Указатели и ссылки

Общая идея: косвенное обращение

```
int x = 0;
int* x_ptr = &x;
int& x_ref = x;

*x_ptr = 42;
std::cout << x << ‘\n’; // x = ?

x_ref = 314;
std::cout << x << ‘\n’; // x = ?
```

Общая идея: косвенное обращение

```
int x = 0;
int* x_ptr = &x;
int& x_ref = x;

*x_ptr = 42;
std::cout << x << ‘\n’; // x = 42

x_ref = 314;
std::cout << x << ‘\n’; // x = 314
```

Немного об определениях

Какие из этих определений скомпилируются?

```
int x1 = 1;
const int x2 = 2;
int const x3 = 3;
int x4 = x3;
const int x5;
extern const int x6;
```

Немного об определениях

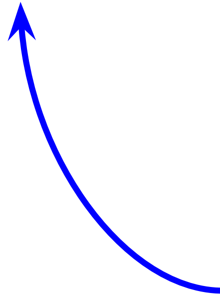
Какие из этих определений скомпилируются?

Определение создает сущность

```
int x1 = 1;             // Ok
const int x2 = 2;       // Ok
int const x3 = 3;       // Ok
int x4 = x3;            // Ok
const int x5;           // Error: uninitialized
extern const int x6;    // Ok
```

Немного об определениях

```
int const x3 = 3;
int x4 = x3;
```



Типы слева и справа отличаются

Что из этого скомпилируется?

```
unsigned int x1 = 1;
int unsigned x2 = 2;
const unsigned int x3 = 3;
unsigned const int x4 = 4;
int const unsigned x5 = 5;
unsigned int const x6 = 6;
const int const x7 = 7;
```

Что из этого скомпилируется?

```
unsigned int x1 = 1;          // Ok
int unsigned x2 = 2;          // Ok
const unsigned int x3 = 3;    // Ok
unsigned const int x4 = 4;    // Ok, но не пишите так
int const unsigned x5 = 5;    // Ok, но не пишите так
unsigned int const x6 = 6;    // Ok
const int const x7 = 7;       // Error: duplicate const
```

Что из этого скомпилируется?

```
int v = 42;
const int c = 13;

int* pv1 = &v;
const int* pv2 = &v;
int* pc1 = &c;
const int* pc2 = &c;
```

Что из этого скомпилируется?

```
int v = 42;
const int c = 13;

int* pv1 = &v;          // Ok
const int* pv2 = &v;    // Ok
int* pc1 = &c;          // Error
const int* pc2 = &c;    // Ok
```

Что из этого скомпилируется?

```
int v = 42;
int* pv1 = &v;

const int* pv2 = pv1;
int* pv3 = pv2;
int* const pv4 = pv1;
int* pv5 = pv4;
const int* pv6;
int* const pv7;
```

Что из этого скомпилируется?

```
int v = 42;
int* pv1 = &v;

const int* pv2 = pv1;   // Ok
int* pv3 = pv2;         // Error: invalid conversion
int* const pv4 = pv1;   // Ok
int* pv5 = pv4;         // Ok
const int* pv6;         // Ok
int* const pv7;         // Error: uninitialized
```

Top-level const

```
const int* pv2 = ...;
int* pv3 = pv2;          // Error: invalid conversion

int* const pv4 = pv1;
int* pv5 = pv4;          // Ok
```

При копировании const верхнего уровня игнорируется

Что из этого скомпилируется?

```
int v1 = 1;
const int v2 = 2;

int& rv1 = v1;
int& rv2 = v2;
const int& rv3 = v1
const int& rv4 = v2
const int& rv5;
int& const rv6;
```

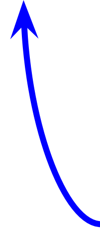
Что из этого скомпилируется?

```
int v1 = 1;
const int v2 = 2;

int& rv1 = v1;          // Ok
int& rv2 = v2;          // Error
const int& rv3 = v1;    // Ok
const int& rv4 = v2;    // Ok
const int& rv5;         // Error
int& const rv6;         // Error
```

Некоторые свойства ссылок

- Ссылки обязаны быть инициализированы
- У ссылок не может быть top-level const
- Ссылки связываются единожды



В этом ссылки похожи на константные указатели

```
T* const ptr = ...;
```

Ссылки прозрачны для операций

```
int x = 42, y = 314;

int& ref = x;
int* ptr1 = &x;
int* ptr2 = &y;

ref = y;        // x = ?   y = ?
ptr1 = ptr2;    // x = ?   y = ?
```

Ссылки прозрачны для операций

```
int x = 42, y = 314;

int& ref = x;
int* ptr1 = &x;
int* ptr2 = &y;

ref = y;        // x = 314 y = 314
ptr1 = ptr2;    // x = 314 y = 314
```

