Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2020

Содержание

Указатели

Указатель – это специальный тип, производный от другого типа, чьё значение является одним из:

- указателем на объект или функцию (представляет собой адрес в памяти функции или первого байта объекта)
- указателем на “после” объекта (представляет адрес в памяти на первый байт, следующий за объектом)
- нулевым указателем
- неопределенным

*специф-ия типа [имя класса] * [cv-квалиф-ор] декларатор*

Примеры объявлений

```
 1  int * a;
 2
 3  const double * b;
 4
 5  char ** c;
 6
 7  void (*d)(int);
 8
 9  int A::* e;
10
11  void (B::* f)(int);
```

Реализация указателей

Размер указателя – зависит от платформы/реализации.

С т.з. языка значение указателя и адрес в памяти, которое оно представляет – не эквивалентные вещи.

Указатель связан не с конкретным объектом, а с местом его хранения. Период существования места хранения определяется типом размещения (автоматический, статический, тред-локальный, динамический), за пределами периода существования места хранения значение указателя на него является неопределенным.

Значения указателей

Разыменование или освобождение неопределенного указателя является UB, любое другое использование неопределенного указателя является поведением, зависящим от реализации.

Нулевой указатель - это специальное значение указателя конкретного типа, служащее для указания отсутствия сущности, на которую указатель ссылается. Разыменование нулевого указателя – UB.

std::nullptr_t

`std::nullptr_t` ⇔ `nullptr`

- `std::nullptr_t` – не является указателем
- `nullptr` – неявно приводится к нулевому указателю любого типа

```
1  int * x = nullptr; // null pointer to int
2  const C * c = nullptr; // null pointer to const C
3  void * u = nullptr; // null pointer to void
```

Указатели, приведения типов и NULL

```
 1    #include <iostream>
 2
 3    int f() { return 10; }
 4
 5    template <class T>
 6    T g(T t) { return t; }
 7
 8    int main()
 9    {
10      auto * pf = f;
11      int * x = f; // error
12      int * x = static_cast<int *>(f); // error
13      int * x = NULL;
14      int a = NULL; // maybe a warning
15      int a = nullptr; // error
16      int a = g(NULL);
17      std::cout << f << std::endl;
18      std::cout << pf << std::endl;
19      std::cout << x << std::endl;
20      std::cout << a << std::endl;
21    }
```

Указатели на объекты

Значение указателя на объект можно получить с помощью оператора взятия адреса:

```
1  A a;
2  A * p_a = &a;
3  A ** pp_a = &p_a;
4  struct S { int n; } s;
5  int * p_i = &s.n;
```

Разыменование указателя: оператор разыменования * возвращает lvalue выражение, идентифицирующее объект, на который указывает разыменованный указатель.

```
1  A a;
2  A * p_a = &a;
3  *p_a = A();
4  A aa = *&a;
```

Разыменование указателей сложных типов

Оператор -> используется для упрощения доступа к членам сложных типов через указатель на значение типа и эквивалентен последовательному разыменованию и затем доступу к элементу:

```
1  A a;
2  A * p_a = &a;
3
4  p_a->foo();
5  (*p_a).foo();
6  a.foo();
```

Указатели на экземпляры классов и наследование

```
1  class A {};
2  class B {};
3  class C : public A, public B {};
4
5  C c;
6  A * a = &c;
7  B * b = &c;
8  C * c_from_a = static_cast<C *>(a);
9  C * c_from_b = static_cast<C *>(b);
```

Указатели на экземпляры классов и наследование, ограничения

```
 1  class D : A, B {};
 2  class E : public A, public C {};
 3  class F : public B {};
 4
 5  D d;
 6  A * a_from_d = &d; // compilation error
 7
 8  E e;
 9  C * c_from_e = &e;
10  A * a_from_e = &e; // compilation error
11
12  C c;
13  B * b_from_c = &c;
14  F * f_from_b = static_cast<F *>(b_from_c); // wrong, but
        ↪ compiles
```

Безтиповые указатели

Возможен указатель на `void` (в т.ч. cv-квалифицированный). Это безтиповый указатель, к которому неявно приводятся обычные указатели. Обратная операция приведения должна быть явной (`static_cast`).

```
1  struct S {};
2  S s;
3  const void * p = &s;
4  const S * ps = static_cast<const S *>(p);
```

