Godafoss reference

Contents

1 background processing

from basics/gf-background.hpp

The background class provides a hook for run-to-completion style background processing.

```
struct background : public not_copyable {
   // This function will be called to do background work for its object.
   virtual void work() = 0;
};
```

A class that needs background processing must inherit from background and implement the work function. This work function will be called when plain wait functions (the ones that allow background processing) are called.

When an application contains background work, all plain wait functions can take longer than the specified time, up to the run time of the longest runtime of the work() functions.

No background work will be done from wait calls made while a work() function is running.

For all background jobs: be careful to preserve the object, or your servicing will end. This is not UB: the background destructor removes itself from the list of background jobs.

When the application would terminate (exit from its main()), background::run() can be called instead, which will serve the background processing (it will never return).

1.1 example

```
#include "godafoss.hpp"
namespace gf  = godafoss;
using target  = gf::target<>;
using timing  = target::timing;

struct background_work: gf::background {

   timing::ticks_type last = timing::now_ms();

   void work() override {
      auto now = timing::now_ms();
      if( now > last + 1'000 ){
         gf::cout << "Another second has passed\n";
      }
   }

};

int main(){

   {
      background_work annnoying;
      for( int i = 0; i < 10; ++i ){
         timing::wait_ms( 2'800 );
         gf::cout << "[" << i << "] 2.8 seconds passed\n"
      }
      // annnoying is destructed here, so it will finally shut up
   }

   for( int i = 0; i < 10; ++i ){
      tinming::wait_ms( 2'100 );
      gf::cout << "[" << i << "] 2.1 seconds passed\n"
   }

};
```

2 box, pipe

from item/gf-item.hpp

A box and a pipe are two kinds of item. The difference is their semantics: a box behaves like a variable that holds a single value, a pipe behaves like a sequence of values.

2.1 box

```
template< typename T, typename VT = T::value_type >
concept is_box = requires {
      T::_box_marker;
   }
   && is_item< T, VT >;
```

```
template< typename VT >
struct box_root :
   item_root< VT >
{
   static const bool _box_marker = true;
};
```

A box is an item that has or contains (at any point in time) a single value. A box has value semantics: when you read from a box twice in rapid succession, you will get the same value. Writing to an item overwrites its old value in the box.

2.2 pipe

```
template< typename T, typename VT = T::value_type >
concept is_pipe = requires {
      T::_pipe_marker;
   }
   && is_item< T, VT >;
```

```
template< typename VT >
struct pipe_root :
   item_root< VT >
{
   static const bool _pipe_marker = true;
};
```

A pipe is an item that holds a sequence of values. A write to a pipe adds a new value the sequence. Hence all write to a stream matter, including repeated write of the same value. Reading from a pipe is destructive: it consumes the value that was read from the sequence. Writing to a pipe adds a value to the sequnce.

3 buffered

from item/gf-item-buffered.hpp

The buffered<> decorator buffers read, write or direction operations, necessitating appropriate refresh or flush calls.

```
template< typename T >
concept can_buffered =
   is_item< T >;
```

```
template< can_buffered T >
struct buffered ... ;
```

4 cto

from item/gf-item.hpp

A cto is a Compile Time Object: it has the role of an object, but it is 'created' at compile time. It is implemented as a struct that has only static functions and static attributes.

A cto always exists: it is just 'waiting' to be used. The features of a cto that are not used will be elimiated by the linker, Hence the mere presence of a cto in the source doen not increase the size of the executable image.

A cto, being a type, is never instantiated. Instead, each cto provides an init() function. Before any of its functions or attributes are used at run-time, a cto must be initialized by calling its init() function. Failing to do so can cause unpredictable results.

As a cto has only static functions and attributes it can be used directly, or the cto can be passed as a template parameter.

For cto, and for each more specific cto, a concept exists (with the name of the cto), and a root struct from which all such cto's are derived (with _root appended to the name of the cto).

The concept checks both for a specific marker, which serves no other purpose than to identify the specific cto, and for the features that the cto is obliged to offer. The concept is used to constrain templates that want to accept only a cto that implements a specific set of features.

The root can be a plain struct, but it is often a template. For more complex cto's the CRTP pattern is used so the root can provide both base properties and enrichment based on the provided implementation.

```
template< typename T >
concept is_cto = requires {
   T::_cto_marker;
   { T::init() } -> std::same_as< void >;
};
```

A cto has a static init() function that can be called without arguments.

```
struct cto_root {
   static const bool _cto_marker = true;
};
```

The struct cto_root is the root type of all cto's: all cto's inherit (in most cases indirectly) from this struct.

