The JSONIZER

This Jupyter Notebook file describes the Jsonizer, i.e. the C++ source code *jsonizer.cpp*, and specifically its version 1.0 (commented at the beginning of the source).

DO THIS FIRST

Run this Notebook in JupyterLab, not Jupyter Notebook, as the "hide code" seems to work better in JupyterLab.

You'll need to pre-install at least graphviz and ipywidgets, possibly tweak some settings.

TODO: LET'S FIGURE OUT WHAT PREPARATIONS NEED TO BE DONE IN A VIRGIN ENVIRONMENT.

The first thing to do after opening is to **View / Collapse All Code**.

The second thing to do after opening is to **Run / Run All Cells**.

This is to keep the document easier to view. You can later open and view the code as you please.

Document Info

This notebook contains action buttons for transferring the source code into the Compiler Explorer (https://godbolt.org/) web page for a quick compile and demo run, and some other action buttons. However this is draft design and at it's present state, breaks easily.

This functionality, plus the "hide code" functionality seems to work best in JupyterLab (3.5.2), whereas the Jupyter Notebook (6.5.2) doesn't seem to work as well.

The buttons reside at the end of this file for the whole application and possibly at select locations for runnable code snippets. Not every detail gets its own transfer button, but just copy/paste the code you'd like to experiment with into the Compiler Explorer or whichever other environment you prefer, and make your own *int main()* for the experiment.

Note that I won't always bother to copy code snippets in question into the following chapters. You might want to get two side-by-side browser windows for keeping the source code location and the discussion in sync.

Should you edit this file, the functionally relevant details are:

- the C++ source code snippet cells should be in Markdown format, every line indented with at least 4 spaces. This is Markdown's idea of "source code". Initially Raw format was used instead, and it works fine in the context of the Notebook itself, and for transmitting code snippets into Compiler Explorer. However the Raw format gives trouble when Notebook gets converted to PDF or HTML.
- the C++ snippets to get transferred with the buttons must have the tags "C++" and some other snippet-identifier like "snippet_1"

There are several ways to convert this Notebook into PDF. One of the best is using nbconvert from e.g. Anaconda initiated PowerShell. One way to get nbconvert into action is:

- open PowerShell Prompt from the Anaconda.Navigator
- pip install nbconvert
- pip install pyppeteer
- jupyter nbconvert --no-input --to webpdf --allow-chromium-download .\the_notebook.ipynb.

That *--allow-chromium-download* is needed only the first time. The *--no-input* omits Code segments from the PDF. However this way constructed PDF is not 100% pretty. The action button things and some other details produce a bit of unwanted text. Not too much to have this method unusable though.

The trick to get PDF without **most** of the unwanted Code output is to modify the **first entry** in this Notebook: set *enable_extra* to *False*. This leaves some of the button specific usage texts unprinted and possibly unperforms some other PDF incompatible chores, thus making the PDF a little more readable. Note thought that the PDF formatting is very crude, so don't expect publish quality document.

Purpose

As the name might imply, *jsonizer.cpp* creates JSON objects. It is intended for forming large and complex JSON objects with semi-random keys and values.

However, this is just the surface of what jsonizer.cpp is. It serves also other purposes:

- to provide SUT for a threaded application debugging session, with gdb
- to demonstrate a hefty list of (mostly) modern C++ features and some interesting and perhaps even useful implementations of said features
- to experiment using Jupyter Notebook as a crude publishing application.

The following chapters outline the features of interest roughly in the order where they appear in the source code.

std::chrono_literals

The *chrono_literals* is a namespace that enables us to use timing related postfixes to numbers. For instance, you might want to sleep for *500ms*. The common usage of *chrono_literals* is to include *chrono* and then use the namespace in a .cpp file (never in a header as namespace pollution is just a bad practice):

```
using namespace std::chrono_literals;
```

enum class

The "old" *enum*s are essentially just a form or named integer constants. The *enum class* on the other hand is in addition an actual type, which doesn't do automatic conversion to integer. Type safety is one of the strong points of C++, so embrace enum classes wherever "just a form or named integer constants" isn't what you really need from enums.

Initializer lists

The initializer list is a convenient and compact way to intialize containers. Jsonizer uses them all over the place. Example:

```
std::vector<int> v{1,2,3};
```

The LOG macro

The LOG macro is intended for thread safe debug or other *stdout* targeting logging. Essentially:

```
LOG("string: " << myString << " and int: " << myInt)
```

equals:

```
std::cout << "string: " << myString << " and int: " << myInt << std::endl;
```

The jsonizer.cpp substantives

"What's life without whimsy" said a wise man once (who?). The Jsonizer uses semi random names as the keys of the key-value pairs it outputs to JSON. These names are hand-picked and sometimes "substantized" from the venerable **Webster's Encyclopedic Unabridged Dictionary of the English Language**. With its over 1800 large pages of small print, this book is a remarkable specimen from the era when everybody's nose wasn't yet glued to a smartphone screen.

Note that the key generation also utilizes a "Base-N" template for postfixing the substantives in order to create a sufficient amount of unique names. More about that one later.

Random numbers

The classic C library provides the *rand()* library function, which has a number of shortcomings, and which probably shouldn't be used for critical application.

The more modern way to get random numbers is to use *std::random_device* as a non-deterministic random number generator, or due to performance issues, more often as the seed generator for other random number generators like the Mersenne Twister (*std::mt19937*) pseudo random generator.

Tuples

Tuples are lists of predetermined sizes of heterogeneous items. The Jsoniser uses them quite a lot. See the template functions named *tie*N*()*. These are tool functions for the totally awesome *init()* variadic lambda, one of the cherries-on-top of the Jsonizer.

Even though the source code walkthrough has now advanced to the *init()* implementation, I'll postpone the introduction of the variadic lambda until a bit later.

The DisNDat<> class

The *DisNDat* template class is nothing special language-wise. But it is a convenient way to aid producing comma (or other) separated text, or otherwise act as a differentiator of "first" and "the rest".

Just as an example, consider which of the following is more convenient:

```
while(condition)
   {
   if(first)
      {
      std::cout << ",";
      first=false;
      }
   std::cout << something;
   possibly_alter(condition);
   }
```

or:

```
DisNDat<> c("",",");
while(condition)
   {
   std::cout << c << something;
   possibly_alter(condition);
   }
```

The getFrom() pattern

The *getFrom()* template function and it's tools of *std::string* and templatized *conv()* functions is a common and convenient way of mixing and matching generic and specific types handling.

The Part class

The *Part* class is one of the key components of Jsonizer. It models a unique, hierarchical, named or nameless "item", which in practice is a simple type like an integer, a double, a string, or a container like an array or an object.

The *Part* class also offers a bookkeeping helper for keeping count of the contained simple typed items in the form of the *valueCount()* member. This in turn is needed for the thread terminate exit condition handling.

Note that the *valueCount()* could use some optimization, as now it is implemented as "*recurse for the same piece of information over and over again*".

Note also that the *ostream operator* for *Part* offers a prime example of the usefullness of the *DisNDat* template class.

Note also the usage of *std::optional<>*. This is a gread addition to the language for those cases where e.g. *nullptr* vs. non-null just doesn't provide enough information of what a value represents. A "not exists" and "null" just aren't the same thing.

Mutexes

The C++ standard comes with a few different ways to ensure a critical section in a code gets serialized. The Jsonizer uses *std::shared_mutex*, which models a "*single writer, multiple readers*". For our queue based data transfer between threads, this seems an obvious choice.

The *shared_mutex* is used by the thread safe *MuxPart* wrapper of the *Part* class.

Jsonizer also uses *std::mutex* within the *LOG* macro and in the context of *std::condition_variable*s. More about that later.

The BaseN<> class

The *BaseN* template class offers a simple implementation for converting integers (i.e. Base-10) into more compact integer representations. As the implementation doesn't jump over the "unpretty" characters between ASCII digits and capital letters nor capitals and lowercase letters, it's readably usable only until Base-26. Furthermore, the implementation offers only pre- and post increments, plus a dereference operator. While many parts of the Jsonizer may be overengineered, this class is an exception.

The KeyGetter class

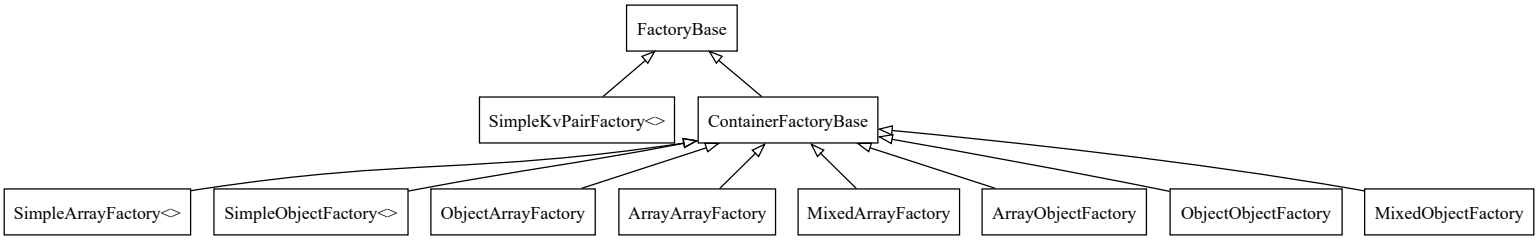
The virtual *KeyGetter* class brings together a container of base names and an automatically extending postfix. It utilizes the *BaseN*<> class. This class also slices the namespace according to the count of name-needing clients.

The SimpleValueGenerator<> class

The *SimpleValueGenerator* template class randomly picks values from the given list of values. The implementation uses this class only for the JSON base types of ints, doubles and strings.

Note that this would be a good candidate to be a *virtual* class, as it offers only a single *get()* interface to clients. However currently it is also bound to the *Part* class. This could instead be a template argument.