Указатели на функции

Выражение, идентифицирующее функцию, неявно приводится к указателю на функцию. Указатель на функцию может непосредственно являться левым операндом оператора функционального вызова.

```
1  double foo(int, char);
2  double (*p) (int, char) = foo;
3  double (*pp) (int, char) = &foo;
4
5  assert(p == pp);
6  p(x, y);
7  (*p)(x, y);
```

Указатели на нестатические методы

```
 1  struct S {
 2      bool check(int, char);
 3  };
 4
 5  bool (S::* p) (int, char) = S::check; // error
 6  bool (S::* p) (int, char) = &S::check;
 7
 8  int i = 0;
 9  char c = 'a';
10  S s, *ps = &s;
11  bool x = (s.*p) (i, c);
12  bool y = (ps->*p)(i, c);
13  auto * pf = s.check; // error
14  auto * pf = &s.check; // error
```

Указатели на нестатические методы, продолжение

```
 1  struct A {
 2      int get_n(char);
 3  };
 4  struct B : A {
 5      bool empty();
 6  };
 7  int (A::*pa) (char) = &A::get_n;
 8  int (B::*pb) (char) = pa;
 9  A a;
10  B b;
11  (a.*pa)('a');
12  (b.*pb)('b');
13  bool (B::*eb) () = &B::empty;
14  bool (A::*ea) () = static_cast<bool (A::*) ()>(eb);
15  (b.*ea) (); // OK
16  (a.*ea) (); // UB
```

Указатели на нестатические поля

Аналогично указателям на нестатические методы (включая вопросы наследования):

```
1  struct A { int n; };
2
3  int A::* p = &A::n;
4  A a{101};
5  A * pa = &a;
6
7  int x = a.*p; // 101
8  int y = pa->*p; // 101
```

Массивы

Массивы – агрегаты, состоящие из конечного числа элементов одного типа. Занимают непрерывную область в памяти. Элементы массива считаются его подобъектами.

```
1  int x[10];
2  const char str[] = "Hello"; // implicit size
```

Элементы массива нумеруются с 0, доступ к ним возможен через оператор индекса:

```
1  for (int i = 0; i < 10; ++i) {
2      x[i] = i;
3  }
```

Массив неявно преобразуется к указателю на первый элемент массива.

Ограничения массивов

Массив, как целое – иммутабелен, хотя его элементы можно менять, если их тип – не `const`.

Массивы неопределенной длины являются неполным типом:

```
1  extern int x[];
```

Массивы нельзя возвращать из функций.

Размер объекта массива:

```
1  int x[10];
2  size_t n = sizeof(x) / sizeof(x[0]); // 10
```

Арифметика указателей

Указатели могут выступать операндами некоторых арифметических операций, при этом указатель на отдельный объект считается указателем на элемент массива единичной длины.

Допустимые операции:

- сложение – указатель и целое число (в любом порядке)
- вычитание – указатель (слева) и целое число
- вычитание двух указателей одного типа
- инкремент
- декремент

Правила арифметики указателей: указатель и число

Если P – указатель на i элемент массива x, то

$P + n \Leftrightarrow x[i + n]$

$n + P \Leftrightarrow x[i + n]$

$P - n \Leftrightarrow x[i - n]$

Результат $x[j]$ – должен быть корректным элементом массива, либо элементом после последнего.

Правила арифметики указателей: вычитание указателей

Оба указателя должны быть одного типа не считая cv-квалификаторов.

Если P – указатель на i элемент массива x, а Q – указатель на j элемент массива x, то

$$P - Q = i - j$$

Результирующее значение имеет тип `std::ptrdiff_t`.

Любые другие арифметические операции над указателями – UB.

Полиморфизм в стиле C

```
 1    #include <cstdlib>
 2
 3    int cmp(const void * lhs, const void * rhs)
 4    {
 5        int arg1 = *static_cast<const int *>(lhs);
 6        int arg2 = *static_cast<const int *>(rhs);
 7        if (arg1 < arg2) return -1;
 8        if (arg1 > arg2) return 1;
 9        return 0;
10    }
11
12    // void qsort(void *ptr, std::size_t count, std::size_t size, /*c-
          ↪ compare-pred*/* comp);
13
14    int main()
15    {
16        int a[] = {-2, 99, 0, -743, 2, INT_MIN, 4};
17        std::qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]), cmp);
18        return a[0];
19    }
```

