Программирование на языке C++

Шаблов Анатолий
anatoliishablov@gmail.com

ИТМО, весенний семестр 2025

Указатели

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

```
int * a;

const double * b;

char ** c;

void( * d)(int);

int A:: * e;

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

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

Указатели, приведения типов и NULL

```
#include <iostream>

int f() { return 10; }

template <class T>
T g(T t) { return t; }

int main() {
  auto * pf = f;
  int  *  x = f; // error
  int  *  x = static_cast < int * > (f); // error
  int  *  x = NULL;
  int     a = NULL; // maybe a warning
  int     a = nullptr; // error
  int     a = g(NULL);
  std::cout << f << std::endl;
  std::cout << pf << std::endl;
  std::cout << x << std::endl;
  std::cout << a << std::endl;
}
```

Указатели на объекты

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

```
A a;
A * p_a = &a;
A ** pp_a = &p_a;
struct S { int n; } s;
int * p_i = &s.n;
```

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

```
A a;
A * p_a = &a;
* p_a   = A();
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:

```
A a;
A* p_a = &a;

p_a->foo();
(*p_a).foo();
a.foo();
```

Указатели на экземпляры классов и наследование

```
class A {};
class B {};
class C : public A, public B {};

C c;
A *a = &c;
B *b = &c;
C *c_from_a = static_cast<C *>(a);
C *c_from_b = static_cast<C *>(b);
```

Ограничения

```
class D : A, B {};
class E : public A, public C {};
class F : public B {};

D d;
A *a_from_d = &d;  // compilation error

E e;
C *c_from_e = &e;
A *a_from_e = &e;  // compilation error

C c;
B *b_from_c = &c;
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).

```
struct S {};
S s;
const void *p = &s;
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.

```
double foo(int, char);
double (*p)(int, char) = foo;
double (*pp)(int, char) = &foo;

assert(p == pp);
p(x, y);
(*p)(x, y);
```

Указатели на нестатические методы

```
struct S {
  bool check(int, char);
};

bool (S::*p)(int, char) = S::check;  // error
bool (S::*p)(int, char) = &S::check;

int i = 0;
char c = 'a';
S s, *ps = &s;
bool x = (s.*p)(i, c);
bool y = (ps->*p)(i, c);
auto* pf = s.check;   // error
auto * pf = &s.check;  // error
```

Указатели на нестатические методы

```
struct A {
  int get_n(char);
};
struct B : A {
  bool empty();
};
int (A::*pa)(char) = &A::get_n;
int (B::*pb)(char) = pa;
A a;
B b;
(a.*pa)('a');
(b.*pb)('b');
bool (B::*eb)() = &B::empty;
bool (A::*ea)() = static_cast<bool (A::*)()>(eb);
(b.*ea)();  // OK
(a.*ea)();  // UB
```

Указатели на нестатические поля

Аналогично указателям на нестатические методы (включая вопросы наследования):

```
struct A {
  int n;
};

int A::*p = &A::n;
A a{101};
A* pa = &a;

int x = a.*p;    // 101
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.

```
int x[10];
std::array<int, 10> x; // Use this instead
const char str[] = "Hello";  // implicit size
```

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

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

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:

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

Массивы нельзя возвращать из функций.

Размер объекта массива:

```
int x[10];
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.

Ссылки

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

```
int                                                    x;
int&                    rx                     =       x;
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.

Примеры

```
X& x;      // error
void& v;   // error
X& x[3];   // error
X&* px;    // error
X&& xx;    // error

X& foo() {
  X x;
  return x;
};
X& x = foo();  // dangling reference
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.

Примеры

```
const int& a = 1 + 2;                                  // extended
int&& a = 2 + 3;                                       // extended
const int &l = foo().bar;                              // extended
const int &l = dummy<int[3]>{1,2,3}[1];                // extended

const double& bar() { return 0.5 * 1.5; }
const double& d = bar();  // dangling reference

const std::string& pass(const std::string& s) {
  return s;  // OK
}
const std::string& s = pass("hello");
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

```
using X = T &;
using Y = T &&;

T t;

X &x = t;                  // decltype(x)  == T &
X &&xx = t;                // decltype(xx) == T &
Y &y = t;                  // decltype(y)  == T &
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:

```
struct A {
  int* x;
};

struct B {
  int& x;
};

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.

```
const int x = 10;
int const x = 10;

const char *str;  // points to a const object
char const *str;  // pointer itself is non-const

char *const str_c;  // points to a non-const object
using X = char *;
const X str_c;  // pointer itself is const
X const str_c;

const std::string &s;  // reference a const object
std::string const &s;  // reference itself is inherently immutable
```

Квалификаторы и многоуровневые указатели

```
int ***x;
int const ***y;
int *const **z;
int **const *w;
int ***const v;

using X = const int;
using Y = const X *;
using Z = const Y *;
const Z *p;
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

```
struct S {};
void f1(S s) { s = S(); }
void f2(const S* ps) {
  if (ps != nullptr) {
    std::cout << *ps << std::endl;
  }
}
void f3(const S& rs) { std::cout << rs << std::endl; }
```

Возврат результатов через аргументы функции

```
bool f(int a, int* b) {
  *b = a;
  return true;
}

bool g(int a, int& b) {
  b = a;
  return true;
}

bool h(int a, int b) {
  b = a;
  return true;
}

int a = 10, b = -1;
f(a, &b);
g(a, b);
h(a, b);
```

Работа с динамической памятью

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

```
X* x = new X;
X* x = ::new X;
X* x = new X();
X* x = new X(a, b, c);
X* x = new X{a, b, c};
X* x = new X[10];
X* x = new X[10]{x1, x2, x3};
X* x = new X[0];  // OK, zero element array

auto p = new (int (*[10])());  // array of function pointers

auto m = new double[n][5][10][100];
auto mm = new double[5][n][10];  // error
```

Размещающее new

```
struct S {
  ...
};

std::byte* ptr = new std::byte[sizeof(S)];

S* ps = new (ptr) S(...);

ps->~S();

delete[] ptr;
```

delete

```
X* x = new X;
delete x;

X* x = nullptr;
delete x;

X* x = new X[10];
delete[] x;

struct A {};
struct B : A {};
A* a = new B;
delete a;
```

Вспомогательные классы

std::string_view

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

```
#include <string>
#include <string_view>

std::string x("Some␣string");
std::string_view y = x;

if (const auto pos = y.find("me"); pos != y.npos) {
  std::string_view z = y.substr(pos, 4);
  assert(z.size() == 4);
  assert(z == "me␣s");
}

for (char c : y) {
  std::cout << c;
}
```

std::reference_wrapper

```
#include <functional>

void f(S &);
void cf(const S &);

S s;
auto r = std::ref(s);
auto cr = std::cref(s);
f(r);
f(r.get());
cf(cr);

std::vector<std::reference_wrapper<const S>> v;
v.push_back(cr);
v.push_back(cr);
auto vv = v;
```