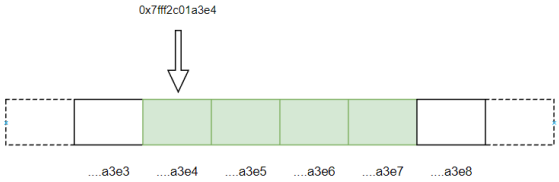
Указатели в языке C++

Экзаменационные вопросы → Указате...

Что такое указатель

- Указатель - это переменная которая хранит адрес
- Адрес можно представить, как номер байта в оперативной памяти.
  Как правило адреса записывают в шестнадцатеричном виде для компактности
  - int i;
    std::cout << &i;
    - Вывод:

      0x7fff2c01a3e4
- Оперативная память в C++ - это линейная последовательность байт (массив)
  -



- Получить адрес конкретного бита в байте нельзя

Объявление

- Переменная-указатель объявляется как:
  Тип_данных * Имя_указателя;
  - void *i;
    int *j;
    string *k;
    vector<string> *m;
- Количество пробелов слева и справа от звёздочки значение не имеет
  - int* i;
    int * i;
    int *i;
- При объявлении нескольких переменных-указателей одновременно, звёздочка должна находится у каждой, иначе переменная воспринимается как переменная-значение
  - int *a, b, c; // a - указатель, b и c - обычные int-ы
    int *i, j, *k; // i и k - указатели

Указатели и константность

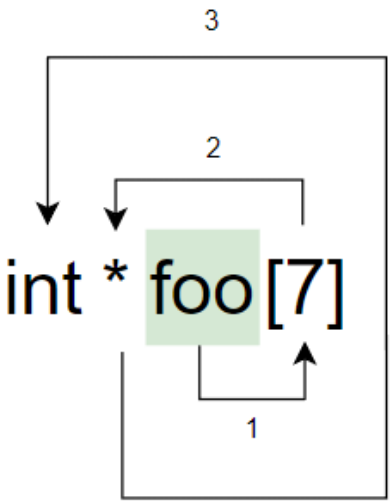
- Данные, адрес которых хранит указатель могут быть как константой так и не константой, поэтому указатель на константу и на обычную переменную обьявляются по разному
  - const int one = 1;
    int* i = &one;
    - Ошибка, т.к. i не гарантирует константности данных и через неё можно можно было бы изменить one, что нарушает условие константности
- Чтобы создать указатель на константу, нужно к типу переменной на которую указываем тоже добавить const
  - const int one = 1;
    int two = 2;

    const int *i = &one;
    const int *j = &two;
    - Оба варианта допустимы, но через указатель j мы не сможем изменить two, хотя сама переменная не константа
- С другой стороны указатель - это переменная, поэтому к нему тоже можно применить модификатор const, в результате чего после инициализации изменить хранимый указателем адрес уже будет нельзя
  - int one;
    int two;
    int* const i = &one;
    i = &two;
    - Ошибка в последней строке, т.к. i - константа
  - const one = 1;
    const int const *i = &one;
    - Здесь константа и сам указатель и данные на которые он указывает

Декларация типов С

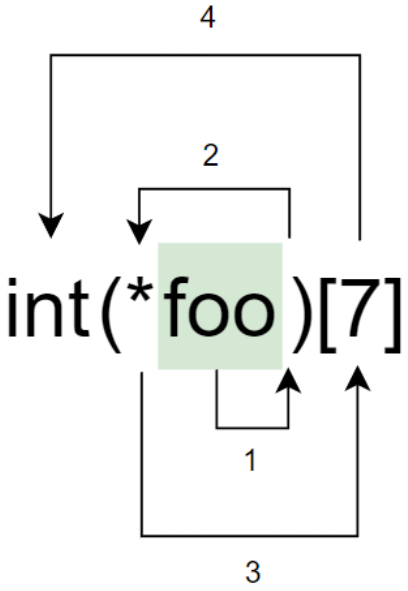
- Когда появляются указатели типы становится очень сложно читать.
  Чтобы научится это делать, нужно запомнить специальное правило:
  Направо, налево, наружу, повторить