The Factory class hierarchy



The *Factory* class hierarchy is responsible for creating *Part*s. A *Part* is an entity that is convertible to JSON data, i.e. a simple key-value pair, a keyed array containing non-keyed values, or an object. The concept is that of a components assembly line where first appears bare values, which then a *SimpleKvPairFactory* instance picks up and joins with a key, thus forming an elementary *Part*, which then either gets recircled back to the components assembly line to be a component of a larger *Part*, or which then gets moved to a products assembly line for further processing.

There are separate *Factory* classes for each of the possible JSON components: key-value pairs of simple values (ints, doubles and strings), arrays of simple kvpairs, arrays of arrays, arrays of objects, objects of simple kvpairs, objects of arrays, and objects of objects, plus arrays and objects of mixed subtypes. All of the container handling factories, based on *ContainerFactoryBase* pure virtual class, are constructed with min and max contained elements counts, and the processing algorithm then chooses randomly the count of elements based on that range.

The factories are constructed with a ratio for recirculating back to the components assembly line versus moving to the products assembly line. This should affect the complexity of the resulting JSON object.

The factories are also constructed with a weigth parameter. The assembly line algorithm is such that the HEAD *Part* of the *std::deque* modeled queue (i.e. "assembly line") is offered randomly to each of the *Factory* instances, until one of them consumes that *Part*. The weigth increases or decreases the chance of a kind of factory to receive the offer.

Consider this offering procedure: if the HEAD is a simple string valued kvpair and it gets offered to a factory for arrays of simple integer valued kvpairs, the kvpair can't be accepted by that specific *Factory* instance. Consider also that the container handling factories will usually require several *Part*s from the assembly line to get their "job order" fulfilled. The following explains the acceptance procedure:

1. *SimpleKvPairFactory* accepts integers
2. *SimpleKvPairFactory* accepts doubles
3. *SimpleKvPairFactory* accepts strings
4. *SimpleArrayFactory* accepts integers and INT kvpairs (drops key if such exists)
5. *SimpleArrayFactory* accepts doubles and DOUBLE kvpairs (drops key if such exists)
6. *SimpleArrayFactory* accepts strings and STRING kvpairs (drops key if such exists)
7. *SimpleObjectFactory* accepts INT kvpairs
8. *SimpleObjectFactory* accepts DOUBLE kvpairs
9. *SimpleObjectFactory* accepts STRING kvpairs
10. *ObjectArrayFactory* accepts objects
11. *ArrayArrayFactory* accepts arrays
12. *ArrayObjectFactory* accepts arrays
13. *ObjectObjectFactory* accepts objects
14. *MixedArrayFactory* accepts anything (drops key if such exists)
15. *MixedObjectFactory* accepts anything with a key

Only the first 6 "simple-consuming" factories, plus the *MixedArrayFactory*, can accept the "raw material". Only the factories 4.-9., plus the "Mixed" factories accept "second round" *Part*s. The rest will require "third round" or more complex *Part*s. Thus, it gets complicated to form arrays of objects of arrays of arrays or the like. The weigth may allow more chances for the more advanced factories to get their job completed, combined possibly with a greater ratio of recirculation for the simpler factories, and combined possibly with a suitable selection of min and max container sizes for each of the factories. The weigth may also be used to the other direction: for instance an array of mixed subtypes might not be very useful as a JSON entity, so it can be configured to be a nonexistent or rare *Part*.

A few notes about the implementation details of these factories. The interface for these is the *get(MuxParts&)* override. The *MuxParts* param is a thread safe accessor for the assembly line. The *get()* returns either null *std::shared_ptr* or a *std::shared_ptr* of a complete *Part*. Every factory checks the suitability of the assembly line received *Part* candidate, and for the container factories, they use the *bool match(PartPtr)* override for this purpose. The object factories need the extra step of ensuring that all the contained values (sub-parts one level down) have unique keys, as duplicate keys in objects are frowned upon by the JSON specification.

So, all in all, quite a complex machinery is required for "random complex JSON objects". There probably exists more sensible implementation ways for the end result, but the intent here was not only the end result but also the application of a producer-consumer threading model and a well-rounded utilization of C++ features.

The ProducerParams class

The command line interface to Jsonizer is a late addition and perhaps not as well structured as the earlier code.

The *Producer* class requires quite a complicated initialization, and the intent was to be able to give (part of) the initialization data from command line. Instead of adding multi-argument constructor(s) for it, the code uses a separate "params" class, thus splitting the maintenance requirements. Now the command line -> params and params -> *Producer* construction are separated. Although any change to the *ProducerParams* will need corresponding fix to the *Producer* construction, and vice versa, at least the constructor interface to *Producer* stays intact. Sometimes such don't-break-interfaces approach is beneficial for maintenance, although here it doesn't matter that much.

The *ProducerParams* collects the min/max/recirc/weigth parameters for all the *Factory* classes, which the *Producer* then instantiates, and the keys and values available for the application.

The Producer class

A *Producer* class instance is run in a dedicated thread. It takes in raw material in the form of ints, doubles and *std::string*s and produces key-value pairs and more complex container structures from this raw material, utilizing earlier discussed code portions.

Let's have a look at the *Producer* constructor, which is more involved than most normal C++ constructors.

First of all, the different types of base values given as parameters (via the *ProducerParams* param) get stored to member containers.

Second, the *KeyGetter* object, intended for forming the keys of the JSON key-value pairs, gets initialized from given parameters.

Third, the *init()* variadic lambda gets used. The next chapter discusses this in detail.

Fourth, the *KeyGetter* object gets "activated", meaning it is then able to slice the namespace for each of the object instances needing names.

Fifth, the *Factory* objects, all based on the pure virtual *FactoryBase*, get stored onto a *ConsumerProducer* container, which *Producer* then shall use for consuming and producing *Parts*.

The *Producer::order()* is the receiver of the raw material of integers, doubles and *std::string*s.

The *Producer::recirculate()* enables a special "zero waste" functionality for the *Assembly* threads. The *Assembly* objects construct JSON objects, and normally a JSON object should use unique keys for each of the contained kvpairs. Unfortunately the production algorithm might give an *Assembly* instance a kvpair it already has. Thus, the *Assembly* instance can recirculate a *Product* it already has for other *Assembly* objects or other *Producer Factory* objects to consume.

The *Producer::done()* is used for signaling the Producer thread that there are no more *Assembly* threads requiring its service, i.e. it is used for setting the thread terminate condition.

The *Producer::produce()* is the implementation of the actual "parts assembly line". In a loop, it consumes, recirculates and productizes *Part* objects until told to stop by the using threads. It picks a *Part* from the input queue and randomly offers this to the consumer *Factories*, until one of them accepts it, then it recirculates or productizes the output from that *Factory*. The *Producer* thread also goes to sleep if there's nothing to do so as not to consume CPU resources, utilizing condition variable design for this.

The init() variadic lambda

The Producer instance contains great many *Factory* members. Although the *Factories* are all part of the same virtual class hierarchy, their initialization is similar to a totally heterogenous group of objects. The constructors of these objects take in various amount of potentially differing types of values. The *init()* variadic lambda is essentially a one-statement initializer for all these objects. This greatly condenses the otherwise quite verbose class instance construct statements.

The only common denominator for all of the heterogenous objects given to *init()* is that they get constructed into *std::shared_ptr*s.

To dissect what *init()* has eaten, let's play with a generic example:

```
struct Curly {Curly(std::string const& c): mC(c){} std::string mC;};
struct Moe {Moe(std::string const& s, std::tuple<int,int,int> const& t): mS(s), mT(t){}
std::string mS; std::tuple<int,int,int> mT;};
struct Larry {Larry(int i, double d, Curly const& c): mI(i), mD(d), mC(c){} int mI;
double mT; Curly mC;};
struct Shemp {operator char const*() {return "Shemp";}};

using CurlyPtr = std::shared_ptr<Curly>;
using MoePtr = std::shared_ptr<Moe>;
using LarryPtr = std::shared_ptr<Larry>;
using ShempPtr = std::shared_ptr<Shemp>;
```

The init() needs to be explicitly informed of the **count** of constructor arguments but the argument types info is implicit. So:

```
CurlyPtr pc;
MoePtr pm;
LarryPtr pl;
ShempPtr ps;

init(tie2(pc,"stooge 1")
    ,tie3(pm,"stooge 2",(7873,7877,7879))
    ,tie4(pl,4597,42.1978,Curly("stooge 3"))
    ,tie1(ps));
```

With some clever additional compile time recursion, it might be possible to make even the constructor argument count implicit. Let's leave that as an exercise to the reader.

The *tieN()* helpers are a bit of boilerplate, but provided the argument count stays within reasonable bounds, writing them shouldn't be too big of a job:

```
template<typename T>
std::tuple<T&> tie1(T&& t)
{
return {t};
}

template<typename T, typename U>
std::tuple<T&,U> tie2(T&& t, U&& u)
{
return {t,u};
}

template<typename T, typename U, typename V>
std::tuple<T&,U,V> tie3(T&& t, U&& u, V&&v)
{
return {t,u,v};
}

template<typename T, typename U, typename V, typename W>
std::tuple<T&,U,V,W> tie4(T&& t, U&& u, V&& v, W&& w)
{
return {t,u,v,w};
}
```

Essentially the *tieN()* routines just store the heterogenous parameters into *std::tuple*s and return those tuples. Note that the first value must be a reference. That will be the returned, initialized *std::shared_ptr*. The other parameters are **universal references**. Do look that concept up from other literature.

The other set of boilerplate are the actual object constructions into those *std::shared_ptr*s. Note the usage of partial specialization for the classes:

```
template<std::size_t N,typename T> struct Tuple
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>();
}
};

template<typename T> struct Tuple<2,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t));
}
};

template<typename T> struct Tuple<3,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t),std::get<2>(t));
}
};

template<typename T> struct Tuple<4,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(
    std::get<1>(t),std::get<2>(t),std::get<3>(t));
}
};
```

Essentially these *Tuple* classes offer *init()* static factory functions for constructing the actual target objects.

