

Лекция

Allocator

Подходы управления памятью

- ручной (*manually*)
 - `new/delete`, `malloc/free`(C)
- полуавтоматический (*semi-manually*)
 - с использованием подсчёта ссылок (*referencing count*)
`std::shared_ptr`/`std::unique_ptr`(C++11)
 - с использованием фиксированного объёма (*memory pool*)
`boost::pool`, `std::pmr::monotonic_buffer_resource`(C++17)
- автоматический при помощи сборщика мусора (*garbage collection*)
 - Boehm-Demers-Weiser Garbage Collector
(<https://github.com/ivmai/bdwgc/>)

Bjarne Stroustrup:

«Actually, what I said was something like "When (not if) automatic garbage collection becomes part of C++, it will be optional"».

Почему возникает необходимость в управлении памятью

возможности и тонкостях его работы
менеджера памяти (*memory manager*)

- Скорость выделения областей памяти
- Скорость освобождения областей памяти
- Фрагментация памяти
- Накладные расходы по памяти
- Поведение в многопоточной среде
- Поведение при кончающейся памяти
- Размещение кэша
- Стабильная работа в режиме реального времени

Ручное управление памятью

```
gcc -shared -fPIC malloc.c -o malloc.so
```

```
LD_PRELOAD=/path/to/malloc.so  
export LD_PRELOAD
```

- environment variable `LD_PRELOAD` используется динамическим линковщиком, который выбирает использующиеся библиотеки при сборке программы

Аналоги `libc malloc`:

- **DLMalloc** – подмножество аллокаторов: оригинальные *Doug Lea*, *GNU libc* и *ptmalloc*.
- **BSDmalloc** – включён во FreeBSD, размещает в памяти объекты из заранее созданного пула. Поддерживает секторы размера степени двойки (минус данные для очистки). Простая реализация, издержки по памяти.
- **Hoard** – цель оптимальная работа в многопоточной среде, блокировки и синхронизации доступа.

Ручное управление памятью

```
1. int main() // Как-то надо бороться с исключениями?
2. {
3.     void *p = malloc(sizeof(C));
4.     if (!p) return 1;
5.
6.     C *c;
7.     try {
8.         c = new(p) C{ 123 }; // размещающий оператор new
9.     } catch (...) {
10.         free(p);
11.         throw;
12.     }
13.
14.     try {
15.         // ...
16.     } catch (...) {
17.         c->~C();
18.         free(p);
19.         throw;
20.     }
21. }
```

Ручное управление памятью

```
1. int main() // Как-то надо бороться с исключениями?
2. {
3.     void *p = operator new(sizeof(C));
4.
5.     C *c;
6.     try {
7.         c = new(p) C{ 123 }; // размещающий оператор new
8.     } catch (...) {
9.         operator delete(p);
10.        throw;
11.
12.        if (p != NULL) {
13.            p->~C();
14.            operator delete(p);
15.        }
16.    }
17. }
```

Ручное управление памятью

```
1. class C {  
2.     // Можно перегрузить так:  
3.     static void* operator new (size_t sz, bool) {  
4.         // ...  
5.         return ::operator new(sz);  
6.     }  
7.     // Но тогда нужно написать парный delete  
8.     static void operator delete(void *ptr) {  
9.         // ...  
10.        ::operator delete(ptr);  
11.    }  
12.};  
13.// placement new:  
14.void* operator new (size_t sz, void *ptr) noexcept  
15. { return ::operator new(sz, ptr); }  
  
16.void* operator new(size_t sz, std::string &str, int a)  
17. { return ::operator new(sz); }  
  
18.// Как тогда их вызвать?  
19.C *p = C::operator new(sizeof(C), true);  
20.operator delete(p);
```

Memory pool

Преимущества:

- Простое понятная структура памяти – непрерывный блок
- Выделение и освобождение памяти происходит **максимально быстро!**
- Существуют стандартные реализации (*obstack* из GNUlibc)
- Эффективно для программы, которые можно явно разделить на стадии

Недостатки:

- Часто невозможно подключить к сторонней библиотеке
- Если изменится структура программы, то пулы памяти также потребуют реконструкции
- Нужно помнить, из какого пула следует выделять память, если ошибётесь, это будет трудно обнаружить.

Arena allocator

помещает все мелкие объекты в заранее выделенную память

```
1.  class Arena {
2.      char* mem_;
3.      size_t size_;
4. public:
5.     Arena(size_t size) : size_(size) { mem_ = new char[size_]; }
6.     ~Arena() { delete[] mem_; }
7.     void* allocate(size_t size);

8.     template <typename T, typename ...Args>
9.     T* New(Args&&... params) {
10.         auto mem = allocate(sizeof(T));
11.         return new(mem) T(std::forward<Args>(params)...);
12.     }
13. };

14. void main() {
15.     Arena arena;
16.     auto ptr = arena.New<Obj>(param1, param2);
17.     // ...
18. }
```

вся память освобождается одновременно. ВОПРОС: Что здесь может пойти не так?

Garbage collection

- Полностью автоматическое определение и удаление *неиспользуемых объектов* из памяти
- Запускается, когда *объём свободной памяти* опускается *ниже определённого порога*
- Первым делом проходит по базовым данным
 - стэк (*stack*)
 - глобальные переменные (*global variables*)
 - куча (*heap*)
- Следит за использованием всех данных программы
- Если ссылки на данные найдены, то эти данные не очищает, в противном случае память освобождается, чтобы быть использованой повторно.
- Имеет таблицу всех указателей программы, чтобы корректно работать должны быть *средством языка*

Garbage collection in C and C++ are both difficult topics

1. С указателями можно обращаться как с целыми числами. **GC** не может знать о блоке, который недоступен не по одному имеющемуся указателю.
2. Указатели прозрачны программисту. Многие **GC** могут перемещать объекты в памяти и упаковывать их для экономии места. Нужно быть уверенным, что никто не обратится по данному адресу (даже изменив тип целого числа), когда перемещаете блок в памяти.
3. Механизм управления памятью доступен для явных операций. **GC** должен учитывать, что пользователь может в любой момент явно освободить блоки памяти.
4. В **C++** существует разделение между выделением/освобождением памяти и созданием/уничтожением объекта. **GC** должен знать, освобождая память, вызывать ли деструктор какого-либо объекта.
5. Память может быть выделена из разных областей. В **C++** можно использовать память из встроенного freestore (**malloc/free**), либо от ОС через **mmap** или другие системные вызовы. Хороший **GC** должен иметь возможность отслеживать ссылки в этих и других источниках и (возможно) должен нести ответственность за их очистку.
6. Указатели могут указывать на середину массивов или объектов. Во многих языках со сборкой мусора ссылки на объекты всегда указывают на начало объекта.