- Это значит, что когда вы читаете тип переменной, то читаете его как бы по спирали начиная от имени:
  0) найти имя переменной;
  1) Посмотреть что стоит сразу справа от имени и прочитать 1 элемент иначе:
  - Если ничего, то переходим на (2)
  - Если скобочка, то переходим на (2)
  - Если квадратные скобки, то читаем все стоящие рядом
  2) Посмотреть что стоит сразу слева от имени и прочитать 1 элемент иначе.
  3) Выйти на следующий круг и начать с пункта (1)



- foo - это (1) массив из 7 (2) указателей на (3) int

```
int a0, a1, a2, a3, a4, a5, a6;
int* foo[7];
foo[0] = &a0;  // Указатель на int
foo[1] = &a1;   // Указатель на int
....
foo[6] = &a6;   // Указатель на int
```



- foo - это (1-2) указатель на (3) массив из 7 (4) int-ов

  ```
  int array[7];  // Массив
  int(*foo)[7] = &array;  // Указатель на массив
  ```

- Здесь на шаге (1) справа от имени скобочки и через них мы пройти не можем пока не прочитали всё что в них есть, после чего мы выходим из скобочек наружу и читаем дальше

- int(*foo[2])[3][3];

  - foo - это массив из 2 указателей на двумерный массив 3x3 с элементами типа int

    ```
    int mat1[3][3];  // Матрица
    int mat2[3][3];  // Матрица

    int(*foo[2])[3][3];
    foo[0] = {&mat1};
    foo[1] = {&mat2};
    ```

Инициализация

- Указатель - это обычная переменная фундаментального типа, поэтому НЕ инициализированный указатель содержит мусорный адрес

- Инициализировать указатель можно либо адресом существующей переменной либо специальным значением (nullptr или NULL).
  Специальное значение применяется как признак того, что указатель не указывает на что-то определённое (пустой указатель)

- int one; // Обычная переменная-значение
  int* i = &one; // Берём адрес one и сохраняем его
  int* j = nullptr; // Спец. значение, говорит о том, что указателю не присвоен конкретный адрес
  int* k = NULL; // Тоже что и выше, но в стиле C
- int* i = new int[10]; // Пытаемся выделить блок памяти
  if (i == nullptr) cout << "Ошибка, память не выделена";

Способы получения адреса

- Адрес можно получить у любого объекта который имеет своё постоянное место в памяти.
  У временных объектов получить адрес нельзя
  - int two = 2; // У two адрес есть
    2; // Адреса нет
    two + 2; // У результата адреса нет

    void foo(int x){} // У foo адрес есть
- Для получения адреса существует оператор взятия адреса - & (амперсанд). Чтобы получить адрес он располагается слева (перед) от переменной адрес которой хотим получить
  - int one;
    &one; // Адрес переменной one

    int* i;
    &i; // Адрес переменной i

    void foo(int x){}
    &foo; // Адрес функции
    foo; // Имя функции автоматически преобразуется в её адрес
- Тип указателя определяется как имя типа переменной, адрес которой берём, и звёздочка после него
  - int one;
    auto i = &one; // Тип i - int*

    auto j = &i; // Тип j - int**

Виды указателей

- Сырой
  - Хранит адрес чего угодно, не обладает никакой дополнительной информацией о том, на что указывает

- void* i; // Не инициализированный
  void* j = nullptr; // Спец. значение, говорит о том, что указателю не присвоен конкретный адрес

  int one; // Просто переменная целого типа
  double two; // Просто переменная вещественного типа
  void* k = &one; // Берём адрес и сохраняем его в k
  void* x = &two; // Берём адрес и сохраняем его в x
- Не позволяет получить доступ к памяти на которую указывает, т.к. не обладает информацией от типе данных на которые указывает
- В C++ применяется для совместимости с функциями C.
  В C применяется как аргумент или возвращаемое значение функций работающих напрямую с памятью, например: malloc, memcpy, free и др.
  - void* mem = malloc(4 * sizeof(int)); // Получили 16 байт памяти
    int* array = (int*) mem; // Преобразуем сырую память в типизированную
    array[0] = 1; // Используем
    free(array); // Освобождаем
- Типизированный
  - Хранит адрес только переменной определённого типа. Знает тип данных на которые указывает
    - int one;
      int* i = &one; // Получаем адрес и сохраняем в переменную i