Ссылки

```
const auto& group_name = students[i].group.full_label;

// use group_name
```

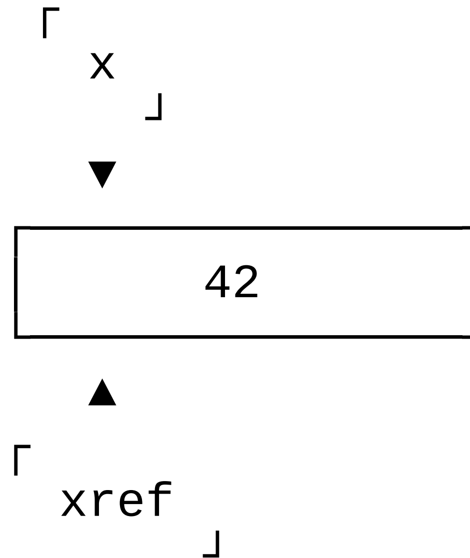
Ссылки удобны как синонимы глубоко вложенных объектов

Неформальные определения

Указатель — это **объект**, хранящий адрес другого объекта[1].

Ссылка — это синоним (**имя**) объекта[1].

```
int x = 42;
int& xref  = x;
```



Объясните, что здесь записано

```
int*& a = ...;
int&* b = ...;
```

Объясните, что здесь записано

```
int*& a = ...;  // Ok: ссылка на указатель
int&* b = ...;  // Error: у ссылки нет адреса
```

Массив VS указатель

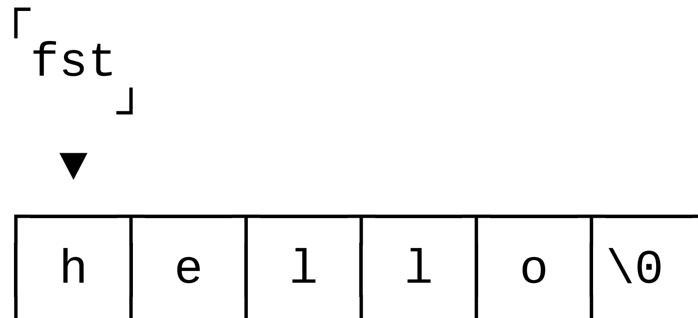
```
const char         fst[] = “hello”;

const char* const snd    = “world”;
```

В чем разница?

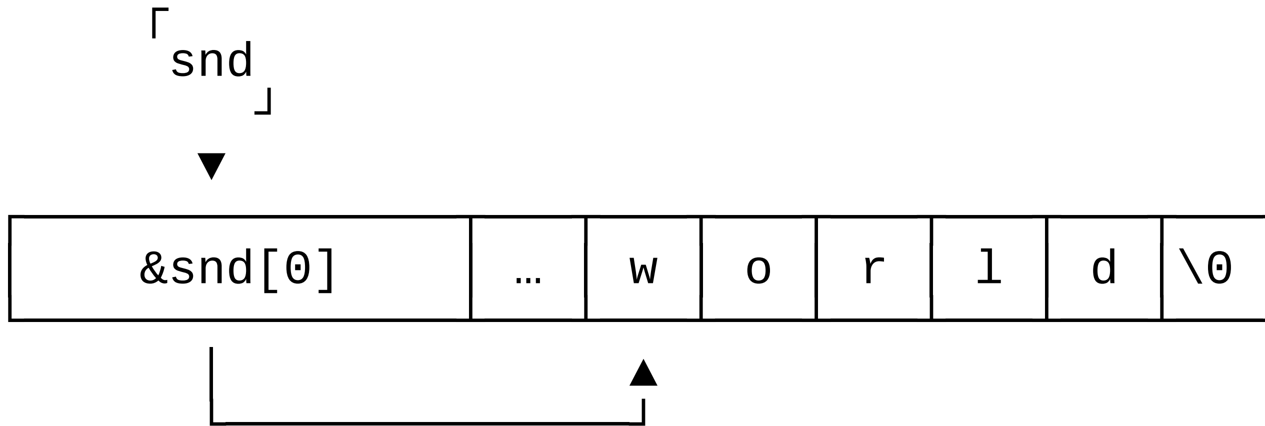
Имя массива — не объект

```
const char fst[] = “hello”;
```



Указатель — это объект

```
const char* const snd = “world”;
```



Имя массива — не объект

```
const char fst[] = “hello”;
const char* const snd = “world”;

&fst    = 0x7ffc085ec43a

&fst[0] = 0x7ffc085ec43a

&snd    = 0x7ffc085ec430

&snd[0] = 0x402004
```


https://godbolt.org/z/T6javz54b

Массивы и адресная арифметика

Пусть

```
int pi[] = {3, 14, 15, 92, 65};
```

Тогда следующие выражения эквивалентны:

```
pi[2]    *(pi + 2)

2[pi]    *(2 + pi)
```


https://godbolt.org/z/9f1rd3v6q

Массивы и адресная арифметика

Пусть

```
int pi[] = {3, 14, 15, 92, 65};
```

Ссылки прозрачны для операций:

```
int& ref = pi[0];
++ref;                  // pi[] = {4, 14, 15 ,92, 65}
```