The main fun(ction) is the actual variadic lambda:

```
    auto init{[](auto&&... t)
    {
    auto impl{[](auto&& me, auto&& head, auto&&... tail)
        {
        using T=typename std::remove_reference_t<
            std::tuple_element_t<0,
                std::remove_reference_t<decltype(head)>>>::element_type;

        Tuple<std::tuple_size_v<
            std::remove_reference_t<decltype(head)>>,T>::init(head);

        if constexpr(sizeof...(tail)>0)
            me(me,tail...);
        }};
    impl(impl,t...);
    }
    };
```

The main *init()* receives a variadic argument, essentially a list of *std::tuple*s, all of which have as head a *std::shared_ptr* of the target class instance and as tail the constructor arguments for that instance. This *init()* lambda declares an *impl()* sub-lambda, which is where the real magic occurs. The *impl()* receives universal reference to itself and the head and tail of the variadic argument list. It then performs some type resolving on the first element of the *std::tuple* in head, initializes the templated *Tuple* object from the head tuple size and the resolved result type and calls *Tuple::init()* for the head. Then *impl()* moves to the next tuple in the list until the list gets exhausted.

VBox(children=(Button(description='Snippet to godbolt', style=ButtonStyle()), Output()))

The full sample snippet:

```
#include <iostream>
#include <memory>
#include <tuple>

template<typename T>
std::tuple<T&> tie1(T&& t)
{
return {t};
}

template<typename T, typename U>
std::tuple<T&,U> tie2(T&& t, U&& u)
{
return {t,u};
}

template<typename T, typename U, typename V>
std::tuple<T&,U,V> tie3(T&& t, U&& u, V&&v)
{
return {t,u,v};
}

template<typename T, typename U, typename V, typename W>
std::tuple<T&,U,V,W> tie4(T&& t, U&& u, V&& v, W&& w)
{
return {t,u,v,w};
}

template<std::size_t N,typename T> struct Tuple
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>();
}
};

template<typename T> struct Tuple<2,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t));
}
};

template<typename T> struct Tuple<3,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t),std::get<2>(t));
}
};
```

```
template<typename T> struct Tuple<4,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(
    std::get<1>(t),std::get<2>(t),std::get<3>(t));
}
};

auto init{[](auto&&... t)
{
auto impl{[](auto&& me, auto&& head, auto&&... tail)
    {
    using T=typename std::remove_reference_t<
        std::tuple_element_t<0,
        std::remove_reference_t<decltype(head)>>>::element_type;

Tuple<std::tuple_size_v<
    std::remove_reference_t<decltype(head)>>,T>::init(head);

if constexpr(sizeof...(tail)>0)
    me(me,tail...);
}};
impl(impl,t...);
}
};

struct Curly {Curly(std::string const& c): mC(c){} std::string mC;};
struct Moe {Moe(std::string const& s, std::tuple<int,int,int> const& t): mS(s), mT(t){}
std::string mS; std::tuple<int,int,int> mT;};
struct Larry {Larry(int i, double d, Curly const& c): mI(i), mD(d), mC(c){} int mI;
double mD; Curly mC;};
struct Shemp {operator char const*() {return "Shemp";}};

using CurlyPtr = std::shared_ptr<Curly>;
using MoePtr = std::shared_ptr<Moe>;
using LarryPtr = std::shared_ptr<Larry>;
using ShempPtr = std::shared_ptr<Shemp>;

int main()
{
CurlyPtr pc;
MoePtr pm;
LarryPtr pl;
ShempPtr ps;

init(tie2(pc,"stooge 1")
    ,tie3(pm,"stooge 2",std::tuple<int,int,int>{7873,7877,7879})
    ,tie4(pl,4597,42.1978,Curly("stooge 3"))
    ,tie1(ps));

std::cout << "Curly: [" << pc->mC
    << "]\nMoe: [" << pm->mS << ", " << std::get<0>(pm->mT) << "/" << std::get<1>(pm->mT)
<< "/"<< std::get<2>(pm->mT)
    << "]\nLarry: [" << pl->mI << ", " << pl->mD << ", " << pl->mC.mC
    << "]\nShemp: [" << *ps << ']'
```

```
        << std::endl;
    }
```

```
VBox(children=(Button(description='Snippet to godbolt', style=ButtonStyle()), Output()))
```

The threading solution

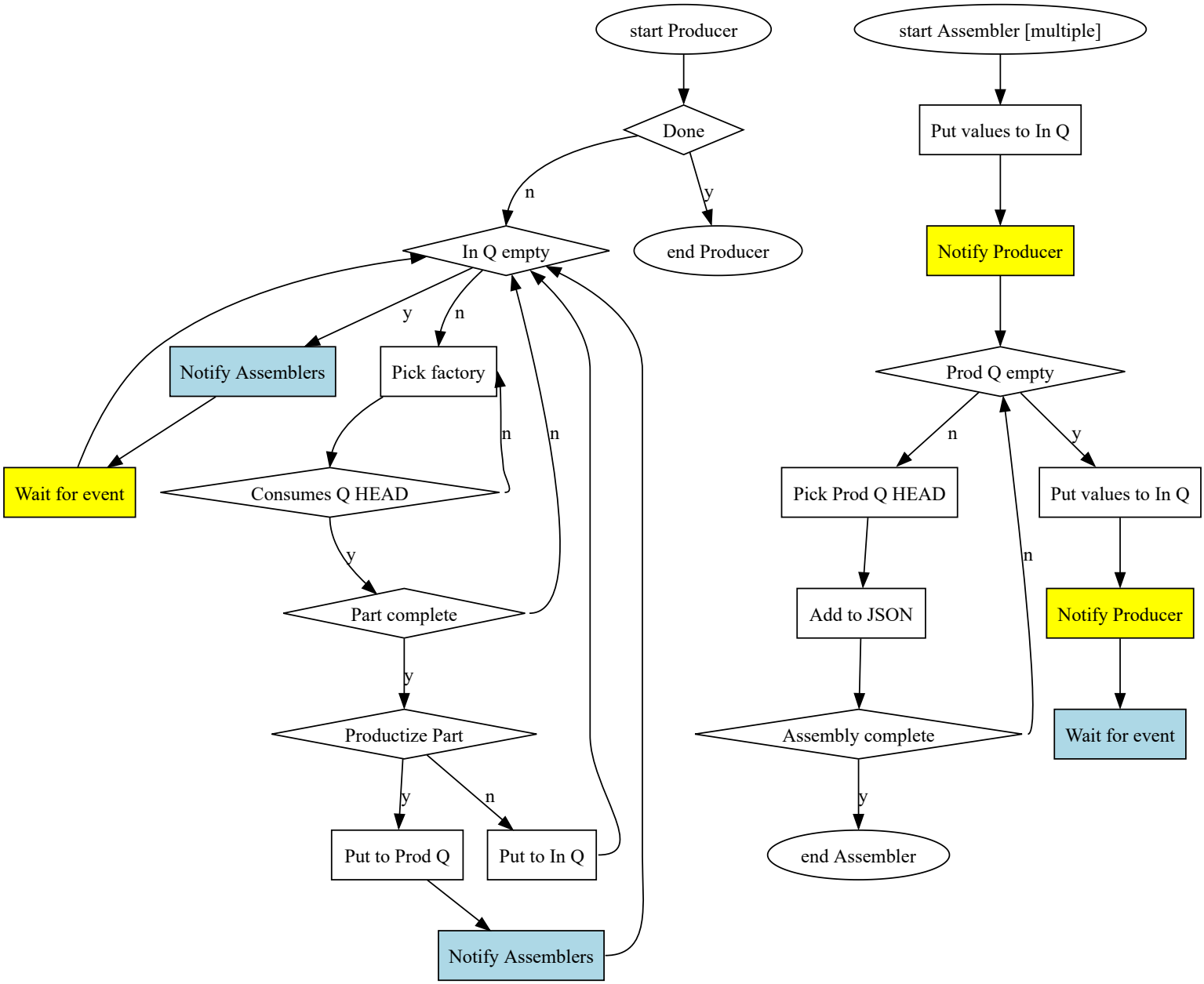
The threading in Jsonizer is modeled after a producer/consumer idea. There exists a Producer thread, managed by the Producer class, which also consumes its own products, and a number of separate Consumer threads, managed by the Assembly classes.

The "business" model here is that there's a factory that produces Parts, and these in turn can form larger and more complex Parts, which eventually form Products. Consider a toy car. It has parts such as wheels, axles, baseplate, engine, chassis, hood, windows, doors, and properties such as dimensions, weigth, color, etc. A factory might get an order of 1,000 wheels. It might also get an order of 100 red sedan chassises having windows and doors, including hood and trunk lids. The Assembly instances - another factories perhaps - then would combine their orders into marketable products. Something of this sort, but all the keys and values of the produced key-value pairs are semi random. No wheels to be found.

Threads share data via queues. There's an input queue with values and Parts on their way to become more complex Parts, and there's an output queue, which are the order fulfillments of the Producer factory.

Both of these queues can get empty, which mean the user of those queues has nothing further to do, until another thread adds items to the queue. This is handled with condition variables, one for the Producer, another for the Assemblies. There are "standard" mutexes protecting access to the condition variables, and there are shared_mutexes for providing single writer - multiple readers access to the queues.

Here is a rough flowchart of the threads interaction:



The Assembly class

The *Assembly* class instances are run each in their own thread. An *Assembly* instance is responsible for constructing the final JSON object from the *Product*s it receives from the corresponding queue.

An *Assembly* object gets instantiated with the minimum amount of integers, doubles and *std::string*s the final JSON object shall at least contain.

The *Assembly::run()* is the thread function. It begins with making an order of so-and so many values. Then it notifies the *Producer* thread to start working, and after that it goes to sleep itself, waiting to be notified by the *Producer* thread of new *Product*s.

