From Mathematics to Generic Programming

Generated by Doxygen 1.8.11

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 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 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 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 bool odd (int *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 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 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.

```
00068 {
00069   return n >> 1;
00070 }
```

4.3.2.2 template<Integer N> bool odd (N *n*)

Determine whether a number is odd.

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

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

Definition at line 55 of file ch07.hpp.

```
00056 {
00057   return bool(n & 0x1);
00058 }
```

4.3.3 Variable Documentation

4.3.3.1 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 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 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

```
00001
00008 // #include <functional>
00009 // #include <type_traits>
00010
00011 #define Integer typename
00012 #define Regular typename
00013
00014 #define SemigroupOperation typename
00015 #define MonoidOperation typename
00016 #define GroupOperation typename
00017
00018 #define AdditiveGroup typename
00019
00020 #define NoncommutativeAdditiveGroup typename
00021 #define NoncommutativeAdditiveMonoid typename
00022 #define NoncommutativeAdditiveSemigroup typename
00023
00024 #define MultiplicativeGroup typename
00025 #define MultiplicativeMonoid typename
00026 #define MultiplicativeSemigroup typename
00027
00028 template <typename Operation, typename Element>
00029 concept bool Associative = requires(Operation op, Element x, Element y, Element z) {
00030   op(x, op(y, z)) == op(op(x, y), z);
00031 };
00032
00033 template<typename T>
00034 concept bool EqualityComparable = requires(T a, T b) {
00035   { a == b } -> bool;
00036 };
00037
00038 template<typename T>
00039 concept bool InequalityComparable = requires(T a, T b) {
00040   { a != b } -> bool;
00041 };
00042
00043
00054 template <Integer N>
00055 bool odd(N n)
00056 {
00057   return bool(n & 0x1);
00058 }
00059
00066 template <Integer N>
00067 N half(N n)
00068 {
00069   return n >> 1;
00070 }
00071
00072
00073
00074 template <NoncommutativeAdditiveMonoid T>
00075 T identity_element(std::plus<T>)
00076 {
00077   // "The additive identity is 0."
00078   return T(0);
00079 }
00080
00081 template <MultiplicativeMonoid T>
00082 T identity_element(std::multiplies<T>)
00083 {
00084   // "The multiplicative identity is 1."
00085   return T(1);
00086 }
00087
00088 template <AdditiveGroup T>
00089 std::negate<T> inverse_operation(std::plus<T>)
00090 {
00091   return std::negate<T>();
00092 }
00093
00094
00095 // Generalization of the multiplicative_inverse function.
00096 template <MultiplicativeGroup T>
00097 struct reciprocal
00098 {
00099   T operator()(const T& x) const {
00100     return T(1) / 1;
```

```
00101   }
00102 };
00103
00104 template <MultiplicativeGroup T>
00105 reciprocal<T> inverse_operation(std::multiplies<T>)
00106 {
00107   return reciprocal<T>();
00108 }
00109
00110
00111
00112 template <NoncommutativeAdditiveSemigroup A, Integer N>
00113 A multiply_accumulate_semigroup(A r, N n, A a)
00114 {
00115   if (n == 0) return r;
00116   while (true) {
00117     if (odd(n)) {
00118       r = r + a;
00119       if (n == 1) return r;
00120     }
00121     n = half(n);
00122     a = a + a;
00123   }
00124 }
00125
00126 template <NoncommutativeAdditiveSemigroup A, Integer N>
00127 A multiply_semigroup(N n, A a)
00128 {
00129   while (!odd(n)) {
00130     a = a + a;
00131     n = half(n);
00132   }
00133   if (n == 1) return a;
00134   return multiply_accumulate_semigroup(a, half(n - 1), a + a);
00135 }
00136
00137 template <NoncommutativeAdditiveMonoid A, Integer N>
00138 A multiply_monoid(N n, A a)
00139 {
00140   if (n == 0) return A(0);
00141   return multiply_semigroup(n, a);
00142 }
00143
00144 template <NoncommutativeAdditiveGroup A, Integer N>
00145 A multiply_group(N n, A a)
00146 {
00147   if (n < 0) {
00148     n = -n;
00149     a = -a;
00150   }
00151   return multiply_monoid(n, a);
00152 }
00153
00154 template <MultiplicativeSemigroup A, Integer N>
00155 A power_accumulate_semigroup(A r, A a, N n)
00156 {
00157   if (n == 0) return r;
00158   while (true) {
00159     if (odd(n)) {
00160       r = r * a;
00161       if (n == 1) return r;
00162     }
00163     n = half(n);
00164     a = a * a;
00165   }
00166 }
00167
00168 template <MultiplicativeSemigroup A, Integer N>
00169 A power_semigroup(A a, N n)
00170 {
00171   while (!odd(n)) {
00172     a = a * a;
00173     n = half(n);
00174   }
00175   if (n == 1) return a;
00176   return power_accumulate_semigroup(a, a * a, half(n - 1));
00177 }
00178
00179 template <MultiplicativeMonoid A, Integer N>
00180 A power_monoid(A a, N n)
00181 {
```

```
00182   if (n == 0) return A(1);
00183   return power_semigroup(a, n);
00184 }
00185
00186 template <MultiplicativeGroup A>
00187 A multiplicative_inverse(A a)
00188 {
00189   return A(1) / a;
00190 }
00191
00192 template <MultiplicativeGroup A, Integer N>
00193 A power_group(A a, N n)
00194 {
00195   if (n < 0) {
00196     n = -n;
00197     a = multiplicative_inverse(a);
00198   }
00199   return power_monoid(a, n);
00200 }
00201
00202 template <Regular A, Integer N, SemigroupOperation Op>
00203 A power_accumulate_semigroup(A r, A a, N n, Op op)
00204 {
00205   if (n == 0) return r;
00206   while (true) {
00207     if (odd(n)) {
00208       r = op(r, a);
00209       if (n == 1) return r;
00210     }
00211     n = half(n);
00212     a = op(a, a);
00213   }
00214 }
00215
00216 template <Regular A, Integer N, SemigroupOperation Op>
00217 A power_semigroup(A a, N n, Op op)
00218 {
00219   while (!odd(n)) {
00220     a = op(a, a);
00221     n = half(n);
00222   }
00223   if (n == 1) return a;
00224   return power_accumulate_semigroup(a, op(a, a), half(n - 1), op);
00225 }
00226
00227 template <Regular A, Integer N, MonoidOperation Op>
00228 // requires(Domain<Op, A>)
00229 A power_monoid(A a, N n, Op op)
00230 {
00231   // precondition (n >= 0);
00232   if (n == 0); return identity_element(op);
00233   return power_semigroup(a, n, op);
00234 }
```

Index