deque

An implementation of a Deque (double-ended queue). The underlying implementation uses a `seq`.

Note: None of the procs that get an individual value from the Deque should be used on an empty Deque.

If compiled with the `boundChecks` option, those procs will raise an `IndexDefect` on such access. This should not be relied upon, as `-d:danger` or `--checks:off` will disable those checks and then the procs may return garbage or crash the program.

As such, a check to see if the Deque is empty is needed before any access, unless your program logic guarantees it indirectly.

Example:

```
import deque
var a = [10, 20, 30, 40].toDeque
doAssertRaises(IndexDefect, echo a[4])
assert not isEmpty(a)

a.addLast(50)
assert $a == "[10, 20, 30, 40, 50]"
assert a.first == 10
assert a.last == 50
assert len(a) == 5
assert capacity(a) == 8

assert a.popFirst == 10
assert a.popLast == 50
assert a.len == 3
assert a.capacity == 8
assert a.high == 2

a.addFirst(11)
a.addFirst(22)
a.addFirst(33)
a.first = 44
assert a == @@[44, 22, 11, 20, 30, 40]

a.shrink(fromFirst = 1, fromLast = 2)
assert a == @@[22, 11, 20]
```

Types

```
Deque[T] = object
```

> A double-ended queue backed with a ringed `seq` buffer.
>
> To initialize an empty Deque, use the initDeque proc.

Procs

```
proc `$`[T](deq: Deque[T]): string
```

Turns a Deque into its string representation.

Example:

```
let a = [10, 20, 30].toDeque
assert $a == "[10, 20, 30]"
```

```
proc `&`[T](x, y: sink Deque[T]): Deque[T] {.noSideEffect.}
```

Returns the concatenation of two Deques.

See also:

- [addLast(var Deque[T], sink Deque[T])](#)
- [&= template](#)

```
proc `&`[T](x: sink Deque[T]; y: sink openArray[T]): Deque[T] {.noSideEffect.}
```

Returns seq y appended to the end of Deque x.

See also:

- [addLast(var Deque[T], sink openArray[T])](#)
- [&= template](#)

```
proc `&`[T](x: sink Deque[T]; y: sink T): Deque[T] {.noSideEffect.}
```

Returns element y appended to the end of Deque x.

See also:

- [addLast(var Deque[T], T)](#)
- [&= template](#)

```
proc `&`[T](x: sink openArray[T]; y: sink Deque[T]): Deque[T] {.noSideEffect.}
```

Returns seq x prepended to the beginning of Deque y.

See also:

- [addFirst(var Deque[T], sink openArray[T])](#)

```
proc `&`[T](x: sink T; y: sink Deque[T]): Deque[T] {.noSideEffect.}
```

Returns element x prepended to the beginning of Deque y.

See also:

- [addFirst(var Deque[T], T)](#)

```
func `==`[T](deq1, deq2: Deque[T]): bool
```

The == operator for Deque. Returns `true` if both Deques contains the same values in the same order.

Example:

```
var a, b = initDeque[int]()
a.addFirst(2)
a.addFirst(1)
b.addLast(1)
b.addLast(2)
doAssert a == b
```

proc `[]`[T; U, V: Ordinal](target: Deque[T]; x: HSlice[U, V]): Deque[T] {.
    systemRaisesDefect.}

Slice operation for Deques. Returns the inclusive range [target[start .. stop]. If the slice indices are reversed, so will be the data.

```
var s = @[1, 2, 3, 4].toDeque
assert $s[0..2] == "[1, 2, 3]"
assert $s[1..<4] == "[2, 3]"
assert $s[^1..0] == "[4, 3, 2, 1]"
```

proc `[]`[T](deq: Deque[T]; i: BackwardsIndex): T {.inline.}

Accesses the backwards indexed i-th element.

deq[^1] is the last element.

Example:

```
let a = [10, 20, 30, 40, 50].toDeque
assert a[^1] == 50
assert a[^4] == 20
doAssertRaises(IndexDefect, echo a[^9])
```

proc `[]`[T](deq: Deque[T]; i: Natural): T {.inline.}

Accesses the i-th element of deq.

