Модель памяти в C++ — конспект темы

Хранение объектов в памяти

Память используется для хранения кода программы и обработки её данных.

С точки зрения C++ память компьютера состоит из одной или нескольких непрерывных последовательностей ячеек. Эти ячейки называются **байтами**.

Байт — минимальная адресуемая единица памяти. Каждый байт в памяти имеет уникальный адрес — числовое значение, задающее его местоположение в памяти.

Программы на C++ работают не с содержимым ячеек памяти напрямую, а с объектами — создают, разрушают их, считывают и модифицируют состояние. Объект в C++ — это регион в памяти, который обладает такими свойствами:

- Размер в байтах. Типы `char`, `unsigned char`, `int8_t`, `uint8_t` и `std::byte` занимают один байт памяти. Чтобы узнать, сколько байтов занимает тип или переменная, используйте оператор `sizeof`.
- Требования к выравниванию в памяти. Оператор `alignof` возвращает значение выравнивания для заданного типа на целевой платформе. Оно может отличаться от размера объекта, возвращаемого `sizeof`.
- Тип. Позволяет программе правильно работать с областью памяти, которую объект занимает.
- Значение. Определяется содержимым области памяти, занимаемой объектом.
- Продолжительность времени жизни. Например, время жизни локальных переменных ограничено блоком, внутри которого они объявлены, а глобальных переменных — продолжительностью работы программы.
- Опциональное имя. Например, имя переменной.

Указатель — переменная, которая хранит адрес объекта в памяти программы.

Указатели объявляются как обычные переменные, только с символом `*` после типа. Так выглядит указатель, способный хранить адрес объекта типа `int`:

```
int* p;
```

Переменная `p` может хранить адрес целого числа. Переменная-указатель `p` не инициализирована, и использовать её для доступа к объекту нельзя. Объявление указателя выделяет память для хранения адреса на платформе, но не инициализирует эту область памяти.

Размер указателей равен размеру адреса на конкретной платформе и не зависит от размера самих объектов.

Чтобы использовать указатель, нужно присвоить ему адрес существующего объекта. Для этого есть унарный оператор `&`:

```
int value = 42;
int* value_ptr;  // Указатель value_ptr ещё не инициализирован
value_ptr = &value;
// Теперь в value_ptr хранится адрес переменной value
```

Указателю можно присвоить только адрес объекта совместимого типа. Присвоить адрес переменной типа `double` указателю на тип `int` нельзя:

```
int int_value = 42;
double double_value = 1.2345;

int* ptr;
// Следующая строка не скомпилируется,
// так как по адресу &double_value располагается объект типа double
ptr = &double_value; // error: cannot convert 'double*' to 'int*' in assignment
```

Объявление указателя лучше объединить с его инициализацией — и запись короче, и неинициализированных указателей в программе не будет:

```
int value = 42;
int* value_ptr = &value;
```

Оператор взятия адреса можно применять не только к отдельным переменным, но и к полям структур и классов:

```
#include <string>

using namespace std;

struct Point {
```

```
    double x;
    double y;
};

int main() {
    Point p;
    // y_ptr хранит адрес координаты Y точки p
    double* y_ptr = &p.y;
}
```

В C++ ссылки — не объекты. Они вводят новое имя для доступа к существующему объекту. Поэтому оператор `&`, применённый к ссылке, возвращает не указатель на ссылку, а указатель на сам объект:

```
int main() {
    int answer = 42;
    int& answer_ref = answer;

    // answer_ptr хранит адрес переменной answer
    int* answer_ptr = &answer_ref;
}
```

Оператор `<<` может вывести в поток значение указателя:

