

Шаг 0. Установка ttgLib и создание нового проекта

Цель

В данном примере показывается процесс установки и настройки библиотеки ttgLib, а также описываются действия, необходимые для подключения заголовочных файлов в новый проект для среды Microsoft Visual Studio 2005/2008.

Получение и установка ttgLib SDK

Последнюю версию библиотеки ttgLib можно скачать на сайте <http://ttglib.org>. Полученный исполняемый файл необходимо запустить и следовать инструкциям установщика, после выполнения которых ttgLib будет установлена и готова к использованию.

Другим способом установки является сборка библиотеки из исходных кодов. Данный вариант позволяет использовать более новую версию библиотеки (для которой ещё не создан установщик), но при этом требует наличие в системе среды Microsoft Visual Studio 2005. Для ручной сборки библиотеки необходимо зайти на портал проекта <http://ttglib.codeplex.com> и скачать нужную версию исходных кодов. Полученный zip-архив необходимо распаковать и определить в переменных окружения Microsoft Windows переменную TTGLIB_HOME, содержащую путь к директории с содержимым архива.

Примечание: Чтобы определить переменную окружения необходимо зайти в *Мой Компьютер* -> *Свойства Системы* -> Вкладка *Дополнительно* -> *Переменные среды*. После этого нужно создать новую переменную, указав её имя и значение.

Следующим действием является компиляция библиотеки. Для этого необходимо запустить из командной строки утилиту *nmake*, поставляемую со средой Microsoft Visual Studio, указав ей в качестве аргументов путь к файлу *makefile* (расположенному в директории %TTGLIB_HOME%\Src) и цель компиляции VC80. Требуемая команда может иметь следующий вид:

```
nmake.exe %TTGLIB_HOME%\Src\makefile VC80
```

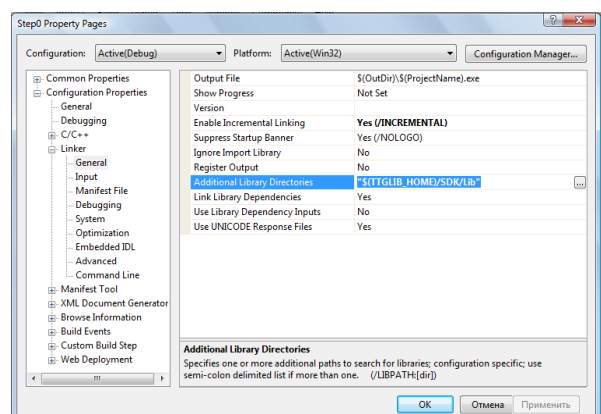
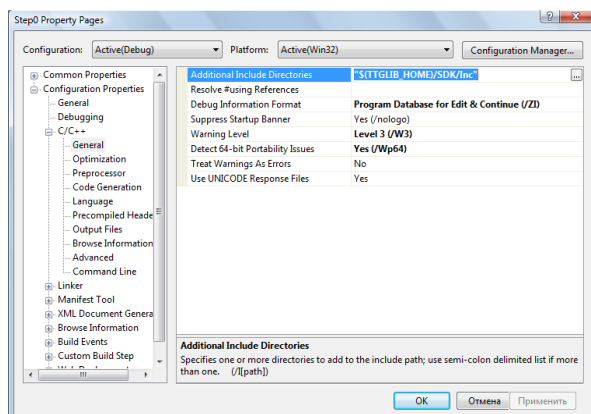
В случае, если в системе также установлена среда Microsoft Visual Studio 2008, то цель VC80 можно не указывать, в результате чего будут созданы также дополнительные утилиты:

```
nmake.exe %TTGLIB_HOME%\Src\makefile
```

В случае успешного выполнения этой команды в директории %TTGLIB_HOME%\SDK будут созданы необходимые заголовочные файлы, а также статические и динамические библиотеки.

Создание нового проекта

После создания чистого проекта в среде Microsoft Visual Studio необходимо указать пути к заголовочным файлам и статическим библиотекам. Для этого нужно зайти в свойства проекта (*Project* -> *Properties*), после чего присвоить полю *Additional Include Directories* (вкладка *Configuration Properties* -> *C/C++* -> *General*) значение «\$(TTGLIB_HOME)/SDK/Inc». Аналогично устанавливается путь к статическим библиотекам: полю *Additional Library Directories* (вкладка *Configuration Properties* -> *Linker* -> *General*) необходимо присвоить значение «\$(TTGLIB_HOME)/SDK/Lib».



Примечание: Вышеприведенная установка путей применяется только к текущему типу конфигурации (которым по умолчанию является *Debug*). Если тип конфигурации был изменён (например, был выбран *Release*), то процесс установки путей необходимо повторить заново.

