Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 6
12.10.2018

Операторы new и delete

Зачем они нужны?

```
#include <iostream>
class C {
 public:
  int arr[100];
  C(int a) { /*...*/}
};
int main() {
  C * c = new C(123);
  // ...
  delete c;
};
```

Операторы new и delete

Зачем они нужны?

```
#include <iostream>
class C {
 public:
  int arr[100];
  C(int a) { /*...*/}
};
int main() {
  C * c = new C(123);
  // ...
  delete c;
};
```

- Создание и удаление динамических объектов

Проблема: Временем жизни таких объектов приходится управлять вручную.

Проблемы new и delete

1. Можно забыть написать delete.
2. Можно написать лишний delete.
3. Утечки памяти при исключениях и т.п.
4. delete / delete[].

Проблемы new и delete

1. Можно забыть написать delete.
2. Можно написать лишний delete.
3. Утечки памяти при исключениях и т.п.
4. delete / delete[].

Как решать?

- Оставить delete умным указателям.
- Оставить new make-функциям.

Но всё-таки про new/delete

Оператор **new** состоит из двух частей:

1. Выделение сырой (свободный кусок динамической) памяти. Может возникнуть исключение.
2. Конструирование объекта в сырой памяти.

Оператор **new** гарантирует, что если в конструкторе произошло исключение, то выделенная динамическая память автоматически очистится.

Но всё-таки про new/delete

Оператор **new** состоит из двух частей:

1. Выделение сырой (свободный кусок динамической) памяти. Может возникнуть исключение.
2. Конструирование объекта в сырой памяти.

Оператор **new** гарантирует, что если в конструкторе произошло исключение, то выделенная динамическая память автоматически очистится.

Оператор **delete** делает все наоборот:

1. Вызывается деструктор.
2. Освобождается память.

Размещающий оператор new

Как было раньше:

```
int * p = (int*)(malloc(sizeof(int));
// ...
free(p);
```

Размещающий оператор new

Как было раньше:

```
int * p = (int*)(malloc(sizeof(int));
// ...
free(p);
```

Два способа нельзя смешивать (malloc + delete, new + free)

Размещающий оператор new

Как было раньше:

```
int * p = (int*)(malloc(sizeof(int));
// ...
free(p);
```

Два способа нельзя смешивать (malloc + delete, new + free)

Способ с new предпочтительнее, и его реализацию можно перегружать:

```
void *p = malloc(sizeof(C));
C * c = new (p) C(123); // placement new;
// ...
c -> ~C();
free(p);
```

Размещающий оператор new

Как-то надо бороться с исключениями:

```
void * p = malloc(sizeof(C));
if (!p) return 1;
C * c;
try {
  c = new (p) C(123);
} catch (...) {
  free(p);
  throw;
}
try {
  // ...
} catch (...) {
  c -> ~C();
  free(p);
  throw;
}
c -> ~C();
free(p);
```

operator new, operator delete

```
//new C(x)
  void *p = operator new(sizeof(C));
  C * c;
  try {
    c = new(p) C(x);
  } catch (...) {
    operator delete(p);
    throw;
  }
//delete p
  if (p!=NULL) {
    p->~C();
    operator delete(p);
  }
```

Перегрузка

```
void * operator new (size_t sz) {
  std::cout << "operator new with " << sz << std::endl;
  void * p = malloc(sz);
  if (!p) throw std::bad_alloc();
  return p;
}

void operator delete(void * p) {
  std::cout << "operator delete" << std::endl;
  free(p);
}
```

Перегрузка

Можно перегрузить так:

```
void*operator new (size_t sz, double a, int x) {
    // ...
    return ::operator new(sz);
}
```

Но тогда нужно написать парный ему

```
void operator delete(void * p, double a, int x) {
    // ...
    ::operator delete(p);
}
```

Как тогда их вызвать?

Перегрузка

Можно перегрузить так:

```
void*operator new (size_t sz, double a, int x) {
    // ...
    return ::operator new(sz);
}
```

Но тогда нужно написать парный ему

```
void operator delete(void * p, double a, int x) {
    // ...
    ::operator delete(p);
}
```

Как тогда их вызвать?

```
C* p = new(1.23, 123) C(111);
delete p;
```

Перегрузка

Оператор new/delete внутри класса обязан быть статическим. static можно не писать.

```
class A {
    int param;
  public:
    A(int a): param(a) {
        cout << "A::A(" << a << ")" << std::endl;
    }
    virtual ~A() { cout  << "A::~A()" << std::endl; }
    static void* operator new(size_t sz) {
        cout << "A::operator new" << std::endl;
        return ::operator new(sz);
    }
    static void operator delete(void* ptr) {
        cout << "A::operator delete" << std::endl;
        ::operator delete(ptr);
    }
};
```

Аллокаторы

Класс, который выделяет память каким-то специальным образом.

```
map<int, string> m;
// map<int, string, less<int>, allocator<pair<int, string>> > m;
```

Обязательно определяются:

- `value_type` — тип, для которого работает аллокатор
- `allocate` — выделение памяти под n объектов, но не конструирование
- `deallocate` — особождение памяти

Необязательно:

- `construct` — инициализация заданной памяти заданным значением
- `destroy` — уничтожение памяти без освобождения

Smart pointers

Чем плохи обычные встроенные указатели?