5 date and time

from adts/gf-date-and-time.hpp

```
struct date_and_time { ... };
```

This is a datatype for representing a date-and-time, intended for use with timekeeping chips or peripherals.

5.1 attributes

```
uint8_t seconds, minutes, hours;
uint8_t day, month, year;
uint8_t weekday;
```

The atributes of data_and_time are:

- seconds (0-59), minutes (0-59), hours (0-23)
- day (1..28/29/30/31), month (1-12), year (0-99)
- weekday (1..7)

Fields that are unknow (some chips don't have a weekday) are set to 0.

5.2 non-member functions

```
stream & operator<<( stream & lhs, by_const< date_and_time > dt ){ ... }
```

The operator<< prints a data_and_time in the format YY-MM-DD HH:MM.SS dW.

6 direct

from item/gf-item-direct.hpp

The direct<> decorator accepts an item and decorates it by inserting the appropriate refresh or flush before or after each read, write, or direction change operation, and replacing the refresh and flush by empty functions.

The effect is that such a decorated item can be used without refresh or flush calls.

```
template< typename T >
concept can_direct =
   is_item< T >;
```

```
template< typename T >
   requires can_direct< T >
struct direct< T > : ... ;
```

6.1 example

```
#include "godafoss.hpp"
namespace gf      = godafoss;
using target      = gf::target<>;
using pin         = gf::pin_out_from< target::d13 >;
using direct_pin  = gf::direct< pin >;

int main(){

   pin::init();

   // write, followed by an explicit flush
   pin::write( 1 );
   pin::flush();

   // write to a direct<> pin implies an implicit flush(),
   // so no explicit fluish() is needed
   direct_pin::write( 1 );

};
```

7 function and class attributes

from basics/gf-attributes.hpp

```
#define GODAFOSS_INLINE ...
```

GODAFOSS_INLINE forces a function to be inline. It is used when the function body is very simple, for instance when it calls only one deeper function. This serves (only) to reduce code size and execution time.

```
#define GODAFOSS_NO_INLINE ...
```

GODAFOSS_NO_INLINE forces a function to be not inline. This is used to preserve low-level properties of a function, like the number of cylces taken by the function preable and postamble. This can be important to get predictable timing.

```
#define GODAFOSS_NO_RETURN ...
```

GODAFOSS_NORETURN indicates that a function will not return. It is used for functions that contain a never-ending loop. This can reduce code size.

```
#define GODAFOSS_IN_RAM ...
```

GODAFOSS_IN_RAM places the function body in RAM (instead of FLASH). On some targets, this is necesarry to get predicatable timing, or faster execution.

```
#define GODAFOSS_RUN_ONCE ...
```

GODAFOSS_RUN_ONCE causes the remainder of the function (the part after the macro) to be executed only once.

```
struct not_constructible { ... };
```

Inheriting from not_constructible makes it impossible to create objects of that class.

```
struct not_copyable { ... };
```

Inheriting from not_copyable makes it impossible to copy an object of that class.

8 hd44780

from chips/gf-hd44780.hpp

The hd44780 template implements a charcater terminal on an hd44780 character lcd.

```
template<
   can_pin_out   rs,
   can_pin_out   e,
   can_port_out  port,
   xy<>          size,
   typename      timing
> using hd44780_rs_e_d_s_timing = { ... };
```

The rs, e and port must connect to the corresponding pins of the lcd. The lcd is used in 4-bit mode, so the port must connect to the d0..d3 of the lcd, the d4..d7 can be left unconnected. Only write to the lcd are used. The _r/w pin must be connected to ground.

The size of the lcd must be specified in characters in the x and y direction. Common sizes are 16x1, 16x2, 20x2 and 20x4.

The timing is used for the waits as required by the hd44780 datasheet.

8.1 example

bla blas

9 hx711

from chips/gf-hx711.hpp

This template implements an interface to the hx711 24-Bit Analog-to-Digital Converter (ADC). This chip is intended to interface to a load cell (force sensor).

```
template<
   can_pin_out  _sck,
   can_pin_in   _dout,
   typename            timing
>
struct hx711 {
   enum class mode {
      a_128  = 1,  // A inputs, gain 128
      b_32   = 2,  // B inputs, gain 32
      a_64   = 3   // A inputs, gain 64
   };
   static void init( mode m = mode::a_128 ){ ... }
   static int32_t read(){ ... }
   static void power_down(){ ... }
   static void mode_set( mode m ){ ... }
};
```

The chip interface consist of a master-to-slave clock pin (sck), and a slave-to-master data pin (dout).