Garbage collection: типы

- **Копирующий (*Copring*)**: делит память на 2 части, позволяет размещать объекты только в 1 из частей. Периодически копирует объекты из одной части в другую. Часть памяти, в которую перемещены данные становится *активной*, всё остальное считается *мусором*. После перемещения данных обновляются все ссылки на объекты.
- **Маркирующий (*Mark and sweep*)**: объекты помечаются специальным тэгом, изначально тэг равен нулю. С каким-то промежутком проходит по всем ссылкам и прибавляет 1 к тэгу. Память объектов с меньшим тэгом может быть использована для выделения в дальнейшем.
- **Инкрементальный (*Incremental*)**: не требует обходов всех данных, осуществляет короткие проходы по части неактуальных объектов при необходимости освобождения памяти, избавляет от долгих задержек в работе программы.
- **Консервативный (*Conservative*)**: ничего не знает о структуре данных, знает только значения всех ссылок и указателей.

Garbage collection: недостатки

- **Всегда работают медленнее**, чем другие техники управления памятью; порождает неочевидные задержки в неизвестных частях программы;
- Неизвестно в какой момент память реально будет очищена; То есть вызов **деструкторов становится небезопасным** и **управление всеми иными ресурсами** (кроме памяти) ложится на плечи программистов;
- **Не избавляет от утечек памяти**, если внимательно не следить за уничтожением ссылок на объекты.
- К коду, использующему сборщик мусора, предъявляются достаточно жёсткие требования.
- Проблема взаимодействия неуправляемых объектов с управляемыми.
- Особенности применения сборки мусора в многопоточных приложениях.

Garbage collection: Boehm-Demers-Weiser

- Консервативный (инкрементальный) сборщик мусора
 - <https://www.hboehm.info/gc>
- Является свободным ПО
- Работает под управлением Linux, Windows и пр.
- Можно использовать подменой системного аллокатора – интерфейсы совпадают.
- Если производительность некритична, то можно не задумываться о времени жизни объекта.
- Performance is competitive with malloc/free.
 - Usually wins for threads + small objects.
- Tracing performance is very close to best commercial JVMs.

Example: Boehm-Demers-Weiser

Lisp S-expressions

```
1. #include "gc.h"  
  
2. typedef union se { struct cons * cp; int i; } sexpr;  
3. struct cons { union se head; union se tail; };  
  
4. #define car(s) (s).cp->head  
5. #define cdr(s) (s).cp->tail  
6. #define from_i(z) ({sexpr tmp; tmp.i=z; tmp;})  
7. #define to_i(s) (s).i  
  
8. sexpr cons(sexpr a, sexpr b) {  
9.     sexpr tmp = { GC_MALLOC(sizeof(struct cons)) };  
10.    car(tmp) = a; cdr(tmp) = b;  
11.    return (tmp);  
12. };  
  
13. int main() {  
14.     return to_i(car(cons(from_i(0), from_i(1))));  
15. }
```

Garbage collection: Microsoft Managed C++

- На сегодняшний момент MC++ (managed C++) от Microsoft поддерживает атрибут `__gc`, который служит признаком того, что объекты управляются GC.
- `__gc` требует поддержки CLR, т.е. двухэтапной компиляции, трансляции в байт-код, который интерпретируется при выполнении.

Упражнение

1. Разобраться с поддержкой GC в стандартной библиотеке C++ **<memory>**

C++ bugs

- **Ошибки доступа в память**
- **Ошибки синхронизации** (многопоточность)
- **Другие ошибки**
(UB) undefined behavior != undefined result
- **Bugs – это дорого:**
 - больше времени,
 - больше ресурсов,
 - меньше безопасности

buffer overflow

```
1. % cat bug.cc
2. #include <iostream>
3. using namespace std;
4. void bad() {
5.     cout << "I am BAD\n";
6. }
7. int main() {
8.     long a, b, ar[10];
9.     cin >> a >> b;
10.    ar[a] = b;
11. }

12. % g++ bug.cc
13. % ./a.out
14. 1 2
15. % ./a.out
16. 13 4196196
17. I am BAD
18. Segmentation fault (core dumped)
19. % nm ./a.out | grep bad
20. 0000000004007f5 t _GLOBAL__sub_I__Z3badv
21. 000000000400764 t _Z3badv
22. % i 0x000000000400764
23. 4196196 0x400764 020003544
```

use-after-free

privilege excalation

```
1. struct Thing { bool has_access; };

2. int main() {
3.     Thing *security_check = new Thing;
4.     security_check->has_access = false;
5.     delete security_check;
6.     int *x = new int(42);
7.     if (security_check->has_access)
8.         cout << "access granted\n";
9. }
```

Integer overflow

иногда цикл выполняется бесконечно (gcc)

```
1. void f(int *arr) {
2.     int val = 0x03020100;
3.     for (int i = 0; i < 64; i++) {
4.         arr[i] = val;
5.         val += 0x04040404;
6.     }
7. }
```

Порядок выполнения

lack of sequence point

```
1. int i = 0;  
2. i = ++i + i++;  
  
3. std::map<int, int> m;  
4. m[10] = m.size();
```

Uninitialized memory

плавающая ошибка

```
1. int Sum(int n) {  
2.     int sum, i;  
3.     for (i = 0; i < n; ++i) sum = sum + 1;  
4.     return sum;  
5. }  
  
6. Гейзен-баг - программная ошибка, которая исчезает или  
7. меняет свои свойства при попытке обнаружения.  
  
8. struct Foo {  
9.     int bar;  
10. };  
11. struct Foo2 {  
12.     int bar;  
13.     Foo2() {}  
14. };  
15. struct Foo3 {  
16.     int bar;  
17.     Foo3() : bar(0) {}  
18. };  
  
19. Foo *foo = new Foo(); // will initialize bar to 0  
20. Foo *foox = new Foo; // will NOT initialize bar to 0  
  
21. Foo2 *fooc = new Foo2(); // will NOT initialize bar to 0  
22. Foo2 *fooxc = new Foo2; // will NOT initialize bar to 0
```

Memory leaks (утечки памяти)

всё правильно?

```
1. #include <iostream>

2. struct A
3.     A() { std::cout << "A()" << std::endl; }
4.     ~A() { std::cout << "~A()" << std::endl; }
5. };

6. class B : public A {
7. public:
8.     B() { std::cout << "B()" << std::endl; }
9.     ~B() { std::cout << "~B()" << std::endl; }
10.};

11. int main() {
12.     A *a = new B;
13.     delete a;
14.     return 0;
15. }
```

Memory leaks (утечки памяти)

всё правильно?

```
1. #include <iostream>

2. struct A
3.     A() { std::cout << "A()" << std::endl; }
4.     ~A() { std::cout << "~A()" << std::endl; }
5. };

6. class B : public A {
7. public:
8.     B() { std::cout << "B()" << std::endl; }
9.     ~B() { std::cout << "~B()" << std::endl; }
10.};

11. int main() {
12.     A *a = new B;
13.     delete a;
14.     return 0;
15. }
```

output:

A()

B()

~A()

Memory leaks (утечки памяти)

RAII

```
1. #include <iostream>

2. struct A
3.     A() { std::cout << "A()" << std::endl; }
4.     virtual ~A() { std::cout << "~A()" << std::endl; }
5. };

6. class B : public A {
7. public:
8.     B() { std::cout << "B()" << std::endl; }
9.     ~B() { std::cout << "~B()" << std::endl; }
10.};

11. int main() {
12.     A *a = new B;
13.     delete a;
14.     return 0;
15. }
```

Три вида утечек памяти

1. выделен фрагмент памяти, стал недостижим
2. незапланированно долгоживущие выделения памяти
3. свободная, но неиспользуемая или непригодная для использования память
4. **tcmalloc** (visual leak detector)
5. **AddressSanitizer**
6. **Valgrind**

Самая опасная функция в C/C++

- **memset()** по версии создателей статического анализатора кода PVS-studio
- 329 ошибок в исходниках open-source проектов, около 3,6% от всех ошибок
- **printf()** и её разновидности – почётное 2 место

```
void* memset(void* ptr, int value, size_t num);
```

- **ptr** - Pointer to the block of memory to fill.
- **value** - Value to be set, it is passed as an int, but the function fills the block of memory using the unsigned char conversion of this value.
- **num** - number of bytes to be set to the value of unsigned integral type.

