1 Новые возможности C++17

В этой главе:

- применение структурированных привязок (декомпозиции) для распаковки набора возвращаемых значений;
- ограничение области видимости переменных в выражениях `if` и `switch`;
- новые правила инициализатора с фигурными скобками;
- разрешение конструктору автоматически вывести полученный тип класса шаблона;
- упрощение принятия решений во время компиляции с помощью `constexpr-if`;
- подключение библиотек, перечисленных в заголовочных файлах, с использованием встраиваемых переменных;
- реализация вспомогательных функций с помощью выражений свертки.

Введение

Функциональность языка C++ значительно расширилась с выходом C++11, C++14 и недавней версии C++17. На текущий момент он совсем не похож на себя образца десятилетней давности. Стандарт C++ упорядочивает не только язык, но и STL.

В этой книге на большом количестве примеров показаны наилучшие способы использования возможностей STL. Но для начала в текущей главе мы сконцентрируемся на самых важных особенностях языка. Изучив их, вы сможете писать легко читаемый, удобный в сопровождении и выразительный код.

Мы рассмотрим, как получить доступ к отдельным элементам пар, кортежей и структур с помощью структурированных привязок и ограничить область видимости переменных благодаря новым возможностям по инициализации переменных внутри выражений `if` и `switch`. Синтаксические двусмысленности, появившиеся в C++11 из-за нового синтаксиса инициализатора с фигурными скобками, кото-

рый выглядит так же, как синтаксис списков инициализаторов, были исправлены в *новых правилах инициализатора с фигурными скобками*. Точный *тип* экземпляра шаблонного класса может быть *определен* по аргументам, переданным его конструктору, а если разные специализации шаблонного класса выполняются в разном коде, то это легко выразить с помощью `constexpr-if`. Обработка переменного количества параметров в шаблонных функциях значительно упростилась благодаря новым *выражениям свертки*. Наконец, стало гораздо удобнее определять доступные глобально статические объекты в библиотеках, указанных в заголовочных файлах, благодаря новой возможности объявлять встраиваемые переменные, что ранее было выполнимо только для функций.

Отдельные примеры данной главы могут оказаться более интересными для тех, кто реализует библиотеки, нежели для тех, кто пишет приложения. Для полноты картины мы рассмотрим несколько свойств, но вам не обязательно разбираться со всеми примерами главы прямо сейчас, чтобы понять остальной материал этой книги.

Применяем структурированные привязки (декомпозицию) для распаковки набора возвращаемых значений

В C++17 появилась новая возможность, объединяющая синтаксический сахар и автоматическое определение типа, — *структурированные привязки*. Эта функция помогает присваивать отдельные значения пар, кортежей и структур отдельным переменным. В других языках программирования этот механизм называется *распаковкой*.

Как это делается

Применение декомпозиции для присвоения значений нескольким переменным на основе одной упакованной структуры всегда выполняется за один шаг. Сначала рассмотрим, как это делалось до появления C++17. Затем взглянем на несколько примеров, в которых показаны способы воплощения этого в C++17.

1. Получаем доступ к отдельным значениям `std::pair`. Представьте, что у нас есть математическая функция `divide_remainder`, которая принимает в качестве параметров *делимое* и *делитель* и возвращает частное и остаток в `std::pair`.

   ```
   std::pair<int, int> divide_remainder(int dividend, int divisor);
   ```

   Рассмотрим следующий способ получения доступа к отдельным значениям полученной пары.

   ```
   const auto result (divide_remainder(16, 3));
   std::cout << "16 / 3 is "
             << result.first << " with a remainder of "
             << result.second << '\n';
   ```

Вместо выполнения действий, показанных во фрагменте выше, мы теперь можем присвоить отдельные значения конкретным переменным с говорящими именами, что более удобочитаемо:

```
auto [fraction, remainder] = divide_remainder(16, 3);
std::cout << "16 / 3 is "
          << fraction << " with a remainder of "
          << remainder << '\n';
```

2. Структурированные привязки работают и для `std::tuple`. Рассмотрим следующий пример функции, которая возвращает информацию о ценах на акции:

```
std::tuple<std::string,
           std::chrono::system_clock::time_point, unsigned>
stock_info(const std::string &name);
```