At the main loop of *run()*, the required amount of values gets collected. If receiving of *Product*s stops (the *Product* queue gets starved), the *Assembly* object orders more values, notifies the *Producer* and again starts waiting on its condition variable.

Sometimes the *Assembly* object receives a new kvpair having a key it already owns, in which case it send the new kvpair back to the *Producer* to be consumed by somebody else.

After being satisfied the *Assembly* object collects and logs some performance statistics using *std::chrono::steady_clock* and *std::chrono::duration_cast*.

The threadize() function

The threadize() function sets up the stage for the whole show. It creates futures of the *Producer* and *Assembly* objects, then waits on the futures and after seeing all the Assembly futures being ready, it signals the Producer to get finished. The threadize() will create as many Assembly futures as parametrized to get done in parallel.

The command line handling

The *parseCmdLine()* and *usage()* functions, along with the *initPredefined()* function, get the rather complicated run setup done.

A command line processing is something the C++ standard helps very little in. The Boost library has some nice support but this excercise is all about standard C++. The *parseCmdLine()* contains a comma-separator lambda (*splitz*) and otherwise a decent, but verbose and unpretty processing of the rather complex command line syntax.

The *initPredefined()* contains as a curiosity a "lambda redirection map" in which a *std::string* key provides access to a function. This time all the lambdas do the same thing, calling *producerParams::setConsumerParams()*, but note that the *std::function* definition compliant lambdas could do anything. This is at times very powerful and convenient pattern. Note also the modern *for* loop pattern for *std::map*, which complements the redirection pattern quite nicely.

The main() function

Finally we arrive at the end of the source code of *jsonizer.cpp*. The *main()* has nothing very exotic C++ wise. The duration logging it adds to that of the *Assembly* is interesting though, as it gives statistics of the usefullness of dividing that JSON object construction into parallel threads.