Указатели, ссылки и функции

- Передача тяжелых объектов без копирования
- out-параметры

Передача тяжелых значений без копирования

Массивы:

```
int strcmp(const char *s1, const char *s2);

char s1[] = “...”, s2[] = “...”;
strcmp(s1, s2);  // array to pointer decay
```

Тяжелые объекты:

```
print(const Matrix* matrix);
```

Что вы можете сказать о ts?

```
void f(const Triangle* ts);
```

Что вы можете сказать о ts?

```
void f(const Triangle* ts);
```

Это адрес одного элемента или целый массив?

Что вы можете сказать о ts?

```
void f(const Triangle* ts);
void f(const Triangle* ts, size_t n);
```

Это адрес одного элемента или целый массив?

Если это массив, то нужен размер или sentinel

Что вы можете сказать о ts?

```
void f(const Triangle* ts);
void f(const Triangle* ts, size_t n);

void f(const Triangle& t)
void f(const std::vector<Triangle>& ts);
```

В C++ предпочитаем контейнеры

Array to pointer decay

```
void f(const char* s);



char str[] = “Hello, very very long string!”;

f(str);
f(“Hello, very very long string!”);
```

Оба вызова «бесплатны»

Создание временного объекта

```
void f(const std::string& s);


std::string str = “Hello, very very long string!”;

f(str);
f(“Hello, very very long string!”);
```

При вызове конструируется std::string

Вариант 1: перегрузка

```
void f(const std::string& s);
void f(const char* s);

std::string str = “Hello, very very long string!”;

f(str);
f(“Hello, very very long string!”);
```

Вариант 2: std::string_view

```
void f(std::string_view s);


std::string str = “Hello, very very long string!”;

f(str);
f(“Hello, very very long string!”);
```

Вариант 2: std::string_view

string_view — пара {size_t len, const char* str}

Out и in-out параметры функций

```
int SDL_GetRendererOutputSize(
  SDL_Renderer* renderer,
  /* Out */ int* w, int* h);


// In-out
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

Результат работы функции

```
void split_to(
  const std::string& s,
  std::vector<std::string>* words);

void split_to(
  const std::string& s,
  std::vector<std::string>& words);
```

Результат работы функции

```
// Видно, что это out-параметр
split_to(“a b c”, &words);

// words не может быть “nullptr”
split_to(“a b c”, words);
```

Оба варианта хуже

Результат работы функции

```
std::vector<std::string> split(const std::string& s);

auto words = split_to(“a b c”)
```

Здесь не будет тяжелого копирования вектора

Передача параметров в функцию

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don’t know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td rowspan="2">f(X&)</td></tr>
<tr><td>In/Out</td><td colspan="2">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2" rowspan="2">f(const X&)</td></tr>
<tr><td>In & retain “copy”</td></tr>
</table>

“Cheap” ≈ a handful of hot int copies

“Moderate cost” ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

CppCoreGuidelines

В чем разница?

```
void fp(int* p) {
    *p = 0xAB;
}

void fr(int& r) {
    r = 0xAB;
}
```

```
int main() {
    int g = 0;
    fp(&g);
    fr(g);
}
```


https://godbolt.org/z/rqdv8Mqc9

Семантика отсутствия объекта

```
FILE *fopen(const char *pathname, const char *mode);
```

RETURN VALUE

Upon successful completion [...] return a FILE pointer.

Otherwise, **NULL** is returned

Какого типа NULL?

MSVC: Определение NULL

```
// vcruntime.h
#ifndef NULL
    #ifdef __cplusplus
        #define NULL 0
    #else
        #define NULL ((void *)0)
    #endif
#endif
```

В C++ NULL не может быть void*

Почему в C++ NULL == 0

Хотим написать:

```
char* str = NULL;
```

C++ строже относится к преобразованию указателей

Корректно в C, Некорректно в C++:

```
char* str = (void*)0;
```

Корректно в C и в C++:

```
char* str = 0;
```

nullptr

В C++ рекомендуется использовать **`nullptr`**:

```
// gcc
typedef decltype(nullptr) nullptr_t;
```

ES.47: Use nullptr rather than 0 or NULL

Мотивация nullptr: разрешение перегрузки

