```
/*============================================================================*/
/*                      Перечислимые типы и оператор switch                   */
/*============================================================================*/
```

Иногда возникает необходимость иметь переменную, принимающую значения из заранее известного ограниченного набора.

Предположим, например, что нам необходимо написать функцию, обрабатывающую запросы ко множеству:

```
void ProcessRequest(
  set<int>& numbers,
  /* ??? */ request_type,
  int request_data) {
  if (/* запрос на добавление */) {
    numbers.insert(request_data);
  } else if (/* запрос на удаление */) {
    numbers.erase(request_data);
  } else if (/* запрос на умножение на -1 */) {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, /* добавить */, 8);
ProcessRequest(numbers, /* умножить на -1 */, 8);
ProcessRequest(numbers, /* удалить */, -8);
```

Каким стоит выбрать тип параметра request_type? Рассмотрим несколько вариантов, а также их плюсы и минусы.

Способ 1. Тип запроса - строка

```
void ProcessRequest(
  set<int>& numbers,
  const string& request_type,
  int request_data) {
  if (request_type == "ADD") {
    numbers.insert(request_data);
  } else if (request_type == "REMOVE") {
    numbers.erase(request_data);
  } else if (request_type == "NEGATE") {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, "ADD", 8);
ProcessRequest(numbers, "NEGATE", 8);
ProcessRequest(numbers, "REMOVE", -8);
```

Безусловно, такой код легко читается. Однако он обладает следующими серьёзными недостатками:

- Тип string предназначен для хранения произвольных строк, мы же храним в нём несколько заранее известных строк. Как следствие, мы тратим лишнюю память на хранение наборов символов, а также лишнее время на сравнение строк в функции ProcessRequest.
- При вызове функции легко опечататься в типе запроса. В этом случае вместо, например, ошибки компиляции запрос просто проигнорируется. Эту проблему можно было бы решить выбрасыванием исключения "Unknown request" из функции ProcessRequest, но и в этом случае ошибка не будет обнаружена на этапе компиляции. (Важно помнить, что чем раньше обнаружена ошибка, тем лучше. Идеально, когда ошибки удаётся обнаружить на этапе компиляции.)

Способ 2. Тип запроса – число

```
void ProcessRequest(
  set<int>& numbers,
  int request_type,
  int request_data) {
  if (request_type == 0) {
    numbers.insert(request_data);
  } else if (request_type == 1) {
    numbers.erase(request_data);
  } else if (request_type == 2) {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, 0, 8);
ProcessRequest(numbers, 2, 8);
ProcessRequest(numbers, 1, -8);
```

Такой вариант выигрывает у предыдущего в эффективности, но заметно проигрывает в понятности: без комментариев очень непросто понять, что означают числа 0, 1 и 2 во втором аргументе функции ProcessRequest.

Большей понятности кода легко достичь, объявив именованные константы для различных типов запросов:

```
const int REQUEST_ADD = 0;
const int REQUEST_REMOVE = 1;
const int REQUEST_NEGATE = 2;

void ProcessRequest(
  set<int>& numbers,
  int request_type,
  int request_data) {
  if (request_type == REQUEST_ADD) {
    numbers.insert(request_data);
  } else if (request_type == REQUEST_REMOVE) {
    numbers.erase(request_data);
  } else if (request_type == REQUEST_NEGATE) {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, REQUEST_ADD, 8);
ProcessRequest(numbers, REQUEST_NEGATE, 8);
ProcessRequest(numbers, REQUEST_REMOVE, -8);
```

(Префикс REQUEST_ в названиях констант необходим для того, чтобы не занимать популярные названия ADD, REMOVE и NEGATE. Например, ADD и NEGATE могут быть использованы в качестве названий арифметических операций.)

Этот вариант читается лучше предыдущего, но и у него есть свои проблемы:

- При вызове функции ProcessRequest можно опечататься и перепутать второй и третий аргументы: передать тип запроса в качестве значения и наоборот. В этом случае вместо ошибки компиляции мы получим обработку совершенно другого запроса, возможно, формально корректного. Ошибки такого рода очень тяжело искать.
- Компилятор не будет препятствовать использованию вместо констант чисел в явном виде: вызов `ProcessRequest(numbers, 0, 8)` будет считаться корректным.

- Константу, например, REQUEST_REMOVE можно умножить на 2 как обычное число.

Эти проблемы мы решим использованием для типа запроса отдельного перечислимого типа данных (enum).

Способ 3. Тип запроса – перечислимый тип (enumeration)

```
enum class RequestType {
  ADD,
  REMOVE,
  NEGATE
};

void ProcessRequest(
  set<int>& numbers,
  RequestType request_type,
  int request_data) {
  if (request_type == RequestType::ADD) {
    numbers.insert(request_data);
  } else if (request_type == RequestType::REMOVE) {
    numbers.erase(request_data);
  } else if (request_type == RequestType::NEGATE) {
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
  }
}

ProcessRequest(numbers, RequestType::ADD, 8);
ProcessRequest(numbers, RequestType::NEGATE, 8);
ProcessRequest(numbers, RequestType::REMOVE, -8);
```

Мы объявили тип RequestType, имеющий три возможных значения: ADD, REMOVE, NEGATE. Обратите внимание на три аспекта использования этого типа:

1. Объявление типа осуществляется с помощью ключевых слов enum class. В фигурных скобках указываются идентификаторы возможных значений типа.
2. Сам тип называется RequestType: его можно использовать в качестве типа параметра функции, типа переменной и т. д.
3. Значения типа RequestType должны предваряться префиксом "RequestType::". Соответственно, вызов функции **ProcessRequest**(numbers, ADD, 8) не скомпилируется и имя ADD остаётся свободным.

