Информатика. Семинар №9

19.04.2016

https://dl.dropboxusercontent.com/u/96739039/sem4/infa_s09.pdf

Операторы new, delete

- Отличие от malloc, calloc и т.п. в том, что new = malloc + n вызовов конструктора string::string()
- Массивы простых типов (int, float) не инициализированы, т.к. у них нет конструкторов
- delete [] s;

```
int n = 42;
std::string* s = new std::string[n];
```

Зачем нужно выделять память на в куче?

- Размер кучи >> размера стека
- Время жизни данных в куче – вплоть до delete
- Время жизни данных на стеке – до соответствующей }

Как и кто должен выделять память?

- Правило: тот, кто выделил память должен её удалить.

```
void f(float x, float y, float a[2])
{
  a[0] = x + y;
  a[1] = x - y;
}

int main() {

  // float b[2];
  float* b = new float[2];
  f(1.0f, 2.0f, b);

  std::cout << b[0] << " " << b[1];
  delete[] b;
```

Откуда такое правило?

- Если вы пользуетесь «чужой» ф-ей, которая выделяет некоторый объём памяти, вам необходимо помнить/знать о необходимости подчищать за другими.
- Если используете библиотеки, собранные другими версиями компиляторов, то при попытке почистить память, выделенную в ней, программа упадёт.

«Умные» указатели

```
#include·<memory>

struct·A·{
··A()··{·std::cout·<<·"Hi,·I'm·A"·<<·std::endl;·}
··~A()·{·std::cout·<<·"Bye.·A."···<<·std::endl;·}
};

void·f(A*·a)·{
··a·=·new·A();
}

int·main()·{
··{
····A*·a·=·nullptr;·/*·в·C++11·для·нулевого·указателя·
·····сделали·специальную·константу·вместо·целочисленного·0·или·NULL·*/
····f(a);
··}
```

«Умные» указатели

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

shared_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0

http://archive.kalnitsky.org/2011/11/02/smart-pointers-in-cpp11/

«Умные» указатели

```
#include <memory>

struct A {
  A(int x, int y) { std::cout << "Hi, I'm A" << std::endl; }
  ~A() { std::cout << "Bye. A." << std::endl; }
  void Do() { std::cout << "Do work" << std::endl; }
};

void f(A* a) {
  a->Do();
}

int main() {
  std::shared_ptr<A> a0;
  {
    std::shared_ptr<A> a = std::make_shared<A>(1, 2);
    f(a.get());
    // a0 = a;
  }
```

Как быть с массивами?

- http://stackoverflow.com/questions/13061979/shared-ptr-to-an-array-should-it-be-used

```
std::shared_ptr<int> a0(new int[10], std::default_delete<int[]>());
std::shared_ptr<int> a1(new int[10], [](int *p) { delete[] p; });
```

P.S. В этом случае лучше использовать std::vector. Если необходим массив из малого числа элементов и область видимости позволяет, то лучше выделить массив на стеке.

Детали реализации std::vector

- Амортизированная стоимость (среднее число операций в худшем случае – T(n) / n) операции push_back: O(1).

Детали реализации std::vector

- Знаем, что у std::vector есть **capacity** (выделено) и **size** (использовано).
- Реализовать на основе T* data = new T[n] не хочется, т.к. лишний раз будет вызван конструктор «по умолчанию»
- Т.е. процесс выделения памяти от процесса вызова конструктора надо отделить: выделить память можно с помощью malloc

Детали реализации std::vector

- Placement new - вызов конструктора на уже выделенной памяти:

```
void SimplePushBack(const T& t) {
        new (data + size) T(t);
        size++;
}
```

- Как удалить объект, который так создали?

T* p = …

Явно вызываем деструктор:

p->~T();

Детали реализации std::vector

- Если не получается выделить память нужного размера, то желательно как-то оповещать пользователя об этом -> бросаем исключение.

```
void PushBack(const value_type& x)
{
  // without reserving
  try
  {
    new (data_ + size_) value_type(x);
    ++size_;
  }
  catch (...)
  {
    throw;
  }
}
```

Детали реализации std::vector

- Зачем нужны списки инициализации у конструкторов?

```
char* data_;

explicit vector(size_type n):
  capacity_(n),
  size_(0),
  data_(new char[capacity_ * sizeof(value_type)])
{
}
```

Детали реализации std::vector

- Как реализовать оператор присваивания?
- Правило **commit or rollback**.
- Идиома **copy and swap** – простой способ избежать утечек памяти, если неожиданно память кончилась и оператор new не отработал до конца (конструктор копирования должен быть реализован).

P.S. https://ru.wikipedia.org/wiki/Copy-and-swap

Упражнение 1

- Реализовать свой класс String фиксированного размера, реализующий конструктор по умолчанию, конструктор копирования, String(const char* s), String(size_t length, char value = 0), деструктор, оператор [] константный и неконстантный, Length, Swap, оператор = и ==, оператор !=
- Продвинутый вариант: реализовать подобие шаблонного класса Vector (размер динамически увеличивается и уменьшается)