The full source code of jsonizer.cpp

```
VBox(children=(Button(description='C++ app to godbolt', style=ButtonStyle()), Output()))
```

```
 /**
JSON object factory.

VERSION 1.0

This is a threading and other C++ features demo application that acts
as a factory for JSON data.

The logic of the application models a kind of a "parts and products" processing
factory (or factories).

A Producer thread generates semi random JSON key-value pairs, here
called Parts, in the fashion of "basename_a":<value> ... "basename_zzz":<value>,
where values are basic JSON value types of ints, doubles and strings.

These Parts get pushed into a queue.

Then consumer factories within the same thread are given the chance of getting
these Parts from the queue in order to form larger Parts consisting of
JSON arrays and objects, provided that the queue head Part meets the required
criteria (e.g. "array of ints" factory only accepts int kvpairs).
These more complex Parts in turn can be pushed back to the queue to form
even larger Parts.

This process is a kind of a modeling of an assembly line.
All types of Parts can be determined to be final products, in which case they
get moved into another queue and become Products.

The Products in turn are consumed by Assembly threads, which form the final
JSON objects, one per Assembly thread.

The setup enables creating large and complex JSON objects.

The criteria for a complete JSON object is that the count of simple JSON
compatible values (integers, doubles and strings) equals or exceeds the given
parameters. At Assembly thread startup, the thread adds the requested amount of
simple values to the work queue of the Producer. However, these values aren't
"earmarked" to that particular Assembly thread but instead can be consumed by
any other Assembly thread as well. As the consuming of the Products consists of
simple key-value pairs as well as arrays, objects, or multi-dimensional
combinations of those, an Assembly thread will likely consume more values than
it originally fed to the work queue, thus potentially starving other Assembly
threads, or the not-ready Parts (like arrays, which have predetermined min/max
ranges for their sizes) within the Producer thread. Thus, when an Assembly
thread can't seem to receive new Products, it feeds some more values to the
system.

The parametrization of the Producer object greatly affects what kind of
JSON objects get created.
```

```
This application is kind of meant to act as a service, i.e. clients request
JSON data, and the data then gets generated parallelly per request for fast
response time, but the service triggering and response giving mechanisms
(e.g. REST API) have yet to be implemented.
*/

#include <atomic>
#include <chrono>
#include <condition_variable>
#include <cstdlib>
#include <cstring>
#include <deque>
#include <functional>
#include <future>
#include <iostream>
#include <map>
#include <memory>
#include <mutex>
#include <optional>
#include <random>
#include <set>
#include <shared_mutex>
#include <sstream>
#include <string>
#include <thread>
#include <tuple>
#include <vector>

using namespace std::chrono_literals;

namespace
{
using V_S=std::vector<std::string>;
using V_I=std::vector<int>;
using V_D=std::vector<double>;

using D_S=std::deque<std::string>;
using D_I=std::deque<int>;
using D_D=std::deque<double>;

using D_S_Ptr=std::shared_ptr<D_S>;
using D_I_Ptr=std::shared_ptr<D_I>;
using D_D_Ptr=std::shared_ptr<D_D>;

using OptString=std::optional<std::string>;
using Key=OptString;
using Value=OptString;

using Serial=int;

const std::size_t DEFAULT_MINSIZE{1};
const std::size_t DEFAULT_MAXSIZE{2};
const std::size_t DEFAULT_RECIRC{50};
const std::size_t DEFAULT_WEIGTH{1};

enum class CT
{
 KI,KD,KS
```

```
,AI,AD,AS,AA,AO,AM
,OI,OD,OS,OA,OO,OM
};

enum ERRORS
{
NO
, USAGE
, CMDLINE_INVALID_PREDEFINED
, CMDLINE_EXCEPTION
};

std::mutex muxCvProd,muxCvAsse,muxLog;
std::condition_variable cvProd,cvAsse;

void print(std::string const& s)
{
std::cout << s << std::endl;
}

#define LOG(data) ({std::lock_guard lock(muxLog); \
std::stringstream ss__LINE__; ss__LINE__ << data; print(ss__LINE__.str());})

V_S g_substantives {
"abaxiator","adscititiouser","affranchiser","aoristicor","athwarter"
,"beaconacor","bheestier","biconcaver","blitherer","buckrammer"
,"centuplicator","chicanerer","coarticulor","cribriformer","ctenidiumer"
,"dactyolizer","delabializator","diminuendor","dubitator","dwindler"
,"eccentrizer","elasticizer","enantiotrophier","eosinophiler","equiprobabilizer"
,"fenestrator","firnificator","flagellator","foliculator","foppisher"
,"gesticulator","ghoulizer","gimcrackerizer","glaciator","gobbledegooker"
,"haplographier","hemistitcher","hierarchizer","horologizer","hyalogizer"
,"illminator","inviolator","iotacer","isomorpher","itemizer"
,"jangler","jettisoner","jibber","jotter","jurisprudenter"
,"katamorpher","kinaestethor","knaverer","kottabosser","kyoodler"
,"laborizer","legitimizer","ligaturer","listlessor","locator"
,"maculator","merchandizor","mimesizer","modalator","multifarier"
,"namablor","negligor","nicher","nocturner","nuncupator"
,"oblanceolator","octamerer","officializer","omitter","oxymoronizer"
,"parasynthesizer","pedimentor","phantastronizer","pickler","plagiotropisizer"
,"quacker","quaererizor","quantumizer","quarreler","quaternator"
,"rachiformer","readjustor","rinser","rollicker","ruinator"
,"salienator","scatterer","segmentalizer","shaper","sinuouser"
,"tanstaafler","tediumizer","thougher","tillyvallier","toilsomizer"
,"ubiquitter","ultimator","umbriferouser","unconformer","upsurger"
,"valuator","vehiculumizer","vinculumizer","vorticer","vulganizer"
,"wackier","whammier","wiggler","wreather","wrought-upper"
,"xanthiciser","xerarchizer","x-unitizer","xylographer","xylotomizer"
,"yarner","yerker","yielder","yonderer","yummizer"
,"zagger","zanizer","zonator","zoomer","zymosizer"};

Serial serialGenerator{};

std::random_device rd;
std::mt19937 mt{rd()};

template<typename T, typename U>
std::tuple<T&,U> tie2(T&& t, U&& u)
```

```
{
return {t,u};
}

template<typename T, typename U, typename V>
std::tuple<T&,U,V> tie3(T&& t, U&& u, V&&v)
{
return {t,u,v};
}

template<typename T, typename U, typename V, typename W>
std::tuple<T&,U,V,W> tie4(T&& t, U&& u, V&& v, W&& w)
{
return {t,u,v,w};
}

template<typename T, typename U, typename V, typename W, typename X>
std::tuple<T&,U,V,W,X> tie5(T&& t, U&& u, V&& v, W&& w, X&& x)
{
return {t,u,v,w,x};
}

template<std::size_t N,typename T> struct Tuple
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>();
}
};

template<typename T> struct Tuple<2,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t));
}
};

template<typename T> struct Tuple<3,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(std::get<1>(t),std::get<2>(t));
}
};

template<typename T> struct Tuple<4,T>
{
template<typename U> static void init(U& t)
{
std::get<0>(t)=std::make_shared<T>(
    std::get<1>(t),std::get<2>(t),std::get<3>(t));
}
};

template<typename T> struct Tuple<5,T>
{
template<typename U> static void init(U& t)
```

```
{
std::get<0>(t)=std::make_shared<T>(
    std::get<1>(t),std::get<2>(t),std::get<3>(t),std::get<4>(t));
}
};

auto init{[](auto&&... t)
    {
    auto impl{[](auto&& me, auto&& head, auto&&... tail)
        {
        using T=typename std::remove_reference_t<
            std::tuple_element_t<0,
                std::remove_reference_t<decltype(head)>>>::element_type;

        Tuple<std::tuple_size_v<
            std::remove_reference_t<decltype(head)>>,T>::init(head);

        if constexpr(sizeof...(tail)>0)
            me(me,tail...);
    }};
    impl(impl,t...);
    }
};
} // unnamed

//-----------------------------------------------------------------------------
template<typename T=std::string>
class DisNDat
{
public:

DisNDat(T const& dis, T const& dat)
    : mDis(dis)
    , mDat(dat)
{}

T const& get() const
{
    if(!mDissed)
    {
        mDissed=true;
        return mDis;
    }
    return mDat;
}

private:

    T mDis;
    T mDat;
    mutable bool mDissed{};

friend std::ostream& operator<<(std::ostream& os, DisNDat const& rhs)
{
    os << rhs.get();
    return os;
}
};
```

```
//---------------------------------------------------------------------------
template<typename T> std::string conv(T& t)
{
return std::to_string(t);
}

std::string conv(std::string const& t)
{
return "\"" + t + "\"";
}

template<typename T> std::string getFrom(
    T const& t,
    std::size_t start,
    std::size_t size)
{
    auto ix{std::min(t.size()-1,start+mt()%size)};
    return conv(t[ix]);
}

//---------------------------------------------------------------------------
struct Part;
using PartPtr=std::shared_ptr<Part>;
using D_PartPtr=std::deque<PartPtr>;
using OptD_PartPtr=std::optional<D_PartPtr>;

struct Part
{
enum class SimpleType
    {
     INT
    ,DOUBLE
    ,STRING
    };

enum class Type
    {
     INT
    ,DOUBLE
    ,STRING
    ,ARRAY
    ,OBJECT
    };

static Type T2T(SimpleType const& type)
{
return type==SimpleType::INT
    ? Type::INT
    : type==SimpleType::DOUBLE
        ? Type::DOUBLE
        : Type::STRING;
}

bool isSimple() const
{
return mType==Type::INT || mType==Type::DOUBLE || mType==Type::STRING;
}
```

```
Part(
    Type type
    ,Key const& key
    ,Value const& val=OptString()
    ,PartPtr sub=nullptr)
    : mSerial(++serialGenerator)
    , mType(type)
    , mKey(key)
    , mValue(val)
    , mValueCount(isSimple() ? 1 : 0)
{
if(sub)
    mSubs=D_PartPtr{sub};
}

Part(
    Type type
    ,OptD_PartPtr const& subs
    ,Key const& key
    ,Value const& val=OptString())
    : mSerial(++serialGenerator)
    , mType(type)
    , mKey(key)
    , mValue(val)
    , mSubs(subs)
    , mValueCount(isSimple() ? 1 : 0)
{}

explicit Part(int val, OptString const& key=OptString())
    : mSerial(++serialGenerator)
    , mType(Type::INT)
    , mKey(key)
    , mValue(conv(val))
    , mValueCount(1)
{}

explicit Part(double val, OptString const& key=OptString())
    : mSerial(++serialGenerator)
    , mType(Type::DOUBLE)
    , mKey(key)
    , mValue(conv(val))
    , mValueCount(1)
{}

explicit Part(std::string const& val, OptString const& key=OptString())
    : mSerial(++serialGenerator)
    , mType(Type::STRING)
    , mKey(key)
    , mValue(conv(val))
    , mValueCount(1)
{}

bool match(SimpleType type) const
{
return type==SimpleType::INT
    ? mType==Type::INT
    : type==SimpleType::DOUBLE
```

```
        ? mType==Type::DOUBLE
        : mType==Type::STRING;
}

std::size_t valueCount(SimpleType type) const
{