Example:

```
let a = [10, 20, 30, 40, 50].toDeque
assert a[0] == 10
assert a[3] == 40
doAssertRaises(IndexDefect, echo a[8])
```

proc `[]`[T](deq: var Deque[T]; i: BackwardsIndex): var T {.inline.}

Accesses the backwards indexed i-th element and returns a mutable reference to it.

deq[^1] is the last element.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
inc(a[^1])
assert a[^1] == 51
```

proc `[]`[T](deq: var Deque[T]; i: Natural): var T {.inline.}

Accesses the i-th element of deq and returns a mutable reference to it.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
inc(a[0])
assert a[0] == 11
```

proc `[]=`[T; U, V: Ordinal](target: var Deque[T]; x: HSlice[U, V];
                            source: openArray[T]) {.systemRaisesDefect.}

Slice assignment for Deques from sequences.

If source is longer than the slice, a slice of source is taken to fit. If source is shorter than the slice, target is shortened. If the slice indices are reversed, so will be the data.

Example:

```
var s = @[1, 2, 3, 4, 5].toDeque
s[1 .. ^2] = @[10, 20]
assert s == @[1, 10, 20, 5].toDeque
s[1..0] = @[100, 200]
assert s == @[200, 100, 20, 5].toDeque
```

proc `[]=`[T; U, V: Ordinal](target: var Deque[T]; x: HSlice[U, V];
                            source: Deque[T]) {.systemRaisesDefect.}

Slice assignment for Deques from Deques.

If source is longer than the slice, a slice of source is taken to fit. If source is shorter than the slice, target is shortened. If the slice indices are reversed, so will be the data.

Example:

```
var s = @[1, 2, 3, 4, 5].toDeque
s[1 .. ^2] = @[10, 20].toDeque
assert s == @[1, 10, 20, 5].toDeque
```

proc `[]=`[T](deq: var Deque[T]; i: BackwardsIndex; x: sink T) {.inline.}

Sets the backwards indexed i-th element of deq to x.

deq[^1] is the last element.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a[^1] = 99
a[^3] = 77
assert $a == "[10, 20, 77, 40, 99]"
```

proc `[]=`[T](deq: var Deque[T]; i: Natural; val: sink T) {.inline.}

Sets the i-th element of deq to val.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a[0] = 99
a[3] = 66
assert $a == "[99, 20, 30, 66, 50]"
```

proc addFirst[T](deq1: var Deque[T]; deq2: sink Deque[T])

Adds `deq2` to the beginning of `deq1` as concatenation.

See also:

- [addLast proc](#)

Example:

```
var a = [1, 2, 3].toDeque
let b = [10, 20, 30].toDeque
a.addFirst(b)
assert $a == "[10, 20, 30, 1, 2, 3]"
```

proc [addFirst](#)[T](deq1: var Deque[T]; seq2: sink openArray[T])

Adds `deq2` to the beginning of `deq1` as concatenation.

See also:

- [addLast proc](#)

Example:

```
var a = [1, 2, 3].toDeque
let b = [10, 20, 30]
a.addFirst(b)
assert $a == "[10, 20, 30, 1, 2, 3]"
```

proc [addFirst](#)[T](deq: var Deque[T]; item: sink T)

Adds an `item` to the beginning of `deq`.

See also:

- [addLast proc](#)
- [& proc](#)

Example:

```
var a = initDeque[int]()
for i in 1 .. 5:
  a.addFirst(10 * i)
assert $a == "[50, 40, 30, 20, 10]"
```

proc [addLast](#)[T](deq1: var Deque[T]; deq2: sink Deque[T])

Adds `deq2` to the end of `deq1` as concatenation.

See also:

- [addFirst proc](#)
- [& proc](#)
- [&= template](#)

Example:

```
var a = [1, 2, 3].toDeque
let b = [10, 20, 30].toDeque
```

```
      a.addLast(b)
      assert $a == "[1, 2, 3, 10, 20, 30]"
```

proc addLast[T](deq1: var Deque[T]; seq2: sink openArray[T])

> Adds deq2 to the end of deq1 as concatenation.
>
> **See also:**
>
> - addFirst proc
> - & proc
> - &= template
>
> **Example:**
>
> ```
> var a = [1, 2, 3].toDeque
> let b = @[10, 20, 30]
> a.addLast(b)
> assert $a == "[1, 2, 3, 10, 20, 30]"
> ```

proc addLast[T](deq: var Deque[T]; item: sink T)

> Adds an item to the end of deq.
>
> **See also:**
>
> - addFirst proc
> - & proc
> - &= template
>
> **Example:**
>
> ```
> var a = initDeque[int]()
> for i in 1 .. 5:
>   a.addLast(10 * i)
> assert $a == "[10, 20, 30, 40, 50]"
> ```

proc contains[T](deq: Deque[T]; item: T): bool {.inline.}

> Returns true if item is in deq or false if not found.
>
> Usually used via the in operator. It is the equivalent of deq.find(item) >= 0.
>
> **Example:**
>
> ```
> let q = [7, 9].toDeque
> assert 7 in q
> assert q.contains(7)
> assert 8 notin q
> ```

proc delete[T; U, V: Ordinal](target: var Deque[T]; x: HSlice[U, V]) {.
    systemRaisesDefect.}