Присваивание результата ее работы отдельным переменным выглядит так же, как и в предыдущем примере:

```
const auto [name, valid_time, price] = stock_info("INTC");
```

3. Декомпозицию можно применять и для пользовательских структур. В качестве примера создадим следующую структуру.

```
struct employee {
    unsigned id;
    std::string name;
    std::string role;
    unsigned salary;
};
```

Теперь можно получить доступ к ее членам с помощью декомпозиции. Мы даже можем сделать это в цикле, если предполагается наличие целого вектора таких структур:

```
int main()
{
    std::vector<employee> employees {
        /* Инициализируется в другом месте */};
    for (const auto &[id, name, role, salary] : employees) {
        std::cout << "Name: "   << name
                  << "Role: "   << role
                  << "Salary: " << salary << '\n';
    }
    }
```

Как это работает

Структурированные привязки всегда применяются по одному шаблону:

```
auto [var1, var2, ...] = <выражение пары, кортежа, структуры или массива>;
```

- Количество переменных `var1`, `var2`... должно точно совпадать с количеством переменных в выражении, в отношении которого выполняется присваивание.

- Элементом `<выражение пары, кортежа, структуры или массива>` должен быть один из следующих объектов:
  - `std::pair`;
  - `std::tuple`;
  - структура. Все члены должны быть *нестатическими* и определенными в *одном базовом классе*. Первый объявленный член присваивается первой переменной, второй член — второй переменной и т. д.;
  - массив фиксированного размера.
- Тип может иметь модификаторы `auto`, `const auto`, `const auto&` и даже `auto&&`.

При необходимости пользуйтесь ссылками, а не создавайте копии. Это важно не только с точки зрения производительности.

Если в квадратных скобках вы укажете *слишком мало* или *слишком много* переменных, то компилятор выдаст ошибку.

```
std::tuple<int, float, long> tup {1, 2.0, 3};
auto [a, b] = tup; // Не работает
```

В этом примере мы пытаемся поместить кортеж с тремя переменными всего в две переменные. Компилятор незамедлительно сообщает нам об ошибке:

```
error: type 'std::tuple<int, float, long>' decomposes into 3 elements, but
only 2 names were provided
auto [a, b] = tup;
```

Дополнительная информация

С помощью структурированных привязок вы точно так же можете получить доступ к большей части основных структур данных библиотеки STL. Рассмотрим, например, цикл, который выводит все элементы контейнера `std::map`:

```
std::map<std::string, size_t> animal_population {
    {"humans",   7000000000},
    {"chickens", 17863376000},
    {"camels",   24246291},
    {"sheep",    1086881528},
    /* … */
};

for (const auto &[species, count] : animal_population) {
    std::cout << "There are " << count << " " << species
              << " on this planet.\n";
}
```

Пример работает потому, что в момент итерации по контейнеру `std::map` мы получаем узлы `std::pair<const key_type, value_type>` на каждом шаге этого

процесса. Именно эти узлы распаковываются с помощью структурированных привязок (`key_type` представляет собой строку с именем `species`, а `value_type` — переменную `count` типа `size_t`), что позволяет получить к ним доступ по отдельности в теле цикла.

До появления C++17 аналогичного эффекта можно было достичь с помощью `std::tie`:

```
int remainder;
std::tie(std::ignore, remainder) = divide_remainder(16, 5);
std::cout << "16 % 5 is " << remainder << '\n';
```

Здесь показано, как распаковать полученную пару в две переменные. Применение контейнера `std::tie` не так удобно, как использование декомпозиции, ведь нам надо *заранее* объявить все переменные, которые мы хотим связать. С другой стороны, пример демонстрирует преимущество `std::tie` *перед* структурированными привязками: значение `std::ignore` играет роль переменной-пустышки. В данном случае частное нас не интересует и мы отбрасываем его, связав с `std::ignore`.