Немного о reinterpret_cast

```
1   template <class T>
2   void a(T x) {
3     if constexpr(std::is_integral_v<T> || std::is_enum_v<T> || std::is_pointer_v<T> || std::is_member_pointer_v<T>){
4       assert(x == reinterpret_cast<T>(x);
5     }
6   }
7
8   template <class I, class P>
9   void b(P x) {
10    if constexpr (std::is_pointer_v<P> && sizeof(P) <= sizeof(I)) {
11      I i = reinterpret_cast<I>(x);
12      assert(x == reinterpret_cast<P>(i));
13      assert(I{0} == reinterpret_cast<I>(nullptr));
14      reinterpret_cast<std::nullptr_t>(0); // error
15      reinterpret_cast<std::nullptr_t>(nullptr); // error
16    }
17  }
18
19  template <class P, class I>
20  void c(I x) {
21    if constexpr ((std::is_integral_v<I> || std::is_enum_v<I>) && std::is_pointer_v<P>) {
22      P p = reinterpret_cast<P>(x);
23      assert(static_cast<P>(nullptr) ??? reinterpret_cast<P>(I{0}));
24      assert(static_cast<P>(nullptr) ??? reinterpret_cast<P>(NULL));
25    }
26  }
27
28  template <class T1, class T2>
29  void d(T1 * x) {
30    T2 * y = reinterpret_cast<T2 *>(x);
31    const T2 * cy = reinterpret_cast<const T2 *>(x);
32    T2 * z = static_cast<T2 *>(static_cast<void *>(x));
33    const T2 * cz = static_cast<const T2 *>(static_cast<const void *>(x));
34    assert(y == z);
35    assert(cy == cz);
36    assert(x == reinterpret_cast<T1 *>(y));
37    T1 * n1 = nullptr;
38    T2 * n2 = nullptr;
39    assert(n1 == reinterpret_cast<T1 *>(n2));
40  }
```

Содержание

Ссылки

Ссылка – это ассоциация (“связывание”) имени с каким-либо объектом или функцией, псевдоним. Реализация ссылок в стандарте не оговорена.

Объявление:

спецификация типа & декларатор

спецификация типа && декларатор

```
1  int x;
2  int & rx = x;
3  int && rrx = 55;
```

Ограничения ссылок

- обязательная инициализация
- иммутабельность
- не является объектом и не имеет своего значения
- всегда указывает на какой-то объект или функцию
- невозможность cv-квалификации
- необязательность наличия размера
- время жизни эквивалентно области видимости

Если время жизни объекта закончится раньше времени жизни ссылки, ссылка становится “висящей” и её использование является UB.

Ограничения ссылок, примеры

```
1  X & x; // error
2  void & v; // error
3  X & x[3]; // error
4  X &* px; // error
5  X & & xx; // error
6
7  X & foo() { X x; return x; }
8  X & x = foo(); // dangling reference
9  X y = x; // UB
```

Типы ссылок и их связывание

- lvalue и rvalue
- `const` и не-`const`

- lvalue ссылка связывается с lvalue выражением
- `const` lvalue ссылка связывается с lvalue или xvalue
- rvalue ссылка связывается с xvalue

`auto &&` – будет выведена в самую подходящую ссылку.

Продление времени жизни временного объекта

Если временный объект или его подобъект связывается со ссылкой, его время жизни продлевается на время жизни ссылки.

Исключения:

- временный объект в инструкции return, если функция возвращает ссылку, разрушается в конце исполнения инструкции
- временный объект, связывающийся со ссылкой внутри выражения (например, к ссылочному параметру функции в выражении её вызова), разрушается в конце исполнения полного выражения.

Продление времени жизни временного объекта, примеры

```
1  const int & a = 1 + 2; // extended
2  int && a = 2 + 3; // extended
3  const int & l = std::array<int, 3>{1, 2, 3}[1]; //
       ↪ extended
4
5  const double & bar() {
6      return 0.5 * 1.5;
7  }
8  const double & d = bar(); // dangling reference
9
10 const std::string & pass(const std::string & s) {
11     return s; // OK
12 }
13 const std::string & s = pass("hello");
14 std::cout << s; // error, dangling reference
```

Схлопывание ссылок

Допускается путём манипуляций над псевдонимами типов и шаблонами получать ссылку на ссылку, в этом случае применяются правила “схлопывания” ссылок (reference collapsing) и результатом всегда является обычная ссылка:

+		=
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

Пример схлопывания ссылок

```
1  using X = T &;
2  using Y = T &&;
3
4  T t;
5
6  X & x = t; // decltype(x) == T &
7  X && xx = t; // decltype(xx) == T &
8  Y & y = t; // decltype(y) == T &
9  Y && yy = std::move(t); // decltype(yy) == T &&
```

Реализация ссылок

Реализация ссылок не определена стандартом. Ссылки не обязаны существовать в результате трансляции программы.