```
const char* f(double*) { return "f(double*)\n"; }

const char* f(long)    { return "f(long)\n"; }

int main() {
  std::cout
    << "f(NULL) -> "    << f(NULL)     // f(long)
    << "f(nullptr) -> " << f(nullptr); // f(double*)
}
```


https://godbolt.org/z/dMWKT5M3s

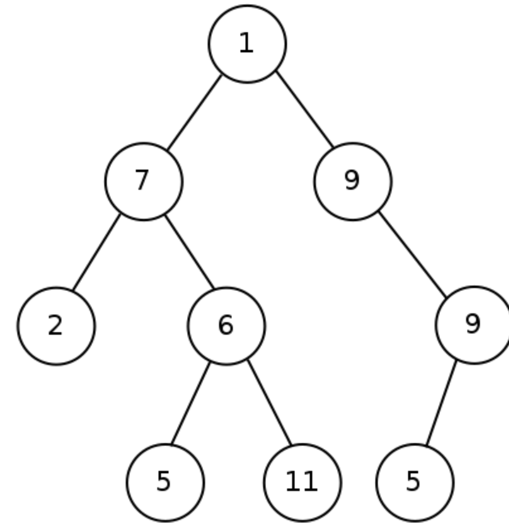
std::optional

У ссылок нет **nullref**, используйте `std::optional<T>`

```
std::optional<Widget> maybe_create_wigdet();
```

Динамические структуры данных

```
struct TreeNode {
  TreeNode* left_;
  TreeNode* right_;
};
```



Применение указателей

1. Массивы и адресная арифметика
2. Передача тяжелых значений в функции без копирования
3. Out и in-out параметры функций
4. Семантика отсутствия объекта
5. Создание динамических структур данных

Это очень много для единственной абстракции

Используйте альтернативы

1. Массивы и адресная арифметика ➡ **STL**
2. Передача тяжелых значений в функции ➡ **const&**
3. Out и in-out параметры функций ➡ **return, &**
4. Семантика отсутствия объекта ➡ **std::optional**
5. **Создание динамических структур данных**

Да, но…

this

Многомерность

```
int x = 42;
int* px = &x;
int** ppx = &px;
int*** pppx = &ppx;
```

```
int x = 42;
int& rx1 = x;
int& rx2 = rx1;
int& rx3 = rx2;
```

Ссылки на ссылку не бывает

Многомерность в прикладном коде

```
$ grep -ER '\w+\*\*' .
./test/AllTests.cpp:int main(int argc, char** argv)
...
```

Указатели и ссылки на функцию

```
void call_by_ptr(void (*fptr)(int)) {
  fptr(42);
}

void call_by_ref(void (&fref)(int)) {
  fref(314);
}
```


https://godbolt.org/z/aETh3TKd3

Провисание указателей

```
int* x_ptr = nullptr;
{
  int x = 42;
  x_ptr = &x;
}
*x_ptr; // oops… dangling pointer
```

В каких случаях функция корректна?

```
Widget* create_widget();
```

В каких случаях функция корректна?

```
Widget* create_widget();
```

Ok:

- `return new Widget(...);`
- `static Widget w; return &w;`
- Возвращает указатель на глобальный объект

Error:

- Возвращает указатель на локальный объект

Возврат ссылки из свободной функции

```
Widget& create_widget();
```

Как правило, плохая идея

А так можно

```
class IndexBuilder {
public:
  const Index& index() const & { return index_; }

private:
  Index index_;
};
```

Ограничение ссылок

Нельзя создать массив или контейнер ссылок.

Нельзя создать указатель на ссылку.

Все это — ошибка компиляции:

```
int& rarr[10];
std::vector<int&> refs;
```

Как вы думаете, почему?

Ссылки и константность

```
void f(int& x);
void g(const int& x);

int a = 42;
const int b = 314;

f(a);
f(b);
g(a);
g(b);
```

Объясните, что здесь могло бы происходить

```
void f(int& x) {
    ++x;
}

int main() {
    double x = 42;
    f(x);
}
```

Это не скомпилируется

Временные объекты связываются с const ref

```
void f(const int& x) {
  // ++x;
}

int main() {
  double x = 42;
  f(x);
}
```

Это ~~как~~ special case

Кажется, здесь нужен специальный механизм

Осталось за кадром

- Полиморфизм подтипов
- rvalue-ссылки: `int&&`
- lvalue/rvalue-квалификаторы:
```
void f() &;
void f() &&;
```
- Умные указатели: `std::unique_ptr<T>`, `std::shared_ptr<T>`
- Указатели на поля и методы (`int C::* p`), операторы `.*` и `->*`

Задача

Дан контейнер строк: std::vector<std::string>.

Нужно сгруппировать их по длине: std::size_t -> ?

Вариант 1: std::size_t -> std::vector<**std::string***>

Вариант 2: std::size_t -> std::vector<**std::size_t**> (len -> [word_idx])

Вариант 3: std::vector<std::string> -> std::vector<std::string*> +
std::size_t -> std::vector<**std::string***>

https://godbolt.org/z/4eWrW5obc

Q&A