

VII

Продолжаем изучать возможности для
определения пользовательских типов
На очереди — **объединения** (**union**)

Объединение представляет собой составной тип данных, в котором в один момент времени может храниться только единственное значение конкретного поля. Технически, при создании переменной-объединения C++ выделяет хранилище, размер которого совпадает **с самым большим размером из возможных подхранилищ**.

Скорее всего, при решении прикладных задач именно этот составной тип понадобится редко. Тем не менее, если кому-нибудь придётся разбираться/играться/пробовать программирование под сильно ограниченные по ресурсам системы (микроконтроллеры, различные автономные датчики) данный тип может пригодиться. Да и в целом будет полезно иметь представление, что за варианты язык предоставляет для собственных типов.

Общий синтаксис определения **объединения**:

```
union [<название_нового_типа>]
{
    <тип_1> <поле_1> [, <поле_2>, ...];
    <тип_2> <поле_1> [, <поле_2>, ...];
    ...
    <тип_n> <поле_1> [, <поле_2>, ...];
} [переменная1, переменная2, ...];
```

Есть одно ограничение: в качестве типов полей нельзя использовать **ССЫЛКИ**.

И практический пример:

```
1 union IntWithDouble
2 {
3     int integer;
4     double real;
5 };
6
7 IntWithDouble var;
```

После структур в самом определении типа никаких особенностей нет. А вот, что происходит при объявлении переменной **var**:

- компилятор смотрит, какой из типов полей создаёт хранилище большего размера;
- очевидно, побеждает тип **double**;
- поэтому размер хранилища под переменную **var** будет равен размеру типа **double**.

Составные типы. Объединения I

Чтобы работать с переменной данного типа, необходимо задать значение для одного из полей. Иначе возникает ситуация *неопределённого поведения* (undefined behaviour), когда сама спецификация языка C++ ничего говорит о том, какие значения должны быть получены. В этом случае всё зависит от разработчика компилятора — какие алгоритмы были заложены в него.

Пример сохранения значений в переменной-объединении:

```
1 IntWithDouble var;  
2  
3 //Делаем "активным" поле integer  
4 var.integer = 6;  
5 printf("int value is %d\n", var.integer);  
6  
7 // Неопределённое поведение  
8 double r = var.real + 3.0;  
9
```

Составные типы. Объединения II

```
10 //Делаем "активным" поле real  
11 var.real = 1.5;  
12 printf("real value is %.2f\n", var.real);
```

Служка за тем, какое поле активно у переменной в данный момент, полностью прерогатива того, кто пишет код. Компилятор, в общем случае, не даёт никаких подсказок по корректной работе с переменными-объединениями.

Составные типы. Объединения

Переменные-объединения можно инициализировать аналогично структурам, но с одним ограничением: **нельзя присваивать значения** сразу обоим полям.

```
1 IntWithDouble var2{101}; // Всё ок
2
3 // Ошибка компиляции
4 // IntWithDouble var3{101, 4.5};
5
6 printf("int value is %d\n", var2.integer);
```

В дополнение:

- копирование осуществляется аналогично фундаментальным переменным: значение одного хранилища копируется в другое;
- также существует возможность создания *анонимных объединений*. Синтаксис аналогичен структурному типу;
- без проблем могут выступать в виде параметров функций.

Относительно интересный исторический пример для демонстрации объединения: получить значение каждого байта у значения типа **double** в двоичном виде.

Идея реализации заключается в следующем:

- создадим объединение, состоящее из поля типа **double** и массива беззнакового **char**, размер которого совпадает со значением **sizeof(double)**;
- записав в переменную какое-нибудь число с плавающей точкой, проанализируем второе поле;
- поскольку каждый элемент массива представляет собой один байт, надо сравнить его со степенями двойки от 7 до нуля. Если соответствующие биты байта и степени двойки совпадают, то выводим единицы, иначе — ноль. Таким образом получаем двоичное представление конкретного байта.

Составные типы. Объединения

Код следующий:

```
1 union Double
2 {
3     double value;
4     unsigned char bytes[sizeof(double)];
5 };
6
7 void show_bits(const Double item)
8 {
9     for (int j = sizeof(double) - 1; j >= 0; j--) {
10         printf("%d bytes in binary: ", j + 1);
11         for (size_t i = 128; i >= 1; i /= 2) {
12             if (i & item.bytes[j]) { printf("1"); }
13             else { printf("0"); }
14         }
15         printf("\n");
16     }
17 }
```


Применение:

```
1 Double var {-444.23323};  
2  
3 show_bits(var);
```

К сожалению, этот трюк строится на неопределённом поведении: по стандарту мы не можем ожидать при обращении к массиву **char**'ов, что нам вернут именно значения байтов, которые были записаны при инициализации переменной-объединения конкретным числом. Тем не менее, во-первых, большинство современных компиляторов так работают, во вторых, пример демонстрирует получение всех битов конкретного байта.

