OTUS

Онлайн образование

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в учебной группе в
Telegram

Задаем
вопрос в чат

Вопросы вижу в чате,
могу ответить не сразу

Тема вебинара

Разработчик C++ - базовый курс

Работа с динамической памятью

Пальчуковский Евгений

Разработчик ПО
Развиваю технологии финансовых услуг с помощью C++

Email: eugene@palchukovsky.com
Telegram: @palchukovsky

Цели вебинара

После занятия вы сможете

1. Выделять и освобождать памяти
2. Находить и избегать ошибки при работе с памятью
3. Понимать идиому RAII
   Resource **A**cquisition **I**s **I**nitialization

Какую проблему решаем?

Пример. Игра типа стратегии

- на карте есть юниты
- юниты строятся и уничтожаются
- игрок может указателем выделить юнит
- игрок может отдать команду выделенному юниту

Дизайн системы

- класс **Unit** с
  - текущими координатами и единицами здоровья
  - методами **move** и **attack**
- класс **Selection** с
  - текущим выбранным **Unit** (может быть пустым)
  - методом **set** для выделения активного юнита
  - метод **unset** для снятия выделения
  - методом **click**, который вызывает move у выделенного юнита
- функции
  - make_unit_and_select - для создания нового юнита
  - destroy - для уничтожения существующего юнита

Дизайн системы

- класс **Unit** с
  - текущими координатами и единицами здоровья
  - методами **move** и **attack**
- класс **Selection** с
  - текущим выбранным **Unit** (может быть пустым)
  - методом **set** для выделения активного юнита
  - метод **unset** для снятия выделения
  - методом **click**, который вызывает move у выделенного юнита
- функции
  - make_unit_and_select - для создания нового юнита
  - destroy - для уничтожения существующего юнита

Как в Selection хранить Unit?

Указатели

- *указатели* - **переменные**, которые хранят **адрес памяти**, по которому храниться **значение** нужного **типа**

```
int ans = 42;    // переменная типа int
int *ptr = &ans; // указатель на переменную типа int
int **pptr = &ptr; // указатель на указатель на переменную типа int
```

Пример: strategy.hpp

Операции с указателями

```
int a = 1, b = 2;
int *ptr = &a;
*ptr = 3;
b = *ptr;
ptr[0] = 3;
ptr[1] = 4;
0[ptr] = 22;
```

```
int c[] = {1, 2, 3};
ptr = c;
b = *c;
b = ptr[2];
ptr += 2;
b = ptr - c;
ptr = nullptr;
```

Создаем юнит и выделяем его

```
void make_unit_and_select(Selection &selection) {
    Soldier soldier{0, 0};
    selection.set(&soldier); // void Selection::set(Unit *unit);
}
```

Что в этом коде не так?

Создаем юнит и выделяем его

```
void make_unit_and_select(Selection &selection) {
    Soldier soldier{0, 0};
    selection.set(&soldier); // void Selection::set(Unit *unit);
}
```

Что в этом коде не так?

- переменная soldier существует только пока не завершится функция
- компилятор не увидит в этом коде какой-либо проблемы
- при вызове selection.click() программа будет обращаться к указателю на удаленный объект
- такое обращение может привести к порче значений других переменных
- в отладочной копии простой программы ошибку можно не заметить runtime

Висячий указатель

- ситуация, когда хранится указатель на объект, который был уничтожен
- для поиска можно использовать AddressSanitizer
  - добавить флаг компиляции -fsanitize=address (/fsanitize=address для MSVC)
  - добавить флаг компоновки -fsanitize=address (/fsanitize=address для MSVC)

