Модульное программирование

Пространства имен

Пространства имен

Области действия и пространства имен

Каждый программный объект имеет область действия и время жизни, которые определяются видом и местом его определения. Существуют следующие разновидности областей действия:

- блок;
- прототип функции;

Пространства имен

- функция;
- файл;
- группа файлов, в пределе включающая все файлы программного проекта (глобальная область действия);
- класс;
- пространство имен (часть глобальной области).

Пространства имен

Функция. Программные объекты, определенные в блоке функции, имеют область действия и время жизни точно такие же как в обычном блоке.

Параметры функции, передаваемые по значению, имеют областью действия всю функцию, а время жизни – время работы функции.

Параметры функции, передаваемые по ссылке, имеют область действия и время жизни, определяемое соответствующими аргументами в вызове функции.

Пространства имен

Локальные объекты, объявленные в теле функции, действуют в пределах конкретного блока. Время жизни – время работы функции. В период работы функции эти объекты хранятся в программном стеке. От вызова к вызову их значения не сохраняются.

Если есть необходимость сохранить значение локальных объектов, их необходимо объявить с модификатором класса памяти static.

Пространства имен

В этом случае переменные будут храниться в программном сегменте и время их жизни совпадает со временем работы программы.

Пример:

```
double func(double d)
{
    static double temp = 3.5;
    cout << " Temp: " << temp++ << endl;
    return d*temp;
}
```

Переменная static double temp будет сохраняться от одного вызова функции к другому.

Пространства имен

Файл. Программный объект, определенный с описателем класса static вне любого блока, функции, класса, имеет областью действия, начинающуюся в точке его объявления и заканчивается в конце файла. В область действия попадают вложенные блоки.

Если во внутреннем блоке определен объект с таким же именем, тогда внешний объект становится невидимым.

Пространства имен

Но обратиться к нему можно через оператор разрешения области видимости -::.

Класс. Компоненты класса (поля, методы), за исключением статических, имеют областью действия класс. Время жизни компонентов класса определяется промежутком времени от создания объекта до его разрушения.

Пространства имен

Пространства имен (именованные области). В C++ есть возможность явным образом задать область действия имен как часть глобальной области с помощью оператора namespace.

Пространства имен

Пространства имен

Пространство имен (именованная область) служит для логического группирования определений, объявлений и ограничения доступа к ним. Чем больше объем программы, тем актуальнее использование именованных областей. Их удобно использовать в больших программных проектах.

Пространства имен

Общий формат объявления именованной области следующий:

namespace [имя_области]

{ /* определения и объявления }

Одно и то же пространство имен может объявляться многократно, причем все последующие будут пониматься как продолжения предыдущих.

Пространства имен

Продолжение именованных областей можно делать в различных файлах.

Рассмотрим простой пример:

```
namespace demo
{
    int I = 1;   //    определение переменной
    int k = 0;  //    определение переменной
    //      прототип функции
    void fun_1(int);
    //      определение функции
    int fun_2(int I, int j)
    {
      //
    }
}
```

Пространства имен

Дальнейшее расширение пространства

```
namespace demo
{
   // int i = 2; ошибка, повторное объявление
   void func_1(double);
}
```

Если имя области не задано (анонимная область), компилятор определяет его самостоятельно с помощью уникального идентификатора.

Пространства имен

Нельзя получить доступ из именованной области одного файла к неименованной области другого файла.

В именованной области логичнее всего помещать объявления объектов, а их определения выносить в файлы реализации, например,

```
void demo:: func_1(double d)
{
   //   тело функции
}
```

Пространства имен

Такой прием обеспечивает разделение интерфейса и реализации.

Объекты программы, определенные внутри пространства имен, становятся доступными с момента объявления пространства. Обратиться к ним можно с помощью имени области и оператора доступа к области видимости, например, `demo:: i == 100;`

Пространства имен

Если какое-либо имя из именованной области используется часто, его можно сделать доступным с помощью оператора using, например, using demo:: I;.

После чего к нему можно обращаться без указания области видимости.

Пространства имен

Если требуется открыть всю область видимости, используется оператор

using namespace demo;

Вспомните, например,

using namespace std;

Именованные области могут быть вложены друг в друга.

Преобразования в стиле языка C++

Преобразования в стиле C++

При выполнении программы производятся явные и неявные преобразования величин из одного типа в другой. Неявные преобразования типов происходит в соответствие со стандартом языка, то есть, от «меньшего» типа к «большему». Например, величины типа bool, char будут автоматически приведены к типу int, а int и float – к типу double.