Следующим шагом является включение заголовочных фалов в код программы. Самым простым вариантом является использование заголовочного файла *<ttg/all.h>*, содержащего все объекты библиотеки ttgLib и подключающего необходимые статические библиотеки:

```
#include <ttg/all.h>
```

Для проверки работоспособности программы можно воспользоваться любой функцией библиотеки ttgLib. Наиболее простым вариантом является получение версии библиотеки:

```
printf("Hello World! (using ttgLib v%s)\n", ttg::Version::toString());
```

Если все предыдущие шаги были выполнены успешно, то после компиляции и запуска программы на экран будет выведена надпись примерно следующего вида:

```
Hello World! (using ttgLib v0.01-pb1)
```

```
Для продолжения нажмите любую клавишу . . .
```

Исходный код

Исходный код данного примера можно найти в *SDK/Sample/Tutorial/Step0*.



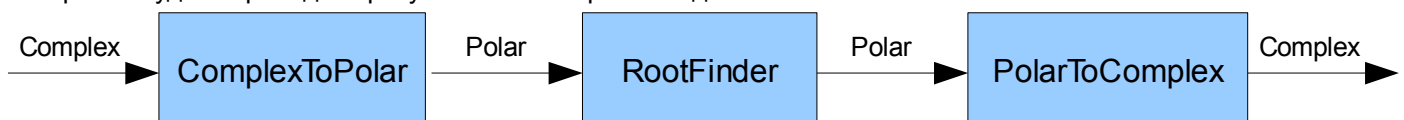
Шаг 1. Создание и запуск конвейера

Цель

На примере нахождения корня из комплексного числа показывается процесс создания звеньев, их «склейки» в простой конвейер и его последующий запуск. Для нахождения корня используется формула $\sqrt[n]{\rho(\cos \phi + i \sin \phi)} = \rho^{\frac{1}{n}}(\cos(\frac{\phi}{n} + \pi n) + i \sin(\frac{\phi}{n} + \pi n))$, где $n \in \{0, 1\}$

Создание звеньев конвейера

Каждое звено должно производить однотипную обработку данных и, по возможности, быть независимым от других звеньев. В дальнейшем это позволит использовать ранее созданные звенья в новых программах, а также повысит расширяемость уже существующих программ. Для решения данной задачи с использованием выбранного алгоритма потребуется три звена. Первое из них будет переводить комплексное число в полярный вид, второе будет находить корень по заданной формуле, а третье будет переводить результат из полярного вида в классический:



Для определения звена конвейера необходимо создать класс, наследующий от шаблонного класса `ttg::pipeline::Node<InType, OutType>`. Тип `InType` определяет, какие данные звено должно получать на вход, а `OutType` — во что они будут преобразованы. В данном классе необходимо переопределить виртуальный метод `virtual void process(InType data)`, отвечающий за обработку одной порции данных. Соответственно, реализация звена `ComplexToPolar` будет иметь следующий вид:

```
class ComplexToPolar :public ttg::pipeline::Node<Complex, PolarComplex> {
    virtual void process(Complex data) {
        //Converting to polar coordinate system.
        double rho = sqrt(data.re * data.re + data.im * data.im);
        double phi = math::isZeroEps(rho, 0.0001)
            ? 0.0
            : acos(data.re/rho) * math::getSign(asin(data.im/rho));
        //Sending result to next nodes.
        sendNext(PolarComplex(rho, phi));
    }
};
```

Отметим, что здесь используется метод `sendNext(...)` (определённый в классе `Node<>`), с помощью которого производится отправка обработанных данных следующим звеньям, о которых данному звену может быть ничего неизвестно. Схожим образом реализуется звено `RootFinder`:

```
class RootFinder :public Node<PolarComplex, PolarComplex> {
    virtual void process(PolarComplex data) {
        //Calculating roots.
        PolarComplex root1(sqrt(data.rho), data.phi / 2.0);
        PolarComplex root2(sqrt(data.rho), data.phi / 2.0 + TTG_PI);
        //Sending roots to next nodes.
        sendNext(root1);
        sendNext(root2);
    }
};
```

Так как для каждого комплексного числа существует два корня, то данное звено отправляет сразу два числа, каждое из которых будет независимо обрабатываться последующими звеньями. Звено `PolarToComplex` реализуется аналогично:

```
class PolarToComplex :public Node<PolarComplex, Complex> {
    virtual void process(PolarComplex data) {
        //Converting.
        double im = data.rho * sin(data.phi);
```

```

        double re = data.rho * cos(data.phi);
        //Sending result to next nodes.
        sendNext(Complex(re, im));
    }
};

```

Данное звено является последним звеном конвейера, поэтому все отправленные из него данные будут получены пользователем.