if(isSimple())
    return mType==T2T(type) ? mValueCount : 0;

std::size_t count{};
if(mSubs)
    for(auto const& i: *mSubs)
        count+=i->valueCount(type);

return count;
}

Type type() const
{
return mType;
}

Key const& key() const
{
return mKey;
}

void setKey(Key const& key)
{
mKey=key;
}

Value const& value() const
{
return mValue;
}

OptD_PartPtr& subs()
{
return mSubs;
}

Serial const& serial() const
{
return mSerial;
}

private:

Serial mSerial;
Type mType;
Key mKey;
Value mValue;
OptD_PartPtr mSubs;
std::size_t mValueCount{};

friend std::ostream& operator<<(std::ostream& os, Part const& rhs)
{
```

```
if(rhs.mKey)
    os << *rhs.mKey << ':';

switch(rhs.mType)
    {
    case Type::INT:
    case Type::DOUBLE:
    case Type::STRING:
        os << *rhs.mValue;
        break;
    case Type::ARRAY:
        os << '[';
        if(rhs.mSubs)
            {
            DisNDat<> c("",",");
            for(auto& i:*rhs.mSubs)
                if(i)
                    os << c << *i;
            }
        os << ']';
        break;
    case Type::OBJECT:
        os << '{';
        if(rhs.mSubs)
            {
            DisNDat<> c{"",","};
            for(auto& i:*rhs.mSubs)
                if(i)
                    os << c << *i;
            }
        os << '}';
    }
return os;
}

friend std::ostream& operator<<(std::ostream& os, Part::Type const& rhs)
{
switch(rhs)
    {
    case Part::Type::INT:
        os << "INT";
        break;
    case Part::Type::DOUBLE:
        os << "DOUBLE";
        break;
    case Part::Type::STRING:
        os << "STRING";
        break;
    case Part::Type::ARRAY:
        os << "ARRAY";
        break;
    case Part::Type::OBJECT:
        os << "OBJECT";
    }
return os;
}
};
```

```
//-----------------------------------------------------------------------------
struct MuxParts
{
void push_back(PartPtr p)
{
std::unique_lock lock(mMux);
mParts.push_back(p);
}

void pop_front()
{
std::unique_lock lock(mMux);
mParts.pop_front();
}

bool empty()
{
std::shared_lock lock(mMux);
return mParts.empty();
}

std::size_t size()
{
std::shared_lock lock(mMux);
return mParts.size();
}

PartPtr const& front()
{
std::shared_lock lock(mMux);
return mParts.front();
}

PartPtr get()
{
std::unique_lock lock(mMux);
if(mParts.empty())
    return PartPtr();

auto part{mParts.front()};
mParts.pop_front();
return part;
}

private:

std::shared_mutex mMux;
D_PartPtr mParts;
};

//-----------------------------------------------------------------------------
template<std::size_t BASE> struct BaseN
{
explicit BaseN(char zero)
    : mZero(zero)
{}

BaseN& operator++()
```

```
{
++mVal;
return *this;
}

BaseN operator++(int)
{
BaseN base{*this};
++mVal;
return base;
}

std::string operator*()
{
auto val{mVal};
std::string s;
while(val)
    {
    auto digit{static_cast<std::size_t>(val%BASE)};
    val=(val-digit)/BASE;
    s=std::string(1,mZero+digit)+s;
    }
return s;
}

private:

char mZero;
std::size_t mVal{};
};

//----------------------------------------------------------------------------
struct KeyGetterBase
{
using Token=std::size_t;

virtual ~KeyGetterBase() = default;
virtual std::size_t keyCount(Token) const = 0;
virtual std::string get(Token) const = 0;

virtual Token reg()
{
return 0;
}

virtual void activate(){}
};

using KeyGetterBasePtr = std::shared_ptr<KeyGetterBase>;

//----------------------------------------------------------------------------
struct KeyGetter : public KeyGetterBase
{
KeyGetter(V_S names, std::size_t count)
{
if(!count)
    mKeys.swap(names);
else
```

```
    {
    mKeys=names;
    BaseN<1+'z'-'a'> b{'a'};
    for(; count; --count, ++b)
        {
        auto s{*b};
        if(!s.empty())
            s="_"+s;
        for(auto const& i: names)
            mKeys.emplace_back(i+s);
        }
    }
mSlice=mKeys.size();
}

std::size_t keyCount(Token tok) const override
{
auto next{tok*mSlice+mSlice};
auto tail{mKeys.size()-next};
return tail>0 && tail<mSlice ? mSlice+tail : mSlice;
}

std::string get(Token tok) const override
{
return getFrom(mKeys,tok*mSlice,keyCount(tok));
}

Token reg() override
{
return mToken++;
}

virtual void activate()
{
mSlice=mKeys.size()/mToken;
}

private:

V_S mKeys;
Token mToken{};
std::size_t mSlice;
};

//----------------------------------------------------------------------------
template<typename T> struct SimpleValueGenerator
{
using DPtr=std::shared_ptr<std::deque<T>>;

SimpleValueGenerator(DPtr&& values)
    : mValues(values)
{}

PartPtr get() const
{
if(!mValues || mValues->empty())
    return PartPtr();
```

```
return std::make_shared<Part>((*mValues)[mt()%mValues->size()]);
}

private:

DPtr mValues;
};

//-----------------------------------------------------------------------------
struct FactoryBase
{
virtual ~FactoryBase()=default;
virtual PartPtr get(MuxParts& queue)=0;
};

using FactoryBasePtr=std::shared_ptr<FactoryBase>;

//-----------------------------------------------------------------------------
template<Part::SimpleType N> struct SimpleKvPairFactory
    : public FactoryBase
{
using Ptr=std::shared_ptr<SimpleKvPairFactory>;

SimpleKvPairFactory(KeyGetterBasePtr keys)
    : mpKeys(keys)
{
if(keys)
    mTok=keys->reg();
}

PartPtr get(MuxParts& queue) override
{
if(queue.empty())
    return PartPtr();

auto part{queue.front()};
if(!part)
    return PartPtr();

queue.pop_front();
auto keys{mpKeys.lock()};
if(keys && part->match(N) && !part->key() && part->value())
    return std::make_shared<Part>(Part::T2T(N),keys->get(mTok),part->value());

return PartPtr();
}

private:

std::weak_ptr<KeyGetterBase> mpKeys;
KeyGetter::Token mTok{};
};

//-----------------------------------------------------------------------------
struct ContainerFactoryBase : public FactoryBase
{
ContainerFactoryBase(
    std::size_t minLen,
```

```
    std::size_t maxLen,
    bool autoClear,
    Part::Type partType,
    KeyGetterBasePtr keys)
    : mMinLen(minLen)
    , mMaxLen(maxLen)
    , mExpectedLen(maxLen<=minLen ? minLen : (mt()%(1+maxLen-minLen)+minLen))
    , mAutoClear(autoClear)
    , mPartType(partType)
    , mpKeys(keys)
{
if(keys)
    mTok=keys->reg();
}

virtual bool match(PartPtr p) const=0;

PartPtr get(MuxParts& queue) override
{
if(mSubs.size()<mExpectedLen)
    {
    PartPtr p;
    if(!queue.empty())
        p=queue.front();

    if(match(p))
        {
        if(mPartType==Part::Type::ARRAY)
            p->setKey(Key());
        mSubs.push_back(p);
        queue.pop_front();
        }
    if(mSubs.size()<mExpectedLen)
        return PartPtr();
    }
auto keys{mpKeys.lock()};
if(!keys)
    return PartPtr();

auto part{std::make_shared<Part>(mPartType,mSubs,keys->get(mTok))};
mExpectedLen=mMaxLen<=mMinLen ? mMinLen : (mt()%(1+mMaxLen-mMinLen)+mMinLen);
if(mAutoClear)
    mSubs.clear();

return part;
}

bool uniqueKey(std::string const& rhs) const
{
for(auto const& i: mSubs)
    if(i && i->key() && *(i->key())==rhs)
        return false;

return true;
}

private:
```

```
D_PartPtr mSubs;
std::size_t mMinLen;
std::size_t mMaxLen;
std::size_t mExpectedLen;
bool mAutoClear{};
Part::Type mPartType;
std::weak_ptr<KeyGetterBase> mpKeys;
KeyGetter::Token mTok{};
};

//------------------------------------------------------------------------------
template<Part::SimpleType N> struct SimpleArrayFactory
    : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<SimpleArrayFactory>;

SimpleArrayFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::ARRAY,keys)
{}

bool match(PartPtr p) const override
{
return p && p->match(N) && !p->key() && p->value();
}
};

//------------------------------------------------------------------------------
template<Part::SimpleType N> struct SimpleObjectFactory
    : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<SimpleObjectFactory>;

SimpleObjectFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::OBJECT,keys)
{}

bool match(PartPtr p) const override
{
return p && p->match(N) && p->key() && p->value() && uniqueKey(*(p->key()));
}
};

//------------------------------------------------------------------------------
struct ObjectArrayFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<ObjectArrayFactory>;

ObjectArrayFactory(
    std::size_t minLen,
    std::size_t maxLen,
```

```
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::ARRAY,keys)
{}

bool match(PartPtr p) const override
{
return p && p->type()==Part::Type::OBJECT;
}
};

//------------------------------------------------------------------------------
struct ArrayArrayFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<ArrayArrayFactory>;

ArrayArrayFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::ARRAY,keys)
{}

bool match(PartPtr p) const override
{
return p && p->type()==Part::Type::ARRAY && p->subs()
   && (p->subs()->empty()
       || ((*p->subs())[0]
           && (*p->subs())[0]->type()!=Part::Type::ARRAY));
}
};

//------------------------------------------------------------------------------
struct MixedArrayFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<MixedArrayFactory>;

MixedArrayFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::ARRAY,keys)
{}

bool match(PartPtr p) const override
{
return p!=nullptr;
}
};

//------------------------------------------------------------------------------
struct ArrayObjectFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<ArrayObjectFactory>;

ArrayObjectFactory(
```

```
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::OBJECT,keys)
{}

bool match(PartPtr p) const override
{
return p && p->key()  && uniqueKey(*(p->key())) && p->type()==Part::Type::ARRAY;
}
};

//------------------------------------------------------------------------------
struct ObjectObjectFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<ObjectObjectFactory>;

ObjectObjectFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::OBJECT,keys)
{}

bool match(PartPtr p) const override
{
return p && p->key() && uniqueKey(*(p->key())) && p->type()==Part::Type::OBJECT;
}
};

//------------------------------------------------------------------------------
struct MixedObjectFactory : public ContainerFactoryBase
{
using Ptr=std::shared_ptr<MixedObjectFactory>;

MixedObjectFactory(
    std::size_t minLen,
    std::size_t maxLen,
    bool autoClear,
    KeyGetterBasePtr keys)
    : ContainerFactoryBase(minLen,maxLen,autoClear,Part::Type::OBJECT,keys)
{}

bool match(PartPtr p) const override
{
return p && p->key() && uniqueKey(*(p->key()));
}
};

//------------------------------------------------------------------------------
