Ссылки. Инкапсуляция.

Введение объектно-ориентированное программирование

13.02.2025

Немного об указателях

- Если указатель это просто расстояние, может быть и нулевое расстояние?
- Нулевой указатель это специальный "маркер ничего". По нему ничего не лежит.
- Не надо путать 0, NULL и nullptr. if (!p) { smth(); } // сработает во всех трёх случаях
- В языке C++ наш выбор nullptr и мы поймём почему это так когда дойдём до перегрузки функций.

Синтаксис ссылок

- Базовый синтаксис lvalue ссылок это одинарный амперсанд.

```
int x;
int &y = x; // теперь y это просто ещё одно имя для x
```

- Чем же ссылки удобнее указателей?

```
int x[2] = {10, 20};
int &xref = x[0];
int *xptr = &x[0];
xref += 1;
xptr += 1;
assert(xref == 11);
assert(*xptr == 20);
```

Правила для ссылок

- Единожды связанную ссылку нельзя перевязать.

```
int x, y;
int &xref = x; // теперь нет возможности связать имя xref с
переменной y
xref = y; // то же, что x = y
```

- Ссылки прозрачны для операций, включая взятие адреса.

```
int *xptr = &xref; // то же самое, что &x
```

- Сами ссылки не имеют адреса. Нельзя сделать указатель на ссылку.

```
int &*xrefptr = &xref; // ошибка
int *& xptrref = xptr; // ok, ссылка на указатель
```

Константность для ссылок

- Все ли помнят правила константности для указателей? const char *s1; // ?
char const *s2; // ?
char * const s3; // ?
char const * const s4; // ?

Константность для ссылок

- Все ли помнят правила константности для указателей?

const char *s1; // указатель на константные данные (west-const)
char const *s2; // указатель на константные данные (east-const)
char * const s3; // константный указатель на (изменяемые) данные
char const * const s4; // константный указатель на константные данные

- Правила для ссылок гораздо проще.

char &r1 = r; // неконстантная ссылка (на изменяемые данные)
const char &r2 = r1; // константная ссылка (на константные данные)

Использование ссылок

• Представим некую функцию, которой нужно читать два тяжёлых объекта.
• Эта сигнатура плоха (все ли понимают чем?)
int foo(Heavy fst, Heavier snd) {// fst.x
• Эта сигнатура куда лучше но придётся разыменовывать указатели.
int foo(const Heavy *fst, const Heavier *snd) { // fst->x
• Эта сигнатура использует указатели неявно.
int foo(const Heavy &fst, const Heavier &snd) { // fst.x

Использование ссылок

• Синонимы внутри больших объектов.

```
void mytype::change_internal(some_big_obj &obj) {
  int &internal = obj.somewhere[5].guts.internal;
// код, активно изменяющий internal
}
```

• Здесь разница заметнее. Указатель был бы ячейкой памяти. Ссылка это просто имя.

• Кроме того, указатель всегда может быть изменён.

```
int *internal = &obj.somewhere[5].guts.internal;
internal += 5; // не имеет смысла тут, но всегда возможно!
```

Немного священных войн

• Многие считают, что ссылка это плохой out-параметр так как она не очевидна при вызове.
• Другие считают, что указатель плох как out-параметр, т.к. он двусмысленен.

```
void foo(int &);
void bar(int *); // не очевидно, что это не массив
int x;
foo(x); // не очевидно, что x это out-param
bar(&x);
```

•Что вы думаете?

Немного священных войн

• Дополнительный аргумент это состояние внутри функции.

```
void foo(int &x) {
// очевидно, что x содержит int
}
void bar(int *x) {
// не очевидно, что x не nullptr
}
```

• Более ограниченный интерфейс ссылок часто позволяет сократить рантайм- проверки.

Обсуждение

- Как вы думаете, почему this это указатель а не ссылка?

Case study: список

• Давайте ещё разок напишем его на C++.

```
struct list_t {
  struct node_t {
  node_t *next_, *prev_;
  int data_;
  };
  node_t *top_, *back_;
};
```

• Можем ли мы написать для него метод length?

Case study: список

```
size_t list_t::length() const {
  size_t len = 0;
  node_t *cur = top_;
  while(cur != nullptr) {
    len += 1;
    cur = cur->next_;
  }
  return len;
};
```

- Что не так с этим методом?

Case study: список

```
size_t list_t::length() const {
  size_t len = 0;
  node_t *cur = top_;
  while(cur != nullptr) { // а с чего мы взяли, что нет петли?
    len += 1;
    cur = cur->next_;
  }
  return len;
};
```

- Он может иметь недетерминированное время работы.