Преобразования в стиле языка C++

Простой пример:

```
int var_int =22;
double var_double = 3.45;
//
cout << var_int * var_double << cout;
```

То есть, выражение var_int * var_double, следует понимать как

```
(double)var_int * var_double;
```

Преобразования в стиле языка C++

В действительности такие преобразования не делаются, поскольку компилятор делает их автоматически.

Преобразование от «большего» типа к «меньшему» необходимо указывать явно, например,

```
var_int * (int)var_double;
```

Необходимо помнить, что в данном случае будет потеря точности.

Преобразования в стиле языка C++

Общий формат преобразования в стиле C следующий:

тип(выражение);

(тип)выражение;

Необходимость в преобразовании типов возникает, например, в случае когда функция возвращает указатель на тип void, который необходимо присвоить переменной конкретного типа для последующих действий:

```
float *p = (float *)malloc(100 * sizeof(float));
```

Преобразования в стиле языка C++

Еще один пример – это переопределение операции new, которая всегда возвращает указатель на тип void.

Явные преобразования типа являются источником возможных ошибок, поскольку вся ответственность за его результат возлагается на программиста. Поэтому в C++ введены операции, позволяющие выполнять частичный контроль над преобразованиями.

Преобразования в стиле языка C++

Операция const_cast

Эта операция служит для удаления модификатора const. Как правило она используется при передаче в функцию константного указателя на место формального параметра , не имеющего данного модификатора.

Преобразования в стиле языка C++

Общий формат операции следующий:

const_cast<тип>(выражение)

Обозначенный тип должен быть таким же, как и тип выражения, за исключением модификатора const. Обычно это указатель. Операция формирует результат указанного типа.

Преобразования в стиле языка C++

Необходимость введения этой операции обусловлена тем, что программист, реализующий функцию, не обязан описывать формальные параметры как неизменяемые (const), хотя это рекомендуется. Правила C++ запрещают передачу константного указателя на место обычного.

Преобразования в стиле языка C++

Рассмотрим пример:

```
void func(int *arg)
{
   cout << " Arg: " << *arg << endl;
}
//
const int var_int = 10;
func(&var_int);     // Ошибка !!!
func(const_cast<int *>(&var_int));   // Ошибки нет
```

Преобразования в стиле языка C++

Еще один пример, более близкий к ООП:

```
class Test
{
    int test;
public:
    Test(){};
    Test(int t):test(t){};
    void Out()
    { cout << " Test: " << test  << endl; }
};
```

Преобразования в стиле языка C++

Объявим константный указатель на объект данного класса:

```
const Test *ptr_Test = new Test(100);
```

Попытка следующего обращения приведет к ошибке

```
ptr_Test->Out();, но
```

вызов

```
const_cast<Test *>(ptr_Test)->Out();
```

будет абсолютно верным.

Преобразования в стиле языка C++

Ошибку подобного рода можно избежать, объявив метод Out() как константный (безопасный):

```
void Out() const
{
   cout << " Test: " << test  << endl;
}
```

Преобразования в стиле языка C++

Операция static_cast

Операция static_cast используется на этапе компиляции между:

- целыми типами;
- целыми и вещественными;
- целыми и перечислимыми;
- указателями и ссылками на объекты одной иерархии, при условии, что оно однозначно и не связанно с понижающим преобразованием виртуального базового класса.

Преобразования в стиле языка C++

Формат операции:

static_cast<тип>(выражение)

Результат операции имеет указанный тип, который может быть ссылкой, указателем, арифметическим или перечислимым типом.

При выполнении операции внутреннее представление может быть модифицировано, хотя численное значение остается неизменным.

Преобразования в стиле языка C++

Например,

```
float f = 100;
int i = static_cast<int>(f);
```

Преобразования подобного рода должны иметь серьезное основание. Результат преобразования остается на совести программиста.

Перечисленные преобразования попробуйте самостоятельно.

Преобразования в стиле языка C++

Мы же рассмотрим пример преобразований в иерархии родственных классов, что для нас имеет больший интерес.

Рассмотрим пример простой иерархии.

Преобразования в стиле языка C++

Базовый класс:

```
class Base
{
protected:
    int base;
public:
    Base(){};
    void Out();
};
void Base::Out()
{
cout << " Base class " << endl;
}
```

Преобразования в стиле языка C++

Производный класс:

```
class Derived : public Base
{
    int derived;
public:
    Derived():Base(){};
    void Out();
};
void Derived::Out()
{
    cout << " Derived class " << endl;
}
```

Преобразования в стиле языка C++

Рассмотрим несколько примеров преобразований между этими классами.