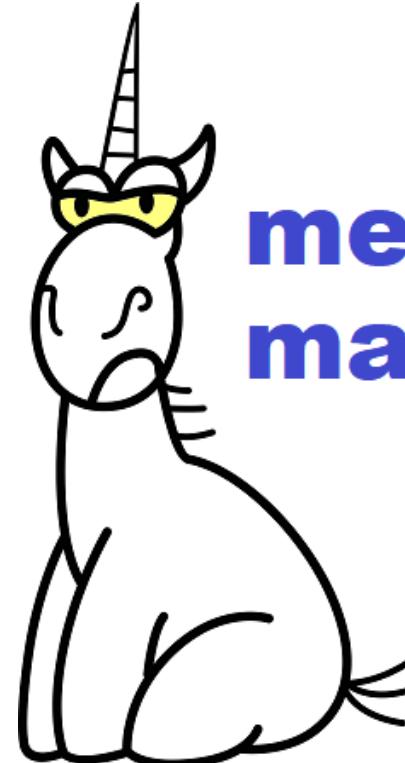
memset

Пример N1 (проект ReactOS)

```
1. void Mapdesc::identify(REAL dest[MAXCOORDS][MAXCOORDS]) {  
2.     memset(dest, 0, sizeof(dest));  
3.     for (int i = 0; i != hcoords; i++)  
4.         dest[i][i] = 1.0;  
5. }
```

Пример N2 (проект Wolfenstein 3D)

```
1. typedef struct cvar_s {  
2.     char *name;  
3.     // ...  
4.     struct cvar_s *hashNext;  
5. } cvar_t;  
  
6. void Cvar_Restart_f(void) {  
7.     cvar_t *var;  
8.     // ...  
9.     memset(var, 0, sizeof(var));  
10.    // ...  
11. }
```



**memset
malloc**

memset

Пример №3 (проект SMTP Client)

```
1. void MD5::finalize() {
2.     // ...
3.     uint1 buffer[64];
4.     // ...
5.     // zeroize sensitive information
6.     memset(buffer, 0, sizeof(*buffer));
7.     // ...
8. }
```

memset

Пример N3 (проект SMTP Client)

```
1. void MD5::finalize() {
2.     // ...
3.     uint1 buffer[64];
4.     // ...
5.     // zeroize sensitive information
6.     memset(buffer, 0, sizeof(*buffer));
7.     // ...
8. }
```

Пример N4 (проект Notepad++)

```
1. const int CONT_MAP_MAX = 50;
2. int _iContMap[CONT_MAP_MAX];

3. DockingManager::DockingManager() {
4.     // ...
5.     memset(_iContMap, -1, CONT_MAP_MAX);
6.     // ...
7. }
```

memset

Пример №9 (проект IPP Samples)

```
1. class MediaDataEx {
2.     class _MediaDataEx {
3.         // ...
4.         virtual bool TryStrongCasting(
5.             pDynamicCastFunction pCandidateFunction) const;
6.         virtual bool TryWeakCasting(
7.             pDynamicCastFunction pCandidateFunction) const;
8.     };
9. };
10. Status VC1Splitter::Init(SplitterParams& rInit) {
11.     MediaDataEx::_MediaDataEx *m_stCodes;
12.     // ...
13.     m_stCodes = (MediaDataEx::_MediaDataEx *)
14.         ippMalloc_8u(START_CODE_NUMBER * 2 * sizeof(Ipp32s) +
15.                     sizeof(MediaDataEx::_MediaDataEx));
16.     // ...
17.     memset(m_stCodes, 0,
18.            (START_CODE_NUMBER * 2 * sizeof(Ipp32s) +
19.             sizeof(MediaDataEx::_MediaDataEx)));
20.     // ...
21. }
22. Memset is evil! Страна переходит на сторону тьмы!
https://habr.com/ru/post/272269/
```

malloc

никогда не возвращает NULL, а когда возвращает уже ничего сделать нельзя

1. Почему важно проверять, что вернула функция malloc
2. <https://www.viva64.com/ru/b/0558/>

```
3. void* xmalloc(size_t s) {  
4.     void* p = malloc(s);  
5.     if (!p) {  
6.         fprintf(stderr, "fatal: out of memory (xmalloc(%zu)).\n", s);  
7.         exit(EXIT_FAILURE);  
8.     }  
9.     return p;  
10. }
```



Упражнение

найди все баги

```
1. #include <thread>

2. int main() {
3.     int *a = new int[4];
4.     int *b = new int[4];
5.     std::thread t([&]() { b++; });
6.     delete a;
7.     t.detach();
8.     return *a + (*++b) + b[3];
9. }
```

Memory management

```
template <class T>
T* allocate(size_t n, T*);  
  
template <class T>
void deallocate(T* buffer);  
  
template <class T>
Pair<T*, size_t> getTemporaryBuffer(size_t n, T*);  
  
template <class T>
void construct(T* p, const T& value);  
  
template <class T>
void destroy(T* pointer);  
  
template <class T>
void destroy(T* first, T* last);
```

std::allocator

установка аллокатора контейнера STL

```
1. // Аллокатор по умолчанию
2. list<int> b;
3. list<int, allocator<int>> c;

4. // Пользовательский аллокатор
5. #include "MyAlloc.h"
6. list<int, MyAlloc<int>> d;
```

std::allocator

полный прототип некоторых контейнеров в STL

```
1. #include <memory>

2. template<
3.     class T,
4.     class Allocator = std::allocator<T>
5. > class vector;

6. template<
7.     class Key,
8.     class T,
9.     class Compare = std::less<Key>,
10.    class Allocator = std::allocator<std::pair<const Key, T>>
11. > class map;
```

allocator C++03

- **Аллокаторы** – это объекты отвечающие за управление памятью (memory management) и скрытие реализации этого управления от потребителей (структур и контейнеров)
- Контейнеры должны иметь дело с неинициализированной памятью и только создавать объекты в этой необработанной памяти по требованию. Контейнерам нужен интерфейс более низкого уровня, чем **new** и **delete**, ближе к **malloc()** и **free()**
- Аллокаторы разделяют выделение памяти и конструирование, уничтожение объекта и очистку памяти в независимые шаги
- В STL все контейнеры включают параметр Allocator, который по умолчанию равен **std::allocator**
- **Declaration:**

```
template <class T> class allocator;
```

Из книги [Josuttis 1999]

“...Аллокаторы изначально были введены в *STL* для обработки отвратительной проблемы разных видов указателей на ПК (*near/far/huge pointers*).

