

Godafoss reference



Godafoss reference

# Contents

<b>1 background processing</b>	<b>4</b>
1.1 example	5
<b>2 box, pipe</b>	<b>6</b>
2.1 box	6
2.2 pipe	6
<b>3 buffered</b>	<b>8</b>
<b>4 cto</b>	<b>9</b>
<b>5 date and time</b>	<b>10</b>
5.1 attributes	10
5.2 non-member functions	10
<b>6 direct</b>	<b>11</b>
<b>7 function and class attributes</b>	<b>12</b>
<b>8 hd44780</b>	<b>14</b>
8.1 example	14
<b>9 hx711</b>	<b>15</b>
<b>10 inherit_*</b>	<b>16</b>
10.1 inherit_init	16
10.2 inherit_read	16
10.3 inherit_write	16
10.4 inherit_direction	16
<b>11 input, output</b>	<b>17</b>
11.1 input	17
11.2 output	17
11.3 input_output	18
11.4 direction	18
<b>12 ints specified by number of bits</b>	<b>20</b>
12.1 example	20
<b>13 invert</b>	<b>21</b>
<b>14 item</b>	<b>22</b>
<b>15 item adapters</b>	<b>23</b>
15.1 item_input	23
<b>16 no_inline</b>	<b>24</b>
<b>17 passing a readonly parameter</b>	<b>25</b>
<b>18 pins</b>	<b>26</b>
18.1 pin_in	26

18.2	pin_out	26
18.3	pin_in_out	27
18.4	pin_oc	27
<b>19</b>	<b>random</b>	<b>29</b>
<b>20</b>	<b>string</b>	<b>30</b>
20.1	attributes	30
<b>21</b>	<b>xy&lt;&gt;</b>	<b>31</b>
21.1	attributes	31
21.2	methods	31
21.3	non-member functions	32

# 1 background processing

implemented in: library/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 annoying;
        for( int i = 0; i < 10; ++i ){
            timing::wait_ms( 2'800 );
            gf::cout << "[" << i << "]" 2.8 seconds passed\n"
        }
        // annoying 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

implemented in: library/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

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.

```
template< typename T >
concept box = requires {
    item< T >;
    T::_box_marker;
};
```

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

---

### 2.2 pipe

A pipe is an [item](#) that holds a sequence of values. A write to a pipe adds a new value the sequence. Hence all writes to a stream matter, including repeated writes 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 sequence.

```
template< typename T >
concept pipe = requires {
    item< T >;
    T::_pipe_marker;
};
```

```
template< typename T >
struct pipe_root :
    item_root< T >
{
```

## Godafoss reference

```
static const bool _pipe_marker = true;  
};
```

## 3 buffered

implemented in: library/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 = requires {
    item< T >;
};
```

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



## 4 cto

implemented in: `library/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 eliminated by the linker, Hence the mere presence of a cto in the source does 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 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

implemented in: library/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 attributes 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 unknown (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

implemented in: library/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 operations 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 = requires {
    item< T >;
};
```

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

## 7 function and class attributes

implemented in: library/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 cycles taken by the function preamble 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 necessary to get predictable 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 { ... };
```

## Godafoss reference

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

## 8 hd44780

implemented in: library/chips/gf-hd44780.hpp

---

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

```
template<
    pin_out_compatible    rs,
    pin_out_compatible    e,
    port_out_compatible    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 writes 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

implemented in: library/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<
    pin_out_compatible    _sck,
    pin_in_compatible     _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 monitor the battery voltage. The mode is set at the initialization (the default 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\_\*

implemented in: library/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

implemented in: library/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 >
concept input = requires {
    item< T >;
    T::_input_marker;
    { T::refresh() } -> std::same_as< void >;
    { T::read() } -> std::same_as< typename T::value_type >;
};
```

```
template< typename T >
struct input_root :
    item_root< T >
{
    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 >
concept output = requires (
    typename T::value_type v
){
    item< T >;
    T::_output_marker;
    { T::write( v ) } -> std::same_as< void >;
};
```

```
template< typename T >
struct output_root :
    item_root< T >
{
    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 >
concept input_output = requires {
    input< T >;
    output< T >;
};
```

```
template< typename T >
struct input_output_root :
    input_root< T >,
    output_root< T >
{};
```

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 >
concept duplex = requires {
    input_output< T >;
    T::_duplex_marker;
};
```

```
template< typename T >
struct duplex_root :
    input_output_root< T >
{
    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 >
concept simplex = requires {
```

```

input_output< T >;
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 >;
};

```

```

template< typename T >
struct simplex_root :
    input_output_root< T >
{
    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

implemented in: library/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 necessarily 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

implemented in: library/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
) {
    item< T >;
    { T::invert( v ) } -> std::same_as< typename T::value_type >;
};
```

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

## 14 item

implemented in: library/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 >
concept item = requires {
    cto< T >;
    T::_item_marker;
};
```

An item is a [cto](#) that holds one or more data elements of a specific type.

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

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

## 15 item adapters

implemented in: library/item/gf-item-adapters.hpp

---

These adapter adapts a [item](#) to be (only) an [input item](#), (only) an [output item](#), or an [input\\_output item](#) (if such adaption is possible).

---

### 15.1 item\_input

```
template< typename T >
concept can_input =
    input< T >
    || 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.

```
template< typename T >
concept can_output =
    output< T >
    || 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.

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

```
template< 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.

## 16 no\_inline

implemented in: library/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< item T >
using no_inline = ... ;
```



## 17 passing a readonly parameter

implemented in: library/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 pins

implemented in: library/pins/gf-pin.hpp

---

A pin is a `box< bool > cto` that is used to abstract 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 physical pin is active-low, an `invert` decorator is used to create the internal active-high representation of the pin.

---

### 18.1 pin\_in

```
template< typename T >
concept pin_in = requires {
    box< bool >;
    input< bool >;
    T::_pin_in_marker;
};
```

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`.

---

### 18.2 pin\_out

```
template< typename T >
concept pin_out = requires {
    box< bool >;
    output< bool >;
    T::_pin_out_marker;
};
```

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`.

---

## 18.3 pin\_in\_out

```
template< typename T >
concept pin_in_out = requires {
    box< bool >;
    simplex< bool >;
    T::_pin_in_out_marker;
};
```

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 `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`.

---

## 18.4 pin\_oc

```
template< typename T >
concept pin_oc = requires {
    box< bool >;
    duplex< bool >;
    T::_pin_oc_marker;
};
```

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, referring 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 wire low. When no chip does so, a common pull-up resistor pulls the line high. 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.

## 19 random

implemented in: library/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](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 truly random source) to start a truly random random sequence.

## 20 string

implemented in: library/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.

---

### 20.1 attributes

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

The `maximum_length` is the maximum 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 currently 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.

## 21 xy<>

implemented in: library/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 two 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: \*/

---

### 21.1 attributes

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

The x and y values are freely accessible.

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

The origin is the (0,0) value.

---

### 21.2 methods

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

The default constructor initializes 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 lhs, V b ){
        { lhs.x == b } -> std::same_as< bool >;
      }
constexpr bool operator==( const xy< V > & rhs ) const { ... }
```

```
template< typename V >
      requires requires( xy_value_type lhs, V b ){
        { lhs.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.

## 21.3 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 & >;
      }
}
```