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    int value = 42;
    int* value_ptr = &value;
    cout << "value_ptr:" << value_ptr << endl;
}
```

Формат вывода адреса зависит от компилятора и платформы.

Неинициализированный указатель содержит неопределённое значение. Использовать такой указатель для доступа к объекту нельзя, это вызовет неопределённое поведение.

Инициализируйте указатель при его объявлении. Присвойте ему адрес существующего объекта совместимого типа или специальное значение `nullptr` — нулевой указатель.

Нулевой указатель хранит значение `nullptr`. C++ гарантирует, что по адресу `nullptr` не будет размещаться ни один объект программы.

Чтобы получить доступ к объекту в C++, используют унарную операцию **разыменования указателя**. Она обозначается символом `*`. Если её применить к указателю, она вернёт ссылку на объект, адрес которого хранит указатель.

Вот как указатели дают доступ к переменной:

```
#include <cassert>

using namespace std;

int main() {
    int value = 1;

    // Указатель value_ptr1, ссылающийся на переменную value
    int* value_ptr1 = &value;
    // Указатель value_ptr2, копия указателя value_ptr1, также ссылается на переменную value
    int* value_ptr2 = value_ptr1;

    // Значения указателей на один и тот же объект будут равны
    assert(value_ptr1 == value_ptr2);

    // Значение объекта value, полученное напрямую и через указатель на него, будет одно и то же
    assert(*value_ptr1 == value && *value_ptr2 == value);

    // Изменение value видно через указатели на него
    value = 2;
    assert(*value_ptr1 == value && *value_ptr2 == value);

    // Изменяем значение value через указатель
    *value_ptr2 = 3;

    // Ожидаемо изменённое значение будет видно как при прямом доступе к объекту по его имени,
    // так и при косвенном обращении через указатель value_ptr1
    assert(*value_ptr1 == value && *value_ptr2 == value);
}
```

Операцию разыменования `*` и операцию доступа к полям и методов `->` можно применять только к указателям, которые хранят адрес существующего объекта в памяти. Использовать их с неинициализированным или нулевым указателем нельзя — это приведёт к неопределённому поведению.

Прежде чем применять указатель, который может потенциально иметь нулевое значение, сделайте проверку на равенство `nullptr`:

```
#include <cassert>
#include <iostream>

int main() {
    using namespace std;

    int* p = nullptr;

    //------
    int value = 0;
    cin >> value;
    if (value >= 0) {
        p = &value;
    }
    //------

    if (p != nullptr) {
        // Использовать p можно
        cout << *p << endl;
    }
}
```

Указатели и константность

Операция `&` возвращает указатель на константный объект — его ещё называют **указателем на константу**. Такой указатель разрешает читать значение объекта, но не модифицировать его:

```
#include <cassert>

int main() {
    const int value = 42;

    // Ошибка: неконстантная ссылка не может ссылаться на константный объект
    // int& value_ref = value;

    // А вот так можно
    const int& const_value_ref = value;

    // Ошибка: указатель на неконстантное значение не может хранить адрес константного объекта
    // int* value_ptr = &value;

    // Указатель на константу типа int.
    const int* const_value_ptr = &value;
    // можно также объявить как int const* - это одно и то же

    // Указатель на константу можно использовать только для чтения значения объекта
    assert(*const_value_ptr == 42);
    // Выполнить модификацию объекта с его помощью нельзя.
    // Следующая строка не скомпилируется:
```

```
    // *const_value_ptr = 43;
}
```

Здесь компилятор не разрешает задать указателю типа `int*` значение адреса константного объекта. Такой указатель позволил бы изменить состояние объекта. В этом плане указатели на константу похожи на константные ссылки.

Изменение значения указателя

В отличие от ссылок, указатели могут в процессе жизни менять своё значение, храня в разные моменты времени адреса разных объектов. Простейший способ изменить значение указателя — присвоить ему адрес другого объекта:

```
#include <cassert>
#include <iostream>
#include <string>

using namespace std;

int main() {
    int value = 1;

