Указатели

Напоминание: модель памяти

- Память в C++ представляет собой нумерованную последовательность байт.
- Номер байта в памяти называется его *адресом*.
- Существует несколько областей памяти: *глобальная (статическая), автоматическая (стековая), динамическая, потоковая*.
- *Динамическую память* обсудим в этой лекции.
- *Потоковую память* не обсуждали и не будем.

Операция взятия адреса

Как узнать адрес, по которому лежит объект в памяти? - Унарная операция `&`.

```
int x;
std::cout << &x << '\n';  // отображается в 16-ричном виде
```

Операцию можно применять только к *lvalue* значениям. Более того, **возможность взятия адреса можно использовать в качестве критерия lvalue**:

```
// так можно
&x, &(x = 8), &(++x);

// а так нельзя
&5, &(x + 8), &(x++);
```

Операция взятия адреса

Адреса в C++ имеют специальный тип - *указатель на тип* объекта, у которого был взят адрес.

```
int x;
const float y = 0;

&x;  // указатель на int [int*]
&y;  // указатель на const float [const float*]
```

Указатели

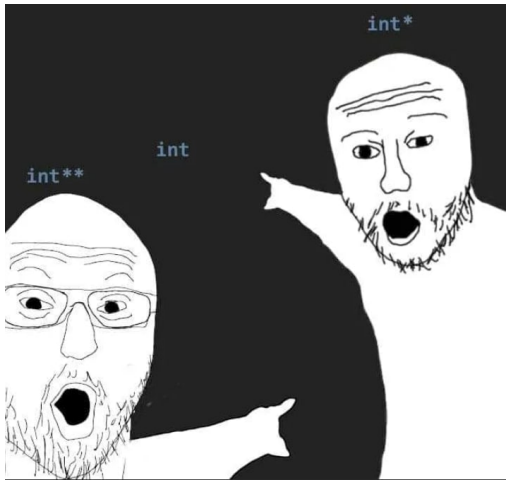
Указатели

Указатель - тип данных позволяющий хранить адрес другого объекта в памяти.

Пусть `T` - некоторый тип, тогда `T*` - указатель на `T`.

Таким образом, указатели - это целое семейство типов.

```
int*           // указатель на int
float*         // указатель на float
long long*     // указатель на long long
const char**   // указатель на указатель на константный char
double***      // указатель на указатель на указатель на double
```



Указатели: примеры

```
int x = 0;
const float y = 0;
```

```
int* px  = &x;

const float* cpy = &y;
const int*   cpx = &x;

int** ppx         = &px;
const int** cppx = &cpx;
```

Указатели учитывают константность!

```
float* py = &y;  // CE: py нарушает константность y
```

Указатели: разыменование

```
int x = 1;
float y = 2.5;

int* px = &x;
int* py = &y;
```

По указателю можно получить значение объекта и даже изменять его.

Для этого воспользуемся операцией *разыменования* (`*`):

```
*px = 11;
*py = 3.5;

std::cout << *px << ' ' << *py;  // 11 3.5
std::cout << x << ' ' << y;      // 11 3.5
```

Указатели: разыменование

```
*px = 11;
*py = 3.5;

std::cout << *px << ' ' << *py;
```

Формально: операция разыменования - унарная операция, применяемая к указателям и возвращающая *lvalue* низлежащего типа.

Арифметика указателей

Как известно, адреса в C++ представляют собой целые числа.

Означает ли это, что к ним применима целочисленная арифметика?