> Deletes the elements at slice x, moving down all elements higher than that.
>
> **Example:**
>
> ```
> var a = [10, 20, 30, 40, 50].toDeque
> a.delete(1..2)
> ```

```
        assert $a == "[10, 40, 50]"
        assert a.len == 3
```

proc delete[T](deq: var Deque[T]; where: BackwardsIndex) {.systemRaisesDefect.}

Deletes the element at backwards index `where`, moving down all elements higher than that.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a.delete(^2)
assert $a == "[10, 20, 30, 50]"
assert a.len == 4
```

proc delete[T](deq: var Deque[T]; where: Natural) {.systemRaisesDefect.}

Deletes the element at `where`, moving down all elements higher than that.

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
delete(a, 3)
assert $a == "[10, 20, 30, 50]"
assert a.len == 4
```

proc dropFirst[T](deq: var Deque[T]) {.inline.}

Removes the first element of the `deq`.

See also:

- popFirst proc
- dropLast proc

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
a.dropFirst
assert $a == "[20, 30, 40, 50]"
```

proc dropLast[T](deq: var Deque[T]) {.inline.}

Removes the last element of the `deq`.

See also:

- dropFirst proc
- popLast proc

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
a.dropLast
assert $a == "[10, 20, 30, 40]"
```

proc first[T](deq: Deque[T]): T {.inline.}

Returns the first element of `deq`, but does not remove it from the Deque.

See also:

- [first proc](#) which returns a mutable reference
- [last proc](#)

Example:

```
let a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
assert a.first == 10
assert len(a) == 5
```

proc `first`[T](deq: var Deque[T]): var T {.inline.}

Returns a mutable reference to the first element of `deq`, but does not remove it from the Deque.

See also:

- [first proc](#)
- [last proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
first(a) = 99
assert $a == "[99, 20, 30, 40, 50]"
inc a.first
assert $a == "[100, 20, 30, 40, 50]"
```

proc `first=`[T](deq: var Deque[T]; item: sink T) {.inline.}

Alters the first element of `deq`.

See also:

- [first proc](#)
- [last proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a.first = 99
assert $a == "[99, 20, 30, 40, 50]"
```

func `hash`[T](deq: Deque[T]): Hash

Hashing of Deque.

proc `initDeque`[T](initialSize: Natural = defaultInitialSize): Deque[T]

Creates a new empty Deque of capacity `initialSize`. The length of a newly created Deque will be 0. Capacity is always a power of two, with a minimum of two.

(default and capacity: [defaultInitialSize](#)).

See also:

Example:

```
var deq1 = initDeque[int](6)
assert capacity(deq1) == 8
assert len(deq1) == 0
```

proc [insert](#)[T](target: var Deque[T]; source: sink Deque[T]; pos: Natural)

Insert source Deque into target Deque in front of position pos.

proc [insert](#)[T](target: var Deque[T]; source: sink openArray[T]; pos: Natural)

Insert source sequence into target Deque in front of position pos.

proc [insert](#)[T](target: var Deque[T]; source: sink T; pos: Natural)

Insert source element into target Deque in front of position pos.

proc [last](#)[T](deq: Deque[T]): T {.inline.}

Returns the last element of deq, but does not remove it from the Deque.

See also:

- [last proc](#) which returns a mutable reference
- [first proc](#)

Example:

```
let a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
assert a.last == 50
assert len(a) == 5
```

proc [last](#)[T](deq: var Deque[T]): var T {.inline.}

Returns a mutable reference to the last element of deq, but does not remove it from the Deque.

See also:

- [first proc](#)
- [last proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a.last() = 99
assert $a == "[10, 20, 30, 40, 99]"
inc a.last
assert $a == "[10, 20, 30, 40, 100]"
```

proc [last=](#)[T](deq: var Deque[T]; item: sink T) {.inline.}

Alters the last element of deq.

See also:

- [first proc](#)
- [last proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
a.last = 99
assert $a == "[10, 20, 30, 40, 99]"
```

proc [newDeque](#)[T](initialSize: Natural = defaultInitialSize): Deque[T]

Creates a new empty Deque of capacity `initialSize`. The length of a newly created Deque will be 0. Capacity is always a power of two, with a minimum of two.

