

Как запустить проект Code::Blocks с примером в Windows

1. Скачать и установить Code::Blocks: <https://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/codeblocks-17.12mingw-setup.exe/download>
2. Создать копию проекта с примером, т.е. скопировать себе папку `ParallelTreeExample`. Она находится в `example/parallel_CodeBlocks_Windows`. Данный пример подробно рассмотрен ниже.
3. Запустить Code::Blocks, выбрать «Open an existing project» и выбрать файл `ParallelTreeExample.cbp` в созданной копии папки `ParallelTreeExample`.
4. Дополнительно ничего настраивать не нужно. Библиотеки `ParallelTree` и `OpenMP` уже подключены в настройках проекта.

Как запустить проект Code::Blocks с примером в Linux

1. Установить gcc версии 5 или выше и установить Code::Blocks. Как это сделать — зависит от дистрибутива Linux.
2. Создать копию проекта с примером, т.е. скопировать себе папку `ParallelTreeExample`. Она находится в `example/parallel_CodeBlocks_Linux_x86_64` для 64-битного Linux, либо в `example/parallel_CodeBlocks_Linux_x86` для 32-битного Linux. Данный пример подробно рассмотрен ниже.
3. Запустить Code::Blocks, выбрать «Open an existing project» и выбрать файл `ParallelTreeExample.cbp` в созданной копии папки `ParallelTreeExample`.
4. Дополнительно ничего настраивать не нужно. Библиотеки `ParallelTree` и `OpenMP` уже подключены в настройках проекта.

Как собрать и запустить пример с помощью make

1. Установить gcc версии 5 или выше и утилиты GNU для разработки. Как это сделать — зависит от операционной системы.
2. Открыть папку `example/parallel_Makefile` в консоли и набрать команду `make`
3. Запустить `./example`

Сборка вручную или в другой среде разработки

Для сборки проектов, использующих библиотеку `ParallelTree`, компилятору gcc обязательно нужно указать следующие флаги:

1. `-std=c++11` для использования стандарта C++11 (можно и `-std=c++14` для использования стандарта C++14, но не обязательно).
2. `-fopenmp` для включения поддержки OpenMP.
3. `-I <путь к ParallelTree.hpp>` для указания пути к заголовочному файлу библиотеки. Он находится в папке `include`.
4. `-L<путь к библиотеке> -lParallelTree -Wl,-rpath=<путь к библиотеке>` для подключения библиотеки `ParallelTree`. Версии этой библиотеки для 32-битных и 64-битных Windows и Linux находятся в папке `lib`. Обратите внимание, что `<путь к библиотеке>` пишется после `-L` и после `-Wl,-rpath=` без пробела.

Например: для сборки примера в `example/parallel_Makefile` в 64-битном Linux нужно использовать следующую команду:

```
1 g++ main.cpp -o example -std=c++11 -fopenmp -I ../../include -L../../lib/Linux_x86_64
   ↪ -lParallelTree -Wl,-rpath=../../lib/Linux_x86_64
```

Флаг `-o example` указывает, что имя выходного файла должно быть `example`.

Как пользоваться библиотекой `ParallelTree`

Библиотека `ParallelTree` предназначена для распараллеливания древовидных алгоритмов для задач минимизации и максимизации.

Чтобы воспользоваться библиотекой, нужно включить соответствующий заголовочный файл (`ParallelTree.hpp`) и вызвать функцию `parallelTree`:

```
1 void parallelTree(
2     std::vector<std::unique_ptr<Node>> (*processNode)(std::unique_ptr<Node>,
3                                                       Result&, const Params*),
4     std::unique_ptr<Node> node,
5     Result& result,
6     const Params* params = nullptr,
7     bool (*higherPriority)(const std::unique_ptr<Node>& n1,
8                           const std::unique_ptr<Node>& n2,
9                           const Params* params) = nullptr);
```

Она принимает следующие параметры:

1. `processNode` — функция обработки узлов дерева вариантов. Принимает текущий узел (`node`), ссылку на рекорд (`result`) и указатель на дополнительные параметры (`params`) и возвращает потомков данного узла (или пустой вектор, если потомков нет). Рекорд — наилучший найденный результат. В примере ниже `params` не используется (класс

`TreeParams` приведен просто для демонстрации). В данную функцию всегда передается копия `result` (создается с помощью метода `clone`), причем создается по копии для каждого потока, поэтому беспокоиться о многопоточном доступе к нему не нужно. По завершении работы функции `processNode` измененная копия `result` автоматически сравнивается с самим `result` (с помощью метода `betterThan`) и `result` заменяется на копию, если копия лучше.