      double two;
      int* j = &two; // Ошибка. Не правильный тип данных

      int* k = nullptr; // Указатель не указывает на переменную (пустой)
  - Позволяет получить доступ к памяти на которую указывает, при этом работа с памятью происходит по правилам типа указателя
    Для доступа к памяти используется оператор разыменования (звёздочка перед именем указателя)
    - int one = 1;
      int* i = &one;

      *i += 1;
      cout << one;
      - Ответ:
        2

- string one = 1;
  string *i = &one;

  *i += 1;
  cout << one;
  - Ответ:
    Ошибка, т.к. тип string не поддерживает операцию сложения с числами

Преобразование типа указателя

- Для всех типов кроме Классов
  - Преобразование указателя типа A к указателю типа B - это НЕ безопасная операции, поэтому неявно она не выполняется.
    Т.е. программист явно должен прописать преобразование в коде, чтобы оно выполнилось
    - int* i;
      double* j;

      1) i = j;
      2) j = i;
      3) i = (int*)j;
      - Ответ:
        1) Ошибка, т.к. указатели разных типов
        2) Ошибка, т.к. указатели разных типов
        3) Нормально. Волья программиста - закон!!!
  - Для сырых указателей сделано исключение, т.к. это их основная задача - хранить адреса чего угодно
    - void* i;
      int* j;

      1) i = j;
      2) j = i;
      3) j = (int*) i;
      - Ответ:

        1) Всё нормально
        2) Ошибка, j принимает только адреса int
        3) Нормально. Программист так решил!!!
  - Преобразование типа указателя можно сделать двумя способами:
    - в стиле C;
    - в стиле C++

- void* i;
  int* j;

  j = (int*) i; // В стиле C
  j = static_cast<int*>(i); // В стиле C++

- В процессе преобразования типа указателя, значение которое он хранит не изменяется, т.е. он продолжает хранить адрес той же ячейки в памяти что и раньше.
  Разница будет после применения операции разыменования

  - int one = 1;
    int* i = &one;

    cout << i << endl;
    cout << (double*) i;

    - Вывод:

      0x7ffe5903203c
      0x7ffe5903203c

  - int one = 1;
    int* i = &one;

    cout << *i << endl;
    cout << *(double*) i;

    - Вывод:

      1
      9.61056e+128

- Для Классов
  - Преобразование типа для классов происходит по специальным правилам, поэтому описано далее в отдельном разделе

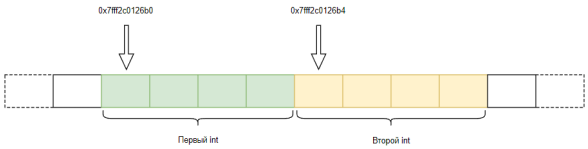
Логические операции с указателями

- Логические операции работают с указателями, так же как и с числами, то есть их можно сравнивать на >, <, ==, != и т.д.
  Но как правило используют == и !=

Арифметика указателей

- С адресами арифметические операторы работают НЕ так как с числами
- Операторы * и / НЕ работают с указателями
- Операторы ++ и --

- Увеличивают или уменьшают адрес хранимый в указателе:
  - в байтах - на размер типа указателя;
  - в элементах - на единицу;
  - int one = 1;
    int* i = &one;
    cout << sizeof(int) << endl;
    cout << i << endl;
    ++i;
    cout << i;
    - Вывод:

      4
      ....26b0
      ....26b4
  - 

- Операторы +, -, +=, -=
  - Случай сложения/ вычитания адреса и целого числа
    - Сложение адреса и целого числа N смещает адрес вперёд на:
      - N * sizeof(тип) байт;
      - N элементов вперёд.
      Вычитание, соответственно, смещает адрес назад
      - double one = 1;
        double* i = &one;
        cout << sizeof(double) << endl;
        cout << i << endl;
        i += 10;
        cout << i;
        - Вывод:

