From Mathematics to Generic Programming

Generated by Doxygen 1.8.14

Thu Jan 1 1970 00:00:01

Contents

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

3 Class Documentation

3.1 reciprocal< T > Struct Template Reference

Public Member Functions

- T **operator()** (const T &x) const

3.1.1 Detailed Description

template<MultiplicativeGroup T>
struct reciprocal< T >

Definition at line 97 of file ch07.hpp.

The documentation for this struct was generated from the following file:

- ch07.hpp

4 File Documentation

4.1 ch02.hpp File Reference

Egyptian Multiplication.

Functions

- int multiply0 (int n, int a)
- bool odd (int n)
- int half (int n)
- int multiply1 (int n, int a)

4.1.1 Detailed Description

Egyptian Multiplication.

Author

Eric Bailey

Date

2016-12-11

Definition in file ch02.hpp.

4.1.2 Function Documentation

4.1.2.1 half()

```
int half (
            int n )
```

Return half of a given number.

$$\text{half}(n) = \frac{n}{2}$$

Definition at line 45 of file ch02.hpp.

```
00046 {
00047   return n >> 1;
00048 }
```

4.1.2.2 multiply0()

```
int multiply0 (
            int n,
            int a )
```

Add instances of `a` together `n` times.

Efficiency: $\mathcal{O}(n)$

Parameters

n	the number of instances of `a` to add.
a	the number to add `n` times.

Returns

$n \times a$

$$1a = a \tag{2.1}$$

$$(n+1)a = na + a \tag{2.2}$$

Definition at line 16 of file ch02.hpp.

```
00017 {
00019   if (n == 1) return a;
00021   return multiply0(n - 1, a) + a;
00022 }
```

4.1.2.3 multiply1()

```
int multiply1 (
              int n,
              int a )
```

"Egyptian multiplication" aka the "Russian Peasant Algorithm"

$$\begin{aligned} 4a &= ((a+a)+a)+a \\ &= (a+a)+(a+a) \end{aligned}$$

The law of associativity of addition:

$$a + (b + c) = (a + b) + c$$

Power of 2	1-bits	Doublings
1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59)$$

Efficiency: $\mathcal{O}(\log_2 n)$

$$\begin{aligned} \#_{+}(n) &= \lfloor \log_2 n \rfloor + (v(n) - 1) \\ v(n) &= \text{the number of 1s in the binary representation of } n\text{, i.e. the } \textit{population count} \end{aligned}$$

Definition at line 83 of file ch02.hpp.

```
00084 {
00085   if (n == 1) return a;
00086   int result = multiply1(half(n), a + a);
00087   if (odd(n)) result = result + a;
00088   return result;
00089 }
```

4.1.2.4 odd()

```
bool odd (
            int n )
```

Determine whether a number is odd.

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \implies \mathsf{odd}(n)$$

$$\mathsf{odd}(n) \implies \mathsf{half}(n) = \mathsf{half}(n-1)$$

Definition at line 34 of file ch02.hpp.

```
00035 {
00036   return n & 0x1;
00037 }
```

4.2 ch02.hpp

```
00001
00016 int multiply0(int n, int a)
00017 {
00019   if (n == 1) return a;
00021   return multiply0(n - 1, a) + a;
00022 }
00023
00034 bool odd(int n)
00035 {
00036   return n & 0x1;
00037 }
00038
00045 int half(int n)
00046 {
00047   return n >> 1;
00048 }
00049
00083 int multiply1(int n, int a)
00084 {
00085   if (n == 1) return a;
00086   int result = multiply1(half(n), a + a);
00087   if (odd(n)) result = result + a;
00088   return result;
00089 }
```

4.3 ch07.hpp File Reference

Deriving a Generic Algorithm.