struct ProducerParams
{
struct ConsumerParams
{
std::size_t min{};
std::size_t max{};
```

```
std::size_t recirc{};
std::size_t weigth{};
};

using M_ConsumerParams=std::map<CT,ConsumerParams>;
using D_D_I=std::vector<D_I>;
using D_D_D=std::vector<D_D>;
using D_D_S=std::vector<D_S>;

D_D_I ints() const
{
return mInts;
}

D_D_D doubles() const
{
return mDoubles;
}

D_D_S strings() const
{
return mStrings;
}

void addInts(D_I const& rhs)
{
mInts.push_back(rhs);
}

void addDoubles(D_D const& rhs)
{
mDoubles.push_back(rhs);
}

void addStrings(D_S const& rhs)
{
mStrings.push_back(rhs);
}

ConsumerParams const& operator[](CT i)
{
return mCons[i];
}

void setKeys(V_S keys)
{
mKeys.swap(keys);
}

void setKeyMultiplier(std::size_t rhs)
{
mMultiplier=rhs;
}

V_S const& keys() const
{
return mKeys;
}
```

```
std::size_t keyMultiplier() const
{
return mMultiplier;
}

void setConsumerParams(M_ConsumerParams rhs)
{
mCons.swap(rhs);
}

void setConsumerParam(CT ct, ConsumerParams const& cp)
{
mCons[ct]=cp;
}

private:

V_S mKeys;
std::size_t mMultiplier{};
D_D_I mInts;
D_D_D mDoubles;
D_D_S mStrings;
M_ConsumerParams mCons;
};

//----------------------------------------------------------------------------
struct Producer
{
enum IX
{
 FACTORY
,PERCENTAGE
,WEIGTH
};

public:

Producer(ProducerParams par)
{
for(auto const& i: par.ints())
    mValueFIs.push_back(D_I_Ptr{new D_I{i}});

for(auto const& i: par.doubles())
    mValueFDs.push_back(D_D_Ptr{new D_D{i}});

for(auto const& i: par.strings())
    mValueFSs.push_back(D_S_Ptr{new D_S{i}});

mKeyGetter=std::make_shared<KeyGetter>(par.keys(),par.keyMultiplier());

init(
     tie2(mKvpFI,mKeyGetter)
    ,tie2(mKvpFD,mKeyGetter)
    ,tie2(mKvpFS,mKeyGetter)
    ,tie5(mArrayFI,par[CT::AI].min,par[CT::AI].max,true,mKeyGetter)
    ,tie5(mArrayFD,par[CT::AD].min,par[CT::AD].max,true,mKeyGetter)
    ,tie5(mArrayFS,par[CT::AS].min,par[CT::AS].max,true,mKeyGetter)
```

```
    ,tie5(mObjectFI,par[CT::OI].min,par[CT::OI].max,true,mKeyGetter)
    ,tie5(mObjectFD,par[CT::OD].min,par[CT::OD].max,true,mKeyGetter)
    ,tie5(mObjectFS,par[CT::OS].min,par[CT::OS].max,true,mKeyGetter)
    ,tie5(mObjArray,par[CT::AO].min,par[CT::AO].max,true,mKeyGetter)
    ,tie5(mArrayArray,par[CT::AA].min,par[CT::AA].max,true,mKeyGetter)
    ,tie5(mMixedArray,par[CT::AM].min,par[CT::AM].max,true,mKeyGetter)
    ,tie5(mArrayObj,par[CT::OA].min,par[CT::OA].max,true,mKeyGetter)
    ,tie5(mObjObj,par[CT::OO].min,par[CT::OO].max,true,mKeyGetter)
    ,tie5(mMixedObj,par[CT::OM].min,par[CT::OM].max,true,mKeyGetter));

mKeyGetter->activate();

mConsumers={
     {mKvpFI,par[CT::KI].recirc,par[CT::KI].weigth}
    ,{mKvpFD,par[CT::KD].recirc,par[CT::KD].weigth}
    ,{mKvpFS,par[CT::KS].recirc,par[CT::KS].weigth}

    ,{mArrayFI,par[CT::AI].recirc,par[CT::AI].weigth}
    ,{mArrayFD,par[CT::AD].recirc,par[CT::AD].weigth}
    ,{mArrayFS,par[CT::AS].recirc,par[CT::AS].weigth}

    ,{mObjectFI,par[CT::OI].recirc,par[CT::OI].weigth}
    ,{mObjectFD,par[CT::OD].recirc,par[CT::OD].weigth}
    ,{mObjectFS,par[CT::OS].recirc,par[CT::OS].weigth}

    ,{mObjArray,par[CT::AO].recirc,par[CT::AO].weigth}
    ,{mArrayArray,par[CT::AA].recirc,par[CT::AA].weigth}
    ,{mMixedArray,par[CT::AM].recirc,par[CT::AM].weigth}

    ,{mArrayObj,par[CT::OA].recirc,par[CT::OA].weigth}
    ,{mObjObj,par[CT::OO].recirc,par[CT::OO].weigth}
    ,{mMixedObj,par[CT::OM].recirc,par[CT::OM].weigth}
    };
}

void order(int ints, int doubles, int strings)
{
if(ints>0)
    {
    auto ix{mt()%mValueFIs.size()};
    for(; ints>=0; --ints)
        mParts.push_back(mValueFIs[ix].get());
    }
if(doubles>0)
    {
    auto ix{mt()%mValueFDs.size()};
    for(; doubles>=0; --doubles)
        mParts.push_back(mValueFDs[ix].get());
    }
if(strings>0)
    {
    auto ix{mt()%mValueFSs.size()};
    for(; strings>=0; --strings)
        mParts.push_back(mValueFSs[ix].get());
    }
}

void recirculate(PartPtr p)
```

```
{
if(p)
    mParts.push_back(p);
}

void done()
{
mDone=true;
}

PartPtr get()
{
return mProducts.get();
}

std::string produce()
{
std::vector<std::size_t> key,key2;
for(std::size_t i=0; i<mConsumers.size(); ++i)
    for(std::size_t j=0; j<std::get<IX::WEIGTH>(mConsumers[i]); ++j)
        key2.push_back(i);

key=key2;
std::size_t madeProducts{};
std::map<Part::Type,std::size_t> madeTypes;
while(!mDone)
    {
    if(mParts.empty())
        {
        std::unique_lock lock{muxCvProd};
        cvProd.wait(lock,[&mParts=mParts,&mDone=mDone]
            {
            return !mParts.empty() || mDone;
            });
        continue;
        }
    auto candidate{mParts.front()->serial()};
    while(!key.empty())
        {
        auto ix{mt() % key.size()};
        auto ixx{key[ix]};
        key.erase(key.begin()+ix);
        auto& consumer{mConsumers[ixx]};
        auto part{std::get<IX::FACTORY>(consumer)->get(mParts)};
        if(part)
            {
            if(100-std::get<IX::PERCENTAGE>(consumer) < mt()%100)
                mParts.push_back(part);
            else
                {
                ++madeTypes[part->type()];
                mProducts.push_back(part);
                    {
                    std::lock_guard lock{muxCvAsse};
                    cvAsse.notify_all();
                    }
                if(!(++madeProducts % 100))
                    {
```

```
                    DisNDat<> c("",", ");
                    std::stringstream ss;
                    ss << "Products created: " << madeProducts << " (";
                    for(auto const& [k,v]: madeTypes)
                        ss << c << k << ": " << v;

                    ss << "); queue size: " << mParts.size();
                    LOG(ss.str());
                    }
                }
            }
        }
    key=key2;
    if(mParts.empty())
        {
        std::lock_guard lock{muxCvAsse};
        cvAsse.notify_all();
        }
    else if(candidate==mParts.front()->serial())
        {
        ++mMisses[candidate];
        if(mMisses[candidate]>2)
            {
            LOG("NOT CONSUMED: " << *mParts.front());
            mProducts.push_back(mParts.front());
            mParts.pop_front();
            mMisses.erase(candidate);
            if(mParts.empty())
                {
                std::lock_guard lock{muxCvAsse};
                cvAsse.notify_all();
                }
            }
        else
            {
            mParts.push_back(mParts.front());
            mParts.pop_front();
            }
        }
    }
LOG("Total products created: " << madeProducts
    << "\nLeftover queue size: " << mParts.size()
    << "\nLeftover products: ");
DisNDat<> c("",",");
std::stringstream ss;
while(!mProducts.empty())
    {
    ss << c << *mProducts.front();
    mProducts.pop_front();
    }
return ss.str();
}

private:

std::deque<SimpleValueGenerator<int>> mValueFIs;
std::deque<SimpleValueGenerator<double>> mValueFDs;
std::deque<SimpleValueGenerator<std::string>> mValueFSs;
```

```
SimpleKvPairFactory<Part::SimpleType::INT>::Ptr mKvpFI;
SimpleKvPairFactory<Part::SimpleType::DOUBLE>::Ptr mKvpFD;
SimpleKvPairFactory<Part::SimpleType::STRING>::Ptr mKvpFS;

SimpleArrayFactory<Part::SimpleType::INT>::Ptr mArrayFI;
SimpleArrayFactory<Part::SimpleType::DOUBLE>::Ptr mArrayFD;
SimpleArrayFactory<Part::SimpleType::STRING>::Ptr mArrayFS;

SimpleObjectFactory<Part::SimpleType::INT>::Ptr mObjectFI;
SimpleObjectFactory<Part::SimpleType::DOUBLE>::Ptr mObjectFD;
SimpleObjectFactory<Part::SimpleType::STRING>::Ptr mObjectFS;

ObjectArrayFactory::Ptr mObjArray;
ArrayArrayFactory::Ptr mArrayArray;
ArrayArrayFactory::Ptr mMixedArray;

ArrayObjectFactory::Ptr mArrayObj;
ObjectObjectFactory::Ptr mObjObj;
ObjectObjectFactory::Ptr mMixedObj;

// Tuple items:                    factory       ,recirc% ,weigth*
using ConsumerProducer=std::tuple<FactoryBasePtr,unsigned,unsigned>;
std::vector<ConsumerProducer> mConsumers;

MuxParts mParts;
MuxParts mProducts;
std::map<Serial,int> mMisses;
KeyGetterBasePtr mKeyGetter;
std::atomic<bool> mDone{};
};

//------------------------------------------------------------------------------
struct Assembly
{
Assembly(
    std::shared_ptr<Producer> prod,
    int ints,
    int doubles,
    int strings)
    : mProd(prod)
    , mInts(ints)
    , mDoubles(doubles)
    , mStrings(strings)
{}

std::string run()
{
auto beg{std::chrono::steady_clock::now()};
mProd->order(mInts,mDoubles,mStrings);
{
std::lock_guard lock{muxCvProd};
cvProd.notify_one();
}
{
std::unique_lock lock{muxCvAsse};
cvAsse.wait(lock);
}
```

```
int iCount{};
int dCount{};
int sCount{};
std::stringstream ss;
ss << '{';
DisNDat<> c("",",");
std::map<std::string,Part::Type> keys;
while((iCount<mInts || dCount<mDoubles || sCount<mStrings))
    {
    auto prod{mProd->get()};
    if(!prod)
        {
        mProd->order(mInts-iCount>0 ? 1:0,
            mDoubles-dCount>0 ? 1:0,mStrings-sCount>0 ? 1:0);
        {
        std::lock_guard lock{muxCvProd};
        cvProd.notify_one();
        }
        std::unique_lock lock{muxCvAsse};
        cvAsse.wait(lock);
        continue;
        }
    bool recirc{};
    if(!prod->key())
        recirc=true;
    else
        {
        if(keys.find(*prod->key())!=keys.end())
            recirc=true;
        else
            keys[*prod->key()]=prod->type();
        }
    if(recirc)
        {
            LOG("recirc object: " << *prod->key()
                << " type: " << prod->type()
                << " serial: " << prod->serial());

        mProd->recirculate(prod);
        continue;
        }
    ss << c << *prod;
    iCount+=prod->valueCount(Part::SimpleType::INT);
    dCount+=prod->valueCount(Part::SimpleType::DOUBLE);
    sCount+=prod->valueCount(Part::SimpleType::STRING);
    }
ss << '}';
auto s{ss.str()};
auto end{std::chrono::steady_clock::now()};
auto t{std::chrono::duration_cast<std::chrono::nanoseconds>(end-beg).count()};
double d{1.0*t/1000000.0};
LOG("Created [" << mInts << ',' << mDoubles << ','
    << mStrings << "] in " << d << " ms for JSON of size: " << s.size());
return s;
}

private:
```

```
std::shared_ptr<Producer> mProd;
int mInts{};
int mDoubles{};
int mStrings{};
};

//----------------------------------------------------------------------------
void initProducerKeys(ProducerParams& pp)
{
pp.setKeys(g_substantives);
}

//----------------------------------------------------------------------------
void initProducerKeysMultiplier(ProducerParams& pp)
{
pp.setKeyMultiplier(26*26*2);
}

//----------------------------------------------------------------------------
