(ostream &out, BCls02 &p)

{

BCls01 \*uk=dynamic\_cast<BCls01\*> (&p);

if(uk) uk->output("BCls02: ");

out<<"x="<<p.x<<", y="<<p.y;

return out;

}

class Cls01:public BCls02, public BCls01{

public:

Cls01(int a, int b):BCls02(a,b){}

};

class Cls02:public BCls01{};

class Cls03:public BCls02, public Cls02{

public:

Cls03(int a, int b):BCls02(a,b){}

};

int main(){

BCls02 ob01(1, 2);

cout<<ob01;//x=1, y=2

Cls01 ob02(3, 4);

cout<<ob02;//BCls02: x=3, y=4

Cls03 ob03(5, 6);

cout<<ob03;//BCls02: x=5, y=6

return 0;}

Другие способы для приведения типов используются для работы с невиртуальными классами:

static\_cast<идентификатор\_типа>(выражение)

reinterpret\_cast<идентификатор\_типа>(выражение)