The timing is used for the waits as required by the hx711 datasheet.

The mode offers a choice between the A differential input with a gain of 128 or 64, and the B input with a gain of 32. The A input are meant to be used with a load cell. The datasheet suggest that the B input could be used to monitior the battery voltage. The mode is set at the initialization (the defauylt is a_128), and can be changed by the mode_set() function.

The chip can be powered down. When a read is done the chip is first (automatically) powered up.

10 inherit_*

from item/gf-item-inherit.hpp

Adapters for selectively inheriting only the init, read, write, or direction functions of a item. This is used or instance in the item_input adapter, to 'pass' only the input functionality.

10.1 inherit_init

```
template< typename T >
struct inherit_init = ... ;
```

The inherit_init decorator inherits only the init() function of the decorated item.

10.2 inherit_read

```
template< typename T >
struct inherit_read = ... ;
```

The inherit_read decorator inherits only the refresh and read functions of the decorated item.

10.3 inherit_write

```
template< typename T >
struct inherit_write = ... ;
```

The inherit_read decorator inherits only the write and flush functions of the decorated item.

10.4 inherit_direction

```
template< typename T >
struct inherit_direction = ... ;
```

The inherit_read decorator inherits only the direction_set_input, direction_set_output and direction_flush functions of the decorated item.

11 input, output

from item/gf-item.hpp

An item can be an input (from which you can read) and/or an output (to which you can write).

An input or output item can be buffered. For an output, this means that the effect of write operations can be postponed until the next flush call. For an input, this means that a read operation reflects the situation immediately before that last refresh call, or later. For immediate effect on a buffered item, a read must be preceded by a refresh, and a write must be followed by a flush.

The direct decorator creates an item for which the read() and write() operations have direct effect.

An item can be an input, an output, or both. When it is an input you can read from it, when it is an output you can write to it. (In theory an item could be neither, but that would not be very useful.)

When an item is both input and output it can be simplex (sometimes call half-duplex) or duplex. A duplex box can, at any time, be both read and written.

11.1 input

```
template< typename T, typename VT = T::value_type >
concept is_input = requires {
      T::_input_marker;
      { T::refresh() }  -> std::same_as< void >;
      { T::read() }     -> std::same_as< typename T::value_type >;
   }
   && is_item< T, VT >;
```

```
template< typename VT >
struct input_root :
   item_root< VT >
{
   static const bool _input_marker = true;
};
```

A input is an item that provides a read() function that returns a value of the value_type of the item.

11.2 output

```
template< typename T, typename VT = T::value_type >
concept is_output = requires (
   typename T::value_type v
   ){
      T::_output_marker;
      { T::write( v ) }  -> std::same_as< void >;
      { T::flush()    }  -> std::same_as< void >;
```

```
template< typename VT >
struct output_root :
   item_root< VT >
{
   static const bool _output_marker = true;
};
```

An output is an item that provides a write() function that accepts a value of the value_type of the item.

11.3 input_output

```
template< typename T, typename VT = T::value_type >
concept is_input_output = requires (
      typename T::value_type v
   ){
      T::_input_output_marker;
      { T::refresh() }   -> std::same_as< void >;
      { T::read() }      -> std::same_as< typename T::value_type >;
      { T::write( v ) }  -> std::same_as< void >;
      { T::flush()    }  -> std::same_as< void >;
   }
   && is_item< T, VT >;
```

```
template< typename VT >
struct input_output_root :
   input_root< VT >,
   output_root< VT >
{
   static const bool _input_output_marker = true;
};
```

An input_output is an item that is both an input and an output.

11.4 direction

A duplex item is an input_output that can function both as an input and as an output at the same time.

```
template< typename T, typename VT = T::value_type >
concept is_duplex = requires {
      T::_duplex_marker;
   }
   && is_input_output< T, VT >;
```

```
template< typename VT >
struct duplex_root :
   input_output_root< VT >
```

```
{
   static const bool _duplex_marker = true;
};
```

A simplex item is an input_output that has a current direction, which can be input or output.

```
template< typename T, typename VT = T::value_type >
concept is_simplex = requires {
      T::_simplex_marker;
      { T::direction_set_input() }   -> std::same_as< void >;
      { T::direction_set_output() }  -> std::same_as< void >;
      { T::direction_flush() }       -> std::same_as< void >;
   }
   && is_input_output< T, VT >;
```

```
template< typename VT >
struct simplex_root :
   input_output_root< VT >
{
   static const bool _simplex_marker = true;
};
```

The direction of a simplex item can be changed with a direction_set_input or direction_set_output call. For a successful read, the direction of a simplex box must be input. For a successful write, the direction of a simplex box must be output. Otherwise a write can have no effect at all, or have a delayed effect, and a read returns an unspecified value, and for a stream it can either consume the value or not.