Арифметика указателей

- Указатели одинакового низлежащего типа можно вычитать друг из друга. Результат - количество элементов данного типа, которое поместится в этом промежутке (`разница в байтах / sizeof(T)`).

```
int x, y, z;
std::cout << &z - &x << '\n';  // скорее всего, 2
```

- К указателям можно прибавлять целые числа. Результат - указатель сдвинутый на `n` шагов размера `sizeof(T)` .

```
int32_t* p = ...;
p + 5;  // + 20 байт
p - 4;  // - 16 байт
p += 10;
p -= 12;
++p;
p--;
```

Арифметика указателей

- Указатели одинакового низлежащего типа можно вычитать друг из друга. Результат - количество элементов данного типа, которое поместится в этом промежутке (`разница в байтах / sizeof(T)`).
- К указателям можно прибавлять целые числа. Результат - указатель сдвинутый на `n` шагов размера `sizeof(T)` .
- Остальные арифметические операции недопустимы.

Отметим, что разность указателей на переменные или сдвиг указателя на переменную, с точки зрения стандарта не дает разумного результата.

Разыменование подобных указателей приводит к *UB*.

Для чего *на самом деле* нужна арифметика указателей, узнаем, когда будем говорить о массивах.

Указатели и `const`

Указатели и `const`

Указатели могут быть константными с двух точек зрения.

1. *Указатель на константу*. Константным является объект, на который указывают:

```
int x = 0;
const int y = 0;

const int* px = &x;  // либо int const*
const int* py = &y;  // либо int const*

std::cin >> *px;   // CE: объект считается неизменяемым
std::cout << *py;  // OK: объект доступен для чтения
```

Указатели и `const`

Указатели могут быть константными с двух точек зрения.

2. *Константный указатель*. Константным является сам указатель:

```
int x;
int y;

int* const px = &x;

std::cin >> *px;   // OK
std::cout << *py;  // OK

px = &y;  // CE: указатель неизменяемый
++px;     // CE
```

Указатели и `const`

Константности можно комбинировать:

```
const int* px = ...;  // указатель на константу
int const* py = ...;  // указатель на константу
int* const pz = ...;  // константный указатель
const int* const pa = ....;  // константный указатель на константу
int const* const pa = ...;   // константный указатель на константу
```

Лайфхак: читайте типы указателей справа налево.

`int const* const - константный указатель на константу int`

Указатели на указатели

Указатель - это тип, а значит на него тоже можно создать указатель.

```
float x;
float* px = &x;
float** ppx = &px;
float*** pppx = &ppx;
```

Указатели на указатели тоже дружат с `const`:

```
const float* const** const p = ...;
// константный указатель на указатель на константный указатель на константу float
```

Нулевой указатель

Попытка чтения неинициализированной переменной приводит к *UB*. То же самое касается и указателей.

```
int* p;
std::cout << p << ' ' << *p;  // UB
```

При этом указателям в общем случае нельзя присвоить конкретного начального числового значения:

```
int* p = &x;  // Ok
int* q = p;   // Ok
int* r = 10;  // CE: преобразование из int в int* завпрещено
```

Нулевой указатель

Возникают следующие вопросы:

```
int* p = &x;  // Ok
int* q = p;   // Ok
int* r = 10;  // CE: преобразование из int в int* завпрещено
```

1. Как задать начальное значение "пустого" указателя?
2. Как отличить корректный указатель от некорректного (который не указывает в "разумную область")?

Нулевой указатель

В языке C++ есть исключение. Любому указателю можно присвоить значение 0, которое соответствует нулевому адресу.

Напоминание: по стандарту по нулевому адресу не может находиться ничего, поэтому разыменование подобного указателя приводит к *UB*.

```
int* p = 0;
const float* pp = 0;
```

К сожалению, подобная возможность иногда приводит к проблемам, так как возникает ассиметрия: целые числа присваивать нельзя, а 0, внезапно, можно.

Нулевой указатель

Правильным с точки зрения современного C++ способом присвоить нулевое значение указателю является литерал `nullptr`.

```
int* p = nullptr;
const float* pp = nullptr;
```

`nullptr` значение специального типа (`std::nullptr_t`), которое неявно приводится к нулевому указателю любого типа.

Правило: для обозначения нулевых указателей используйте только `nullptr`.

Указатель на `void`

Напоминание: `void` - специальный тип обозначающий отсутствие объекта.