Далее — **перечисления** (enumerations)


Перечисления - это пользовательский тип данных, состоящий из *ограниченного* набора констант **целого типа**. По умолчанию, *базовым типом* каждой константы является **int**. В современном C++ перечисления делятся на



Открытые (unscoped) -

каждая константа становится доступной глобально по имени и допускается неявное приведение значений констант к значениям целочисленных типов. Ключевое слово для объявления:

enum



Закрытые (scoped) -

каждая константа доступна только через название перечисления с использованием оператора :: и своего имени. Допускаются только **явные** преобразования в значения целочисленных типов. Ключевое слово для объявления:

enum class

Синтаксис определения перечисления:

```
enum <название_типа> [: <целочисленный тип>]  
{  
    <константа_1> [= <значение_1>],  
    [<константа_2>, <константа_3>, ...]  
};
```

- 1 по умолчанию значение первой константы перечисления равно **нулю**;
- 2 каждая константа, кроме первой, получает **на единицу большее значение**, чем предшествующая;
- 3 каждой константе может быть присвоено **произвольное значение целого типа**;
- 4 как только константе присваивается значение, то все следующие за ней меняются по **второму пункту**;
- 5 разные константы могут иметь **одинаковые значения**;

Синтаксис определения перечисления:

```
enum <название_типа> [: <целочисленный тип>]  
{  
    <константа_1> [= <значение_1>],  
    [<константа_2>, <константа_3>, ...]  
};
```

- 6 опционально, можно задать тип («<целочисленный тип>» в определении выше), который будет использован для хранения каждой константы.

Пример типа-перечисления:

```
1 enum ComputingState
2 {
3     NOT_STARTED, // значение — 0
4     STARTED,     // 1
5     COMPLETED   // 2
6 };
```

- в строках **1 — 6** определяется новый тип;
- набор значений этого типа представлен тремя именованными константами: строки **3 — 5**;
- поскольку первой константе не задано никакого значения, по умолчанию она становится равной **нулю**. Следующие константы по порядку получают свои значения, в соответствии с правилом на 14 слайде;
- использование верхнего регистра для названий констант является стилистической особенностью, а не требованием C++.

Составные типы. Открытые перечисления

Согласно названию, добавленные в тип-перечисление константы становятся доступными в коде как по своим прямым идентификаторам, так и через название типа:

```
8 printf("NOT_STARTED = %d\n", NOT_STARTED);  
9 printf("STARTED = %d\n", STARTED);  
10 printf("COMPLETED = %d\n", ComputingState::COMPLETED);
```

- строки **8** — **9** используют прямые названия констант;
- строка **10** демонстрирует использование полного имени: идентификатор типа плюс идентификатор константы;
- сами по себе значения констант полностью аналогичны значениям типа **int**;
- то, что константы становятся доступными в созданной области видимости, является первой тёмной стороной перечислений. Так, если используется два типа-перечисления с одинаковым названием входящей в них константы, то по идентификатору будет доступно то значение, которое объявлено позже (по отношению к строкам исходного кода).

Составные типы. Открытые перечисления

Раз есть тип, можно объявлять и переменные.

```
12 ComputingState my_task = NOT_STARTED;
```

Переменная ведёт себя как обычный целочисленный тип, то есть может использоваться в выражениях.

```
13 int no_sence_here = my_task + 5;
```

```
14 printf("value: %d\n", no_sence_here);
```

И вторая тёмная сторона открытых перечислений: переменной-перечисления может быть присвоено любое значение *базового типа*:

```
16 my_task = 808;
```

```
17 printf("my_task value: %d\n", my_task);
```

Это корректный код с точки зрения C++. Но, справедливости ради, некоторые компиляторы имеют опции, которые выдают предупреждение или ошибку на подобный фрагмент (ведь в перечислении нет ни одной константы со значением **808**). Для примера, компилятор **gcc** ругается на такой код, если добавлена опция **-fpermissive**.

Составные типы. Открытые перечисления

Стоит знать, что с помощью *явного приведения типов* компилятор можно обмануть:

```
19 my_task = ComputingState(777);  
20 printf("my_task value: %d\n", my_task);
```

Даже предупреждения не будет.

Перечисления полезно использовать тогда, когда в программном коде в различных ситуациях нужно делать выбор из немногочисленного набора некоторых вариантов. Далее несколько примеров, когда перечисления будут к месту:

- сохранение набора математических операций в виде символьных констант:

```
1 enum MathOperator : char  
2 {  
3     PLUS      = '+', MINUS  = '-',  
4     MULTIPLY  = '*', DIVIDE = '/',  
5 };
```

Код работает из-за того, что тип **char** совместим с целочисленными типами.

