Лекция по C++

Управление памятью, new, delete. Умные указатели

new и delete

Оператор *new* динамически выделяет память на объект в куче

```
int* a = new int(13);
```

Оператор *delete* освобождает память, выделенную с помощью new

```
delete a;
```

new[] и delete[]

```
int* a = new int[10];
a[0] = 1;
std::cout << a[0] << std::endl;
delete[] a;
```

new и delete

```
class Record {
  public:
    Record(int size) {
        numbers = new int[size];
    }
  private:
    int* numbers;
};

int main() {
    {
        Record r(100);
    }
    // memory leak!
}
```

new и delete

```
class Record {
  public:
    Record(int size) {
        numbers = new int[size];
    }
    ~Record() {
        delete[] numbers;
    }
  private:
    int* numbers;
};

int main() {
    {
        Record r(100);
    } // destructor is called here
}
```

new и malloc

```
int main() {
    int *a = new int[4];
    int *b = static_cast<int*>(malloc(sizeof(int) * 4));
    // static_cast выполняет явно допустимое
    // приведение типов данных
    // static_cast<T>(B) - преобразовать B в T
    // используйте static_cast вместо c-style кастов!
}
```

Если вы пишете на C++, не используйте malloc!

Лирическое отступление

auto позволяет не указывать тип явно. Вместо этого компилятор сам выведет тип на основе типа инициализируемого выражения.

```
int main() {
    std::vector<std::vector<int>> v;

    std::vector<std::vector<int>>* v_ptr = &v;
    auto v_ptr_ = &v;

    std::vector<std::vector<int>>::iterator it = v.begin();
    auto it_ = v.begin();
}
```

new: вызов конструктора и деструктора

```
struct Test {
    Test() {
        std::cout << "constructor of test called\n";
    }
    ~Test() {
        std::cout << "destructor of test called\n";
    }
};

int main() {
    auto t = new Test();  // constructor of test called
    delete t;  // destructor of test called
}
```

Placement new

```
struct Test {
    Test() {
        std::cout << "constructor of test called\n";
    }
    ~Test() {
        std::cout << "destructor of test called\n";
    }
};

int main() {
    char* ptr = new char[sizeof(Test)];
    auto t = new(ptr) Test;  // constructor of test called
    t->~Test();  // destructor of test called
    delete[] ptr;
}
```

Operator new

```
struct Test {
    Test() {
        std::cout << "constructor of test called\n";
    }
    ~Test() {
        std::cout << "destructor of test called\n";
    }
};

int main() {
    void* ptr = ::operator new(sizeof(Test));
    auto t = new(ptr) Test;  // constructor of test called
    t->~Test();  // destructor of test called
    ::operator delete(ptr);
}
```

Перегрузка new и delete

```
class Test {
  public:
    void *operator new(size_t size) {
        return new char[size];
    }
    void operator delete(void *p) {
        delete p;
    }
};

int main() {
    auto ptr = new Test();
    delete ptr;
}
```

Fixed allocator

Аллокатор инкапсулирует управление памятью.
Fixed allocator - аллокатор, выделяющий память блоками заданного размера.

```
const size_t CHUNK_SIZE = 1024; // размер блока
struct Chunk { // структура, описывающая блоки памяти
    char buff[CHUNK_SIZE];
    Chunk* next;
};
```

Fixed allocator

```
class FixedAllocator {
  public:
    void* allocate() {
        if (free == nullptr) new_pool();
        auto result = free;
        free = free->next;
        return static_cast<void*>(result);
    }
    void deallocate(void* ptr) {
        auto node = static_cast<Chunk*>(ptr);
        node->next = free;
        free = node;
    }
  private:
    void new_pool(); // memory allocation magic hides here
    Chunk* free = nullptr;
};
```

Fixed allocator

```
class FixedAllocator {
  public:
    ...
  private:
    void new_pool() {
        // allocate a new pool of chunks
        // add it to 'pools' vector
        // make 'free' point to the top of it
    }
    Chunk* free = nullptr; // current pool
    std::vector<Chunk*> pools;
};
```

New, pointers

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    ~A() {
        std::cout << "A destructor";
    }
};

int main() {
    A* a = new A("asasa");
    // много сложного кода
    // ...
    delete a; // A destructor
}
```

New, pointers

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    ~A() {
        std::cout << "A destructor";
    }
};

int main() {
    A* a = new A("asasa");
    // много сложного кода
    if (some_complicated_condition) {
        return; // Утечка памяти!
    }
    // ...
    delete a; // Не попадем сюда из-за return-а
}
```

Unique pointer

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    ~A() {
        std::cout << "A destructor";
    }
};

int main() {
    std::unique_ptr<A> ptr = std::make_unique<A>("azaza");

    auto ptr1 = ptr; // error

} // A destructor
```

Shared pointer

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    ~A() {
        std::cout << "A destructor";
    }
};

int main() {
  {
    std::shared_ptr<A> ptr = std::make_shared<A>("ab");
  } // ref_cnt = 0
}
```

Shared pointer

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    ~A() {
        std::cout << "A destructor";
    }
};

int main() {
    std::shared_ptr<A> ptr1;
    {
      std::shared_ptr<A> ptr = std::make_shared<A>("ab");
      ptr1 = ptr;
    }
    std::cout << "no destructor yet";
} // A destructor
```

Умные указатели и наследование

```
struct A {
    A(std::string some_str) : some_str(some_str) {}
    std::string some_str;
    virtual ~A() {
        std::cout << "A destructor ";
    }
};

struct B : A {
    B() : A("special_b_string") {}
    ~B() {
        std::cout << "B destructor ";
    }
};
int main() {
    std::unique_ptr<A> ptr = std::make_unique<B>();
} // B destructor A destructor
```

Visitor pattern

```
struct BaseNode;
typedef std::shared_ptr<BaseNode> node_ptr;

struct BaseNode {
    virtual ~BaseNode() = default;
    std::vector<node_ptr> children;
};

node_ptr tree;

void Traverse(node_ptr current_node) {
    for (auto to : current_node->children) {
        Traverse(to);
    }
}
```

Visitor pattern

```
struct BaseNode {
    virtual ~BaseNode() = default;
    std::vector<node_ptr> children;
};
struct Node : BaseNode {};
struct TerminalNode : BaseNode {};

void Traverse(node_ptr cur_node) {
    // разная логика для разных узлов?
    if (std::dynamic_pointer_cast<TerminalNode>(cur_node)
        != nullptr) {
            ...
    }
    for (auto to : cur_node->children) {
        Traverse(to);
    }
}
```

Visitor pattern

```
struct BaseNode {
    virtual ~BaseNode() = default;
    virtual void Visit() = 0;
    std::vector<node_ptr> children;
};
struct TerminalNode : BaseNode {
    void Visit() override {
        // some logic
    }
};

void Traverse(node_ptr current_node) {
    current_node->Visit();
    for (auto to : current_node->children) {
        Traverse(to);
    }
}
```