Висячий указатель

- ситуация, когда хранится указатель на объект, который был уничтожен

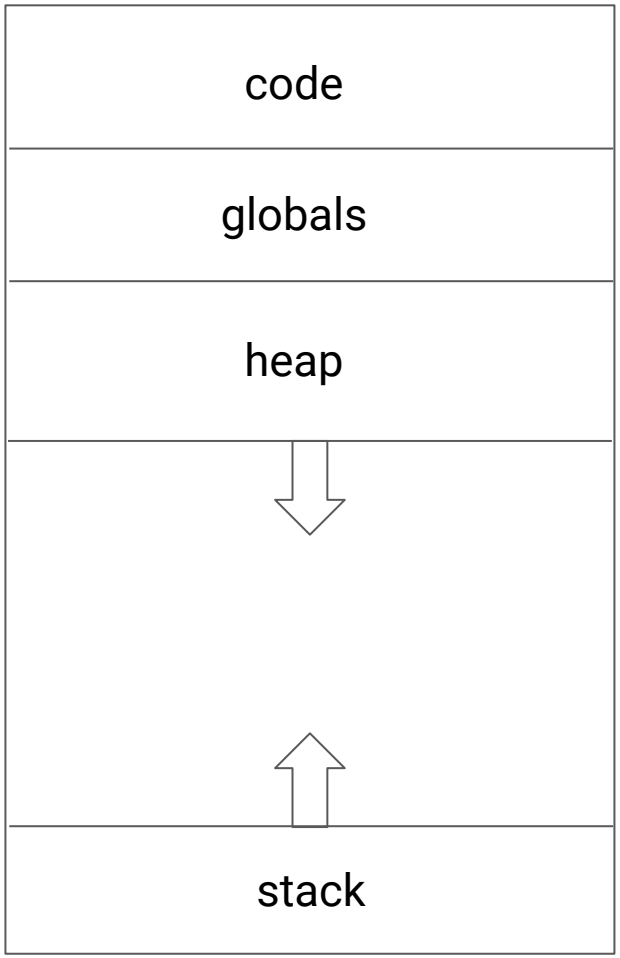
- для поиска можно использовать AddressSanitizer
  - добавить флаг компиляции -fsanitize=address (/fsanitize=address для MSVC)
  - добавить флаг компоновки -fsanitize=address (/fsanitize=address для MSVC)

Пример: dangling_ref.cpp

Память приложения

Сегменты памяти*

- **code** - машинный код
  (может быть общим)
- **globals** - глобальные константы
  (могут быть общим)
  и глобальные переменные
- **stack** - локальные переменные
  (свой на каждый поток)
- **heap** - динамическая память
  (условно - бесконечная, **условно!**)



* упрощенная модель

Куча

- управляется аллокатором памяти из стандартной библиотеки
- аллокатор можно заменить
- аллокатор может увеличивать размер кучи
- память выделяется и освобождается программистом вручную

Выделение и освобождение памяти

Оператор new

- для выделения памяти используется оператор **new**

- он выполняет 2 действия
  - запрашивает необходимый объем памяти у аллокатора
  - вызывает конструктор объекта

- в качестве результата возвращается указатель на объект

Оператор new

- для выделения памяти используется оператор **new**
- он выполняет 2 действия
  - запрашивает необходимый объем памяти у аллокатора
  - вызывает конструктор объекта
- в качестве результата возвращается указатель на объект

```
void make_unit_and_select(Selection &selection) {
   Unit *soldier = new Soldier{0, 0};
   selection.set(soldier);
}
```