The effect of calling a direction_set... function can be delayed up to the next direction_flush() call. Like for read() and write(), direct can be used to get an immediate effect.

12 ints specified by number of bits

from basics/gf-ints.hpp

```
template< uint64_t n > struct uint_bits {
   typedef typename ...
      fast;
   typedef typename ...
      least;
};
```

uint_bits< N >::fast is the smallest 'fast' unsigned integer type that stores (at least) N bits.

uint_bits< N >::least is the smallest (but not necesarrily fast) unsigned integer type that stores (at least) N bits.

As both are unsigned they should be used for bit patterns, not for amounts.

Note that both can be larger than requested, so they should not be used for modulo arithmetic (at least not without masking out excess bits).

Use uint_bits< N >::fast for variables and parameters, use uint_bits< N >::least for arrays.

12.1 example

bla bla

13 invert

from item/gf-item-invert.hpp

The invert<> decorator inverts the value written to or read from an item.

```
// invert is supported for an item that has an invert function
template< typename T >
concept can_invert = requires (
      typename T::value_type v
   ){
      { T::invert( v ) } -> std::same_as< typename T::value_type >;
   }
```

```
template< can_invert T >
struct invert< T > ... ;
```

14 item

from item/gf-item.hpp

An item is the basic cto from which most other cto's are derived.

A summary of terms:

- cto: a compile-time (static) object
- item: holds some data elements(s))
- box: item that always holds one element of the data
- pipe: item that holds a sequence of data elements
- input: item that supports read
- output: item that supports write
- input_output: both input and output
- duplex: both input and output at the same time
- simplex: both input and output, but not at the same time

```
template< typename T, typename VT = T::value_type >
concept is_item = requires {
      T::_item_marker;
      { typename T::value_type() } -> std::same_as< VT >;
   }
   && is_cto< T >;
```

An item is a cto that holds one (or, in case of a pipe, more) data elements of a specific type.

```
template< typename VT >
struct item_root : cto_root {
   static const bool _item_marker = true;
   using value_type = VT;
};
```

All items inherit (in most cases indirectly) from the struct item_root.

15 item adapters

from item/gf-item-adapters.hpp

These adapters adapt an item to be (only) an input item, (only) an output item, or (only) an input_output item (in each case, if such adaption is possible).

These adapters serve, of course, to adapt a given item to the adapted role, but also to ensure that the code that uses the adapted item, doesn't use any features beyond the ones of the adapted role.

15.1 item_input

```
template< typename T >
concept can_input =
   is_input< T >
   || is_input_output< T >;
```

```
template< can_input T >
struct item_input ... ;
```

The item_input<> decorator decorates an item to be an input item, which is possible if the item satisfies the can_input concept, which requires the item to be either an input or an input_output.

15.2 item_output

```
template< typename T >
concept can_output =
   is_output< T >
   || is_input_output< T >;
```

```
template< can_output T >
struct item_output ... ;
```

The item_output<> decorator decorates an item to be an output item, which is possible if the item satisfies the can_output concept, which requires the item to be either an input or an input_output.

15.3 item_input_output

```
template< typename T >
concept can_input_output =
   is_input_output< T >;
```

```
template< can_input_output T >
struct item_input_output ... ;
```

The item_input_output<> decorator decorates an item to be an input_output item, which is possible if the item satisfies the can_input_output concept, which requires the item to an input_output.

16 no_inline

from item/gf-item-no-inline.hpp

The no_inline<> item decorator creates an item for which all functions are not inline.