Сейчас они служат техническим решением выбора моделей памяти, распределённой памяти (*shared memory*), сборки мусора (*garbage collection*) и размещение объектов в БД без изменения интерфейса. Тем не менее, это достаточно молодые проблемы, которые ещё не широко известны...

Аллокаторы представляют собой специальную модель управлению памятью обобщённых алгоритмов, являются абстракцией операции трансляции требования памяти в реальное обращение к памяти.

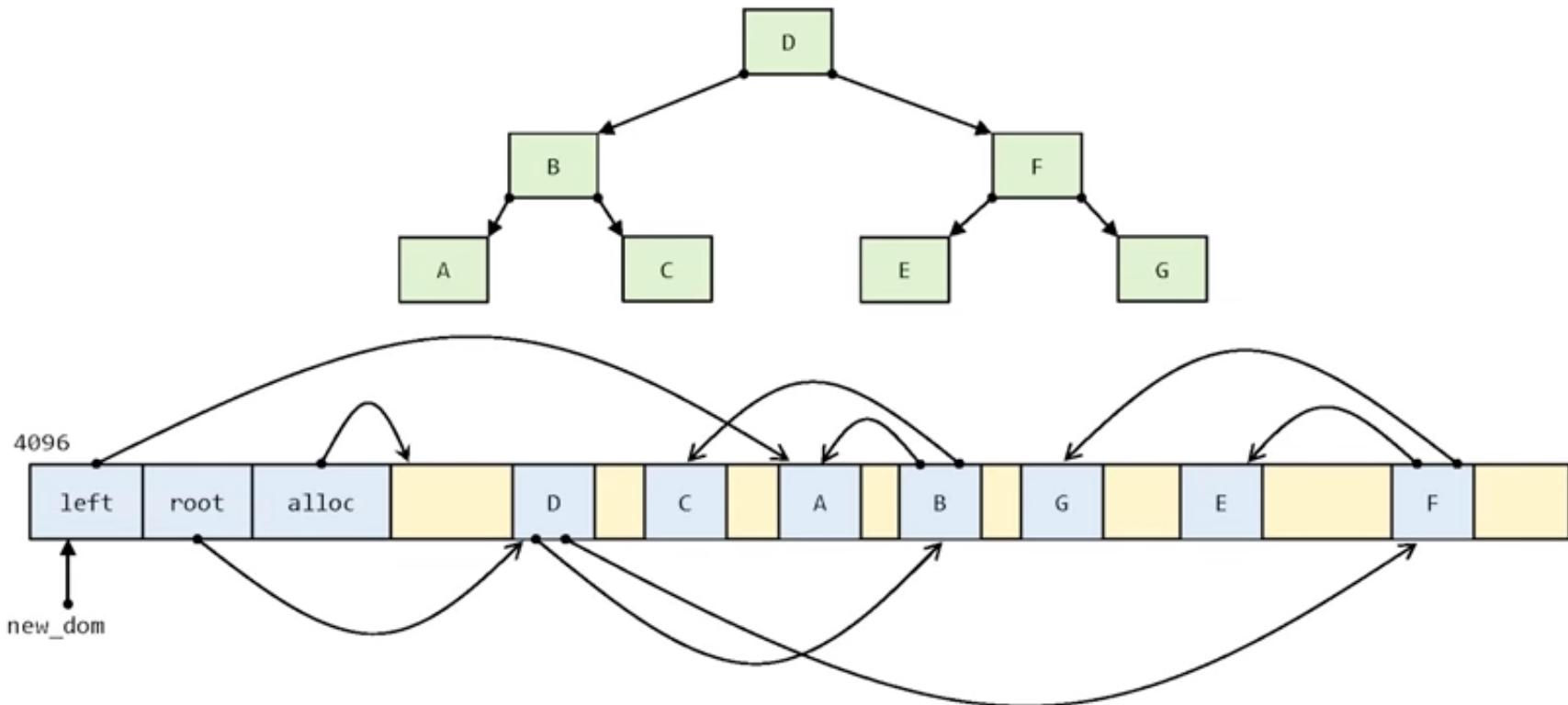
Они представляют собой интерфейс для создания и уничтожения объектов в подготовленной для них области памяти компьютера.

Благодаря аллокаторам контейнеры и алгоритмы могут быть параметризованы на разные виды элементов, хранящихся в разной памяти...”

Область применения аллокаторов

- **Объём:** Оптимизация использования памяти (pools, fixed-size allocators)
- **Скорость:** Уменьшение времени выделения памяти (single-threaded, one-time free)
- **Варианты применения:**
 - Не блокирующий доступ к heap каждый потоком
 - Предотвращение фрагментации памяти
 - Выравнивание объектов в памяти
 - Выделение памяти одного размера для разных объектов
 - Пользовательский список/порядок очисток
 - Возможности отладки при управлении памятью
 - Пользовательская функциональность heap
 - Специфический тип памяти

Motivating Example – Self-Contained DOM



Не могут быть применимы, если

- Нет доступной в системе функции `realloc()`
- Требуется продвинутый компилятор C++
- C++ Standard hand-waving
- В целом зависимы от библиотек (**library-specific**)
 - Изменив библиотеки STL, нужно изменить аллокаторы
- В общем непереносимы (**not cross-platform**)
 - Заменив компилятор, нужно заменить и аллокатор

Ограничения аллокаторов C++03

- Нельзя использовать параметры шаблона шаблона
- **`template<class T, template<class> class Alloc>`**
`class vector { };`
- Аллокаторы обязательно должны быть шаблонами, привязаны к определенному типу - нельзя легко использовать один аллокатор для нескольких типов
- Можно получить копию только из контейнеров, нельзя из исходного объекта-аллокатора
- Ограничения на указатель и другие вложенные типы исключают использование их для поддержки экзотических моделей памяти
- Затруднена поддержка аллокатора с состоянием, не являются равных друг другу (стандарт C++03 требует **равенства всех экземпляров аллокаторов одного типа**)

Как написать свой аллокатор?