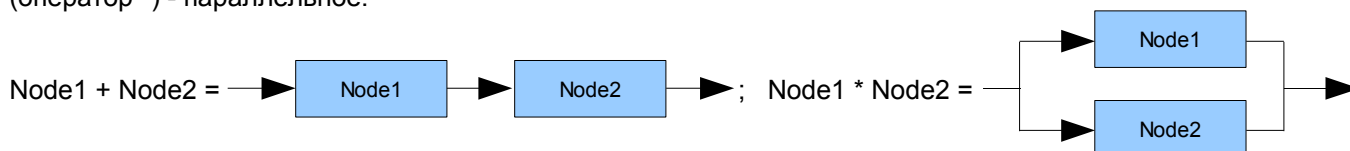
“Склейка” конвейера

Перед первым использованием любой подсистемы библиотеки `ttgLib` необходимо запустить сервисы с помощью следующей команды:

```
ttg::services::ServiceBase::getRef()->start();
```

Примечание: В результате выполнения этой команды будет запущен стандартный набор сервисов, предоставляющих доступ к файлу конфигурации и *log*-файлу, активирующих высокоточный таймер, находящихся доступные вычислительные устройства и т.д.. Подробное описание всех возможностей сервисов будет приведено в примере “Шаг X”.

Для создания конвейера используется класс `ttg::pipeline::Pipeline<InType, OutType>`, содержащий “склепку” из звеньев. Для этого необходимо обернуть каждое звено (используя функцию `wrap(...)`) и указать тип соединения звеньев (используя перегруженные операторы `+` и `*`). Метод `wrap(...)` позволяет “обернуть” указатели или ссылки на звенья, после чего их можно будет умножать и складывать. Также он определяет, должно ли звено быть удалено при уничтожении конвейера. При сложении звеньев (оператор `+`) получается их последовательное соединение, а при умножении (оператор `*`) - параллельное:



Примечание: Стоит учитывать, что у соединяемых звеньев входы и выходы должны быть соответствующего типа. Так, например, попытка последовательно подключить к звену со входом типа *double* звено с выходом типа *bool* приведёт к ошибке компиляции.

Согласно вышеприведённой схеме, конвейер будет задаваться следующим образом:

```

Pipeline<Complex, Complex> pl(wrap(new ComplexToPolar()) +
                               wrap(new RootFinder()) +
                               wrap(new PolarToComplex()));

```

Чтобы запустить созданный конвейер необходимо вызвать метод `run(...)`, аргументом которого являются обрабатываемые данные и который возвращает вектор обработанных данных (полученных из последнего звена):

```
std::vector<Complex> root = pl.run(Complex(0.0, 1.0));
```

В нашем случае вектор результатов будет состоять из двух чисел $\frac{1}{\sqrt{2}}(1+i)$ и $-\frac{1}{\sqrt{2}}(1+i)$.

Исходный код

Исходный код данного примера можно найти в *SDK/Sample/Tutorial/Step1*.



Шаг 2. Синхронизация звеньев конвейера

Цель

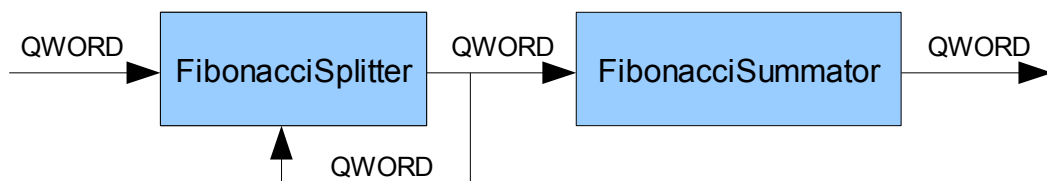
В данном примере создаётся конвейер с простейшей петлёй, а также демонстрируется подход, позволяющий проводить базовую синхронизацию различных звеньев конвейера с помощью событий. В качестве тестовой задачи выбрана реализация рекурсивного алгоритма нахождения чисел Фибоначчи, для которой также будет определено ускорение, полученное по сравнению с последовательной версией.

Создание конвейера

Последовательная реализация рассматриваемого алгоритма имеет достаточно простой вид:

```
QWORD calculateFibonacciNumber(QWORD n) {  
    if (n <= 2)  
        return 1;  
    else  
        return calculateFibonacciNumber(n - 1) + calculateFibonacciNumber(n - 2);  
}
```

Сразу отметим, что единственным достоинством этого алгоритма является исключительно простота его записи, благодаря чему он и был выбран для данного примера. Наиболее простым подходом создания его параллельной реализации является независимое вычисление $n-1$ и $n-2$ чисел на различных вычислительных устройствах. В терминах конвейера это можно записать следующим образом:



Первое звено (*FibonacciSplitter*) получает номер искомого числа Фибоначчи. Если число с таким номером «известно», то оно отсылается дальше по конвейеру, в противном же случае звено посылает самому себе два новых запроса на нахождение двух предыдущих чисел. Второе звено производит суммирование всех пришедших ему значений. Как только станет известно, что числа для суммирования закончились, оно отправит результат дальше по конвейеру (в данном случае — пользователю).