2. `node` — корень дерева вариантов.
3. `result` — класс, который содержит рекорд.
4. `params` — класс, который хранит константные параметры алгоритма, если они нужны. Вместо него можно указать `nullptr`.
5. `higherPriority` — функция приоритетов. Возвращает `true`, если левый узел (`n1`) нужно обрабатывать до правого (`n2`), и `false`, если правый нужно обрабатывать раньше, чем левый. В некоторых ситуациях порядок их обработки может быть нарушен (хотя большинство узлов будут обработаны в правильном порядке), поэтому писать программу с расчетом на определенный порядок обработки узлов нельзя. Вместо `higherPriority` можно передать `nullptr`. Тогда узлы будут обрабатываться в том порядке, в котором получится.

Функция `parallelTree` занимается параллельной обработкой узлов дерева вариантов с помощью функции `processNode`. Для начала ей нужен только корень дерева (`node`). После обработки корня она занимается обработкой потомков, которых вернет функция `processNode` (потомки также передаются в функцию `processNode`). Алгоритм считается завершенным, если все узлы были обработаны. Вместо `params` и `higherPriority` можно указать `nullptr`, если они не нужны.

Классы `Node`, `Result` и `Params` имеют следующий вид:

```
1  class Node {};  
2  
3  class Result  
4  {  
5  public:  
6      /**  
7       * @brief Должен возвращать true, если данный рекорд лучше (меньше в задачах  
8       *           минимизации и больше в задачах максимизации), чем @p other  
9       */  
10     virtual bool betterThan(const Result& other) = 0;  
11  
12     /**  
13     * @brief Должен возвращать копию данного объекта.  
14     */  
15     virtual std::unique_ptr<Result> clone() = 0;  
16
```

```

17     virtual Result& operator= (const Result& other) = 0;
18 };
19
20 class Params {};

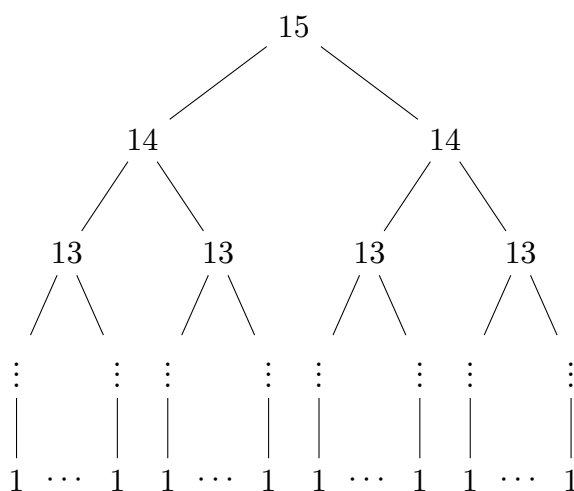
```

От них нужно унаследоваться. Три метода класса `Result` предназначены для внутреннего использования в библиотеке. При наследовании их обязательно нужно переопределить, но вызывать их самому не обязательно.

Пример использования библиотеки

Далее рассмотрим пример решения следующей задачи с помощью библиотеки `ParallelTree` (весь код примера можно найти в папке `example`):

Дано двоичное дерево вариантов, в каждом узле которого находится целое число. Найти минимум этих чисел. На вход подадим дерево следующего вида (ясно, что минимум в нем будет равен 1):



Примечание: чтобы перейти от последовательной версии алгоритма к параллельной с наименьшим количеством изменений кода рекомендуется писать последовательную версию как рекурсивную функцию, или заменить рекурсию на стек (данный код можно найти в папке `example/sequential_Makefile`):

```

1  #include <iostream>
2  #include <limits>
3  #include <stack>
4
5  using namespace std;
6
7  // Рекурсивный обход дерева
8  void minTreeRecursive(int value, int& record)
9  {
10     if(value < record)

```

```

11     record = value;
12     if(value > 1)
13     {
14         // Потомки данного узла
15         minTreeRecursive(value-1, record);
16         minTreeRecursive(value-1, record);
17     }
18 }
19
20 // Замена рекурсии на стек
21 void minTreeStack(int rootValue, int& record)
22 {
23     // Узлы дерева, которые нужно обработать
24     stack<int> nodes;
25     nodes.push(rootValue);
26
27     while(!nodes.empty())
28     {
29         int value = nodes.top();
30         nodes.pop();
31         if(value < record)
32             record = value;
33         if(value > 1)
34         {
35             // Потомки данного узла
36             nodes.push(value-1);
37             nodes.push(value-1);
38         }
39     }
40 }
41
42
43 int main()
44 {
45     int recordRecursive = numeric_limits<int>::max();
46     minTreeRecursive(15, recordRecursive);
47
48     int recordStack = numeric_limits<int>::max();
49     minTreeStack(15, recordStack);
50
51     cout << "Result (recursive): " << recordRecursive << "\n";
52     cout << "Result (stack):      " << recordStack << "\n";
53
54     return 0;
55 }