          8
          ....7f98
          ....7fe8
  - Случай вычитания адреса из адреса. Сложение адресов НЕ работает
    - Вычитание адреса из адреса возвращает НЕ количество байт, а количество элементов типа указателя которое помещается между двумя адресами, т.е. разницу в байтах делённую на размер типа в вайтах

- double one = 1;
  double* i = &one;
  double* j = i+1;
  cout << sizeof(double) << ' ' << j - i << endl;
  cout << sizeof(int) << ' ' << (int*)j - (int*)i << endl;
  cout << sizeof(char) << ' ' << (char*)j - (char*)i << endl;
  - Вывод:

    8 1
    4 2
    1 8

Доступ к памяти на которую указывает переменная-указатель

- Для доступа к памяти через указатель применяется оператор разыменования *. Оператор располагается слева от имени указателя (перед ним)
  - int one = 10;
    int* i = &one;
    *i = 20; // Получаем доступ к памяти
    cout << one;
    - Ответ:
      20
  - int one = 10;
    int two;
    int* i = &one;
    int* j = &two;

    *j = *i + 40; // Получаем доступ к памяти
    cout << two;
    - Ответ:
      50
- Внимание!!! Не путайте звёздочку стоящую в объявлении указателя и оператор разыменования - это разные вещи, они просто внешне похожи
  - Эти звёздочки - разные штуки:

    int * i; // Здесь * - часть типа переменной. Тип i - int*
    *i; // Здесь * - оператор разыменования. Т.е. приказ выполнить действие

    int one = 2 * 2; // Здесь тоже есть звёздочка, но это оператор умножения.

- Ещё один вариант, как можно получить доступ к памяти через указатель - это синтаксис массивов, т.е. использование оператора [ ] вместе с указателем.
  С точки зрения C++ выражение:
  указатель + число
  эквивалентно выражению:
  указатель[число]
  - int array[3] = {10, 20, 30};
    int* i = &array[0];

    cout << *i << ' ' << *(i + 1) << ' ' << *(i + 2) << endl;
    cout << i[0] << ' ' << i[1] << ' ' << i[2] << endl;

    i = &array[2];
    cout << i[-2] << ' ' << i[-1] << ' ' << i[0] << endl;
    - Вывод:

      10 20 30
      10 20 30
      10 20 30
- Синтаксис массива позволяет поменять указатель местами, то есть с точки зрения C++ выражение:
  число + указатель
  эквивалентно выражению:
  число[указатель]
  - int array[3] = {10, 20, 30};
    int* i = &array[0];

    cout << *i << ' ' << *(i + 1) << ' ' << *(i + 2) << endl;
    cout << 0[i] << ' ' << 1[i] << ' ' << 2[i] << endl;
    - Вывод:

      10 20 30
      10 20 30

Связь с массивами

- Массив и указатель это разные вещи!!!
- По правилам C и C++ первое измерение массива может "деградировать" (распадаться) до указателя (Array Decay).
  Т.е. осуществляется:
  1) неявное взятие адреса у массива;
  2) преобразование адреса из адреса массива в адрес одного элемента массива

- Decay - автоматическое преобразование к более простому типу

- int a[10];
  int b[20];

  int* i = a;
  int* j = b;

  - Здесь a и b - массивы int-ов.
    i и j - адреса нулевых элементов этих массивов

- int a[10][5];
  int b[8][4][2];

  int (*i)[5] = a;
  int (*j)[4][2] = b;

  - Здесь a - массив из 10 массивов по 5 int-ов
    i - адрес нулевого массива из 5 элементов

    b - массив из 8-ми матриц 4x2 типа int
    j - адрес нулевой матрицы 4x2 типа int