This can be used as the outermost decorator around an item constructed from a chain of inheritances, in which the chain of function calls is all marked GODAFOSS_INLINE.

```
template< is_item T >
using no_inline = ... ;
```

17 passing a readonly parameter

from basics/gf-passing.hpp

```
// use by_const< T > when passing a T
template< typename T >
using by_const = ...
```

The by_const< type > template is the preferred way to pass a const value of the type passed_type. This will be either a plain (by copy) const, or a const reference, depending (among other things) on the size of the type compared to a the size of a reference.

```
#include "array"
#include "godafoss.hpp"
namespace gf  = godafoss;

void GODAFOSS_NO_INLINE f1(
   gf::by_const< char > p
){ (void) p; }

void GODAFOSS_NO_INLINE f2(
   gf::by_const< std::array< int, 100 > > p
){ (void) p; }

int main(){

   // should probably be passed by value (copy)
   f1( 'x' );

   // should be probably be passed by reference
   f2( std::array< int, 100 >{ 0 } );

};
```

18 pin adapters

from modifiers/gf-modifiers-pins.hpp

These adapters adapt a pin to be (only) an input pin, (only) an output pin, (only) an input_output pin, or (obly) an open collector pin. (in each case, if such adaptation is possible).

These adapters serve, of course, to adapt a given pin to the adapted role, but also to ensure that the code that uses the adapted pin doesn't use any features beyond the ones of the adapted role.

18.1 pin_in

```
template< typename T >
concept can_pin_in =
      is_pin_in< T >
   || is_pin_in_out< T >
   || is_pin_oc< T >;
```

```
template< can_pin_in T > = ...;
```

The pin_in_ decorator decorates a pin to be an input pin, which is possible if the pin satisfies the can_input concept, which requires the pin to be either a pin_in_ or a pin_in_out_.

18.2 pin_out

```
template< typename T >
concept can_pin_out =
      is_pin_out< T >
   || is_pin_in_out< T >
   || is_pin_oc< T >;
```

```
template< can_pin_out T > = ...;
```

The pin_out_ decorator decorates a pin to be an output pin, which is possible if the pin satisfies the can_output concept, which requires the pin to be either a pin_in_, a pin_in_out_, or a pin_oc_.

Note that when a pin_oc_ is adapted to be used as pin_out_, a pull-up resistor is required in order for the pin to reach a high level.

18.3 pin_in_out

```
template< typename T >
concept can_pin_in_out =
      is_pin_in_out< T >
   || is_pin_oc< T >;
```

```
template< can_pin_in_out T  > = ...;
```

The pin_in_out_ decorator decorates a pin to be an input_output pin, which is possible if the pin satisfies the can_input_output concept, which requires the pin to a pin_in_out_, or a pin_oc_.

Note that when a pin_oc_ is adapted to be used as pin_in_out_, a pull-up resistor is required in order for the pin to reach a high level.

18.4 pin_oc

```
template< typename T >
concept can_pin_oc =
      is_pin_in_out< T >
   || is_pin_oc< T >;
```

```
template< can_pin_oc T > = ...;
```

The pin_oc_ decorator decorates a pin to be an open collector pin, which is possible if the pin satisfies the can_input_output concept, which requires the pin to a pin_in_out_ or a pin_oc_.

19 pins

from pins/gf-pin.hpp

A pin is a box_< bool > cto that is used to asbstract a GPIO (general-purpose input-output interface) pin on a micro-controller (or peripheral chip), or the more limited input-only, output-only, or open-collector version.

When a pin represents a physical pin, 0 (or false) means a low voltage level (almost ground), and 1 (or true) means a high voltage level.

When a pin represents a functionality, for instance 'enable', true (or 1) means that the function is enabled, and false (or 0) means that the function is not enabled (disabled).

When the phyiscal pin is active-low, an invert decrorator is used to create the internal active-high representation of the pin.

19.1 pin_in

```
template< typename T >
concept is_pin_in = requires {
      T::_pin_in_marker;
   }
   && is_box< T, bool >
   && is_input< T >;
```

A pin_in_ is a box_< bool > cto that abstracts a single-pin read-only interface to the world outside the target chip. A typical use of a pin_in_ is to read a switch or pushbutton.

```
struct pin_in_root :
   box_root< bool >,
   input_root< bool >
{
   static constexpr bool _pin_in_marker = true;
};
```

All pin_in_ cto's inherit from pin_in_root.

19.2 pin_out

```
template< typename T >
concept is_pin_out = requires {
      T::_pin_out_marker;
   }
   && is_box< T, bool >
   && is_output< T >;
```

A pin_in_ is a box_< bool > cto that abstracts a single-pin write-only interface to the world outside the target chip. A typical use of a pin_in_ is to drive an LED.

```
struct pin_out_root :
   box_root< bool >,
   output_root< bool >
{
   static constexpr bool _pin_out_marker = true;
};
```

All pin_out_ cto's inherit from pin_out_root.