(default capacity: [defaultInitialSize](#)).

See also:

- [initDeque proc](#)
- [toDeque proc](#)

Example:

```
var deq1 = newDeque[int]()
assert capacity(deq1) == defaultInitialSize
assert len(deq1) == 0
```

proc [popFirst](#)[T](deq: var Deque[T]): T {.inline, discardable.}

Removes and returns the first element of the `deq`.

See also:

- [popLast proc](#)
- [shrink proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
assert a.popFirst == 10
assert $a == "[20, 30, 40, 50]"
```

proc [popLast](#)[T](deq: var Deque[T]): T {.inline, discardable.}

Removes and returns the last element of the `deq`.

See also:

- [popFirst proc](#)
- [shrink proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
assert a.popLast == 50
assert $a == "[10, 20, 30, 40]"
```

proc reverse[T](target: var Deque[T])

>   Reverses target in place.

proc reversed[T](source: Deque[T]): Deque[T]

>   Returns a reversed copy of source.

proc rotL[T](deq: var Deque[T]; howMany: Natural = 1) {.inline.}

>   Rotate each element of deq howMany places to the left, wrapping around. Default is one.
>
>   **See also:**
>
>   - rotR proc
>
>   **Example:**

```
var deq = @@[1, 2, 3, 4, 5]
deq.rotL
assert deq == @@[2, 3, 4, 5, 1]
rotL(deq, 3)
assert deq == @@[5, 1, 2, 3, 4]
```

proc rotR[T](deq: var Deque[T]; howMany: Natural = 1) {.inline.}

>   Rotate each element of deq howMany places to the right, wrapping around. Default is one.
>
>   **See also:**
>
>   - rotL proc
>
>   **Example:**

```
var deq = @@[1, 2, 3, 4, 5]
deq.rotR
assert deq == @@[5, 1, 2, 3, 4]
rotR(deq, 2)
assert deq == @@[3, 4, 5, 1, 2]
deq.rotR(0)
assert deq == [3, 4, 5, 1, 2].toDeque
```

proc setCap[T](target: var Deque[T]; length: Natural)

>   Sets the capacity of target to length, shrinking or growing as needed. Capacity will always be a power of two.

proc setLen[T](target: var Deque[T]; length: Natural)

>   Sets the length of target to length, increasing its capacity if needed.

proc shrink[T](deq: var Deque[T]; fromFirst = 0; fromLast = 0)

>   Removes fromFirst elements from the front of the Deque and fromLast elements from the back.
>
>   If the supplied number of elements exceeds the total number of elements in the Deque, the Deque will remain empty.
>
>   **See also:**

- [clear template](#)
- [dropFirst proc](#)
- [dropLast proc](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
a.shrink(fromFirst = 2, fromLast = 1)
assert $a == "[30, 40]"
```

```
proc toDeque[T](x: sink openArray[T]): Deque[T]
```

Creates a new Deque that contains the elements of x (in the same order).

See also:

- [initDeque proc](#)
- [newDeque proc](#)
- [toDeque template](#)
- [@@ template](#)

Example:

```
let a = toDeque([7, 8, 9])
assert len(a) == 3
assert $a == "[7, 8, 9]"
```

Iterators

```
iterator backwards[T](deq: Deque[T]): T {.inline.}
```

Yields each element of deq in reverse order.

See also:

- [items iterator](#)
- [backwardsMut iterator](#)

Example:

```
let thisDeq = [1, 2, 3, 4, 5].toDeque
var thatDeq = [6].toDeque
for item in backwards(thisDeq):
  thatDeq.addLast(item)
assert thisDeq == [1, 2, 3, 4, 5].toDeque
assert thatdeq == [6, 5, 4, 3, 2, 1].toDeque
```

```
iterator backwardsMut[T](deq: var Deque[T]): var T {.inline.}
```

Yields in reverse order a mutable version of every element of deq.

See also:

- [mitems iterator](#)
- [backwards iterator](#)

Example:

```
var thisDeq = [1, 2, 3, 4, 5].toDeque
var thatDeq = [12].toDeque
var otherDeq = @@[6]
for item in backwardsMut(thisDeq):
  otherDeq.addFirst(item)
  item *= 2
  thatDeq.addLast(item)
assert thisDeq == [2, 4, 6, 8, 10].toDeque
assert thatDeq == [12, 10, 8, 6, 4, 2].toDeque
assert otherDeq == @@[1, 2, 3, 4, 5, 6]
```

iterator `backwardsPairs`[T](deq: Deque[T]): tuple[key: int, val: T] {.inline.}

Yields every `(position, value)`-pair of `deq` in reverse order.

See also:

- [pairs iterator](#)

Example:

```
import std/sequtils
let a = [10, 20, 30].toDeque
assert toSeq(a.backwardsPairs) == @[(2, 30), (1, 20), (0, 10)]
```

iterator `items`[T](deq: Deque[T]): T {.inline.}

Yields every element of `deq`.

See also:

- [mitems iterator](#)
- [backwards iterator](#)

Example:

```
let a = [10, 20, 30, 40, 50].toDeque
var b: seq[int]
for item in a: b.add(item)
assert b == @[10, 20, 30, 40, 50]
assert $a == "[10, 20, 30, 40, 50]"
```

iterator `mitems`[T](deq: var Deque[T]): var T {.inline.}

Yields every element of `deq`, which can be modified.

See also:

- [items iterator](#)
- [backwardsMut iterator](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
for x in mitems(a):
  x = 5 * x - 1
```

```
assert $a == "[49, 99, 149, 199, 249]"
```

iterator pairs[T](deq: Deque[T]): tuple[key: int, val: T] {.inline.}

Yields every (position, value)-pair of deq.

See also:

- backwardsPairs iterator

Example:

```
import std/sequtils
let a = [10, 20, 30].toDeque
assert toSeq(a.pairs) == @[(0, 10), (1, 20), (2, 30)]
```

Templates

template `@@`[T](deq: Deque[T]): Deque[T]

Returns a copy of deq.

See also:

- toDeque template

template `@@`[T](x: openArray[T]): Deque[T]

Creates a new Deque that contains the elements of x (in the same order).

See also:

- toDeque proc

Example:

```
let thisDeq = @@[1, 2, 3]
assert thisDeq == [1, 2, 3].todeque
assert $thisDeq == "[1, 2, 3]"
```

template capacity[T](deq: Deque[T]): int

Returns the maximum capacity of the sequence backing deq. Capacity is always a power of two.

Example:

```
var deq = [1, 2, 3, 4].toDeque
assert deq.len == deq.capacity
deq.addLast(5)
assert deq.len == 5
assert deq.capacity == 8
```

template clear[T](deq: var Deque[T])

Resets the Deque so that it is empty, but retains its capacity.

See also:

- [reset template](#)

Example:

```
var a = [10, 20, 30, 40, 50].toDeque
assert $a == "[10, 20, 30, 40, 50]"
clear(a)
assert len(a) == 0
assert capacity(a) == 8
```

template `high`[T](deq: Deque[T]): int

Returns the highest valid index in `deq`. This is the same as len(deq) - 1. If `deq` is empty, will return -1.

template `isEmpty`[T](deq: Deque[T]): bool

Returns true is `deq` is empty, false otherwise.

template `len`[T](deq: Deque[T]): int

Returns the number of elements in `deq`.

template `low`[T](deq: Deque[T]): int

Returns the lowest valid index in `deq`, normally 0. If `deq` is empty, will return -1.

template `pushFirst`[T](deq1: var Deque[T]; deq2: sink Deque[T])

Alias for:
- [addFirst proc](#)

template `pushFirst`[T](deq1: var Deque[T]; seq2: sink openArray[T])

Alias for:
- [addFirst proc](#)

template `pushFirst`[T](deq: var Deque[T]; item: sink T)

Alias for:
- [addFirst proc](#)

template `pushLast`[T](deq1: var Deque[T]; deq2: sink Deque[T])

Alias for:
- [addLast proc](#)

template `pushLast`[T](deq1: var Deque[T]; seq2: sink openArray[T])

Alias for:
- [addLast proc](#)

template `pushLast`[T](deq: var Deque[T]; item: sink T)

Alias for:
- [addLast proc](#)

template `reset`[T](deq: var Deque[T]; maxCap: Natural = defaultInitialSize)

Resets `deq` so it is empty and sets its capacity to `maxCap`. Capacity is always a power of two.

See also:

- [clear template](#)
- [defaultInitialSize constant](#)

```
template rotateLeft[T](deq: var Deque[T]; howMany: Natural = 1)
```

Alias for:
- [rotL proc](#)

```
template rotateRight[T](deq: var Deque[T]; howMany: Natural = 1)
```

Alias for:
- [rotR proc](#)

```
template toDeque[T](deq: Deque[T]): Deque[T]
```

Returns a copy of `deq`.

See also:

- [toDeque proc](#)
- [@@ template](#)