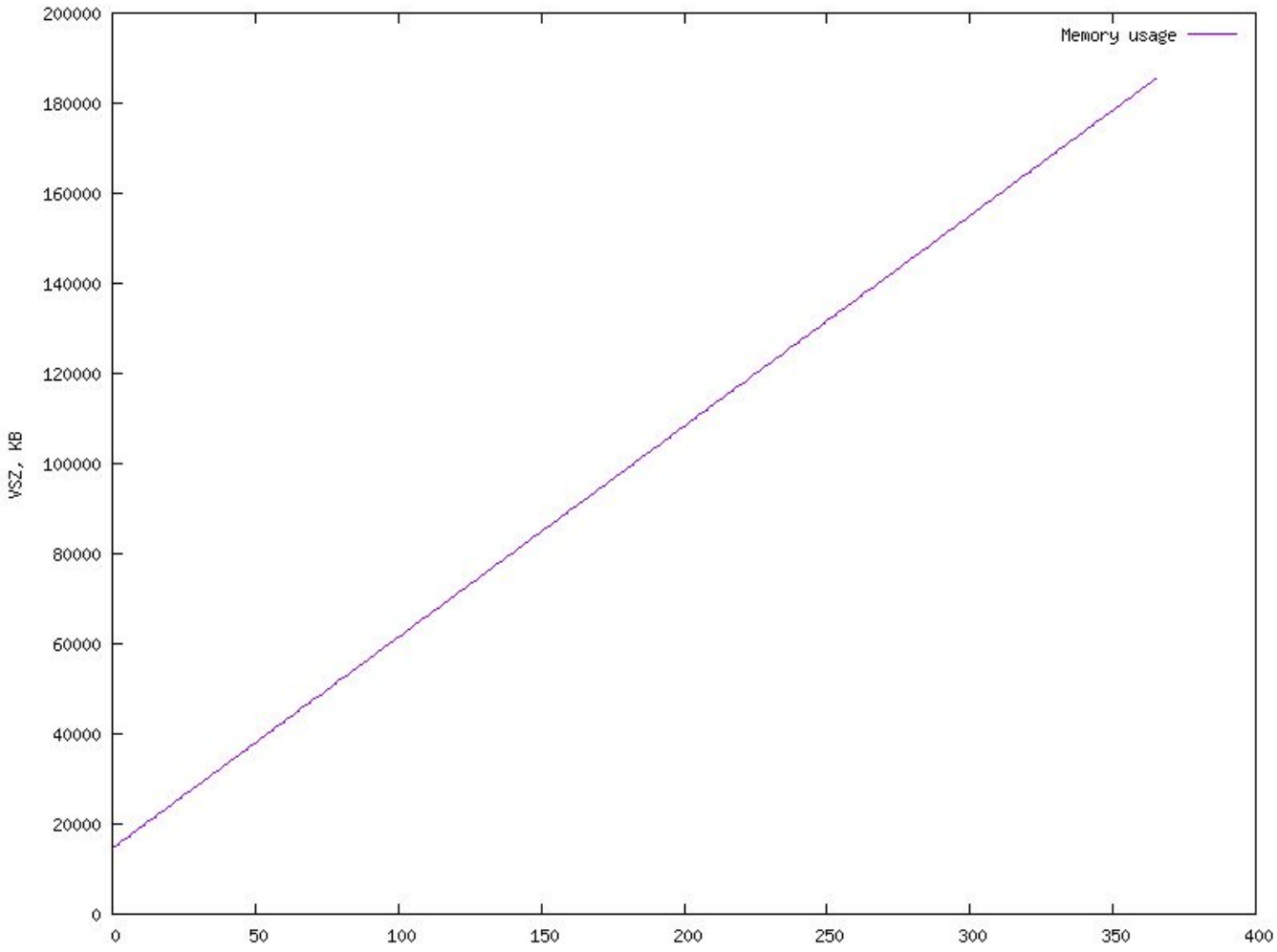
Оператор new

- для выделения памяти используется оператор **new**
- он выполняет 2 действия
  - запрашивает необходимый объем памяти у аллокатора
  - вызывает конструктор объекта
- в качестве результата возвращается указатель на объект

```
void make_unit_and_select(Selection &selection) {
   Unit *soldier = new Soldier{0, 0};
   selection.set(soldier);
}
```

Что не так с этим кодом?