- На второе и последующие измерения это правило не распространяется

  - int a[10][5];

    int** i = a; // Ошибка

- Такие указатели применяются, в основном, для передачи массивов в функцию и внутри функции, благодаря использованию [ ], с указателями можно работать как с обычными массивами

  - ```
    int sum(int* arr,  int size){
      int accum = 0;
      for(int i=0; i<size; i++) accum += arr[i];
      return accum;
    }

    int a[4] = {1, 2, 3, 4};
    cout << sum(a, 4);
    ```

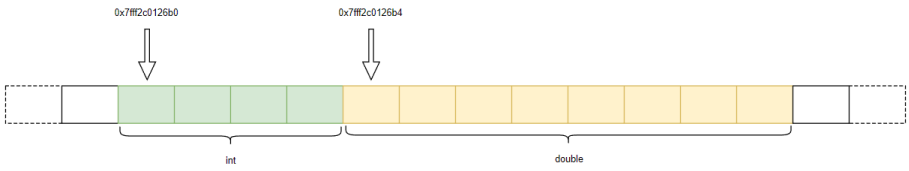
    - Вывод:

      10
    ```

- Все эти 3 объявления - идентичны:

  int sum(int* arr, int size);
  int sum(int arr[], int size);
  int sum(int arr[100], int size);

Размер указателя

- Все указатели занимают одно и тоже количество байт в памяти.
  В зависимости от разрядности системы: на x32 - 4 байта, на x64 - 8 байт
  - sizeof(int *) == sizeof(void *)
    sizeof(int *) == sizeof(double *)
    sizeof(int *) == sizeof(vector<int> *)
    - Ответ:
      true
      true
      true
- Переменная int и vector<int> разного размера, каким образом они помещается в указателе, если все указатели одинакового размера?
  - Указатель хранит информацию только об адресе первого байта, но не об адресе каждого байта переменной на которую указывает.
    Первый байт это точка где переменная начинается, т.к. указатель знает тип на который указывает, то он знает и размер этого типа. Зная точку начала и размер, можно определить где переменная заканчивается.
    - 

Указатель в ООП

- В отличие от других типов данных классы можно связать через наследование.
  Если 2 или больше классов связаны через наследование, то между ними возможно преобразование вниз или вверх по иерархии
  - int* i;
    double* j;

    1) i = j;
    2) j = i;
    - Ответ:
      1) Ошибка, т.к. указатели разных типов
      2) Ошибка, т.к. указатели разных типов

- Преобразование типа указателя вверх по иерархии классов - безопасно, с точки зрения работы с памятью, поэтому оно выполняется неявно.
  Преобразование вниз по иерархии может быть опасно, поэтому такое преобразование можно выполнить только явным образом (программист берёт ответственность на себя)
  - struct A{
    };

    struct B: A{
    };

    A* a;
    B* b;

    1) a = b;
    2) b = a;
    3) b = (B*) a;
    4) b = static_cast<B*>(a);
    - Ответ:
      1) Ошибок нет, т.к. преобразование вверх по иерархии (UpCast)
      2) Ошибка, преобразование вниз по иерархии (DownCast)
      3) Ошибок нет, т.к. используется явное преобразование в стиле C
      4) Ошибок нет, т.к. используется явное преобразование в стиле C++
  - Почему преобразование вверх по иерархии безопасно (с точки зрения памяти)?
    - Что подразумевается под безопасностью с точки зрения памяти?
      - Опасно это:
        - портить данные других переменных, т.к. приведёт к не предсказуемому поведение программы;
        - залезать в память не выделенную для программы. Приведёт к падению программы с ошибкой сегментации
    - Если класс C является наследником классов A и B, то в памяти занятой объектом класса C можно выделить фрагмент занимаемый полями A, фрагмент занимаемый полями B и добавочную часть - личные поля C.

      T.e. C как бы собран из полноценных объектов класса A и B и полей класса C

- struct A{
  int a1;
  int a2;
  };

  struct B{
  int b1;
  int b2;
  int b3;
  };

  struct C: A, B{
  int c;
  };

  C object;
  - Картинка
- Из за такой структуры размещения в памяти, в наследнике, всегда возможно найти точку начиная с которой размещён каждый из предков и установить на него указатель.
  Т.е. в процессе преобразования адреса потомка к адресу предка результирующий адрес может быть сдвинут вперёд
  - struct A{
    int a1;
    };

    struct B{
    int b1;
    int b2;
    };

    struct C: A, B{
    int c1;
    };

    C object;
    A* a = &object;
    B* b = &object;
    cout << &object << '\n' << a << '\n' << b;
    - Вывод:
     60
     60
     64

- Такой вывод объясняется тем, что "подобъект" B внутри C расположен после A и для корректной работы с ним нужно сдвинуть указатель начала на длину обьекта A вперёд

  a1 b1 b2 c1 // Размещение C в памяти
  ^
  B

  - Картинка

- Почему преобразование вниз по иерархии может быть опасно?
  - Как упоминалось выше, в процессе преобразования типа указателя с адресом могут происходить различные преобразования из-за которых мы можем не намеренно получить доступ к чужим данным.