- Скопировать или унаследовать интерфейс от **std::allocator**
- **Core functionality**
 - allocate
 - deallocate
- По необходимости переопределить функции-помощники

Как написать свой аллокатор?

```
1. #include <memory>
2. using namespace std;

3. template <typename T> class my_allocator {
4.     static std::size_t alloc_count = 0u;
5.     std::size_t id_;
6. public:
7.     using value_type = T;
8.     my_allocator() : id_(++alloc_count) {}
9.     template <typename U> my_allocator(const my_allocator<U> &a) : id_(a.id_) {}

10.    T* allocate(std::size_t n) { return allocator<T>().allocate(n); }
11.    void deallocate(T *ptr, std::size_t n) {allocator<T>().deallocate(ptr, n); }

12.    std::size_t id() const { return id_; }
13. };
14. template <typename T, typename U>
15. bool operator==(const my_allocator<T> a, const my_allocator<U>&b)
16. { return (a.id_ == b.id_); }

17. template <typename T, typename U>
18. bool operator!=(const my_allocator<T>&a, const my_allocator<U>&b)
19. { return !(a == b); }
```

Интерфейс std::allocator

```
pointer allocate(size_type n,  
allocator<void>::const_pointer p = nullptr)
```

- **n** число экземпляров **T**, НЕ байтов
- Возвращает указатель на область памяти, достаточную для хранения **n * sizeof(T)** байт
- Возвращает просто байты памяти, НЕ конструирует
- Может бросать (**throw**) исключение (**std::bad_alloc**)
- По умолчанию вызывает **::operator new**
- **p** опциональный трюк; избегайте и опасайтесь

Интерфейс std::allocator

```
void deallocate(pointer p,  
                  size_type n)
```

- **p** должен получать результат функции **allocate()**
- **p** должен быть набором байт; уже деконструированным
- **n** должен соответствовать значению **n**, переданному в функцию **allocate()**
- По умолчанию вызывает **::operator delete(void*)**
- Большинство реализаций разрешают и игнорируют значение **nullptr** аргумента **p**; поэтому и Вам следует

Интерфейс std::allocator

- **allocate()** не вызывает конструкторов
- **deallocate()** не вызывает деструкторов
- **Почему?**

Интерфейс std::allocator

- **allocate()** не вызывает конструкторов
- **deallocate()** не вызывает деструкторов
- **Почему?** Производительность
 - Для вызова конструктора есть функция **construct()**
 - Для вызова деструктора есть функция **destroy()**

Интерфейс std::allocator

```
void construct(pointer p, const T& t)
{ new(static_cast<void*>(p)) T(t); }
```

- Placement new

- НЕ выделяет памяти
- Вызывает конструктор копирования

```
void destroy(pointer p)
{ static_cast<T*>(p) ->~T(); }
```

- Прямой вызов деструктора (destructor invocation)

- НЕ очищает память
- Вызывает деструктор

std::allocator

функции

```
1. #include <iostream>
2. #include <memory>

3. int main() {
4.     std::allocator<int> myAllocator;      // allocator for integer values
5.     int* arr = myAllocator.allocate(5);   // allocate space for five ints
6.     myAllocator.construct(arr, 100);      // construct arr[0] and arr[3]
7.     arr[3] = 10;
8.     std::cout << arr[3] << std::endl;
9.     std::cout << arr[0] << std::endl;
10.    myAllocator.deallocate(arr, 5);      // deallocate space for five ints
11.    return 0;
12. }
```

std::allocator

функции

```
1. #include <iostream>
2. #include <memory>
3. #include <string>

4. int main() {
5.     std::allocator<std::string> myAllocator;
6.     std::string* str = myAllocator.allocate(3);

7.     myAllocator.construct(str, "Geeks");
8.     myAllocator.construct(str + 1, "for");
9.     myAllocator.construct(str + 2, "Geeks");
10.    std::cout << str[0] << str[1] << str[2];

11.    myAllocator.destroy(str);
12.    myAllocator.destroy(str + 1);
13.    myAllocator.destroy(str + 2);
14.    myAllocator.deallocate(str, 3);
15.    return 0;
16. }
```

Rebind (deprecated C++17)

- Аллокаторы не все выделяют память под тип T

```
list<Obj> ObjList; // allocates nodes
```

- Определив вложенный тип Rebind

```
template<typename U> struct rebind
{ typedef allocator<U> other; }
```

- Можем аллоцировать что-то другое

```
Alloc<T> a;
T* t = a.allocate(1); // allocs sizeof(T)
Alloc<T>::rebind<N>::other na;
N* n = na.allocate(1); // allocs sizeof(N)
```

- Требуется, например, чтобы **std::list** работал правильно, т.к. **Allocator<int>**, переданный в список **std::list<int>**, на самом деле, должен выделять память для **std::list<int>::Node<int>**.

To Derive or Not to Derive

- Наследовать свой аллокатор от `std::allocator`
 - Пишите меньше кода, легче увидеть различия
 - Должен предоставлять `rebind`, `allocate`, `deallocate`
 - Гениально реализованные наследников можно найти в заголовочном файле `<xdebug>`
- Написать с нуля
 - Аллокатор не был спроектирован, как базовый класс
 - Лучшее понимание работы
 - Josuttis и Austern написали с нуля
 - Austern, Matt, C/C++ Users Journal
[The Standard Librarian: What Are Allocators Good For?](#)
 - [Nicolai M. Josuttis](#) book
[The C++ Standard Library - A Tutorial and Reference](#)
- Personal preference

Allocator with State

(no problems since C++11)

- **State** = поля данных в аллокаторе
- Аллокатор по умолчанию состояния (данных) не имеет
- Стандарт C++03 (paragraph **20.1.5**):
 - Поставщики сами решают поддерживают ли аллокаторы состояние
 - Контейнеры могут полагаться, что аллокатор не имеет состояния
 - Проблемы совместимости в версиях STL
 - Функции `list::splice()` и `C::swap()` проявят отсутствие поддержки состояния
- В любом случае **тестируйте основательно!**
- Аллокатор – это важнейшая часть языка C++, используется повсеместно. Если что-то сломать в аллокаторе, ошибка Вас скоро обязательно найдёт!

Heap Allocator

something new

```
1.  template <typename T>
2.  class HeapAllocator {
3.      HeapAllocator();
4.      HANDLE m_hHeap{};
5.  public:
6.      explicit HeapAllocator(HANDLE hHeap);
7.      HeapAllocator(const Halloc&); // copy
8.
9.      template <typename U> // templated copy
10.     HeapAllocator(const HeapAllocator<U> &a) :
11.         m_hHeap(a.m_hHeap) {} // error
12.     // ...
13. }
```

Heap Allocator

something new

```
1.  template <typename T>
2.  class HeapAllocator {
3.      HeapAllocator();
4.      HANDLE m_hHeap{};
5.  public:
6.      explicit HeapAllocator(HANDLE hHeap);
7.      HeapAllocator(const Halloc&); // copy
8.
9.      template <typename U> // templated copy
10.     HeapAllocator(const HeapAllocator<U> &a) :
11.         m_hHeap(a.m_hHeap) {} // error
12.     // ...
13.     template <typename U>
14.     friend class HeapAllocator;
```

Heap Allocator

something new

```
1.  template <typename T>
2.  class HeapAllocator {
3.      HeapAllocator();
4.      HANDLE m_hHeap{};
5.  public:
6.      explicit HeapAllocator(HANDLE hHeap);
7.      HeapAllocator(const Halloc&); // copy
8.
9.      template <typename U> // templated copy
10.     HeapAllocator(const HeapAllocator<U> &a) :
11.         m_hHeap(a.m_hHeap) {} // error
12.     // ...
13.     template <typename U>
14.     friend class HeapAllocator;
15.
16.     template <typename T, typename U>
17.     bool operator==(const HeapAllocator<T>& a, const HeapAllocator<U>& b)
18.     { return a.state == b.state; }
```