Когда мы применяем декомпозицию, у нас нет переменных-пустышек tie, поэтому нужно привязывать все значения к именованным переменным. Это может оказаться неэффективным, если позже не задействовать некоторые переменные, но тем не менее компилятор может оптимизировать неиспользованное связывание.

Раньше функцию `divide_remainder` можно было реализовать следующим образом, используя выходные параметры:

```
bool divide_remainder(int dividend, int divisor,
                      int &fraction, int &remainder);
```

Получить к ним доступ можно так:

```
int fraction, remainder;
const bool success {divide_remainder(16, 3, fraction, remainder)};
if (success) {
    std::cout << "16 / 3 is " << fraction << " with a remainder of "
              << remainder << '\n';
}
```

Многие все еще предпочитают делать именно так, а не возвращать пары, кортежи и структуры. При этом они приводят следующие аргументы: код работает *быстрее*, поскольку мы не создаем промежуточные копии этих значений. Но для современных компиляторов это *неверно* — они изначально оптимизированы так, что подобные копии не создаются.

Помимо того что аналогичной возможности нет в языке C, возврат сложных структур в качестве выходных параметров долгое время считался медленным, поскольку объект сначала нужно инициализировать

в возвращающей функции, а затем скопировать в переменную, которая должна будет содержать возвращаемое значение на вызывающей стороне. Современные компиляторы поддерживают оптимизацию возвращаемых значений (return value optimization, RVO), что позволяет избежать создания промежуточных копий.

Ограничиваем область видимости переменных в выражениях if и switch

Максимальное ограничение области видимости переменных считается хорошим тоном. Иногда, однако, переменная должна получить какое-то значение, а потом нужно его проверить на соответствие тому или иному условию, чтобы продолжить выполнение программы. Для этих целей в C++17 была введена инициализация переменных в выражениях `if` и `switch`.

Как это делается

В данном примере мы воспользуемся новым синтаксисом в обоих контекстах, чтобы увидеть, насколько это улучшит код.

- Выражение `if`. Допустим, нужно найти символ в таблице символов с помощью метода `find` контейнера `std::map`:

```
if (auto itr (character_map.find(c)); itr != character_map.end()) {
    // *itr корректен. Сделаем с ним что-нибудь.
} else {
    // itr является конечным итератором. Не разыменовываем.
}
// здесь itr недоступен
```

- Выражение `switch`. Так выглядит код получения символа из пользовательского ввода и его одновременная проверка в выражении `switch` для дальнейшего управления персонажем компьютерной игры:

```
switch (char c (getchar()); c) {
    case 'a': move_left();  break;
    case 's': move_back();  break;
    case 'w': move_fwd();   break;
    case 'd': move_right(); break;
    case 'q': quit_game();  break;

    case '0'...'9': select_tool('0' - c); break;

    default:
        std::cout << "invalid input: " << c << '\n';
}
```

Как это работает

Выражения `if` и `switch` с инициализаторами по сути являются синтаксическим сахаром. Два следующих фрагмента кода эквивалентны:

До C++17:

```
{
    auto var (init_value);
    if (condition) {
        // Ветвь A. К переменной var можно получить доступ
    } else {
        // Ветвь B. К переменной var можно получить доступ
    }
    // К переменной var все еще можно получить доступ
}
```

Начиная с C++17:

```
if (auto var (init_value); condition) {
    // Ветвь A. К переменной var можно получить доступ
} else {
    // Ветвь B. К переменной var можно получить доступ
}
// К переменной var больше нельзя получить доступ
```

То же верно и для выражений `switch`.

До C++17:

```
{
    auto var (init_value);
    switch (var) {
    case 1: ...
    case 2: ...
    ...
    }
    // К переменной var все еще можно получить доступ
}
```

Начиная с C++17:

```
switch (auto var (init_value); var) {
case 1: ...
case 2: ...
  ...
}
// К переменной var больше нельзя получить доступ
```

Благодаря описанному механизму область видимости переменной остается минимальной. До C++17 этого можно было добиться только с помощью дополнительных фигурных скобок, как показано в соответствующих примерах. Короткие жизненные циклы уменьшают количество переменных в области видимости, что позволяет поддерживать чистоту кода и облегчает рефакторинг.