Таблица виртуальных методов — конспект темы

Виртуальные методы

Если позволить объекту иметь размер 0, может возникнуть ситуация, когда у разных объектов будет один и тот же адрес. Размер пустого объекта в C++ равен 1.

Ловкость рук и никакого волшебства

`vftable` — **указатель на виртуальную таблицу** или **указатель на таблицу виртуальных функций**. Он занимает 8 байт.

Таблица виртуальных методов

RTTI (run-time type identification) — **динамическая идентификация типа данных**. Видя указание на этот механизм, можно предположить, что тип данных на момент компиляции ещё не известен, а функция не вызвана. Чтобы вызвать функцию, когда объект станет известен, используют **позднее связывание:**

- Программа видит, что пользователь просит вызвать метод `WhatInside`.
- Программа не находит адреса вызова функции, но видит, что этот метод — виртуальный.
- Программа обращается по адресу указателя `vftable` и находит указатель на метод `WhatInside` внутри таблицы виртуальных методов.
- Вызывает функцию из виртуальной таблицы.

Работая с поздним связыванием, помните: то, что компилятор не проверил до запуска, может сломаться во время работы. В C++ везде, где возможно, используется раннее связывание.

Конструкторы и деструкторы

Кроме оценки количества операций важно уметь оценивать количество затраченной памяти. Принцип оценки тот же: насколько быстро растёт объём

затрачиваемой памяти в зависимости от входных параметров. Это **оценка пространственной сложности**.

Адреса виртуальных методов одинаковы для всех объектов класса. Нет смысла хранить таблицу для каждого объекта — она будет для всех одинакова. Затраченная память не зависит от числа объектов, она константная.

Вот что происходит с `vptr` в конструкторах и деструкторах:

- создадим на стеке три переменных,
- их деструкторы будут вызваны при выходе из функции `main` автоматически.

```
void PrintVptr(void* obj) {
    auto ptr = reinterpret_cast<size_t*>(obj);
    cout << "Vptr : "s << *ptr << endl;
}

class MagicHat {
public:
    MagicHat() {
        cout << "MagicHat::Ctor : "s;
        PrintVptr(this);
    }
    ~MagicHat() {
        cout << "MagicHat::Dtor : "s;
        PrintVptr(this);
    ...
};

class HatWithApples : public MagicHat {
public:
    HatWithApples() {
        cout << "HatWithApples::Ctor : "s;
        PrintVptr(this);
    }
    ~HatWithApples() {
        cout << "HatWithApples::Dtor : "s;
        PrintVptr(this);
    }
    ...
};

class HatWithRabbits : public MagicHat {
public:
    HatWithRabbits() {
        cout << "HatWithRabbits::Ctor : "s;
        PrintVptr(this);
    }
    ~HatWithRabbits() {
```

```
        cout << "HatWithRabbits::Dtor : "s;
        PrintVptr(this);
    ...
};

class HatWithDoves : public MagicHat {
public:
    HatWithDoves() {
        cout << "HatWithDoves::Ctor : "s;
        PrintVptr(this);
    }
    ~HatWithDoves() {
        cout << "HatWithDoves::Dtor : "s;
        PrintVptr(this);
    }
    ...
};

int main() {
    HatWithApples hat1;
    HatWithRabbits hat2;
    HatWithDoves hat3;
}
```

Результат работы программы:

```
MagicHat::Ctor : Vptr : 4220880
HatWithApples::Ctor : Vptr : 4220816
MagicHat::Ctor : Vptr : 4220880
HatWithRabbits::Ctor : Vptr : 4220848
MagicHat::Ctor : Vptr : 4220880
HatWithDoves::Ctor : Vptr : 4220784
HatWithDoves::Dtor : Vptr : 4220784
MagicHat::Dtor : Vptr : 4220880
HatWithRabbits::Dtor : Vptr : 4220848
MagicHat::Dtor : Vptr : 4220880
HatWithApples::Dtor : Vptr : 4220816
MagicHat::Dtor : Vptr : 4220880
```

Внутри каждого объекта прячется объект базового класса. Поэтому каждому объекту вызывается два конструктора, а потом два деструктора соответственно.

Указатель `vptr` перед вызовом конструктора устанавливается на таблицу виртуальных функций того класса, конструктор которого вызывается. Такой механизм гарантирует, что конструктор базового объекта не попытается обратиться к методам класса-наследника, который на данный момент ещё не сконструирован.

С деструкторами то же самое, но есть нюанс: при компиляции не всегда известно, какой динамический тип у удаляемого объекта. Деструктор класса, содержащего виртуальные методы, тоже должен быть виртуальным. Тогда позднее связывание решит задачу, будет вызван деструктор из виртуальной таблицы соответствующего класса. Все ресурсы освободятся.

Pure virtual

Абстрактный метод — метод, который нельзя осмысленно определить для данного класса, но который будет иметь смысл для потомков.

Вот как выглядят абстрактные методы `WhatInside` и `HaveSomethingInside`. Оставим их реализацию во всех классах-наследниках, но объявим их `= 0` в классе `MagicHat`:

```
class MagicHat {
public:
    MagicHat() {
        cout << "MagicHat::Ctor"s;
        if (HasSomethingInside()) {
          WhatInside();
        }
    }
    ~MagicHat() {
        cout << "MagicHat::Dtor"s;
        PrintVptr(this);
    }
    virtual void IsMagicHat() const {
      cout << "MagicHat:: I'm a magic hat"s << endl;
    }
    virtual void WhatInside() const = 0;
    virtual bool HasSomethingInside() const = 0;
};
```

Такой код не скомпилируется, несмотря на то что пользователь вызывает метод, определённый в `MagicHat`:

```
int main() {
    MagicHat hat;
    hat.IsMagicHat();
}
```

Объекты типа `MagicHat` на самом деле существуют внутри объектов классов-наследников. Конструктор этого метода точно будет вызван, если создать объект

класса `HatWithApples`.