Types aliases

псевдонимы типов при подмене аллокаторов, чтобы не запутаться

```
1. // опасно
2. list<int, HeapAllocator<int>> b;

3. // лучше
4. // .h
5. using IntHeapAlloc = HeapAllocator<int>;
6. using IntListOnHeap = list<int, IntHeapAlloc>;
7. // .cpp
8. IntListOnHeap list_container;
```

Types aliases

псевдонимы типов при подмене аллокаторов, чтобы не запутаться

```
1. // containers accept allocators via ctors
2. IntListOnHeap list_container(IntHeapAlloc(x, y, z));

3. // if none specified, you get the default
4. IntListOnHeap list_container_new; // calls IntHeapAlloc()

5. // map/multimap requires std::pairs
6. using PairKTHeapAlloc = HeapAlloc<std::pair<K, T>>;
7. PairKTHeapAlloc alloc;

8. map<K, T, less<K>, PairKTHeapAlloc> m(less<K>(), alloc);

9. // container adaptors accept containers via constructors, not allocators
10. HeapAllocator<T> al;
11. deque<T, HeapAllocator<T>> deq(al);
12. stack<T, deque<T, HeapAllocator<T>>> s(deq);

13. // string example
14. basic_string<T, char_traits<T>, HeapAllocator<T>> str(al);
```

Allocator Testing

- Проверьте нормальный случай использования
- Протестируйте со всеми контейнерами
(не забудьте строки, хэш-контейнеры, стэк и т.д.)
- Протестируйте с различными объектами T ,
особенно с нетривиальными деструкторами
- Протестируйте особые, ошибочные, граничные
(edge cases) сценарии, такие как `list::splice`
- Убедитесь, что ваша версия лучше!
- **Allocator test framework:**
www.tantalon.com/pete.htm

Итог.

Когда писать свой аллокатор?

- Аллокаторы: **последний рубеж оптимизации**
- Основывай свой аллокатор на файле **<memory>**
- Остерегайся портирования кода
(на новый компилятор или версию библиотеки STL)
- **Тестируйте основательно**
- **Убедитесь в уличениях**
скорости или занимаемого места в памяти
- Используйте псевдонимы типов

allocator: C++03

- Стандарт C++03 требовал определить [20.1.5 allocator requirements] [20.4.1 default allocator]
- «указатель на *T*» (**pointer**),
- «константный указатель на *T*» (**const_pointer**),
- «ссылку на *T*»(**reference**),
- «константную ссылку на *T*» (**const_reference**),
- «сам тип *T*» (**value_type**),
- «беззнаковый целочисленный тип, представляет размер самого большого объекта в модели аллокации» (**size_type**),
- «знаковый целочисленный тип представляет разность двух указателей на *T*» (**difference_type**)
- шаблонный член-класс **rebind**, преобразователь типа из **Allocator<T>** в **Allocator<U>**
- метод **address()** возвращает адрес памяти под объект типа *T*
- **allocate/deallocate/construct/destroy**

std::allocator: C++03

```
1.  template<typename T>
2.  class Allocator {
3.  public:
4.      using value_type = T;
5.      using pointer = value_type*;
6.      using const_pointer = const value_type*;
7.      using reference = value_type&;
8.      using const_reference = const value_type&;
9.      using size_type = std::size_t;
10.     using difference_type = std::ptrdiff_t;

11.    // convert an allocator<T> to allocator<U>
16.    template<typename U> struct rebind { typedef Allocator<U> other; };

17.    inline explicit Allocator() {}
18.    inline explicit Allocator(const Allocator&) {}
19.    template<typename U> inline explicit Allocator(const Allocator<U>&) {}
20.    inline ~Allocator() {}
21.    // <-- see next
```

std::allocator: C++03

```
16.     // <-- see prev
17.     inline pointer address(reference r) { return &r; }
18.     inline const_pointer address(const_reference r) { return &r; }

31.     inline size_type max_size() const
32.     { return std::numeric_limits<size_type>::max() / sizeof(T); }

16.     using P = typename std::allocator<void>::const_pointer;
17.     inline pointer allocate(size_type cnt, P a=nullptr)
18.     { return reinterpret_cast<pointer>(::operator new(cnt * sizeof (T))); }

19.     inline void deallocate(pointer p, size_type)
20.     { ::operator delete(p); }

31.     inline void construct(pointer p, const T& t) { new(p) T(t); }
32.     inline void destroy(pointer p) { p->~T(); }

33.     inline bool operator==(const Allocator&) { return true; }
34.     inline bool operator!=(const Allocator &a) { return !((*this)==(a)); }
35. }; // end of class Allocator
```

allocator: C++11

- В C++11 разрешены *аллокаторы с состоянием*, в том числе со *статическим состоянием*
- **std::allocator_traits** предоставляет все шаблоны типов
- Аллокатор контейнера не фиксирован в конструкторе
- Модель **scoped allocator** теперь безопасна
 - Выделяет память из *shared memory*, доступной нескольким потокам

Новые требования стандарта [17.6.3.5 alloc.requirements]

- Должно быть **static_cast** преобразование **pointer** к **void_pointer**
- **construct()** и **destroy()** принимают **raw** указатели, а не тип **pointer**
- **construct(T*, Args&&...)** функция с аргументами **variadic template**, поддерживает **perfect forwarding**
- Больше не нужны **A::reference** и **A::const_reference**

allocator C++11

```
1. template <class T> class allocator {
2. public:
3.     using value_type      =typename T;
4.     using pointer         =typename value_type*;
5.     using const_pointer   =typename pointer_traits<pointer>::rebind<value_type const>;
6.     using void_pointer    =typename pointer_traits<pointer>::rebind<void>;
7.     using const_void_pointer=typename pointer_traits<pointer>::rebind<const void>;
8.     using difference_type =typename pointer_traits<pointer>::difference_type;
9.     using size_type       =typename make_unsigned_t<difference_type>;
10.    template <class U> struct rebind { typedef allocator<U> other; };
11.    allocator() noexcept {} // not required, unless used
12.    template <class U> allocator(allocator<U> const&) noexcept {}
16.    pointer allocate(size_t n, const_void_pointer) { return allocate(n); }
17.    pointer allocate(size_t n)
18.    { return static_cast<value_type*>(::operator new(n * sizeof(value_type))); }
19.    void deallocate(pointer p, size_t) noexcept { ::operator delete(p); }
20.    template <class U, class ...Args> void construct(U* p, Args&& ...args)
21.    { ::new(p) U(forward<Args>(args)...); }
22.    template <class U> void destroy(U* p) noexcept { p->~U(); }
26.    size_t max_size() const noexcept
27.    { return std::numeric_limits<size_type>::max() / sizeof(value_type); }
28.    allocator select_on_container_copy_construction() const { return *this; }
29. }; // end of allocator class
```

```
40. template <class T, class U>
41. bool operator==(allocator<T> const&, allocator<U> const&) noexcept
42. { return true; }

43. template <class T, class U>
44. bool operator!=(allocator<T> const& x, allocator<U> const& y) noexcept
45. { return !(x == y); }
```

```
1. // using propagate_on_container_copy_assignment = std::false_type;
2. // using propagate_on_container_move_assignment = std::false_type;
3. // using propagate_on_container_swap           = std::false_type;
```

allocator C++11

minimal

```
1.  template<typename T>
2.  struct Alloc {
3.      typename T value_type;
4.
5.      Alloc();
6.      template<typename U>
7.      Alloc(const Alloc<U>&);
8.
9.      T* allocate(size_t n);
10.     void deallocate(T*, size_t);
11. };
12.
13. template<typename T>
14. bool operator==(const Alloc<T>&, const Alloc<T>&);
```

allocator: C++11

Указать, что **созданный тип использует allocator**:

- предоставить соответствующие конструкторы
- определить вложенный член **allocator_type**
- специализировать (или частично специализировать) признак **std::used_allocator**

Container implementations are significantly more complicated by the **propagation traits**

No compile-time property to tell if allocators compare **always equal** (C++17)
(swap noexcept when using stateless allocators)

Allocators: The reviews are not good

std::STL allocators are painful to work with. [2]

The C++ Standardization Committee added wording to the Standard that emasculated allocators as objects. [3]

Allocators are one of the most mysterious parts of the C++ Standard library. [4]

It is now accepted by the C++ community that allocators are fairly useless. [1]

1. Chris Baus, *C++ pooled_list class alpha release*, 2006
2. Paul Pedriana, *N2271 EASTL*, 2007
3. Meyers: *Effective C++ Digital Edition*, 2012
4. Matt Austern, *The Standard Librarian: What Are Allocators Good For?*, Dr. Dobbs, 2000

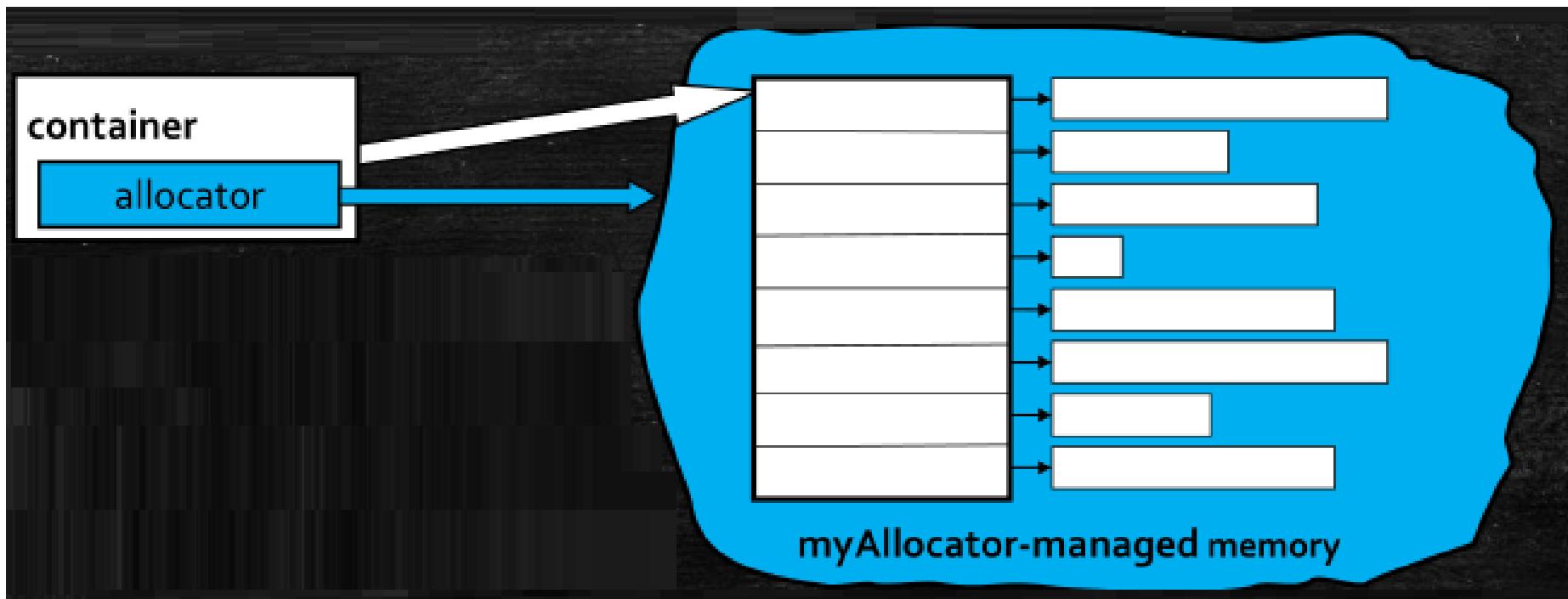
С++11 упростил использование аллокаторов

Но сделал реализацию контейнеров сложнее

- STL контейнеры всегда взаимодействовали с аллокатором с помощью класса `allocator_traits`
- Размножение `traits` добавило гибкости, но также и сложности в сам контейнер
- Представили большую **гранулярность** операций
- Большое число `typedefs`
- Изменение шаблона `rebind`
- Обобщённая модель указателя
- Политика типа-аллокатора для каждого контейнера
- `allocator_traits` для обратной совместимости

С++11 упростил использование аллокаторов

- `vector<string> container(myAllocator);`



std::allocator

C++17

20.7.2 Header <memory> synopsis [memory.syn]

```
namespace std {
// ...

// 20.7.9, the default allocator:
template <class T> class allocator;
template <> class allocator<void>;
template <class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
template <class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

// ...
}
```

20.7.9 The default allocator [default_allocator]

1. All specializations of the default allocator satisfy the allocator completeness requirements 17.6.3.5.1.

```
namespace std {
    template <class T> class allocator;

    // specialize for void:
    template <> class allocator<void>;
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // reference to void members are impossible.
    typedef void value_type;
    template <<class U>> struct rebind { typedef allocator<U> other; };
};

template <class T> class allocator {
public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    template <<class U>> struct rebind { typedef allocator<U> other; };
    typedef true_type propagate_on_container_move_assignment;
    typedef true_type is_always_equal;
    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template <class U> allocator(const allocator<U>&) noexcept;
    ~allocator();
    pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;
    pointer<T*> allocate(
        size_type, allocator<void>::const_pointer const void* hint = 0);
    void deallocate(pointer<T*> p, size_type n);
    size_type max_size() const noexcept;
    template <class U, class... Args>
        void construct(U* p, Args&&... args);
    template <class U>
        void destroy(U* p);
};

}
```

std::allocator (C++17)

```
1. template <class T>
2. class allocator
3. {
4. public:
5.     using value_type = T;
6.     using is_always_equal = std::true_type;
7.     using propagate_on_container_move_assignment = std::true_type;
8.
9.     allocator() noexcept {} // not required, unless used
10.    template <class U> allocator(allocator<U> const&) noexcept {}
11.
12.    pointer allocate(size_t n, const void_pointer) { return allocate(n); }
13.    pointer allocate(std::size_t n)
14.    { return static_cast<value_type*>(::operator new(n*sizeof(value_type))); }
15.
16.    void deallocate(pointer p, std::size_t) noexcept
17.    { ::operator delete(p); }
18. }
```