Classes

- struct reciprocal< T >

Macros

- #define **Integer** typename
- #define **Regular** typename
- #define **SemigroupOperation** typename
- #define **MonoidOperation** typename
- #define **GroupOperation** typename
- #define **AdditiveGroup** typename
- #define **NoncommutativeAdditiveGroup** typename
- #define **NoncommutativeAdditiveMonoid** typename
- #define **NoncommutativeAdditiveSemigroup** typename
- #define **MultiplicativeGroup** typename
- #define **MultiplicativeMonoid** typename
- #define **MultiplicativeSemigroup** typename

Functions

- template<Integer N>
  bool odd (N n)
- template<Integer N>
  N half (N n)
- template<NoncommutativeAdditiveMonoid T>
  T **identity_element** (std::plus< T >)
- template<MultiplicativeMonoid T>
  T **identity_element** (std::multiplies< T >)
- template<AdditiveGroup T>
  std::negate< T > **inverse_operation** (std::plus< T >)
- template<NoncommutativeAdditiveSemigroup A, Integer N>
  A **multiply_accumulate_semigroup** (A r, N n, A a)
- template<NoncommutativeAdditiveSemigroup A, Integer N>
  A **multiply_semigroup** (N n, A a)
- template<NoncommutativeAdditiveMonoid A, Integer N>
  A **multiply_monoid** (N n, A a)
- template<NoncommutativeAdditiveGroup A, Integer N>
  A **multiply_group** (N n, A a)
- template<MultiplicativeSemigroup A, Integer N>
  A **power_accumulate_semigroup** (A r, A a, N n)
- template<MultiplicativeSemigroup A, Integer N>
  A **power_semigroup** (A a, N n)

- template< MultiplicativeMonoid A, Integer N>
  A **power_monoid** (A a, N n)
- template< MultiplicativeGroup A>
  A **multiplicative_inverse** (A a)
- template< MultiplicativeGroup A, Integer N>
  A **power_group** (A a, N n)
- template< Regular A, Integer N, SemigroupOperation Op>
  A **power_accumulate_semigroup** (A r, A a, N n, Op op)
- template< Regular A, Integer N, SemigroupOperation Op>
  A **power_semigroup** (A a, N n, Op op)
- template< Regular A, Integer N, MonoidOperation Op>
  A **power_monoid** (A a, N n, Op op)

Variables

- template< typename Operation , typename Element >
  concept bool **Associative**
- template< typename T >
  concept bool **EqualityComparable**
- template< typename T >
  concept bool **InequalityComparable**
- template< MultiplicativeGroup T>
  reciprocal< T > **inverse_operation** (std::multiplies< T >)

4.3.1 Detailed Description

Deriving a Generic Algorithm.

Author

Eric Bailey

Date

2017-01-13

Definition in file ch07.hpp.

4.3.2 Function Documentation

4.3.2.1 half()

```
template<Integer N>
N half (
            N n )
```

Return half of a given number.

$$\text{half}(n) = \frac{n}{2}$$

Definition at line 67 of file ch07.hpp.

```
{
  return n >> 1;
}
```

4.3.2.2 odd()

```
template<Integer N>
bool odd (
            N n )
```

Determine whether a number is odd.

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \Longrightarrow \text{odd}(n)$$

$$\text{odd}(n) \Longrightarrow \text{half}(n) = \text{half}(n-1)$$

Definition at line 55 of file ch07.hpp.

```
{
  return bool(n & 0x1);
}
```

4.3.3 Variable Documentation

4.3.3.1 Associative

template<typename Operation , typename Element >
concept bool Associative

Initial value:

```
= requires(Operation op, Element x, Element y, Element z) {
  op(x, op(y, z)) == op(op(x, y), z);
}
```

Definition at line 29 of file ch07.hpp.

4.3.3.2 EqualityComparable

template<typename T >
concept bool EqualityComparable

Initial value:

```
= requires(T a, T b) {
  { a == b } -> bool;
}
```

Definition at line 34 of file ch07.hpp.

4.3.3.3 InequalityComparable

template<typename T >
concept bool InequalityComparable

Initial value:

```
= requires(T a, T b) {
  { a != b } -> bool;
}
```

Definition at line 39 of file ch07.hpp.