Что если хочется сохранить адрес ячейки памяти, без учета типа объекта, который там может лежать?

Для этого можно воспользоваться `void*`:

```
int x;
void* p = &x;

std::cout << p;   // Ok
std::cin >> *p;   // CE: не ясно, что записывать
std::cout << *p;  // CE: не ясно, что читать
```

"Висячий" указатель

Какие чувства вызывает у вас этот код?

```
int main() {
  int* p = nullptr;
  {
    int x = 0;
    p = &x;
  }
  return *p;
}
```

"Висячий" указатель

```
int main() {
  int* p = nullptr;
  {
    int x = 0;
    p = &x;
  }
  return *p;
}
```

Область действия объекта заканчивается раньше, чем область действия указателя, указывающего на него.

Чтение или запись данных по такому указателю приводит к *UB*!

Важно: не допускайте провисания указателей!

Динамическое выделение памяти

Статус

До сих пор мы лишь заводили переменные в глобальной области и на стеке.

У хранения данных в глобальной области и на стеке есть ряд ограничений:

1. Они имеют строго фиксированное время жизни. То есть глобальные переменные будут жить до конца программы, а локальные - до конца текущего блока.
2. Как правило, эти области ограничены несколькими мегабайтами. Однако реальные программы часто требуют гораздо больше.

Выражение `new`

Выражения вида `new T` / `new T(...)` / `new T{...}` создают в динамической области объект типа `T` и возвращают указатель на него.

```
int* pa = new int;       // без инициализации
int* pb = new int(11);   // инициализация числом 11
int* pc = new int{13};   // инициализация числом 13
```

Полученный указатель используется для доступа к объекту.

Выражение `new`

Объекты в динамической области живут с момента вызова `new` и до окончания работы программы.

Таким образом, можно создавать объекты внутри блоков, которые будут доступны и вне него:

```
int* p = nullptr;
{
  int x = 11;
  p = new int(13);
}
std::cout << x;   // CE: x уже уничтожен
std::cout << *p;  // Ok
```

Но это ведет к тому, что память постепенно "засоряется".

Выражение `delete`

Память в динамической области можно (и нужно) очистить до завершения программы, как только она перестала быть нужной.

`delete <указатель>` - удаляет объект, возвращает память системе.

```
int* p = new int;
// ...
delete p;
```

Замечание 1: удалять можно только ту память, которая была выделена с помощью `new`. В остальных ситуациях `delete` приводит к *UB*.

Замечание 2: удаление нулевого указателя - корректная операция.

Утечка памяти

При некорректной работе с указателями или несвоевременном очищении динамической памяти может возникать *утечка памяти* - неконтролируемый рост лишней памяти.

```
int* p = new int;
p = nullptr;  // потеряли указаль на динамическую память - утечка

new float;  // выделили память, указатель не сохранили

p = new int;  // забыли удалить после использования - утечка
```

Аккуратное использование `new` и указателей и своевременное использование `delete` - залог успешной борьбы с утечками.

Иные методы выделения динамической памяти

В языке C основным способом выделения динамической памяти была функция `malloc`.

Ключевые отличия:

1. Возвращает указатель `void*`.
2. Требует явного указания количества байт.
3. Не инициализирует память.

```
void* v = std::malloc(11);
int* p = static_cast<int*>(std::malloc(sizeof(int)));
*p = 0;
```

Для очищения используется функция `free`:

```
std::free(v);
std::free(p);
```

Иные методы выделения динамической памяти

В C++ есть аналог функций `malloc` / `free` - функции `operator new` / `operator delete`:

```
void* v = operator new(11);
int* p = static_cast<int*>(operator new(sizeof(int)));
*p = 0;
```

```
operator delete(v);
operator delete(p);
```

Они используются крайне редко и в специфических ситуациях.

Основной способ - выражения `new` / `delete`.

Факт: `new` и `delete` внутри себя вызывают эти функции.