19.3 pin_in_out

```
template< typename T >
concept is_pin_in_out =
 requires {
      T::_pin_in_out_marker;
   }
   && is_box< T, bool >
   && is_simplex< T >;
```

A pin_in_out_ is a box_< bool > cto that abstracts a single-pin simplex read-write interface to the world outside the target chip. A pin_in_out_ is the most versatile of the pin types, because it can be used in any of the roles. In most cases a pin_in_out_ is used as either pin, a a pin_out_, or a pin_oc_, but some communication protocols require a pin to be switched between input and output.

```
struct pin_in_out_root :
   box_root< bool >,
   simplex_root< bool >
{
   static constexpr bool _pin_in_out_marker = true;
};
```

All pin_in_ cto's inherit from pin_in_out_root.

19.4 pin_oc

```
template< typename T >
concept is_pin_oc = requires {
      T::_pin_oc_marker;
   }
   && is_box< T, bool >
   && is_duplex< T >;
```

A pin_oc_ is a box_< bool > cto that abstracts a single-pin duplex read-write interface to the world outside the target chip. The term oc means open-collector, refrerring to the (now somewhat outdated) way this type of pin can be implemented: the output stage has a transistor that can pull the pin low, but unlike a normal output pin it has no transistor to pull the pin high.

Open-collector pins are used in various protocols like i2c and one-wire, where open-collector pins of more than one chip are connected to the same wire. Any chip can pull the write low. When no chip does so, a common pull-up resistor pulls the line low. This arrangement prevents electrical problems which would be caused when one chip drives the line low, and another drives it high.

```
struct pin_oc_root :
   box_root< bool >,
   duplex_root< bool >
{
   static constexpr bool _pin_oc_marker = true;
};
```

All pin_in_ cto's inherit from pin_oc_root.

20 port adapters

from modifiers/gf-modifiers-ports.hpp

These adapters adapt a port to be (only) an input port, (only) an output port, (only) an input_output port, or (only) an open collector port. (in each case, if such adaptation is possible).

The created pin has only the properties required for that pin: other properties of the source pin are not available via the created pin. The exception is pullup and pulldown features: those are available via the created pins.

These adapters serve, of course, to adapt a given port to the adapted role, but also to ensure that the code that uses the adapted port doesn't use any features beyond the ones of the adapted role.

20.1 port_in

```
template< typename T >
concept can_port_in =
      is_port_in< T >
   || is_port_in_out< T >
   || is_port_oc< T >;
```

```
template< can_port_in T > = ...;
```

The port_in<> adapter creates an input port from a source port, which is possible if the source port satisfies the can_port_in concept, which requires it to be either a port_in, a port_in_out, or a port_oc.

20.2 port_out

```
template< typename T >
concept can_port_out =
      is_port_out< T >
   || is_port_in_out< T >
   || is_port_oc< T >;
```

```
template< can_port_out T > = ...;
```

The port_out<> adapter creates an output port from a source port, which is possible if the source port satisfies the can_port_out concept, which requires it to be either a port_in, a port_in_out, or a port_oc.

Note that when a port_oc is adapted to be used as port_out, pull-up resistors are required in order for the pins to reach a high level.

20.3 port_in_out

```
template< typename T >
concept can_port_in_out =
      is_port_in_out< T >
   || is_port_oc< T >;
```

```
template< can_port_in_out T  > = ...;
```

The port_in_out<> adapter creates an input_output port from a source port, which is possible if the source port satisfies the can_port_in_out concept, which requires it to a port_in_out, or a port_oc.

Note that when a port_oc is adapted to be used as port_in_out, pull-up resistors are required in order for the pins to reach a high level.

20.4 port_oc

```
template< typename T >
concept can_port_oc =
   is_port_oc< T >;
```

```
template< is_port_oc T  > = ...;
```

The port_oc<> adapter creates an open collector port from a source port, which is possible if the source port satisfies the can_port_oc concept, which requires it to a port_oc.

It is not possible to create a port_oc from an input-output port, because that would require control over the direction of the individual pins. An input-output provides (only) control over the direection of all pins at once.

21 random

from basics/gf-random.hpp

This is simple 32-bit LCG random function, for demos and games. The random facilities of the standard library are not used because they eat up too much RAM. Do NOT use this for crypto work.