Члены-методы интерфейса аллокатора:

[allocator requirements table](#) [20.1.5.1 allocator requirements]

- **allocate** и **deallocate**: выделение и освобождение памяти
- **address** (deprecated in C++17, removed in C++20)
- **rebind** (deprecated in C++17, removed in C++20)
- **max_size** (deprecated in C++17, removed in C++20)
- **construct** и **destroy** (deprecated in C++17, removed in C++20)
because they're **useless** and **add nothing over the default**
handling **memory alignment** is the **task of allocate**, not construct
- **std::allocator<void>** (removed in C++17) because is useless

Polymorphic memory resources (PMR) (C++17)

- Defined in namespace `std::pmr`
- Provide **runtime polymorphism** with single type argument to containers
- Client allocators store a pointer to a base class **memory resource**
- **No lateral propagation** – an allocator sticks for life

Polymorphic memory resources (PMR) (C++17)

simple base class with allocate and deallocate member functions

```
1.  class pmr::memory_resource
2.  {
3.      static constexpr size_t max_align = alignof(max_align_t);
4.  public:
5.      virtual ~memory_resource();
6.      void* allocate(size_t bytes, size_t alignment = max_align);
7.      void deallocate(void* p, size_t bytes, size_t alignment = max_align);
8.      bool is_equal(const memory_resource& other) const noexcept;
9.  private:
10.     virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
11.     virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
12.     virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
13. }
```

Polymorphic Allocator (C++17)

wrapper around a pointer to `pmr::memory_resource`

for backwards-compatibility with the C++11 (and C++03) allocator model

```
1.  template<class T> class polymorphic_allocator
2.  {
3.      memory_resource* memory_rsrc;
4.  public:
5.      using value_type = T;
6.      polymorphic_allocator() noexcept;
7.      polymorphic_allocator(memory_resource* r);
8.      polymorphic_allocator(const polymorphic_allocator& other) = default;
9.      template <class U>
10.     polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;
11.     polymorphic_allocator& operator=(const polymorphic_allocator& rhs) = delete;
12.     T* allocate(size_t n);
13.     void deallocate(T* p, size_t n);
14.     polymorphic_allocator select_on_container_copy_construction() const;
15.     // ...
16.     memory_resource* resource() const;
17. }
```

PMR Type Aliases (C++17)

A partial list of type aliases now provided in std::pmr

```
1. namespace pmr
2. {
3.     template <class charT, class traits = char_traits<charT>>
4.     using basic_string = std::basic_string<charT, traits, polymorphic_allocator<charT>>;
5.     using string = basic_string<char>;
6. //...
7.     template <class T>
8.     using deque = std::deque<T, polymorphic_allocator<T>>;
9. //...
10.    template <class Key, class Compare = less<Key>>
11.    using set = std::set<Key, Compare, polymorphic_allocator<Key>>;
12. //...
13.    template <class Key, class Hash = hash<Key>, class Pred = equal_to<Key>>
14.    using unordered_set = std::unordered_set<Key, Hash, Pred,
15.                                         polymorphic_allocator<Key>>;
16. }
```

Упрощения C++17 относительно C++11

Task	C++98/C++03	C++11/C++14	C++17 polymorphic_allocator<byte>
Use an allocator	MEDIUM viral templates	MEDIUM viral templates	EASY
Create an allocator	MEDIUM Lots of boilerplate, non-portable state	EASY	EASY just derive from memory_resource
Create a scoped allocator	IMPOSSIBLE	MEDIUM-EASY alias scoped_allocator_adaptor	EASY polymorphic_allocator is scoped
Create a new allocator-aware container	MEDIUM rebinding needed, ignore allocator state?	HARD propagation traits, allocator_traits	EASY skip C++11 complexity

C++17 defines several standard resources

- **`new_delete_resource()`**: allocates using `::operator new`
- **`null_memory_resource()`**: throws on allocation
- **`synchronized_pool_resource`**: Thread-safe pools of similar-sized memory blocks
- **`unsynchronized_pool_resource`**: Non-thread-safe pools of similar-sized memory blocks
- **`monotonic_buffer_resource`**: Super-fast, non-thread-safe allocation into a buffer with do-nothing deallocation
- A memory resource for testing
<https://github.com/phalpern/CppCon2017Code>

Polymorphic memory resources (PMR) (C++17)

authoring a test resource

```
1.  class test_resource : public pmr::memory_resource {
2. public:
3.     explicit test_resource(pmr::memory_resource *parent =
4.                           pmr::get_default_resource());
5.     ~test_resource();
6.     pmr::memory_resource *parent() const;
7.     size_t bytes_allocated() const;
8.     size_t bytes_deallocated() const;
9.     size_t bytes_outstanding() const;
10.    size_t bytes_highwater() const;
11.    size_t blocks_outstanding() const;
12.    static size_t leaked_bytes();
13.    static size_t leaked_blocks();
14.    static void clear_leaked();
15.    // ...
16. }
```

Упрощения C++17 относительно C++11

- For node-based containers, define a node containing an element that is not automatically constructed.
 - Putting the element inside, then call the element's constructor
 - Allocate raw memory for nodes using:
 - **node* new_node = static_cast<node*>(
allocator.resource()->allocate(sizeof(node),
alignof(node)));**
- Construct new elements within nodes using
 - **allocator.construct(std::addressof(new_node->m_value),
std::forward<Args>(args)...);**

C++17 supports a simpler allocator model

- **std::pmr::memory_resource** простой базовый класс с методами **allocate** и **deallocate**.
- **std::pmr::polymorphic_allocator** – обёртка над указателем на **pmr::memory_resource** для обратной совместимости с C++11 (и C++03) моделями аллокаторов.
- **std::pmr::vector<T>** - плевдоним **std::vector<T, std::pmr::polymorphic_allocator<T>>**
- Similarly for the other allocator-aware standard containers.

Упрощения C++17 относительно C++11

Простое использование **polymorphic_allocator<byte>**

- Без шаблонного аргумента аллокатора, аллокатор всегда тот же
- Нет необходимости в **allocator_traits**, аллокатор имеет всегда те же **traits**
- *No propagation traits – allocators don't propagate except on move construction.*
- Не нужен **rebind** – просто аллоцируйте байты
- Простые понятные поля по умолчанию
- Наследуйтесь от **pmr::memory_resource** для создания нового механизма аллоцирования

Упражнения

1. Реализовать простой **memory leaks printer**, переопределением операторов **new/delete**
2. Напишите свой аллокатор, как обёртку над стандартным аллокатором
(проверьте его работу с несколькими стандартными контейнерами)
3. Напишите свой аллокатор, распределяя память с помощью **mmap/HeapAlloc** и печатая выполняемые операции в консоль
(проверьте его работу с несколькими стандартными контейнерами)
4. Реализовать **memory arena allocator** без багов
(проверьте его работу с несколькими стандартными контейнерами)

СПАСИБО ЗА ВНИМАНИЕ!