```

Далее напомним параллельную версию алгоритма. Для этого унаследуемся от классов `Node`, `Result` и `Params`:

```

1  // Узел дерева вариантов. В данном случае каждый узел содержит одно число.
2  class TreeNode : public Node
3  {
4  public:
5      TreeNode(int number) :
6          number(number) {}
7
8      int number;
9  };
10
11 // Класс, который содержит рекорд.
12 class NumericResult : public Result
13 {
14 public:
15     NumericResult(int value) :
16         value(value) {}
17
18     /* Эти три метода нужны для внутреннего использования в библиотеке.
19      * Пользоваться ими самому не обязательно.
20      */
21
22     /* Возвращает true, если данный рекорд лучше other. Поскольку данная задача
23      * - задача минимизации, то здесь стоит оператор <. В задачах максимизации
24      * нужно поставить >.
25      */
26     bool betterThan(const Result& other) override
27     {
28         const NumericResult& o = (NumericResult&) other;
29         return value < o.value;
30     }
31
32     // Создает копию данного рекорда.
33     std::unique_ptr<Result> clone() override
34     {
35         return std::unique_ptr<Result>(new NumericResult(*this));
36     }
37
38     // Присваивает данному рекорду значение other.
39     Result& operator= (const Result& other) override
40     {
41         const NumericResult& o = (NumericResult&) other;
42         value = o.value;
43         return *this;
44     }
45
46     int value;
47 };
48

```

```

49  /* Если алгоритму нужны какие-либо константные параметры, то их нужно хранить
50  * здесь. В данном примере они не нужны. Этот класс приведен просто для
51  * демонстрации.
52  */
53  class TreeParams : public Params
54  {
55  public:
56      const int param1 = 10;
57      const double param2 = 2.5;
58  };

```

Напишем функцию обработки узлов processNode:

```

1  vector<unique_ptr<Node>> processNode(std::unique_ptr<Node> node,
2                                     Result& result, const Params* params)
3  {
4      TreeNode* treeNode = (TreeNode*) node.get();
5      NumericResult& numericResult = (NumericResult&) result;
6
7      // Если число в данном узле меньше рекорда, то обновляем рекорд.
8      if(treeNode->number < numericResult.value)
9          numericResult.value = treeNode->number;
10
11     // Если число в данном узле больше 1, то в его потомках находятся числа на 1 меньше.
12     if(treeNode->number > 1)
13     {
14         vector<unique_ptr<Node>> children;
15         children.emplace_back(new TreeNode(treeNode->number - 1));
16         children.emplace_back(new TreeNode(treeNode->number - 1));
17
18         return children;
19     }
20     // Если число в данном узле равно 1, то у него нет потомков.
21     else
22         return vector<unique_ptr<Node>>();
23 }

```

И напишем функцию приоритетов higherPriority, которая будет отдавать предпочтение узлам с меньшим числом в них:

```

1  bool higherPriority(const std::unique_ptr<Node>& n1,
2                    const std::unique_ptr<Node>& n2,
3                    const Params* params)
4  {
5      // Узлы с меньшим числом в них обрабатываем раньше
6      TreeNode* treeNode1 = (TreeNode*) n1.get();
7      TreeNode* treeNode2 = (TreeNode*) n2.get();

```

```
8     return treeNode1->number < treeNode2->number;
9 }
```

И, наконец, напомним `main`:

```
1  int main()
2  {
3      // Корень дерева
4      unique_ptr<Node> root(new TreeNode(15));
5
6      /* Параметры алгоритма. В данном примере они не нужны. Этот класс приведен
7       * просто для демонстрации.
8       */
9      TreeParams params;
10
11     /* Рекорд.
12      * Мы не можем присвоить int бесконечность, поэтому присваиваем максимальное
13      * возможное значение.
14      */
15     NumericResult result(numeric_limits<int>::max());
16
17     parallelTree(processNode, move(root), result, &params, higherPriority);
18
19     // По окончании работы функции parallelTree результат будет храниться в result.
20     cout << "Result: " << result.value << "\n";
21
22     return 0;
23 }
```

В результате работы функции `parallelTree` будет найдено минимальное число в дереве. Оно будет сохранено в `result`.