The LCG used is the Microsoft Visual/Quick C/C++ variant as explained on https://en.wikipedia.org/wiki/Linear_congruential_generator, but using bits 8..23 rather than 16..30.

```
uint16_t random16(){ ... }
```

The random16() function returns a 16-bit non-negative pseudo-random number.

```
uint32_t random32(){ ... }
```

The random32() function returns a 32-bit non-negative pseudo-random number.

```
template< typename int_type >
int_type random_in_range(
   by_const< int_type >first,
   by_const< int_type > last
){ ... }
```

The random_in_range() function returns a non-negative pseudo-random number in the range [first .. last]. This number is calculated from a number generated by random32 by modulo arithmetic. This is simple and fast, but the distribution is not ideal: the higher values in the range will be somewhat underrepresented. When the width of the range is much smaller than 2^32 this effect will be small.

```
void random_seed( by_const< uint32_t > x ){ ... }
```

The random_seed() function sets the start for the value returned by subsequent random calls. It can be used to re-start a random sequence, or (when you have a truely random source) to start a truely random random sequence.

22 specific pin adapters

from pins/gf-pin-adapters.hpp

These adapters create a pin cto from a specific (same or other) pin cto.

The created pin has only the properties required for that pin: other properties of the source pin are not available via the created pin. The exception is pullup and pulldown features: those are available via the created pins.

These adapters can only be used when the source pin is know. For general use, the pin adapters that accept any (possible) source pin are more covenient.

```
template< is_pin_in T >
struct pin_in_from_pin_in : ... {};
```

```
template< is_pin_in_out T >
struct pin_in_from_pin_in_out : ... {};
```

```
template< is_pin_oc T >
struct pin_in_from_pin_oc : ... {};
```

```
template< is_pin_out T >
struct pin_out_from_pin_out : ... {};
```

```
template< is_pin_in_out T >
struct pin_out_from_pin_in_out : ... {};
```

```
template< is_pin_oc T >
struct pin_out_from_pin_oc : ... {};
```

```
template< is_pin_in_out T >
struct pin_in_out_from_pin_in_out : ... {};
```

```
template< is_pin_oc T >
struct pin_in_out_from_pin_oc : ... {};
```

```
template< is_pin_oc T >
struct pin_oc_from_pin_oc : ... {};
```

23 specific port adapters

from ports/gf-port-adapters.hpp

These adapters create a port cto from a specific (same or other) port cto.

The created port has only the properties required for that port: other properties of the source port are not available via the created port.

These adapters can only be used when the source port is know. For general use, the port adapters that accept any (possible) source port are more covenient.

```
template< is_port_in T >
struct port_in_from_port_in : ... {};
```

```
template< is_port_in_out T >
struct port_in_from_port_in_out : ... {};
```

```
template< is_port_oc T >
struct port_in_from_port_oc : ... {};
```

```
template< is_port_out T >
struct port_out_from_port_out : ... {};
```

```
template< is_port_in_out T >
struct port_out_from_port_in_out : ... {};
```

```
template< is_port_oc T >
struct port_out_from_port_oc : ... {};
```

```
template< is_port_in_out T >
struct port_in_out_from_port_in_out : ... {};
```

```
template< is_port_oc T >
struct port_in_out_from_port_oc : ... {};
```

```
template< is_port_oc T >
struct port_oc_from_port_oc : ... {};
```

24 string

from adts/gf-string.hpp

```
template< std::size_t _maximum_length >
struct string { ... }
```

This is a fixed-maximum-size string. It offers an alternative to std::string and raw 0-terminated char arrays. It doesn't use the heap, and doesn't cause Undefined Behaviour with buffer overflows or out-of-bounds indexes.

The functions that extend the string by appending characters do so up to the maximum length of the string. Appending characters beyond this maximum length has no effect: the excess characters are ignored.

The functions that access a character at an index (a position within the stored string) do so only when the index is valid. When the index is invalid, an undefined character (or a reference to an undefined character) is returned.

24.1 attributes

```
using size_t = std::size_t;
static constexpr size_t maximum_length = _maximum_length;
```

The maxmimum_length is the maxiumum number of character that can be stored by the string.

```
constexpr size_t length() const { ... }
```

The member function length() returns number of characters that are currently stored.

```
constexpr bool valid_index( const size_t n ) const { ... }
```

The member function valid_index(n) returns whether n is a valid index into the curretly stored string of characters.

```
string & append( char c ){ ... }
string & operator+=( char c ){ ... }
string & operator<<( char c ){ ... }
```

The append function, the operator+= and the operator<< all append a single character to the string. If the string is already at its maximum length the character is ignored.