- размеры некоторого буфера, с помощью которого осуществляется чтение данных из внешнего источника (файл, консоль)

```
1 enum BufferSize
2 {
3     SZ_2KiB = 1024 * 2,
4     SZ_4KiB = 1024 * 4,
5     SZ_8KiB = 1024 * 8,
6     SZ_16KiB = 1024 * 16
7 };
```

Можно представить функцию, которая в зависимости от переданного ей значения в переменной-перечислении выделяет динамический массив нужного размера.

- замена текстовых констант на целочисленные представления. Например, палитра цветов, для каких-нибудь целей нужная программе

```
1 enum Color { RED, GREEN, YELLOW, PURPLE };
```

Перечисления, как и другие типы, может без ограничений выступать как параметры и возвращаемые значения функций. Например, функция, печатающая в консоль название цвета:

```
3 void print_color(const Color option)
4 {
5     switch (option) {
6         case Color::RED      : printf("red"); break;
7         case Color::GREEN    : printf("green"); break;
8         case Color::YELLOW   : printf("yellow"); break;
9         case Color::PURPLE   : printf("purple"); break;
10    }
11 }
```

Составные типы. Открытые перечисления

И функция для получения строкового представления цвета:

```
13 const char* color_as_str(const Color option)
14 {
15     switch (option) {
16         case Color::RED      : return "red";
17         case Color::GREEN    : return "green";
18         case Color::YELLOW   : return "yellow";
19         case Color::PURPLE   : return "purple";
20     }
21     return "";
22 }
```

Правило хорошего тона

При использовании конструкции **switch** для работы с перечислениями, следует **обязательно** прописать в ветках **case** все относящиеся к перечислению константы. Кроме того, для открытых перечислений полезным будет использование полного имени константы, с указанием типа. Это позволит избежать потенциального перекрытия названий констант с другими перечислениями.

Пример использования созданных функций:

```
24 Color c1 = Color::GREEN;
25 printf("First color is ");
26 print_color(c1);
27 printf("\n");
28
29 Color c2 = Color::RED;
30 char str[80] = "Color <<";
31
32 strcat(str, color_as_str(c2));
33 strcat(str, ">> is my choice");
34 puts(str);
```

Составные типы. Открытые перечисления

Ещё один способ записать в переменную перечисление некорректное по смыслу значение — воспользоваться пользовательским вводом (например, **scanf**):

```
36 Color c3;  
37 printf("Color values: red - %d, green - %d, "  
38        "yellow - %d, purple - %d\n",  
39        Color::RED, Color::GREEN, Color::YELLOW,  
40        Color::PURPLE);  
41 printf("Enter desired color: ");  
42 scanf("%d", &c3);  
43  
44 printf("Value of c3 is %d\n", c3);
```

Функция **scanf** запишет в переменную **c3** любое целое значение, введённое пользователем. Поэтому, как правило, ввод с консоли корректных значений для перечисления требует ручной проверки.

Синтаксис определения **закрытого перечисления**:

```
enum class <название> [: <целочисленный тип>]
{
    <константа_1> [= <значение_1>],
    [<константа_2>, <константа_3>, ...]
};
```

При определении все значения констант задаются по тем же правилам, как и для *открытых* перечислений на слайде 14.

Составные типы. Закрытые перечисления

Для демонстрации отличий, что нельзя делать с переменной закрытого перечисления:

```
1 enum class Output
2 {
3     CONSOLE_TEXT, FILE_TEXT = 20,
4     FILE_BINARY, FILE_HTML, FILE_XML
5 };
6
7 Output choice;
8 // Недопустимая: нет названия перечисления
9 choice = FILE_XML;
10 // Недопустимая: используется значение типа int
11 choice = 20;
12
13 // Недопустимые: нет неявного приведения к int
14 int some_num = choice + 2;
15 bool equals_to_zero = (choice == 0);
```

Строки **9**, **11**, **14** и **15** вызовут ошибки компиляции программы.

Составные типы. Закрытые перечисления

А теперь допустимые операции:

```
1 enum class Output
2 {
3     CONSOLE_TEXT, FILE_TEXT = 20,
4     FILE_BINARY, FILE_HTML, FILE_XML
5 };
6
7 Output choice = Output::CONSOLE_TEXT;
8 // Всё ок, присваиваем другую константу
9 choice = Output::FILE_TEXT;
10 // явное приведение к int
11 int status = int(choice) * 2;
12 printf("value of choice is %d\n", int(choice));
13
14 printf("Enter output source: ");
15 scanf("%d", &output);
16 printf("value of choice is %d", choice);
```

Строки **15-16** — единственные операции, когда не требуется приведение типов в явном виде. На 16 строку компилятор, возможно, выдаст только предупреждение.

Всё сказанное про **scanf** на слайде 24 применительно и к вводу значений для закрытых перечислений.

Что же выбирать?

Поскольку *закрытые* перечисления дают больше возможностей компилятору для проверки корректности работы с их переменными, их стоит применять всегда, когда тип-перечисление подходит под задачу.