4.4 ch07.hpp

```
// #include <functional>
// #include <type_traits>

#define Integer typename
#define Regular typename

#define SemigroupOperation typename
#define MonoidOperation typename
#define GroupOperation typename

#define AdditiveGroup typename

#define NoncommutativeAdditiveGroup typename
#define NoncommutativeAdditiveMonoid typename
#define NoncommutativeAdditiveSemigroup typename

#define MultiplicativeGroup typename
#define MultiplicativeMonoid typename
#define MultiplicativeSemigroup typename

template <typename Operation, typename Element>
concept bool Associative = requires(Operation op, Element x, Element y, Element z) {
  op(x, op(y, z)) == op(op(x, y), z);
};

template<typename T>
concept bool EqualityComparable = requires(T a, T b) {
  { a == b } -> bool;
};

template<typename T>
concept bool InequalityComparable = requires(T a, T b) {
  { a != b } -> bool;
};


template <Integer N>
bool odd(N n)
{
  return bool(n & 0x1);
}

template <Integer N>
N half(N n)
{
  return n >> 1;
}



template <NoncommutativeAdditiveMonoid T>
T identity_element(std::plus<T>)
{
  // "The additive identity is 0."
  return T(0);
}

template <MultiplicativeMonoid T>
T identity_element(std::multiplies<T>)
{
  // "The multiplicative identity is 1."
  return T(1);
}

template <AdditiveGroup T>
std::negate<T> inverse_operation(std::plus<T>)
{
  return std::negate<T>();
}


// Generalization of the multiplicative_inverse function.
template <MultiplicativeGroup T>
struct reciprocal
{
  T operator()(const T& x) const {
    return T(1) / 1;
```

```
  }
};

template <MultiplicativeGroup T>
reciprocal<T> inverse_operation(std::multiplies<T>)
{
  return reciprocal<T>();
}



template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_accumulate_semigroup(A r, N n, A a)
{
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}

template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_semigroup(N n, A a)
{
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  return multiply_accumulate_semigroup(a, half(n - 1), a + a);
}

template <NoncommutativeAdditiveMonoid A, Integer N>
A multiply_monoid(N n, A a)
{
  if (n == 0) return A(0);
  return multiply_semigroup(n, a);
}

template <NoncommutativeAdditiveGroup A, Integer N>
A multiply_group(N n, A a)
{
  if (n < 0) {
    n = -n;
    a = -a;
  }
  return multiply_monoid(n, a);
}

template <MultiplicativeSemigroup A, Integer N>
A power_accumulate_semigroup(A r, A a, N n)
{
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r * a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a * a;
  }
}

template <MultiplicativeSemigroup A, Integer N>
A power_semigroup(A a, N n)
{
  while (!odd(n)) {
    a = a * a;
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, a * a, half(n - 1));
}

template <MultiplicativeMonoid A, Integer N>
A power_monoid(A a, N n)
{
```

```
  if (n == 0) return A(1);
  return power_semigroup(a, n);
}

template <MultiplicativeGroup A>
A multiplicative_inverse(A a)
{
  return A(1) / a;
}

template <MultiplicativeGroup A, Integer N>
A power_group(A a, N n)
{
  if (n < 0) {
    n = -n;
    a = multiplicative_inverse(a);
  }
  return power_monoid(a, n);
}

template <Regular A, Integer N, SemigroupOperation Op>
A power_accumulate_semigroup(A r, A a, N n, Op op)
{
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = op(r, a);
      if (n == 1) return r;
    }
    n = half(n);
    a = op(a, a);
  }
}

template <Regular A, Integer N, SemigroupOperation Op>
A power_semigroup(A a, N n, Op op)
{
  while (!odd(n)) {
    a = op(a, a);
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, op(a, a), half(n - 1), op);
}

template <Regular A, Integer N, MonoidOperation Op>
// requires(Domain<Op, A>)
A power_monoid(A a, N n, Op op)
{
  // precondition (n >= 0);
  if (n == 0); return identity_element(op);
  return power_semigroup(a, n, op);
}
```

Index