Основным отличием данного конвейера от рассмотренного в предыдущем примере является наличие петли (для реализации рекурсии) и использование событий (для определения, что предыдущие звенья закончили свою работу). Чтобы задать подобную петлю, используется метод *sendThis(...)*, отправляющий этому же звену новые данные на обработку. Соответственно, звено *FibonacciSplitter* можно записать следующим образом:

```
class FibonacciSplitter :public Node<QWORD, QWORD> {  
    public:  
        virtual void process(QWORD data)  
        {  
            //If it small enough, calculating serially and sending to next node.  
            if (data <= 30)  
                sendNext(calculateFibonacciNumber(data));  
            //Otherwise, splitting into sum and sending to this node.  
            else {  
                sendThis(data - 1);  
                sendThis(data - 2);  
            }  
        }  
};
```

Примечание: в данной реализации звена «известными» числами Фибоначчи считаются все числа, меньшие 30. Это вызвано тем, что метод *process()* должен содержать достаточно вычислительноёмкий код, так как в противном случае накладные расходы на синхронизацию потоков нивелируют ускорение, полученное от параллельного выполнения (в результате чего программа может оказаться даже медленней, чем её последовательный аналог). Чтобы этого избежать, необходимо помешать в метод *process()* код, выполнение которого займёт не менее 10 миллисекунд.

Для определения событий необходимо перекрыть виртуальные методы *onBegin()* и *onEnd()*. Гарантируется, что метод *onBegin()* будет вызван перед началом любой обработки данных звеном (т.е. перед первым вызовом метода *process(...)*), а *onEnd()* - после завершения обработки (т.е. после последнего вызова метода *process(...)*). С помощью этих событий удобно реализовывать открытие и закрытие файла, начало и завершение рендеринга очередного кадра 3D сцены и т.д.. В нашем случае их можно использовать при реализации звена *FibonacciSummator* следующим образом:

```
class FibonacciSummator :public Node<QWORD, QWORD> {
private:
    QWORD res;
public:
    //Clears result variable.
    virtual void onBegin()
    { res = 0; }
    //Sums incoming numbers.
    virtual void process(QWORD data)
    { res += data; }
    //Sends result to next node (in our case it is user).
    virtual void onEnd()
    { sendNext(res); }
    //Note: "true" flags means, that this node will work serially.
    FibonacciSummator()
        :Node("FibonacciSummator", true)
    { /*nothing*/ }
};
```

Для хранения суммы всех пришедших звену чисел вводится специальное внутреннее поле-счётчик, которое по событию «начало итерации» обнуляется, а по событию «конец итерации» накопленное в нём значение отправляется дальше по конвейеру. Отдельно стоит остановиться на флаге, которому передаётся значение *true* в конструкторе *Node(...)*. Он определяет, должны ли методы данного звена вызываться последовательно (значение *true*), или же они могут выполняться независимо и параллельно (значение *false*). Так как в методе *process(...)* есть обращение к внутренней переменной, то при параллельной обработке данных может произойти потеря некоторых значений (проблема параллельного программирования, известная как *race condition*).

Примечание: альтернативным решением данной проблемы является использование «быстрых» критических секций *FastCriticalSection* или атомарных переменных *AtomicInt*, определённый в пространстве имён *ttg::threads*. Более подробно об этих примитивах будет рассказано в примере «Шаг X».

Определения полученного ускорения

Для замера времени, потраченного на последовательное и параллельное вычисления *N*-ого числа Фибоначчи, можно воспользоваться сервисом *Timer* (определённым в пространстве имён *ttg::services*). Он является классом-синглтоном, поэтому для него всегда существует единственный экземпляр, ссылку на который можно получить с помощью статического метода *getRef()*. Чтобы определить затраченное время на выполнение, запуск конвейера нужно «обрамить» замерами времени:

```
double t_start = Timer::getRef()->getSeconds();
QWORD res = pl.run(45)[0];
double t_end = Timer::getRef()->getSeconds();
double t_elapsed = t_end - t_start;
```

После проведения подобных замеров ускорение от использования конвейера можно определить как отношение времени выполнения последовательной версии ко времени выполнения параллельной (т.е. во сколько раз всё стало быстрее работать). Отметим, что на тестовой машине с двухядерным процессором ускорение равнялось 1.96.

Исходный код

Исходный код данного примера можно найти в *SDK/Sample/Tutorial/Step2*.

