7. Ссылки

https://github.com/is-itmo-c-22/lectures/blob/main/22.10.24/Lecture 5. Reference%2C initialization%2C function%2C namespace.pdf

https://en.cppreference.com/w/cpp/language/reference

http://mesyarik.ru/19/cplusplus_course_19.pdf

Объясните идею ссылок (references). Для чего они нужны, чем они отличаются от указателей?
Что такое “передача аргументов по ссылке и по значению”? Как в C++03 реализовать функцию swap?

Попросить узнать адрес по ссылке — узнать адрес переменной. Ссылка объявляет именованную переменную как ссылку, то есть псевдоним уже существующего объекта или функции.

Ссылки отличаются от указателей тем, что не являются объектами. Поэтому не может быть массива ссылок, указателя на ссылку, ссылки на ссылку (см. пример ниже).

```
int& a[3]; // error
int&* p;   // error
int& &r;   // error
```

На самом деле можно немного поколдовать. Разрешается формировать ссылки на ссылки с помощью манипуляций с типами в шаблонах или определениях типов, и в этом случае применяются правила свертывания ссылок: ссылка rvalue на ссылку rvalue свертывается до ссылки rvalue, все остальные комбинации образуют ссылку lvalue.

Reference collapsing

It is permitted to form references to references through type manipulations in templates or typedefs, in which case the *reference collapsing* rules apply: rvalue reference to rvalue reference collapses to rvalue reference, all other combinations form lvalue reference:

```
typedef int&  lref;
typedef int&& rref;
int n;

lref&  r1 = n; // type of r1 is int&
lref&& r2 = n; // type of r2 is int&
rref&  r3 = n; // type of r3 is int&
rref&& r4 = 1; // type of r4 is int&&
```

(since C++11)

(This, along with special rules for template argument deduction when T&& is used in a function template, forms the rules that make `std::forward` possible.)

l-value references

Что значит l-value?

lvalue (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес).

rvalue определено путём исключения, говоря, что любое выражение является либо lvalue, либо rvalue. Таким образом из определения lvalue следует, что rvalue — это выражение, которое не представляет собой объект, который занимает идентифицируемое место в памяти.

Элементарные примеры

Термины, определённые выше, могут показаться немного нечёткими. Поэтому стоит сразу рассмотреть несколько простых поясняющих примеров. Предположим, мы имеем дело с переменной целого типа:

```
int var;
var = 4;
```

Оператор присваивания ожидает lvalue с левой стороны, и `var` является lvalue, потому что это объект с идентифицируемым местом в памяти. С другой стороны, следующие заклинания приведут к ошибкам:

```
4 = var;       // ERROR!
(var + 1) = 4; // ERROR!
```

Ни константа 4, ни выражение var + 1 не являются lvalue
(что автоматически их делает rvalue). Они не lvalue, потому что оба являются временным результатом выражений, которые не имеют определённого места в памяти (то есть они могут находится в каких-нибудь временных регистрах на время вычислений). Таким образом, присваивание в данном случае не несёт в себе никакого семантического смысла. Иными словами — некуда присваивать.

Изменяемые lvalue

Изначально, когда понятие *lvalue* было введено в C, оно буквально означало «выражение, применимое с левой стороны оператора присваивания». Однако позже, когда ISO C добавило ключевое слово `const`, это определение нужно было доработать. Действительно:

```
const int a = 10; // 'a' - lvalue
a = 10;           // но ему не может быть присвоено значение!
```

Таким образом не всем lvalue можно присвоить значение. Те, которым можно, называются *изменяемые lvalue* (modifiable lvalues). Формально C99 стандарт определяет изменяемые lvalue как:

> [...] lvalue, тип которого не является массивом, не является неполным, не имеет спецификатор `const`, не является структурой или объединением, содержащими поля (также включая поля, рекурсивно вложенные в содержащиеся агрегаты и объединения) со спецификатором `const`.