Case study: список

```
list_t l;
// тут как-то заполняем
l.top_->next_ = l.top_; // oops
size_t len = l.length();
```

- Можем ли мы проверить, что в списке нет петли?
- Алгоритм Флойда вычисляет количество элементорв даже если петля есть.
- Но что если мы хотим теперь написать метод reverse?
- Надо ли в начале reverse опять вызывать алгоритм Флойда, проверяя нет ли петли и удваивая общее время работы?

Инварианты

• Все методы списка существенно упростятся, если он сможет сохранять свои инварианты.
• Что для этого нужно?
• Есть методы типа, которые пишем мы как разработчики типа. Сохранять инварианты в методах – обязанность разработчика и он обычно с ней справляется.
• Но есть внешние функции, работающие с объектами этого типа. И вот они как раз являются источником проблем.
• Есть ли у нас языковые средства, чтобы запретить всем, кроме методов класса, работать с его состоянием?

Инкапсуляция в языке C++

• В языке C++ для инкапсуляции используется специальный механизм private, позволяющий ограничить видимость полей и методов.

```
struct list_t {
private:
  struct node_t;
  node_t *top_, *back_;
public:
  int length() const;
};
```

• В структуре по умолчанию все поля public.

Инкапсуляция в языке C++

• В языке C++ для инкапсуляции используется специальный механизм private, позволяющий ограничить видимость полей и методов.

```
class list_t {
//private:
  struct node_t;
  node_t *top_, *back_;
public:
  int length() const;
```

• Новое ключевое слово class определяет по умолчанию закрытые поля.

Неконсистентное состояние

- У нас есть линейная модель памяти
- Разве это не значит, что просто приведя указатель объект к char* мы можем нарушить все инварианты?
- Да можем (по крайней мере для standard-layout и для trivially copyable). Идея в том, что мы не хотим этого делать.
- Объект у которого нарушены инварианты это объект в неконсистентном состоянии операции над ним опасны и непредсказуемы.
- Никакой программист, будучи в своём уме, не приведёт свой или чужой объект в неконсистентное состояние по доброй воле.

Обсуждение: ссылки

- Ссылки тоже сохраняют инварианты.

```
int foo(const int *p) { int t = *p; free(p); return t; }
int bar(const int &p) { return p; }
foo(nullptr); // это невозможно проделать с bar
double d = 1.0;
int *q = *reinterpret_cast<int **>(&d);
foo(q); // это невозможно проделать с bar
```

- Инвариант const int reference: правильное и не вам принадлежащее целое число под ней. Именно поэтому побитовое представление ссылки скрыто.

Важное замечание

- Инкапсуляция это свойство типа, а не его объектов. class list {
  node *top_, *back_; public:
  void concat_with(list other) {
    for (auto cur = other.top_; // всё нормально, мы можем
      cur != other.back_; // работать не только с this
      cur = cur->next_) // а с любым объектом list
    push(cur->data_); } };

Конструкторы и деструкторы

- Инкапсуляция делает критически важными конструкторы.
- Теперь состояние объектов просто нельзя установить извне.

```
class list {
  node *top_ = nullptr, *back_ = nullptr;
public:
   list(size_t initial_len); // ctor
   ~list(); // dtor
```

- Но в случае со списком, его нельзя и очистить извне. Поэтому

важными становятся также деструкторы. Синтаксис показан на слайде.

Обсуждение

- Увы, старые добрые malloc и free ничего не знают о конструкторах и деструкторах.
- Созданный с их помощью в динамической памяти объект не будет корректно инициализирован и будет создан в невалидном состоянии.
- Что делать?

Аллокация динамической памяти

В языке C++ аллокация делается через new и delete. Они вызывают конструкторы и деструкторы создаваемых объектов.

- Важно запомнить парность операторов.

```
int *t = new Widget; // выделяем скалярный объект
delete t; // освобождаем скалярный объект
int *p = new Widget[5]; // выделяем массив
delete [] p; // освобождаем массив
```

- Вы не должны пытаться освободить через delete выделенное через new[] и наоборот.
- Вы не должны смешивать new/delete с механизмом malloc/free.

Семантика new и delete

- Парность вызовов крайне важна.

```
using mvi = MyVector<int>;
mvi *pv = new mvi; //ctor
mvi *pvs = new MyVector<int>[5]; //5 ctors
mvi *vpv = static_cast<mvi *> malloc(sizeof(mvi)); // no ctor
delete pv; // dtor
delete[] pvs; // 5 dtors
free(vpv); // no dtor
```

- По типу pv и pvs очень похожи. Как в точке удаления по pvs понять что нужно пять деструкторов?

Обсуждение

- Что вы думаете о ссылке на выделенную память?

```
int *p = new int[5];
int &x = p[3];
```

Обсуждение

- Что вы думаете о ссылке на выделенную память?

```
int *p = new int[5];
int &x = p[3];
```

- Вроде всё хорошо если бы не червячок сомнения. А что будет после delete?