```
struct A{
   int a1;
};

struct B{
   int b1;
   int b2;
};

struct C: A, B{
   int c1;
};

B object;
C* c = static_cast<C*>(&object);
cout << &object << '\n' << c << '\n' << &c->c1;
```

    - Вывод:
     904
     900
     912

    - Как видно, чтобы получить адрес объекта класса C адрес B был сдвинут влево на длину равную размеру первого предка.
      Теперь при попытка обратиться к полям a1 и c1 мы попадём в участки памяти реально нам не принадлежащие. Там могут быть например другие переменные или временно сохранённые значения регистров процессора или вообще всё что угодно

      xx b1 b2 xx // Размещение C в памяти
      ^
      B

- Картинка

- Даже если преобразований адреса не происходит, то вероятность залезть в чужую память остаётся
  - struct A{
    int a1;
    };

    struct C: A{
    int c1;
    };

    A object;
    C* c = static_cast<A*>(&object);
    cout << &object << '\n' << c << '\n' << &c->c1;
    - Вывод:
     600
     600
     604
    - Как видно, смешения адреса не произошло, но при попытке получить доступ к c1 мы выходим за пределы реально выделенной памяти и снова попадаем неизвестно куда

      a1 xx // Размещение C в памяти
      ^
      A
      - Картинка

- Когда преобразование вниз по иерархии безопасно?
  - Чтобы преобразование было безопасно, реально размещённый в памяти объект должен быть ниже или на том же уровне иерархии, что и тип указателя к которому выполняется преобразование

```
struct A{
   int a1;
};

struct B: A{
   int b1;
   int b2;
};

struct C: B{
   int c1;
};

C object;
A* a = &object;    // Здесь OK т.к. UpCast
B* b = static_cast<B*>(a);
cout << &object << '\n' << a << '\n' << b;
```

- Вывод:
 60
 60
 64
- В этом примере класс C наследуется только от B, который наследуется от A.
  Т.е. реальный объект на который указывает A принадлежит классу C, то преобразование вниз к B безопасно. т.к. в памяти присутствуют все данные для корректной работы B

  ```
  a1 b1 b2 c1   // Размещение C в памяти
     ^
     B
  ```

  - Картинка

- Если класс содержит хотя бы один виртуальный метод, то, кроме членов-данных в памяти каждого объекта хранится таблица виртуальных методов, поэтому для преобразования вниз вместо static_cast нужно использовать dynamic_cast, который перед преобразованием сопоставляет таблицу виртуальных методов исходного и конечного классов и в случае их не совместимости возвращает нулевой указатель

```
struct A{
   virtual ~A(){};
};

struct B: A{
};

struct D{
};

struct C:A, D{
};

C object;
A* a = &object;  // OK, т.е. UpCast
B* b1 = static_cast<B*>(a);
B* b2 = dynamic_cast<B*>(a);
C* c = dynamic_cast<C*>(a);

cout << (b1 == nullptr) << '\n' << (b2 ==nullptr) << '\n'
        << (c ==nullptr);
```

- Картинка
- Результат:
  false
  true
  false
- Здесь в классе A присутствует виртуальный деструктор, который наследуется его классами потомками.
  Попытка преобразования из A в B при помощи static_cast проходит успешно, хотя это потенциальная ошибка, т.к. реальный объект под указателем это объект класса C.
  Попытка сделать тоже самое, но через dynamic_cast проваливается и это правильное поведение.
  Преобразование из A обратно в C через dynamic_cast, как и ожидалось проходит успешно