

Использование ключевых нововведений стандарта C++11

№1. Потоки

В C++11 были добавлены потоки выполнения. С помощью создания дополнительных потоков и передачи в них части логики нашей программы мы можем исполнять наш код не последовательно, как раньше, а параллельно, и, как следствие, ускорить выполнение нашей программы. Однако для простых программ не нужно выделять потоки.

Что делает конкретно наша программа? Вне `main` находится функция `DoWork()`, а внутри `main()` находится цикл. Оба участка кода имитируют большой участок кода, которые при запуске выполняются очень долго. Мы решаем данную проблему при помощи потоков: создаем поток `th`, отвечающий за выполнение `DoWork()`. В конце мы используем ключевое слово `join()` для того, чтобы `main()` дождался конца выполнения потока `th`.

Как итог, мы получаем следующий результат, который показывает, что выполнение программы действительно происходило в двух потоках:

```
%1. Потоки.  
main  
DoWork  
main  
DoWork  
main  
main  
DoWork  
main  
DoWork  
main  
main  
DoWork  
main  
main  
DoWork  
main  
DoWork  
DoWork  
DoWork  
DoWork  
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 12792) завершил работу с кодом 0.  
Нажмите любую клавишу, чтобы закрыть это окно:
```

№2. Initializer_list<T>

`std::initializer_list` - это шаблонный класс в стандартной библиотеке C++, который представляет собой список значений одного типа. Он используется для передачи списка значений в функции или конструкторы.

Рассмотрим в нашей программе некоторую задачу, а именно заполнение нашего вектора некоторыми элементами. Разумеется, для данной цели можно использовать `push_back()`, однако при добавлении большого числа элементов возникает проблема не только читаемости кода, но и его низкого качества. В результате нововведений C++11 мы можем заполнить наш пустой вектор элементами, записанными в обычных фигурных скобках. Приятно.

Далее мы используем `std::initializer_list` для передачи списка значений одного типа данных в фигурных скобках напрямую в качестве аргумента функции, что также упрощает наш код. Функция умножает переданные аргументы на 2 и выводит их.

```
%2. Initializer_list<T>
pop metal rap jazz falk blues phonk
2 4 6 8 10
```

```
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 13276) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

№3. constexpr

Ключевое слово `constexpr` позволяет указывать, что функция или переменная может быть вычислена на этапе компиляции, что позволяет использовать в тех ситуациях, где требуется константное выражение, таких как размер массива или значение шаблона. Использование `constexpr` является одним из способов оптимизации производительности программы.

Мы рассмотрим 2 примера. Первый – простой, суть которого заключается в том, что функция возвращает число – размер массива в `main()`, элементы которого затем выводятся. Мы написали `constexpr` перед функцией, теперь компилятор знает, что функция выполняется на этапе компиляции и мы можем использовать возвращаемое значение для инициализации массива. **ВАЖНО ОТМЕТИТЬ**, что в C++11 были строгие ограничения на `constexpr` функций: функция обязательно должна была что-то возвращать, тело состоять из `return`, `expr` должен был состоять из констант или вызовов других `constexpr` функций, нельзя было использовать опережающее объявление функции.

Второй пример – факториал. В процессе подсчета может произойти неприятная ситуация, когда вместо рекурсии будет крутиться цикл. Мы помечаем нашу функцию `constexpr`, компилятор видит, что факториал нужен на этапе компиляции, мы получаем константу без ошибок.

ВАЖНО еще то, что может возникнуть неопределенное поведение, программа компилироваться не будет, например, когда результат факториала типа `int` превышает допустимые промежутки типа `int`. При помощи `constexpr` мы можем получить ошибку компиляции, что поможет нам избежать неправильных вычислений. Полезно.

Результат без использования `constexpr` (неправильный ответ из-за проблем с превышением порога типа данных `int`)

```
// рассмотрим пример использования ключевого слова на примере функции вычисления факториала числа
auto res = Factorial(14);
cout << res << endl;
```

1278945280

Результат с использованием `constexpr` (ошибка выполнения программы)

```
// рассмотрим пример использования ключевого слова на примере функции вычисления факториала числа
constexpr auto res = Factorial(14);
cout << res << endl;
```

 `constexpr int Factorial(int n)`

факториал числа типа `int`

Поиск в Интернете

выражение должно иметь константное значение
значение превышает диапазон "int"
вызвано из:

Поиск в Интернете

№4. Auto

Ключевое слово `auto` позволяет не указывать явно тип данных переменной, которую мы создаем. Ранее мы уже использовали `auto` в предыдущих номерах. Зачем она нужна, где используется? Мы рассматриваем простой пример – проход по элементам вектора. Будем использовать вектор из номера 2.

В нашем примере мы 2 раза прошлись по вектору и вывели его элементы. Использование `auto` позволяет упрощать код, а также предотвращает возможные ошибки, которые произошли бы, например, при изменении типа данных вектора.