Символ "&" играет несколько другую роль в C++ — он позволяет определить ссылочный тип. Его называют «ссылкой на lvalue». Неконстантной ссылке на lvalue не может быть присвоено rvalue, так как это потребовало бы неверное rvalue-в-lvalue преобразование:

```
std::string& sref = std::string();  // ОШИБКА: неверная инициализация
                                    // неконстантной ссылки типа 'std::string&'
                                    // rvalue типа 'std::string'
```

Константным ссылкам на lvalue *можно* присвоить rvalue. Так как они константы, значение не может быть изменено по ссылке и поэтому проблема модификации rvalue просто отсутствует. Это свойство делает возможным одну из основополагающих идиом C++ — допуск значений по константной ссылке в качестве аргументов функций, что позволяет избежать необязательного копирования и создания временных объектов.

Как выглядит l-value и r-value ссылки?

& *attr*(optional) *declarator*	(1)	
&& *attr*(optional) *declarator*	(2)	(since C++11)

1) **Lvalue reference declarator**: the declaration `S& D;` declares `D` as an *lvalue reference* to the type determined by *decl-specifier-seq* S.

2) **Rvalue reference declarator**: the declaration `S&& D;` declares `D` as an *rvalue reference* to the type determined by *decl-specifier-seq* S.

Reference (lvalue reference)

- “Псевдоним” для уже существующего объекта
- Обязательно инициализирован
- Не занимает дополнительную память
- Нельзя сделать указатель на ссылку
- Продлевают “жизнь” временным переменным

```
int main() {
   int i = 10;
   int& j = i;
   //int& k;  // error: ‘k’ declared as reference but not initialized
   j = 20;
   std::cout << i << std::endl;
   std::cout << &i << "  " << &j << std::endl;

   const int& r = i;
   // r = 21; // error assignment of read-only reference ‘r’
   return 0;
}
```

Функции и ссылки

Reference

```
int& foo() {
  int i = 20;
  return i;
}

int main() {
  int& x = foo();
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9 ,10, 11, 12};

  std::cout << x; // !OOps
  return 0;
}
```

Однако, не все присваивания результату вызова функции ошибочны. Например, использование ссылок в C++ делает это возможным:

```
int globalvar = 20;

int& foo()
{
    return globalvar;
}

int main()
{
    foo() = 10;
    return 0;
}
```

Здесь `foo` возвращает ссылку, *которая является lvalue*, то есть ей можно придать значение. Вообще, в C++ возможность возвращать lvalue, как результат вызова функции, существенна для реализации некоторых перегруженных операторов. Как пример приведём перегрузку оператора `[]` в классах, которые реализуют доступ по результатам поиска. Например `std::map` :

```
std::map<int, float> mymap;
mymap[10] = 5.6;
```

Присваивание `mymap[10]` работает, потому что неконстантная перегрузка `std::map::operator[]` возвращает ссылку, которой может быть присвоено значение.

Передача аргументов в функцию

- By reference
  - В общем случае наиболее “дешевый” и простой способ
- By pointer
- By value
  - Для встроенных и небольших типов

Передача аргументов в функцию

```
struct SomeStruct ;

void funcA(const SomeStruct& value); // by reference
void funcB(const SomeStruct& value); // by reference
void funcC(SomeStruct* value);       // by pointer
void funcD(int value);               // by value
```

dangling reference

```
std::string& f()
{
    std::string s = "Example";
    return s; // exits the scope of s:
              // its destructor is called and its storage deallocated
}

std::string& r = f(); // dangling reference
std::cout << r;       // undefined behavior: reads from a dangling reference
std::string s = f();  // undefined behavior: copy-initializes from a dangling reference
```

Функция swap

```
void swap(int& i, int& j) {
   int temp = i;
   i = j;
   j = temp;
}
```

Аналогично следующему коду в плане результата. Только при этом не выделяется память.

```
void swap(int* pi, int* pj) {
   int temp = *pi;
   *pi = *pj;
   *pj = temp;
}
```