Передаём данные в функцию — конспект темы

По ссылке или по значению

Передача по ссылке должна быть обоснована. При прочих равных выбирайте передачу по значению.

Обоснования передачи по ссылке:

- **Объект трудно копировать**. Его размер на стеке больше 16 байт или он владеет динамической памятью, которая будет перевыделена при копировании.
- **Копирование объекта имеет побочный эффект**. Например, класс односвязного списка может уместиться в 16 байт, но копирование приведёт к обходу списка и выделению динамической памяти.
- Класс лёгкий, но планируется расширение.

Иногда назначение функции — поменять объект, переданный в аргументе, но не принимать владение. Тут всё однозначно — принимать нужно по неконстантной ссылке.

Когда параметр функции или метода необязателен, можно задать значение по умолчанию:

```
void PrintStat(std::ostream& out = std::cout);
```

Два таких вызова эквивалентны:

```
PrintStat();
PrintStat(cout);
```

`optional` явно показывает, что передача объекта необязательна:

```
// Можно указать письмо, если его нужно отправить по дороге в магазин.
// Но это необязательно:
void GoToShop(Shop& destination, const vector<Food>& shopping_list,
              std::optional<Letter> letter_to_send = std::nullopt);
```

Указатель похож на ссылку по возможностям и внутреннему устройству. В отличие от ссылки, он позволяет передать `nullptr`, то есть отсутствие объекта.

Совершенный способ — Forwarding reference

С увеличением количества аргументов количество необходимых функций растёт экспоненциально. Но есть в C++ способ исправить эту ситуацию — **Forwarding reference**, или **Forwarding-ссылка**:

```
template <typename Object>
void SaveObject(Object&& object);
```

Правило поглощения ссылок: обычная ссылка побеждает rvalue-ссылку при одновременном применении.

Для Forwarding reference требуется универсальный способ — функция `std::forward` из файла `<utility>`. В отличие от `move`, ей нужно обязательно указывать шаблонный аргумент:

```
#include <utility>

using namespace std;

//...

template <typename T1, typename T2>
void SaveObject(T1&& object1, T2&& object2) {
    object1_ = forward<T1>(object1);
    object2_ = forward<T2>(object2);
}

// ...
```

Такая реализация одинаково хорошо работает для временных объектов, постоянных, перемещаемых и неперемещаемых.

Недостатки Forwarding reference: невнятная сигнатура, запутанные сообщения об ошибках.

Использование Forwarding reference в паре с `forward` — это стандартный паттерн, когда функция должна передать свой аргумент дальше, не задумываясь о его типе, либо сохранить его в классе. Правильная реализация функции, вызывающей конструктор произвольного типа:

```
template <typename T, typename S>
T Construct(S&& arg) {
    return T(std::forward<S>(arg));
}

int main() {
    string s = Construct<string>("abc");
}
```

Если передавать владение объектом не требуется, выбирайте между передачей по ссылке или значению, исходя из легкости копирования объекта. Для неизвестного или шаблонного типа без передачи владения всегда выбирайте ссылку.

C++20 Ranges

Библиотека `ranges` позволяет передавать один объект вместо пары итераторов.

Для работы с `ranges` нужен компилятор, например GCC версии не ниже 10. Скачать GCC для Windows можно по ссылке http://winlibs.com. Протестировать `ranges` можно онлайн на сайте Coliru.

Эта программа сортирует и выводит в `cout` диапазон чисел:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ranges>
#include <string_view>

namespace rng = std::ranges;
using namespace std;

template <rng::input_range Range>
void Print(const Range& range) {
    for (const auto& x : range) {
        cout << x << " "sv;
    }
    cout << endl;
}

int main() {
    vector v = { 4, 1, 7, 2, 3, 8 };
    rng::sort(v);
    Print(v);

    return 0;
}
```

Программа, собранная с ключом `-std=c++20`, выводит такой текст:

```
1 2 3 4 7 8
```

Типовой параметр — параметр шаблона, значение которого — тип данных. В списке параметров шаблона он обозначается словами `class` или `typename`.

Три способа передачи функций

1. Шаблонный параметр

Иногда параметром одного действия бывает другое действие. В программировании это выражается функциональным параметром. Например, алгоритм `sort` допускает компаратор:

```
template <typename RandomIt, typename Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
```

Компаратор может быть любым объектом, допускающим вызов с двумя аргументами нужного типа.

2. Указатель на функцию

Указатели на функции делают сигнатуру понятнее. Например, функция, сортирующая `int`, будет выглядеть так:

```
template <typename RandomIt>
void SortInt(RandomIt begin, RandomIt end, bool (*comp)(int x, int y));
```

Название параметра `comp`, вопреки обыкновению, записывается не после типа, а внутри. У этого параметра будет тип указателя на функцию, возвращающую `bool` и принимающую два параметра `int`. Запись получится более естественной, если создать для типа этого компаратора псевдоним: `bool (*)(int x, int y)`:

```
using IntComparator = bool (*)(int x, int y);

template <typename RandomIt>
void SortInt(RandomIt begin, RandomIt end, IntComparator comp);
```

3. std::function

Класс `std::function` из файла `<functional>` позволяет сделать указатель на функцию универсальным, способным принять любые компараторы и функциональные объекты. Единственный шаблонный параметр класса `std::function` — тип указателя на функцию.

С `std::function` сигнатура универсальной функции, сортирующей диапазон чисел, выглядит так:

```
template <typename RandomIt>
void SortInt(RandomIt begin, RandomIt end, const function<bool(int, int)>& comp);
```

Из неё сразу видно, что третий параметр должен быть функциональным объектом, принимающим два значения `int` и возвращающим `bool`. Такая сигнатура очень выразительна.

Можно сделать функцию `SortFuncPtr` универсальной, добавив в неё `function`:

```
template <typename RandomIt>
void SortFunction(RandomIt first, RandomIt last, const function<ComparatorByIterator<RandomIt>>& comp) {
    sort(first, last, ref(comp));
}
```