void initProducerValues(ProducerParams& pp)
{
pp.addInts({0,1,2,3,4,5,6,7,8,9,});
pp.addInts({10,11,12,13,14,15,16,17,18,19,});
pp.addInts({110,111,112,113,114,115,116,117,118,119,});
pp.addDoubles({0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,});
pp.addDoubles({10.1,10.2,10.3,10.4,10.5,10.6,10.7,10.8,10.9,11.0,});
pp.addDoubles({110.11,110.21,110.31,110.41,110.15,110.61,110.71,110.18
    ,110.19,111.02,});
pp.addStrings({"A-0001","B-0010","C-0100","D-1000","E-1001","F-1010"
    ,"G-1100","H-1101","I-1111",});
pp.addStrings({"3212-ab","4230-bb","4901-cb","9443-db","8444-eg","3300-ff"
    ,"5932-gb","0943-hb","4064-ig",});
}

//----------------------------------------------------------------------------
using V_Counts=std::vector<std::tuple<int,int,int>>;
std::set<std::string> threadize(ProducerParams& pp, V_Counts const& v)
{
std::set<std::string> results;
auto prod{std::make_shared<Producer>(pp)};
auto futProducer{std::async(std::launch::async,
    [&prod]
    {
    auto res{prod->produce()};
    LOG(res);
    })};
std::deque<std::future<std::string>> futs;
for(auto const& i: v)
    futs.push_back(std::async(std::launch::async,
        [&prod,i]
        {Assembly a{prod,std::get<0>(i),std::get<1>(i),std::get<2>(i)};
        return a.run();
        }));

for(auto i{futs.begin()}; i!=futs.end();)
    {
    if(i->wait_for(1ms)==std::future_status::ready)
        {
```

```
        auto res{i->get()};
        results.emplace(res);
        std::swap(*i,futs.back());
        if(std::next(i)==futs.end())
            i=futs.begin();

        futs.pop_back();
        if(futs.empty())
            break;
        }
    else
        if(++i==futs.end())
            i=futs.begin();
    }
prod->done();
{
std::lock_guard lock{muxCvProd};
cvProd.notify_one();
}
while(futProducer.wait_for(1ms)!=std::future_status::ready);
return results;
}

//------------------------------------------------------------------------------
void usage()
{
LOG(
R"(jsonizer usage:
-h      : This help
-s [N]  : Keys multiplier, adds e.g. _a ... _zzz postfix
          Example: -s 52
-p [key]: Use predefined config
          Currently valid are: default, godbolt, complex
          Example: -p godbolt
-c [xx,min,max,recirc%,weigth]
          where min, max, recirc% and weigth are numbers
          having defaults of 1, 2, 50, 1, respectively,
          and xx is one of KI,KD,KS, AI,AD,AS,AO,AA,AM, OI,OD,OS,OA,OO,OM,
          representing a specific type of factory. Legend:
          K=keyed, I=integer, D=double, S=string, A=array, O=object,
          M=mixed type values.
          This param can be given several times.
-t [int values,double values,string values]
          This represents one JSON file production constraints, i.e.
          a minimum of this many values of specified type will exist in
          the produced JSON object. Example: -t 100,100,100
          This param can be given several times.)");
}

//------------------------------------------------------------------------------
int parseCmdline(
    int argc,
    char* argv[],
    ProducerParams& pp,
    V_Counts& counts,
    std::map<std::string,ProducerParams> const& predefined)
{
auto splitz{[&](
```

```
    auto&& splitz,
    std::vector<std::string>& res,
    std::string const& s,
    std::size_t pos=0) -> void
    {
    if(pos!=std::string::npos)
        {
        auto pos2{s.find(",",pos)};
        res.push_back(s.substr(pos,pos2));
        if(pos2!=std::string::npos)
            splitz(splitz,res,s,++pos2);
        }
    }};
try
    {
    const std::set<std::string> KEYS_1{"-h"};
    const std::set<std::string> KEYS_2{"-s","-p","-c","-t"};
    std::map<std::string,std::vector<std::string>> candidates;
    for(int i=1; i<argc; ++i)
        {
        if(KEYS_2.find(argv[i])!=KEYS_2.end())
            {
            int j=i+1;
            if(!strcmp(argv[i],"-h")
               || (j<argc && KEYS_2.find(argv[j])==KEYS_2.end()))
                candidates[argv[i]].push_back(argv[j]);
            }
        else if(KEYS_1.find(argv[i])!=KEYS_1.end())
            candidates[argv[i]];
        else if(argv[i][0]=='-')
            {
            usage();
            return ERRORS::USAGE;
            }
        }
    auto k{candidates.find("-h")};
    if(k!=candidates.end())
        {
        usage();
        return ERRORS::USAGE;
        }
    k=candidates.find("-s");
    if(k!=candidates.end())
        for(auto i: k->second)
            pp.setKeyMultiplier(std::stoi(i));

    k=candidates.find("-p");
    if(k!=candidates.end())
        for(auto i: k->second)
            {
            auto j{predefined.find(i)};
            if(j!=predefined.end())
                pp=j->second;
            else
                {
                usage();
                return ERRORS::CMDLINE_INVALID_PREDEFINED;
                }
```

```
        }
    k=candidates.find("-c");
    if(k!=candidates.end())
        {
        static const std::map<std::string,CT> KEYS{
             {"KI",CT::KI}
            ,{"KD",CT::KD}
            ,{"KS",CT::KS}
            ,{"AI",CT::AI}
            ,{"AD",CT::AD}
            ,{"AS",CT::AS}
            ,{"AO",CT::AO}
            ,{"AA",CT::AA}
            ,{"AM",CT::AM}
            ,{"OI",CT::OI}
            ,{"OD",CT::OD}
            ,{"OS",CT::OS}
            ,{"OA",CT::OA}
            ,{"OO",CT::OO}
            ,{"OM",CT::OM}};
        std::size_t minSize{DEFAULT_MINSIZE};
        std::size_t maxSize{DEFAULT_MAXSIZE};
        std::size_t recirc{DEFAULT_RECIRC};
        std::size_t weigth{DEFAULT_WEIGTH};
        for(auto ii: k->second)
            {
            std::string key;
            std::vector<std::string> v;
            splitz(splitz,v,ii);
            auto vv{v.begin()};
            if(vv!=v.end())
                {
                key=*vv++;
                if(KEYS.find(key)==KEYS.end())
                    continue;
                }
            if(vv!=v.end())
                minSize=std::stoi(*vv++);

            if(vv!=v.end())
                maxSize=std::stoi(*vv++);

            if(vv!=v.end())
                recirc=std::stoi(*vv++);

            if(vv!=v.end())
                weigth=std::stoi(*vv++);

            pp.setConsumerParam(
                KEYS.find(key)->second,{minSize,maxSize,recirc,weigth});
            }
        }
    k=candidates.find("-t");
    if(k!=candidates.end())
        {
        for(auto ii: k->second)
            {
            int i,d,s;
```

```
            std::vector<std::string> v;
            splitz(splitz,v,ii);
            i=d=s=0;
            auto vv{v.begin()};
            if(vv!=v.end())
                if(!vv++->empty())
                    d=s=i=std::stoi(*vv);

            if(vv!=v.end())
                if(!vv++->empty())
                    d=s=std::stoi(*vv);

            if(vv!=v.end())
                if(!vv->empty())
                s=std::stoi(*vv);

            counts.push_back({i,d,s});
            }
        }
    }
catch(...)
    {
    DisNDat<> c("","  ");
    std::stringstream ss;
    ss << "\nSomething wrong with the command line arguments: ";
    for(int i=0; i<argc; ++i)
        ss << c << argv[i];
    ss << '\n';
    LOG(ss.str());
    usage();
    return ERRORS::CMDLINE_EXCEPTION;
    }
return 0;
}

std::map<std::string,ProducerParams> initPredefined()
{
ProducerParams pp,pp2;
initProducerKeys(pp2);
initProducerKeysMultiplier(pp2);
initProducerValues(pp2);

using f=std::function<void()>;
static std::map<std::string,f> CONSUMERS{
    {"godbolt",[&pp]{
        pp.setConsumerParams({
             {CT::KI,{0,0,90,1}}
            ,{CT::KD,{0,0,90,1}}
            ,{CT::KS,{0,0,90,1}}
            ,{CT::AI,{4,12,80,1}}
            ,{CT::AD,{3,11,80,1}}
            ,{CT::AS,{2,10,80,1}}
            ,{CT::AO,{2,6,40,1}}
            ,{CT::AA,{3,5,40,1}}
            ,{CT::AM,{2,4,40,1}}
            ,{CT::OI,{3,5,40,1}}
            ,{CT::OD,{4,5,40,1}}
            ,{CT::OS,{2,5,40,1}}
```

```
            ,{CT::OA,{4,8,30,1}}
            ,{CT::OO,{3,7,30,1}}
            ,{CT::OM,{2,6,30,1}}
            });}}
    ,{"complex",[&pp]{
        pp.setConsumerParams({
             {CT::KI,{0,0,90,1}}
            ,{CT::KD,{0,0,90,1}}
            ,{CT::KS,{0,0,90,1}}
            ,{CT::AI,{4,12,80,1}}
            ,{CT::AD,{3,11,80,1}}
            ,{CT::AS,{2,10,80,1}}
            ,{CT::AO,{2,6,80,1}}
            ,{CT::AA,{3,5,80,1}}
            ,{CT::AM,{2,4,80,1}}
            ,{CT::OI,{3,5,80,1}}
            ,{CT::OD,{4,5,80,1}}
            ,{CT::OS,{2,5,80,1}}
            ,{CT::OA,{4,8,90,1}}
            ,{CT::OO,{3,7,90,1}}
            ,{CT::OM,{2,6,90,1}}
            });}}
    ,{"default",[&pp]{
        pp.setConsumerParams({
             {CT::KI,{1,2,50,1}}
            ,{CT::KD,{1,2,50,1}}
            ,{CT::KS,{1,2,50,1}}
            ,{CT::AI,{1,2,50,1}}
            ,{CT::AD,{1,2,50,1}}
            ,{CT::AS,{1,2,50,1}}
            ,{CT::AO,{1,2,50,1}}
            ,{CT::AA,{1,2,50,1}}
            ,{CT::AM,{1,2,50,1}}
            ,{CT::OI,{1,2,50,1}}
            ,{CT::OD,{1,2,50,1}}
            ,{CT::OS,{1,2,50,1}}
            ,{CT::OA,{1,2,50,1}}
            ,{CT::OO,{1,2,50,1}}
            ,{CT::OM,{1,2,50,1}}
            });}}};
std::map<std::string,ProducerParams> ppp;
for(auto& [k,v]: CONSUMERS)
    {
    pp=pp2;
    v();
    ppp[k]=pp;
    }
return ppp;
}

int main(int argc, char* argv[])
{
V_Counts counts;
std::map<std::string,ProducerParams> predefined{initPredefined()};
ProducerParams pp{predefined.find("default")->second};
auto r{parseCmdline(argc,argv,pp,counts,predefined)};
if(r)
    exit(r);
```

```
    static const V_Counts defaultCounts{
         {70,70,70}
        ,{60,60,60}
        ,{50,50,50}
        ,{40,40,40}
        ,{30,30,30}
        ,{20,20,20}
        };
    if(counts.empty())
        counts=defaultCounts;

    auto beg{std::chrono::steady_clock::now()};
    auto results{threadize(pp,counts)};
    auto end{std::chrono::steady_clock::now()};
    auto t{std::chrono::duration_cast<std::chrono::nanoseconds>(end-beg).count()};
    double d{1.0*t/1000000.0};
    LOG("RUN took: " << d << " ms");
    LOG("created " << results.size() << " JSON files");
    for(auto const& i: results)
        LOG("Result: " << i << '\n');
    }
```

```
VBox(children=(Button(description='C++ app to godbolt', style=ButtonStyle()), Output()))
```

```
VBox(children=(Button(description='Get godbolt languages', style=ButtonStyle()), Output()))
```

```
VBox(children=(Button(description='Get godbolt C++ compilers', style=ButtonStyle()), Output()))
```