Потребление памяти при утечках



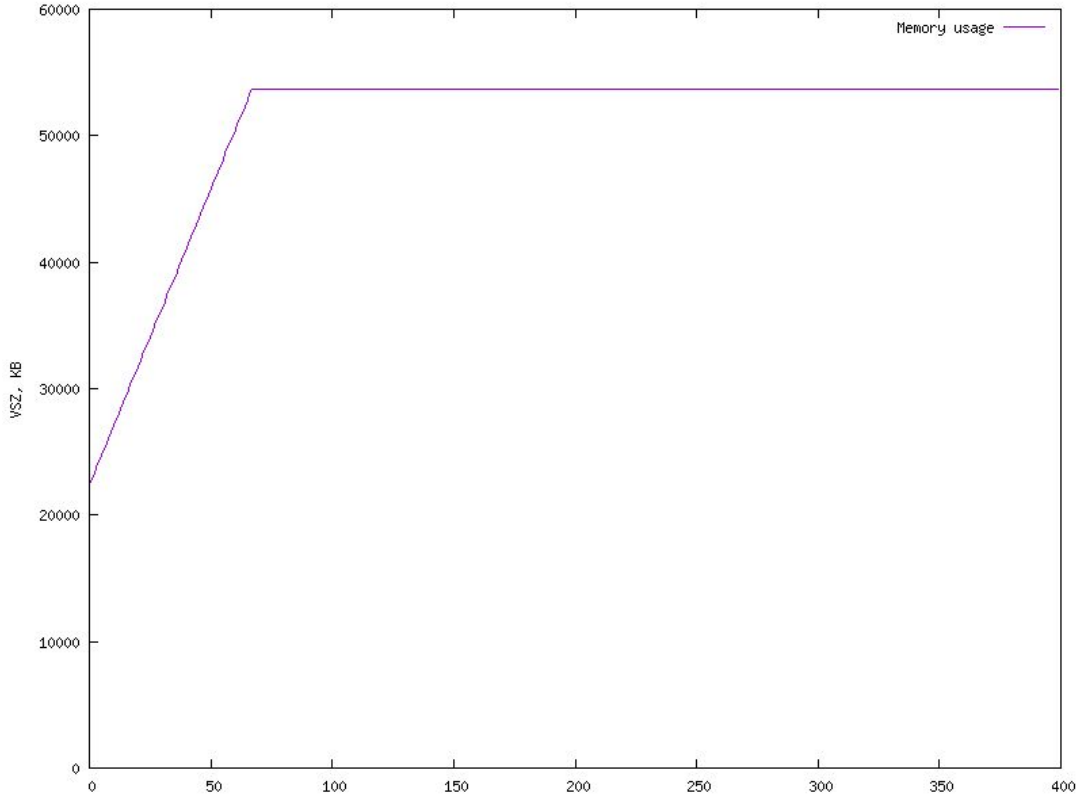
Оператор delete

- для освобождения памяти выделенной с помощью **new**, и **только new**
- выполняет 2 действия (**обратный** порядок)
  - вызывает деструктор объекта
  - возвращает память аллокатору
- принимает указатель в качестве аргумента
- если не вызвать delete, то память не будет возвращена аллокатору - **утечка памяти** и, возможно, **потеря данных**
- если аллокатор постоянно будет запрашивать новую память, то это приведет к ее исчерпанию и **невозможности выделить еще**

Ручное освобождение памяти

- будем сохранять все указатели в массив
- при достижении максимального числа объектов вызовем для них delete

Пример: memory_leak_fixed.cpp



Как искать утечки памяти

- использовать специальные инструменты
  - valgrind - Linux only
  - LeakSanitizer - Linux only
  - _CRTDBG_MAP_ALLOC - Visual Studio
  - Visual Leak Detector for Visual Studio

- проводить нагрузочное тестирование и анализировать потребление памяти

Пример: memory_leak.cpp

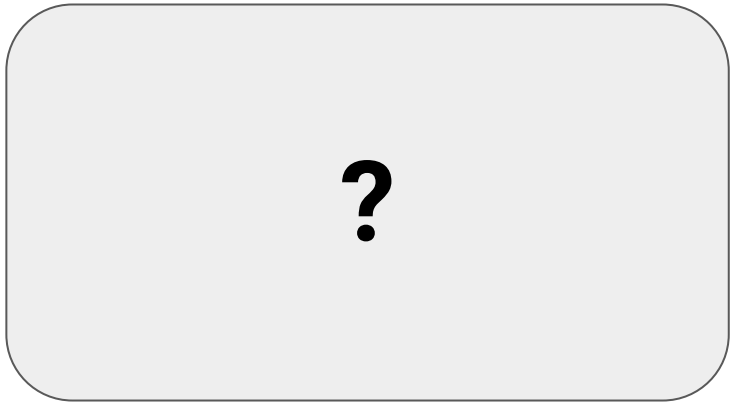
```
=================================================================
==8176==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 12 byte(s) in 1 object(s) allocated from:
    #0 0x7f029a9b5afd in operator new(unsigned long) (/usr/lib/x86_64-linux-gnu/liblsan.so.0+0xfafd)
    #1 0x55f5ff6e099b in make_unit_and_select(Selection&) /home/yc-user/otus/cpp-basic/23_dynamic_memory/memory_leak.cpp:10
    #2 0x55f5ff6e09f9 in main /home/yc-user/otus/cpp-basic/23_dynamic_memory/memory_leak.cpp:17
    #3 0x7f0299c13bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

SUMMARY: LeakSanitizer: 12 byte(s) leaked in 1 allocation(s).
```

Оператор new - ручной контроль

```
Unit *soldier = new(std::nothrow) Soldier{0, 0};
if (soldier == nullptr) {
    throw std::runtime_error{"No out of memory"};
}
selection.set(soldier);
...
delete soldier;
```

Оператор new - совсем ручной контроль

```
Unit *soldier = static_cast<Soldier *>(malloc(sizeof(Soldier)));
if (soldier == nullptr) {
    throw std::runtime_error{"Out of memory"};
}
```




```
free(soldier);
```

Оператор new - совсем ручной контроль

```
Unit *soldier = static_cast<Soldier *>(malloc(sizeof(Soldier)));
if (soldier == nullptr) {
    throw std::runtime_error{"Out of memory"};
}
new(soldier) Unit{0, 0};
selection.set(soldier);

?

free(soldier);
```