Перечислимые типы основаны на целочисленных типах и благодаря этому столь же эффективны. Однако RequestType и целочисленные типы не будут неявно преобразовываться друг к другу. Как следствие, вызов функции **ProcessRequest**(numbers, 8, RequestType::ADD) с перепутанными аргументами не скомпилируется.

Значения одного перечислимого типа (например, RequestType) можно сравнивать друг с другом не только с помощью == и !=, но и с помощью < и >. Благодаря этому значения перечислимых типов можно использовать в качестве элементов множеств или ключей словарей. Порядок между значениями соответствует порядку их определения при объявлении типа. Например, в нашем примере RequestType::ADD < RequestType::REMOVE и RequestType::REMOVE < RequestType::NEGATE.

Оператор switch

На примере функции ProcessRequest рассмотрим более компактную альтернативу цепочке условных операторов – оператор switch. В отличие от if, оператор switch не позволяет проверять произвольные логические выражения. Он позволяет сравнить заданную переменную (или результат выражения) с различными конкретными значениями и выполнить различные действия в зависимости от того, с каким значением произошло совпадение.

Перепишем функцию ProcessRequest, используя оператор switch:

```
void ProcessRequest(
  set<int>& numbers,
  RequestType request_type,
  int request_data) {
  switch (request_type) {
  case RequestType::ADD:
    numbers.insert(request_data);
    break;
  case RequestType::REMOVE:
    numbers.erase(request_data);
    break;
```

```
  case RequestType::NEGATE:
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
    break;
  }
}
```

Обратите внимание, что каждая case-ветка должна оканчиваться оператором break и не нуждается в обрамлении фигурными скобками. Оператор break здесь означает выход из оператора switch и не повлечёт за собой выхода из объемлющего цикла for при его наличии.

Аналог else для оператора switch - ветка default. Предположим, например, что мы хотим перестраховаться от случаев добавления новых типов запросов и добавить вывод предупреждающего сообщения для неизвестного запроса:

```
void ProcessRequest(
  set<int>& numbers,
  RequestType request_type,
  int request_data) {
  switch (request_type) {
  case RequestType::ADD:
    numbers.insert(request_data);
    break;
  case RequestType::REMOVE:
    numbers.erase(request_data);
    break;
  case RequestType::NEGATE:
    if (numbers.count(request_data) == 1) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
    break;
  default:
    cout << "Unknown request" << endl;
  }
}
```

default-ветка выполнится всегда, если не подошла ни одна case-ветка.

Важная особенность оператора switch заключается в том, что при необходимости объявить переменную в одной из его веток всю ветку придётся заключить в блок из фигурных скобок:

```
void ProcessRequest(
  set<int>& numbers,
  RequestType request_type,
  int request_data) {
  switch (request_type) {
  case RequestType::ADD:
    numbers.insert(request_data);
    break;
  case RequestType::REMOVE:
    numbers.erase(request_data);
    break;
  case RequestType::NEGATE: {  // фигурные скобки обязательны
    bool contains = numbers.count(request_data) == 1;
    if (contains) {
      numbers.erase(request_data);
      numbers.insert(-request_data);
    }
    break;
  }
  default:
    cout << "Unknown request" << endl;
  }
}
```

Числовые значения элементов enum

Мы вскользь упоминали о связи перечислимых и целочисленных типов. Продемонстрируем

её, приведя значения типа RequestType к int с помощью static_cast:

```
// Выведет 0
cout << static_cast<int>(RequestType::ADD) << endl;

// Выведет 1
cout << static_cast<int>(RequestType::REMOVE) << endl;

// Выведет 2
cout << static_cast<int>(RequestType::NEGATE) << endl;
```

Как видим, значения типа RequestType кодируются целыми числами подряд, начиная с 0. В нашем случае эта нумерация не имела значения, но в тех случаях, когда числовые значения элементов enum важны, их можно указать явно:

```
enum class RequestType {
  ADD = 9,
  REMOVE = 8,
  NEGATE = 7
};
```

Это удобно в случае, когда типы запросов поступают на вход программы в виде чисел 9, 8 и 7. Тогда получить объект типа RequestType по его числовому коду можно будет простым оператором static_cast:

```
int request_code;
cin >> request_code;
auto request_type = static_cast<RequestType>(request_code);
// Если ввести request_code = 7,
// в переменной request_type окажется RequestType::NEGATE
```

Заметим, что если преобразуемому числовому коду не будет соответствовать ни один элемент перечислимого типа, ошибки компиляции не произойдёт, но дальнейшее поведение программы может оказаться непредсказуемым.