Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

26 мая 2015

Преобразования типов

Операция const_cast.

```
void print(int *p) { cout << *p; }
const int *p;
print(p); //ошибка
int *j = const_cast<int*>(p);
```

Преобразование типов

Операция dynamic_cast. Преобразования бывают двух типов:

- повышающее – приведение производного класса к базовому
- понижающее — из базового класса в производный
- перекрестное — приведение между производными типами

dynamic_cast

Повышающее преобразование.

```
class B{ ... };
class C : public B{ ... };
C *c = new C;
B *b = dynamic_cast<B*>(c); //Эквивалентно B *b = c
    ;
```

dynamic_cast

Понижающее преобразование. Применяется когда компилятор не может проверить правильность приведения. Для использования данного типа преобразований необходимо включить механизм RTTI.

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
C *c = new C;
B *b = new B;
C *temp = dynamic_cast<C*>(b);
if(temp)
temp->f2();
```

dynamic_cast

Перекрестное преобразование.

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
class D : public B{ ... };
D *d = new D;
C *temp = dynamic_cast<C*>(d);
if(temp)
temp->f2();
```

dynamic_cast

Для доступа к RTTI введена операция typeid и класс type_info.

```
class B { ... };
class C : public B { ... };
B *p = new C;
if(typeid(*p) == typeid(C))
{
dynamic_cast<C*>(p)->f2();
cout << typeid(*p).name();
}
```

static_cast

Выполняется на этапе компиляции над:

- целыми типами
- целыми и вещественными типами
- целыми и перечисляемыми типами
- указателями и ссылками одной иерархии

reinterpret_cast

Применяется для преобразования не связных между собой типов, например указателей в целые и наоборот.

```
char *p = reinterpret_cast <char*>(malloc(10));
```

std::string

Класс string входит в стандартную библиотеку C++.

```
string s1("Text");
string s2;

char *str = "Txt";
string s3(str);
```

std::string

Функции:

```
s.at(1);
s1.append(s2); // +
s1.append(s2, 3, 5);
s1.insert(1, str2, 3, 5);
s1.replace(1, 2, str2, 5);
s1.swap(s2);
s1.c_str(); //возвращает const char*
s1.find(s2); //возвращает позицию
s1.compare(1, 2, s2, 1, 2); //аналог strstr
```

Контейнерные классы

Последовательные контейнеры:

- vector — структура, эффективно реализующая произвольныей доступ к элементам, а также добавление в конец и удаление из конца;
- dequeue — структура, эффективно реализующая произвольныей доступ к элементам, а также добавление в оба конеца и удаление из обоих концов;
- list — список, эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

std::vector

```
vector<int> v;
v.push_back(5);
v.push_back(6);
v.push_back(7);
for( vector<int>::iterator i = v.begin(); i != v.
    end(); i++)
{ cout << *i << "␣"; }
```

Функции:

```
insert() //вставка в произвольное место
erase() //удаление из произвольного места
at, [] //произвольный доступ к элементу
```

Ассоциативные контейнеры:

- map — словарь
- multimap — словарь с дубликатами

std::map

```
map<int, int> maps;
maps[100] = 1;
maps[200] = 3;
```