    // Сначала value_ptr ссылается на value
    int* value_ptr = &value;

    cout << "&value: "s << &value << endl;
    cout << "value_ptr: "s << value_ptr << endl;
    assert(*value_ptr == 1);

    int another_value = 2;
    // Затем ссылается на another_value
    value_ptr = &another_value;

    cout << "&another_value: "s << &another_value << endl;
    cout << "value_ptr: "s << value_ptr << endl;
    assert(*value_ptr == 2);
}
```

Значение константного указателя нельзя изменить после инициализации. Чтобы объявить такой указатель, поставьте `const` справа от знака `*`. Как и обычная константа, константный указатель должен быть инициализирован при объявлении:

```
int value = 42;
int* const const_ptr_to_value = &value;

int another_value = 5;
```

```
// Ошибка: нельзя изменить значение константного указателя
// const_ptr_to_value = &another_value;
```

Константные указатели на константу

Константными могут быть и сам указатель, и данные, на которые он ссылается:

```
int value = 42;
const int* const const_ptr_to_const_value = &value;

int another_value = 5;
// Ошибка: нельзя изменить значение константного указателя:
// const_ptr_to_const_value = &another_value;

// Ошибка: нельзя изменить значение данных через указатель:
// *const_ptr_to_const_value = 0;
```

Определение типа указателя

Чтобы запомнить, к чему относится `const` в типе указателя, прочитайте объявление указателя справа налево, заменяя символ `*` на слово «указатель»:

```
// p1 - это указатель на данные типа int
int* p1;

// p2 - это указатель на данные типа const int
const int* p2;

int data = 42;

// p3 - это константный указатель на данные типа int
int* const p3 = &data;

// p4 - это константный указатель на данные типа const int
const int* const p4 = &data;
```

Статическое и автоматическое размещение объектов в памяти

Память для программ на C++ можно разделить на несколько независимых областей:

- для объектов со статическим временем жизни;
- для объектов с автоматическим временем жизни;
- для объектов с динамическим временем жизни;
- для объектов в локальной памяти потока.

Для хранения локальных переменных при каждом вызове функции выделяется кадр стека фиксированного размера. Объекты со статическим размещением существуют в программе в единственном экземпляре.

Динамическое размещение памяти позволяет программе создавать столько объектов, сколько нужно для решения задачи, а потом удалять эти объекты, когда задача решена.

Динамическое размещение объектов в памяти

Часто до запуска программы неизвестно, какой объём памяти ей понадобится.

Стандартные контейнеры, такие как `vector`, `map`, `set`, хранят свои элементы в **куче** — области, предназначенной для динамического выделения и освобождения памяти по запросу программы.

Чтобы создать объект в куче, используют оператор `new`.

Удалить ставший ненужным объект можно оператором `delete`. Он принимает указатель на объект, который создан оператором `new`, удаляет объект, вызывая его деструктор, и освобождает занимаемую объектом память обратно в кучу:

```
int main() {
    // В куче создаётся объект типа int. Адрес этого объекта сохраняется в value_ptr.
    int* value_ptr = new int;

    // Используем созданный объект по указателю на него
    *value_ptr = 42;

    // Удаляем объект и возвращаем занимаемый им блок памяти обратно в кучу
    delete value_ptr;
}
```

В оператор `delete` можно безопасно передавать указатель, равный `nullptr`.

Оператор `new` позволяет передать параметры конструктору создаваемого объекта.

Когда в куче нет свободного места для создания объекта, оператор `new` выбрасывает исключение `std::bad_alloc`.

Память ограничена, поэтому нужно удалять созданные в куче объекты, когда в них нет необходимости.

Утечки памяти вызывают завершение программы. Они возникают, если:

- программист не удалил объект после использования;

- указателю, ссылающемуся на объект в куче, присвоили новое значение, и других ссылок на объект нет. Объект становится недоступным из кода, хотя продолжает занимать память.