Однако, можно ожидать, что:

```
1  struct A {
2      int * x;
3  };
4
5  struct B {
6      int & x;
7  };
8
9  static_assert(sizeof(A) == sizeof(B));
```

Квалификаторы и указатели/ссылки

cv-квалификатор можно ставить как слева, так и справа от того, на что он действует.

```
1  const int x = 10;
2  int const x = 10;
3
4  const char * str; // points to a const object
5  char const * str; // pointer itself is non-const
6
7  char * const str_c; // points to a non-const object
8  using X = char *;
9  const X str_c; // pointer itself is const
10 X const str_c;
11
12 const std::string & s; // reference a const object
13 std::string const & s; // reference itself is inherently
       ↪ immutable
```

Квалификаторы и многоуровневые указатели

```
 1
 2  int *** x;
 3  int const *** y;
 4  int * const ** z;
 5  int ** const * w;
 6  int *** const v;
 7
 8  using X = const int;
 9  using Y = const X *;
10  using Z = const Y *;
11  const Z * p;
12  int const * const * const * p;
```

Передача объектов

- по значению – объект копируется
- по указателю – копируется указатель, можно передать нулевой указатель
- по ссылке – передаётся связь с объектом

```
1  struct S {};
2  void f1(S s) {
3      s = S();
4  }
5  void f2(const S * ps) {
6      if (ps != nullptr) {
7          std::cout << *ps << std::endl;
8      }
9  }
10 void f3(const S & rs) {
11     std::cout << rs << std::endl;
12 }
```

Возврат результатов через аргументы функции

```
 1    bool f(int a, int * b)
 2    {
 3      *b = a;
 4      return true;
 5    }
 6
 7    bool g(int a, int & b)
 8    {
 9      b = a;
10      return true;
11    }
12
13    bool h(int a, int b)
14    {
15      b = a;
16      return true;
17    }
18
19    int a = 10, b = -1;
20    f(a, &b);
21    g(a, b);
22    h(a, b);
```

Содержание

Работа с динамической памятью

Выделение – делает доступным для использования участок памяти заданного размера.

Освобождение – возвращает выделенный ранее участок памяти операционной системе.

`malloc`/`free` – функции выделения/освобождения памяти из C.

`new`/`delete` – операторы выделения/освобождения памяти в C++.

new

```
1  X * x = new X;
2  X * x = ::new X;
3  X * x = new X();
4  X * x = new X(a, b, c);
5  X * x = new X{a, b, c};
6  X * x = new X[10];
7  X * x = new X[10]{x1, x2, x3};
8  X * x = new X[0]; // OK, zero element array
9
10 auto p = new (int (*[10])()); // array of function
       ↪ pointers
11
12 auto m = new double[n][5][10][100];
13 auto mm = new double[5][n][10]; // error
```

Размещающее new

```
1  struct S { ... };
2
3  std::byte * ptr = new std::byte[sizeof(S)];
4
5  S * ps = new (ptr) S(...);
6
7  ps->~S();
8
9  delete [] ptr;
```

delete

```
 1  X * x = new X;
 2  delete x;
 3
 4  X * x = nullptr;
 5  delete x;
 6
 7  X * x = new X[10];
 8  delete [] x;
 9
10  struct A {};
11  struct B : A {};
12  A * a = new B;
13  delete a;
```

Содержание

std::string_view

Указание на участок памяти, интерпретируемый как последовательность символов. Не позволяет менять данные, на которые указывает.

```
 1  #include <string>
 2  #include <string_view>
 3
 4  std::string x("Some␣string");
 5  std::string_view y = x;
 6
 7  if (const auto pos = y.find("me"); pos != y.npos) {
 8      std::string_view z = y.substr(pos, 4);
 9      assert(z.size() == 4);
10      assert(z == "me␣s");
11  }
12
13  for (char c : y) {
14      std::cout << c;
15  }
```

std::reference_wrapper

```
 1  #include <functional>
 2
 3  void f(S &);
 4  void cf(const S &);
 5
 6  S s;
 7  auto r = std::ref(s);
 8  auto cr = std::cref(s);
 9  f(r);
10  f(r.get());
11  cf(cr);
12
13  std::vector<std::reference_wrapper<const S>> v;
14  v.push_back(cr);
15  v.push_back(cr);
16  auto vv = v;
```