25 torsor<>

from adts/gf-torsor.hpp

```
template<
   typename    _data_type,
   _data_type  _zero
>
class torsor final {
public:

   // the type this torsor stores
   using data_type = _data_type;

   // the base (zero) value of this torsor
   static constexpr data_type zero = _zero;
 ... \\n };
```

The torsor<> template expresses and enforces the difference between relative and absolute (anchored) values. Much like a compile-time unit system like boost::units, torsor<> uses the type system to eliminate erroneous operations at compile-time. It also helps to make interfaces simpler and more elegant by making the difference between relative and absolute values explicit in the type system.

For a value type that denotes a ratio scale value (a value for which addition yields a value on the same scale), the torsor of that (base) type is the corresponding interval scale (anchored) type.

The canonical example of a base and its torsor is *distance* (in the vector sense), and its torsor, *location.*

Distances can be added or subtracted, which yields a distance. Locations can't be meaningfully added, but adding a location and a distance is meaningfull and yields a distance. Two locations can be subtracted, yielding a distance.

Whether a scale is a torsor or not has nothing to do with its unit: in a unit system (like SI) a basic (ration) type and its torsor have the same unit.

The operations on the torsor are limited to: - default- or copy-consructing a torsor - assigning a torsor (ssigns the base value) - adding or subtracting a base type value (yields a torsor value) - subtracting two torsors (yields a base type value) - comparing torsors (compares their base values) - printing a torsor (prints @ followed by its base type value)

The base type T of a torsor must have a constructor that accepts a (single) 0 argument.

All operations have the __attribute__((always_inline)), hence there is no need to bother with choosing for copy or reference parameter passing: all passing disappears.

26 xy<>

from adts/gf-xy.hpp

```
template<
   typename xy_value_type = int64_t,
   xy_value_type zero = 0 >
struct xy final { ... };
```

The xy< xy_value_type > ADT class template is a pair of t wo xy_value_type values named x and y. It is used for distances in an xy plane, like on a window or terminal. For a location in an xy plane the torsor_< xy< T > > is used.

The xy<> ADT supports - constructors: default (initializes to zero), from x and y values, copy (from another xy<>) - direct acces to the x and y values - an origin (zero) constant - operators on two xy<>'s: - + == != - operators on an xy<> and a scalar: * /

26.1 attributes

```
using value_t = xy_value_type;
value_t x, y;
```

The x and y values are freely acessible.

```
static constexpr auto origin = xy{};
```

The origin is the (0,0) value.

26.2 methods

```
constexpr xy():x{ zero }, y{ zero }{}
```

The default constructor intializes a and y to the zero value.

```
constexpr xy( value_t x, value_t y ): x{ x }, y{ y }{}
```

The two-value constructor initializes the x and y from the supplied values.

```
template< typename X >
constexpr xy( const xy< X > & rhs ): x( rhs.x ), y( rhs.y ) {}
```

An xy<> object can be constructed from an xy with the same or a different value type.

```
   template< typename V >
//      requires requires( V b ){ { x + b }; }   - GCC 10.0.1 ICE segfault
      requires requires( xy_value_type x, V b ){ { x + b }; } ... }
```

```
template< typename V >
   requires requires( xy_value_type x, V b ){ { x - b }; }
constexpr auto operator-( const xy< V > rhs ) const { ... }
```

Two xy<> values can be added to or subtracted provided that their xy_value_types can be added or subtracted. The resulting xy<> gets the xy_value_type of that addition or subtraction.

```
constexpr xy operator*( const value_t rhs ) const { ... }
constexpr xy operator/( const value_t rhs ) const { ... }
```

An xy<> can be multiplied or divided by a value, provided an xy_value_can be constructed from it. The result is an xy<> value of the same xy<>_value_type.

```
template< typename V >
   requires requires( xy_value_type a, V b ){
      { a == b } -> std::same_as< bool >; }
constexpr bool operator==( const xy< V > & rhs ) const { ... }
```

```
template< typename V >
   requires requires( xy_value_type lhs, V b ){
      { x == b } -> std::same_as< bool >; }
constexpr bool operator!=( const xy & rhs ) const { ... }
```

An xy<> can be compared to another xy<> for equality or inequality, provided that their xy_value_types can be compared.

26.3 non-member functions

```
template< typename stream, typename value >
   requires requires( stream & s, char c, value v ){
      { s << 'c' } -> std::same_as< stream & >;
      { s << v   } -> std::same_as< stream & >;
   }
```