```
%4. Auto
pop metal rap jazz falk blues phonk
pop metal rap jazz falk blues phonk
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 6804) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Также в программе представлен пример использования `auto` для инициализации переменных.

```
a - int
b - double
c - char
d - char const * __ptr64
```

№5. Decltype

Ключевое слово `decltype` используется для выведения типа выражения. Оно особенно полезно в тех ситуациях, когда тип выражения может не быть известен до времени компиляции. Рассмотрим 2 примера – примитивный и простой. 😊

В первом примере мы определяем тип переменной при помощи типа другой переменной. Все достаточно просто.

```
%5. Decltype
y - double
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 21452) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Во втором примере мы делаем то же самое, только используем для этого вектор. `Decltype` позволяет нам работать с векторами различными типами данных, избавляя нас, таким образом, от возможных ошибок. Результат – вывод элементов вектора.

№6. Делегирующие конструкторы

Рассмотрим ситуацию (наш пример в программе), когда мы работаем с классом «Человек», полями которого являются различные человеческие характеристики (у нас это имя, возраст и вес). Нам может понадобиться создать различные объекты класса, в которых некоторое поле мы зададим, а некоторые нет. Получается так, что для каждой ситуации нам необходим свой отдельный конструктор, который будет удовлетворять нашему набору полей. Но это очень неудобно, громоздко и может являться причиной множества ошибок.

Что именно мы сделали в нашей программе? Первый конструктор работает с полем Name, а остальные поля инициализирует нулем. Второй же сначала делегирует свой вызов первому конструктору (с полем Name), первый вызывается, а затем отработывает второй конструктор, изменяя значение поля Age на нужное. И т.д.

ЧТО ВАЖНО! Пусть мы хотим, чтобы наш конструктор везде после имени ставил «!». Нам бы пришлось менять реализацию всех конструкторов, работающих с полем Name. Однако благодаря делегирующим конструкторам, мы можем поменять реализацию лишь в самом первом, что весьма приятно. ☺

```
%6. Делегирующие конструкторы
Name: Bogdan, Age: 19, Weight: 500
Name: Dora, Age: 24, Weight: 0
Name: BOOOM, Age: 0, Weight: 0

C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 16340) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

№7. Default

Благодаря Default компилятор сам формирует тело конструкторов, для конструкторов копирования он сам копирует все поля, деструктор или оператор присваивания сам пишет. Если мы что-то важное забудем написать, компилятор исправит таким образом наши ошибки. Здорово.

В качестве примера будем использовать измененный класс из прошлого номера. Создаем три различных объекта (без полей, с полями и копию второго) и выводим данные в консоль. Ура, все получилось.

```
%7. Default
Name: , Age: -858993460, Weight: -858993460
Name: Bogdan, Age: 19, Weight: 500
Name: Bogdan, Age: 19, Weight: 500

C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 15952) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

№8. Override

Пусть у нас есть класс "пистолет". В нем реализован метод "выстрелить". Он стреляет только одиночными выстрелами. А мы хотим класс "пистолет-пулемет". И он тоже будет стрелять. Но стрелять по-другому. Мы от класса "пистолет" наследуем класс "пистолет-пулемет" и переопределяем метод "выстрелить". Как итог, у нас есть 2 связанных класса.

Теперь мы делаем еще один класс – Игрок, в руку которого при помощи указателя. Далее мы по очереди дадим ему сначала один вид оружия, затем другой и посмотрим на результат.

Override – необязательное ключевое слово, но использовать его полезно. Он следит за тем, чтобы функция переопределялась корректно. Что это значит? Он контролирует то, что методы Shoot() в обоих классах по сути одинаковые. Без override в классе SubmachineGun мы могли написать метод Shoot(int a), например, и ошибки не было бы, т.к. это, по сути, новый метод для этого класса. Однако т.к. мы хотим, чтобы методы в обоих классах были идентичными, override за нас следит за этим и показывает ошибку в противном случае. Также override следит за типом данных параметров функций, что тоже помогает не получить ошибки.

```
%8. Override
Дали пистолет: BANG!
Дали пистолет-пулемет: BANG! BANG! BANG!

C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 13896) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

№9. Nullptr

Nullptr — ключевое слово, введенное для описания константы нулевого указателя. Его также можно использовать для очистки указателя, что происходит в нашем небольшом примере.

В коде при очистке динамической памяти обязательно сначала надо использовать delete, а затем nullptr. В противном случае может произойти утечка памяти. ПОЧЕМУ? Если сначала напишем (pa = nullptr), то мы удалим адрес, по которому могли бы найти динамическую память. Но т.к. адрес удален безвозвратно, то ее мы найти и удалить не можем, а значит от (delete pa) смысла нет, и эта область останется в оперативной памяти навсегда до ее утечки.



№10. Tuple

Tuple представляет собой обобщенный фиксированный размер контейнера, хранящий в себе набор элементов, которые могут быть разного типа данных (а если они не разных типов, тогда зачем вообще использовать tuple?).

В нашем примере просто создается маленький кортеж, элементы которого мы затем выводим при помощи std::get<id>(tuple). Важно отметить, что данные передаются по ссылке, что дает нам возможность изменять значения элементов tuple.

```
%10. Tuple
Bogdan, 19, 500.52, 7
```

```
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 11480) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:|
```

№11. Лямбда функция

Лямбда функция еще называется анонимной, т.к. у нее нет имени. Но как к ней обратиться, если у нее нет имени? В этом и красота, что мы ее можем использовать в качестве параметра, например, в другой функции или цикле.

Наш пример простой – мы сортируем вектор. Лямбда функция является членом `sort` и проверяет отношение элементов на больше-меньше. Как итог – сортированный по возрастанию вектор.

```
%11. Лямбда функции
-777 -8 0 13 39 52 165
```

```
C:\Users\dor04\Homework\LessonWork\x64\Debug\LessonWork.exe (процесс 22008) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:|
```