Множественное наследование и RTTI в C++

Звонарев Никита

СПбГУ, СтатМод

22 сентября 2017 г.

Введение: наследование

Наследование (inheritance) — концепция ООП, заключающаяся в построении новых типов, которые наследуют данные и функциональность уже существующего типа.
C++:

```
class A {}; // Базовый класс / предок /
// суперкласс

class B : public A {}; // Класс-потомок /
// производный класс / подкласс
```

Напоминание: типы наследования

```
class A {};

class B : public A {};
class C : protected A {};
class D : private A {};
```

- public наследование: public-члены базового класса доступны как public-члены производного класса; protected-члены базового класса — доступны как protected-члены производного класса
- protected наследование: public- и protected- члены базового класса доступны как protected-члены производного класса
- private наследование: public- и protected- члены базового класса доступны как private-члены производного класса

Множественное наследование

Множественное наследование (multiple inheritance) — наследование от более чем одного базового класса!

```
class satellite : public task, public displayed {
// ...
};
```

По аналогии: наследование от одного класса — единственное наследование.

Множественное наследование: объединение полей

```
void f(satellite& s)
{
s.draw();      // displayed::draw()
s.delay(10);   // task::delay()
s.xmit();      // satellite::xmit()
}
```

Можно делать так:

```
void highlight(displayed*);
void suspend(task*);

void g(satellite* p)
{
highlight(p);    // highlight((displayed*)p)
suspend(p);      // suspend((task*)p);
}
```

Множественное вхождение базового класса

Конфликта не возникнет: два различных объекта link.

```
class task : public link {
// link используется для связывания всех
// задач в список (список диспетчера)

// ...
};

class displayed : public link {
// link используется для связывания всех
// изображаемых объектов (список изображений)

// ...
};
```

Разрешение неоднозначности

Допустим, есть функции-члены с одинаковыми именами.

```
class task {
// ...
virtual debug_info* get_debug();
};
class displayed {
// ...
virtual debug_info* get_debug();
};
//...
void f(satellite* sp)
{
debug_info* dip = sp->get_debug(); //ошибка
dip = sp->task::get_debug();          // нормально
dip = sp->displayed::get_debug();   // нормально
}
```

Виртуальный базовый класс

Хотим, чтобы базовый класс входил один раз. Как сделать?

```
class window {
virtual void draw();
};
class window_w_border : public virtual window {
// определения, связанные с рамкой
void draw(); };
class window_w_menu : public virtual window {
// определения, связанные с меню
void draw(); };
class window_w_border_and_menu
: public virtual window,
public window_w_border,
public window_w_menu {
void draw();
};
```

draw: неверная реализация

```
void window_w_border::draw()
{
window::draw(); // рисуем рамку
}
void window_w_menu::draw()
{
window::draw(); // рисуем меню
}

void window_w_border_and_menu::draw() // ловушка!
{
window_w_border::draw();
window_w_menu::draw();
// теперь операции, относящиеся только
// к окну с рамкой и меню
}
```

draw: корректная реализация

```
class window {
void _draw();
void draw();
};
class window_w_border : public virtual window {
void _draw();
void draw(); };
void window_w_border::draw() {
  window::_draw();
  _draw();    // рисует рамку
};
class window_w_border_and_menu:
public virtual window,
public window_w_border, public window_w_menu {
void _draw(); void draw();
};
```

draw: корректная реализация, часть 2

```
void window_w_border_and_menu::draw()
{
window::_draw();
window_w_border::_draw();
window_w_menu::_draw();
_draw();   // теперь операции, относящиеся только
// к окну с рамкой и меню
}
```

RTTI

RTTI (*Run-time type identification*) — возможность в C++ узнать тип объекта, имея указатель только на базовый класс
Два синтаксиса:

```
cout << typeid(*s).name() << endl;
```

`typeid()` — выглядит как функция, но реализована компилятором (как и `sizeof()`). Возвращает ссылку на глобальный константный объект типа `std::type_info`.

```
Shape* sp = new Circle;
Circle* cp = dynamic_cast<Circle*>(sp);
if(cp) cout << "cast␣successful";
```

Преобразование `Circle*` к `Shape*` называется *upcast*, `Shape*` к `Circle*` — *downcast*

typeid() и std::type_info

Объекты типа `std::type_info` можно сравнивать друг с другом с помощью == и !=.
Метод `name()` возвращает имя. Будьте осторожны: может понадобиться т.н. demangle, если хотите человечески читаемое имя типа (компиляторо-специфично). См. `abi::__cxa_demangle`.
Работает корректно со встроенными типами:

```
#include <cassert>
#include <typeinfo>
using namespace std;

int main() {
assert(typeid(47) == typeid(int));
assert(typeid(0) == typeid(int));
int i;
assert(typeid(i) == typeid(int));
assert(typeid(&i) == typeid(int*));}
```

typeid() и вложенные классы

```
#include <iostream>
#include <typeinfo>
using namespace std;
class One {
  class Nested {};
  Nested* n;
  public:
    One() : n(new Nested) {}
    ~One() { delete n; }
    Nested* nested() { return n; }
};
int main() {
  One o;
  cout << typeid(*o.nested()).name() << endl;
//One::Nested
}
```

typeid() и неполиморфные типы

```
#include <cassert>
#include <typeinfo>
using namespace std;
class X {
  int i;
};
class Y : public X {
  int j;
};
int main() {
  X* xp = new Y;
  assert(typeid(*xp) == typeid(X));
  assert(typeid(*xp) != typeid(Y));
}
```

RTTI не работает без полиморфизма — использует статическую информацию.

typeid() и ссылки, dynamic_cast

См. листочки

`dynamic_cast` возвращает ссылку на объект, если каст успешен, и 0 в противном случае.
Также кидается исключения типа `bad_cast`.
`typeid` от нулевого указателя сгенерирует `bad_typeid`.

Explicit cast syntax

`static_cast` — for "well-behaved" and "reasonably well-behaved" casts, including things you might now do without a cast (e.g., an upcast or automatic type conversion).
`const_cast` — to cast away `const` and/or `volatile`.
`dynamic_cast` — for type-safe downcasting
`reinterpret_cast` — to cast to a completely different meaning. The key is that you'll need to cast back to the original type to use it safely. The type you cast to is typically used only for bit twiddling or some other mysterious purpose. This is the most dangerous of all the casts.

Хороший стиль в C++ — использовать такое приведение вместо приведения в стиле C.