Преобразования «вверх», то есть от объекта производного типа к типу базового класса, относятся к стандартным преобразованиям и не требуют явных преобразований. Например,

```
Derived der;
Base base = der;
```

Преобразования в стиле языка C++

Выражение Base base = der; вполне оправдано, хотя правильнее (понятней) была бы запись: Base base = (Base)der;,

или Base base = static_cast<Base>(der);.

Как было сказано, что преобразования «вверх» относятся к стандартным, то последнее преобразование не является обязательным.

Преобразования в стиле языка C++

Рассмотренное преобразование типа производного класса возможно, если производный класс описан с обобществленным базовым классом, как в нашем примере:

class Derived : public Base {}

Если объявить базовый класс как защищенный или закрытый

class Derived : protected Base {}

Подобные преобразования будут возможны, но не доступны.

Преобразования в стиле языка C++

Преобразования объектов производного класса к типу базового встречается на практике достаточно часто, например при инициализации объекта базового класса объектом производного.

Более интересный пример – попытка преобразования «вниз», то есть из типа базового класса в производный тип.

Преобразования в стиле языка C++

Следует оговориться, что преобразования допустимы только уровне указателей или ссылок, но не объектов.

Для рассмотренных классов:

```
Base *ptr_Base = new Base();
ptr_Derived = static_cast<Derived *>(ptr_Base);
ptr_Derived->Out();
```

Что можно ожидать в этом случае?

Преобразования в стиле языка C++

Сработает функция производного класса и выведет «Derived class».

А что будет при вызове ptr_Base->Out(); ?

В данном случае вывод будет: «Base class».

Ничего странного, ничего не обычного.

Изменим несколько объявления:

```
Base *ptr_Base = new Derived();
```

Преобразования в стиле языка C++

Такое объявление также допустимо, указатель на базовый класс инициализируется значением указателя на производный.

Вызов ptr_Base->Out(); приведет к активизации функции базового класса.

Для того чтобы активизировать функцию производного класса, функцию базового класса нужно описать как виртуальную: virtual void Out();

Это проявление полиморфизма в C++.

Преобразования в стиле языка C++

Операция dynamic_cast

Эта операция используется в основном для преобразования указателей и ссылок на объекты базового типа в указатели и ссылки на производный тип. При этом во время выполнения программы появляется возможность проверки (контроля) допустимости преобразований.

Преобразования в стиле языка C++

Несложно догадаться, что преобразования будут выполняться в период выполнения программы.

Общий формат операции:

dynamic_cast<тип>(выражение)

Выражение должно быть указателем или ссылкой на класс, тип – базовым или производным для данного класса.

Преобразования в стиле языка C++

В случае успешного выполнения операции формируется результат заданного типа, в противном случае для указателей результат равен нулю, а для ссылок порождается исключение bad_cast. Если заданный тип не относится к одной иерархии, преобразования не допускаются.

Все дальнейшие рассуждения при рассмотрении наследования и полиморфизма.

Преобразования в стиле языка C++

Операция reinterpret_cast

Операция reinterpret_cast применяется для преобразования не связанных между собой типов, например, указателе в целые типы или наоборот, а также указателей типа void в конкретный тип. При этом внутреннее представление данных не меняется, меняется только точка зрения компилятора на данные.

Преобразования в стиле языка C++

Рассмотрим пример:

```
struct Struct
{
    int str_int;
    string str_string;
    Struct(int s_i, string s_s):
    str_int(s_i), str_string(s_s){};
    void Out()
    { cout << str_int << ' ' << str_string << endl; }
};
```

Преобразования в стиле языка C++

Далее использование объекта этого типа через указатель на тип void *

```
Struct str(120, "string");
void *ptr_void = &str;
reinterpret_cast<Struct *>(ptr_void)->Out();
```

Этот пример того, что не следует делать в практическом программировании, поскольку результат операции останется на совести программиста.

Преобразования в стиле языка C++

Практическое использование этого оператора при форматированном вводе-выводе числовых величин.

```
#include<fstream>
#include<iostream>
const int MAX = 100;
int buffer[MAX];
```

Преобразования в стиле языка C++

```
// создаем выходной поток
   ofstream os(“data.dat”, ios::binary);
// записываем в него
   os.write(reinterpret_cast<char *>(buffer),
   Max*sizeof(int));
// закрываем поток
   os.close();
```

Преобразования в стиле языка C++

В данном случае использование оператора reinterpret_cast вполне оправдано, поскольку его действие позволяет существенно сократить ресурсы памяти.