- Указывают на массив или на объект?
- Владеет ли указатель тем, на что указывает?
- Трудно обеспечить уничтожение ровно один раз.
- Обычно сложно определить, является ли указатель висячим.
- Нельзя предоставить информацию компилятору о том, могут ли два указателя указывать на одну область памяти.

«Умный» («интеллектуальный») указатель притворяется обычным указателем с дополнительными функциями.
Обертка над обычными указателями.

Smart pointers

Хочется что-то вроде такого

```
SmartPointer sp(new C);
```

и дальше пользоваться как обычным указателем, не задумываясь об удалении.

Smart pointers

Хочется что-то вроде такого

```
SmartPointer sp(new C);
```

и дальше пользоваться как обычным указателем, не задумываясь об удалении.
А что делать тут?

```
SmartPointer sp2 = sp;
```

Smart pointers

Хочется что-то вроде такого

```
SmartPointer sp(new C);
```

и дальше пользоваться как обычным указателем, не задумываясь об удалении.
А что делать тут?

```
SmartPointer sp2 = sp;
```

Всегда можно обмануть умный указатель:

```
C * ptr = new C;
SmartPointer sp(ptr);
SmartPointer sp2(ptr);
```

Smart pointers: стратегии

- Запрет копирования и присваивания.
- Глубокое копирование.
- Подсчет ссылок в специальных счетчиках.
- Список ссылок.
- Передача владения.

Стратегия передачи владения

Если кто-то пытается скопировать указатель, то ему и передается владение, и исходный умный указатель не указывает больше на объект. Такой указатель не нужно класть в контейнер .

```
std::auto_ptr<int> p (new int(123));
std::vector<std::auto_ptr<int> > v;
v.push_back(p);
```

Умные указатели в C++ 11 / 14

```
std::auto_ptr<>  // deprecated
std::unique_ptr<>
std::shared_ptr<>
std::weak_ptr<>
```

std::unique_ptr

- Реализует семантику исключительного владения
- Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- Копирование не разрешается.
- При деструкции освобождает ресурс, которым владеет.

std::unique_ptr

- Реализует семантику исключительного владения
- Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- Копирование не разрешается.
- При деструкции освобождает ресурс, которым владеет.

Обычное использование – возвращаемый тип фабричных функций для объектов иерархии:

```
template <typename T>
std::unique_ptr<Base> makeObject(T&& params);
```

Некоторые методы std::unique_ptr

- `reset` — заменяет объект;
- `release` — освобождает владение;
- `get` — возвращает указатель на объект, которым владеет;

Некоторые методы std::unique_ptr

- `reset` — заменяет объект;
- `release` — освобождает владение;
- `get` — возвращает указатель на объект, которым владеет;

Запрещены неявные преобразования обычного указателя в умный:

```
std::unique_ptr<Base> p;
p = new A();
```

std::unique_ptr

Две разновидности для индивидуальных объектов и для массивов:

```
std::unique_ptr<T> //  *, ->
std::unique_ptr<T[]>  // []
```

std::unique_ptr

Две разновидности для индивидуальных объектов и для массивов:

```
std::unique_ptr<T> //  *, ->
std::unique_ptr<T[]>  // []
```

`std::unique_ptr` можно присваивать в `std::shared_ptr` (можно не задумываться над тем, как будет использован возвращаемый указатель).

Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...)
    )
}
```

Чего не хватает? Массивов, пользовательских удалителей.

Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...)
    )
}
```

Чего не хватает? Массивов, пользовательских удалителей.

Функция уже есть — `std::make_unique` в C++14.

Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...)
    )
}
```

Чего не хватает? Массивов, пользовательских удалителей.

Функция уже есть — `std::make_unique` в C++14.

```
std::unique_ptr<Base> p(new Base); // дважды пишем Base
auto p1(std::make_unique<Base>()); // make
```

std::shared_ptr

- Реализует семантику совместного владения.
- Использует метод подсчета ссылок. Счетчик ссылок хранится в динамически выделяемой памяти. Объект про счетчик ссылок ничего не знает.
- Тип удалителя не является частью типа указателя.
- Может работать только для указателей на объекты.

std::shared_ptr

- Реализует семантику совместного владения.
- Использует метод подсчета ссылок. Счетчик ссылок хранится в динамически выделяемой памяти. Объект про счетчик ссылок ничего не знает.
- Тип удалителя не является частью типа указателя.
- Может работать только для указателей на объекты.

Что происходит здесь?

```
sp1 = sp2;
```

Некоторые методы std::shared_ptr

- `use_count` — количество shared ptr'ов, ссылающихся на этот же объект;
- `reset` — заменяет объект;
- `unique` — проверяет, что объект контролируется единственным указателем shared ptr;
- `get` — возвращает указатель на объект, которым владеет;

std::shared_ptr

- Перемещение быстрее копирования.
- Счетчик ссылок хранится в динамически выделяемой памяти.
- Пользовательский удалитель не является частью типа указателя.

Управляющий блок

- Функция `std::make_shared` всегда создает управляющий блок.
- Управляющий блок создается, когда указатель `std::shared_ptr` создается из указателя с исключительным владением
- Когда конструктор. `std::shared_ptr` вызывается с обычным указателем — он создает управляющий блок.

Типичная ошибка

```
A* p = new A;
std::shared_ptr<A> sptr1(p);
std::shared_ptr<A> sptr2(p);
```