Оператор new - совсем ручной контроль

```
Unit *soldier = static_cast<Soldier *>(malloc(sizeof(Soldier)));
if (soldier == nullptr) {
    throw std::runtime_error{"Out of memory"};
}
new(soldier) Unit{0, 0};
selection.set(soldier);
...
soldier->~Unit();
free(soldier);
```

new и delete для массивов

- для выделения памяти под массив используется `new[]`
  - `int *arr = new int[10];`

- new для массивов выполняет следующие действия
  - выделяет память под хранение всех элементов в непрерывном блоке
  - для каждого элемента вызывает конструктор

- для освобождения памяти в этом случае используется `delete[]`
  - `delete[] arr;`

- delete для массивов выполняет следующие действия
  - вызывает деструктор для каждого элемента в массиве
  - освобождает весь блок памяти

Пример: buffer.hpp

Важно! Если перепутать применение `new[]/delete` или `new/delete[]`, то это создаст ошибку в программе.

Проблемы?

Какие проблемы очевидны при работе с динамической памятью?

Проблемы?

Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти

Проблемы?

Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти
- ручной контроль
  можно забыть вызвать delete

Проблемы?

Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти
- ручной контроль
  можно забыть вызвать delete
- можно вызвать,
  но не в каждой ветви кода (retun, break, …)

Проблемы?

Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти
- ручной контроль
  можно забыть вызвать delete
- можно вызвать,
  но не в каждой ветви кода (retun, break, …)
- исключение может привести к утечке
  … может выбросить каждая строка!

Проблемы?

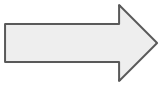
Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти
- ручной контроль
  можно забыть вызвать delete
- можно вызвать,
  но не в каждой ветви кода (retun, break, …)
- исключение может привести к утечке
  … может выбросить каждая строка!
- “голый” указатель не даёт понимания
  - кто владеет объектом
  - и вообще - владеет ли?

Проблемы? Решение есть!

Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти



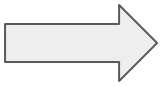
не использовать динамическую память без причины

- ручной контроль
  можно забыть вызвать delete
- можно вызвать,
  но не в каждой ветви кода (retun, break, …)
- исключение может привести к утечке
  … может выбросить каждая строка!
- “голый” указатель не даёт понимания
  - кто владеет объектом
  - и вообще - владеет ли?

Проблемы? Решение есть!

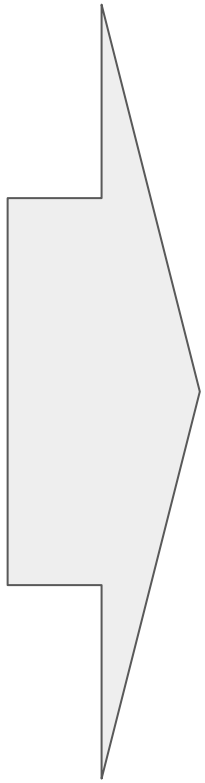
Какие проблемы очевидны при работе с динамической памятью?

- фрагментация памяти



не использовать динамическую память без причины

- ручной контроль
  можно забыть вызвать delete
- можно вызвать,
  но не в каждой ветви кода (retun, break, …)
- исключение может привести к утечке
  … может выбросить каждая строка!
- “голый” указатель не даёт понимания
  - кто владеет объектом
  - и вообще - владеет ли?



писать код на основе идиом

RAII
Resource Acquisition Is Initialization

RAII

Resource **A**cquisition **I**s **I**nitialization

- “получение ресурса есть инициализация”
- идиома языка C++ для автоматического управления ресурсами
- для ресурсов создается класс-обертка
- в конструкторе класса захватывается или передается выделенный ресурс
- в деструкторе класса ресурс освобождается

Пример: smart_ptr.cpp

std::unique_ptr

- умный указатель из стандартной библиотеки
- обеспечивает уникальное владение ресурсом
- может “подарить” контролируемы ресурс
- позволяет указывать произвольные функции для освобождения ресурса
- умеет работать с массивами

Следующий вебинар

7 июля 2023

Умные указатели

Ссылка на вебинар будет в ЛК за 15 минут

Материалы к занятию в ЛК — можно изучать

Обязательный материал обозначен красной лентой

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Заполните, пожалуйста, опрос о занятии по ссылке в чате