```
#include <iostream>
#include <new>
#include <string>

using namespace std;

int main() {
    size_t n = 0;
    try {
        for (; n != 300; ++n) {
            string* p = new string(100'000'000, ' ');
            // Утечка памяти исправлена - объект в куче своевременно удаляется
            delete p;
        }
        cout << "Program completed successfully"s;
    } catch (const bad_alloc&) {
        // Сюда программа, скорее всего, не попадёт,
        // если объём свободной памяти в куче будет больше 100 мегабайт
        cout << "bad_alloc after "s << n << " allocations"s << endl;
    }
}
```

Копирование объектов

Копирующий конструктор инициализирует экземпляр класса, копируя уже имеющийся. Этот конструктор принимает константную ссылку на копируемый объект. Всякий раз, когда в программе явно или неявно создаётся копия существующего объекта, используется копирующий конструктор.

Компилятор генерирует конструктор копирования, а он в свою очередь создаёт копию объекта, копируя все поля оригинала. Это хорошо работает для классов и структур, хранящих объекты-значения, но плохо для копирования классов, которые содержат данные в динамической памяти.

Когда сгенерированный компилятором конструктор копирования не подходит, вы можете написать его самостоятельно.

Умные указатели — классы, которые благодаря перегрузке операций ведут себя как указатели. Они предоставляют операцию разыменования `*` и доступа к членам класса `->`.

В отличие от обычных указателей, которые ещё называют «сырыми», умные указатели реализуют семантику владения объектом.

Ещё умные указатели определяют, что должно происходить при копировании указателя.

Как правило, умные указатели делаются шаблонными, чтобы использовать с объектами разных типов.

В обычной работе вам вряд ли понадобится писать собственные умные указатели и вручную вызывать `new` / `delete`. Скорее всего, вы будете использовать стандартные.

Присваивание объектов

Присваивание — специальная операция C++. Для пользовательских типов компилятор может реализовать её автоматически.

Сгенерированный компилятором оператор `=` присваивает значения полей одного объекта соответствующим полям другого.

Но когда объект владеет другими объектами в динамической памяти, реализация усложняется.

Чтобы решить проблему, переопределите оператор присваивания и реализуйте в нём присваивание значений объектов, а не указателей.

В C++ оператор присваивания можно переопределить только внутри класса в виде метода с именем `operator=`. Левый аргумент операции присваивания — это текущий экземпляр класса, а правый аргумент передаётся через единственный аргумент оператора.

Тип правого аргумента операции может быть любым, а самих операций присваивания может быть определено несколько. Обычно оператор присваивания возвращает ссылку на свой левый аргумент.

Если класс или структура объявляют один из следующих методов, скорее всего, они должны объявить все три:

- деструктор,
- конструктор копирования,
- оператор присваивания.

Строгая гарантия безопасности исключений значит, что операция либо завершается успешно, либо происходит выбрасывание исключения и состояние объекта останется прежним. Такая семантика выполнения также называется “commit or rollback”.

Обеспечить строгую гарантию безопасности исключений в пользовательском операторе присваивания позволяет идиома “copy-and-swap”. В ней оператор присваивания переиспользует функционал конструктора копирования.

```
class Object {
public:
    // Копирующий конструктор
    Object(const Object& other);

    // Копирующий оператор присваивания
    Object& operator=(const Object& rhs) {
        if (this != &rhs) {
            // Реализация операции присваивания с помощью идиомы Copy-and-swap
            // Если исключение будет выброшено, то на текущий объект оно не повлияет
            auto rhs_copy(rhs);

            // rhs_copy содержит копию правого аргумента
            // Обмениваемся с ним данными
            swap(rhs_copy);
            // теперь текущий объект содержит копию правого аргумента,
            // а rhs_copy - прежнее состояние текущего объекта, которое при выходе
            // из блока будет разрушено
        }

        return *this;
    }

    // Обменивает состояние текущего объекта с other без выбрасывания исключений
    void swap(Object& other) noexcept;

    ~Object();
};
```