// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include <cstdlib>

#include <memory>

#include <utility>

#include "arrow/array.h"

#include "arrow/chunked\_array.h"

#include "arrow/datum.h"

#include "arrow/extension/uuid.h"

#include "arrow/ipc/json\_simple.h"

#include "arrow/python/gdb.h"

#include "arrow/record\_batch.h"

#include "arrow/scalar.h"

#include "arrow/table.h"

#include "arrow/type.h"

#include "arrow/util/debug.h"

#include "arrow/util/decimal.h"

#include "arrow/util/key\_value\_metadata.h"

#include "arrow/util/logging.h"

#include "arrow/util/macros.h"

namespace arrow {

using extension::uuid;

using extension::UuidType;

using ipc::internal::json::ArrayFromJSON;

using ipc::internal::json::ChunkedArrayFromJSON;

using ipc::internal::json::ScalarFromJSON;

namespace gdb {

// Add a nested `arrow` namespace to exercise type lookup from GDB (ARROW-15652)

namespace arrow {

void DummyFunction() {}

} // namespace arrow

namespace {

class CustomStatusDetail : public StatusDetail {

public:

const char\* type\_id() const override { return "custom-detail-id"; }

std::string ToString() const override { return "This is a detail"; }

};

std::shared\_ptr<Array> SliceArrayFromJSON(const std::shared\_ptr<DataType>& ty,

std::string\_view json, int64\_t offset = 0,

int64\_t length = -1) {

auto array = \*ArrayFromJSON(ty, json);

if (length != -1) {

return array->Slice(offset, length);

} else {

return array->Slice(offset);

}

}

} // namespace

void TestSession() {

// We define local variables for all types for which we want to test

// pretty-printing.

// Then, at the end of this function, we trap to the debugger, so that

// test instrumentation can print values from this frame by interacting

// with the debugger.

// The test instrumentation is in pyarrow/tests/test\_gdb.py

#ifdef \_\_clang\_\_

\_Pragma("clang diagnostic push");

\_Pragma("clang diagnostic ignored \"-Wunused-variable\"");

#elif defined(\_\_GNUC\_\_)

\_Pragma("GCC diagnostic push");

\_Pragma("GCC diagnostic ignored \"-Wunused-variable\"");

#endif

arrow::DummyFunction();

// Status & Result

auto ok\_status = Status::OK();

auto error\_status = Status::IOError("This is an error");

auto error\_detail\_status =

error\_status.WithDetail(std::make\_shared<CustomStatusDetail>());

auto ok\_result = Result<int>(42);

auto error\_result = Result<int>(error\_status);

auto error\_detail\_result = Result<int>(error\_detail\_status);

// String views

std::string\_view string\_view\_abc{"abc"};

std::string special\_chars = std::string("foo\"bar") + '\x00' + "\r\n\t\x1f";

std::string\_view string\_view\_special\_chars(special\_chars);

// Buffers

Buffer buffer\_null{nullptr, 0};

Buffer buffer\_abc{string\_view\_abc};

Buffer buffer\_special\_chars{string\_view\_special\_chars};

char mutable\_array[3] = {'a', 'b', 'c'};

MutableBuffer buffer\_mutable{reinterpret\_cast<uint8\_t\*>(mutable\_array), 3};

auto heap\_buffer = std::make\_shared<Buffer>(string\_view\_abc);

auto heap\_buffer\_mutable = \*AllocateBuffer(buffer\_abc.size());

memcpy(heap\_buffer\_mutable->mutable\_data(), buffer\_abc.data(), buffer\_abc.size());

// KeyValueMetadata

auto empty\_metadata = key\_value\_metadata({}, {});

auto metadata = key\_value\_metadata(

{"key\_text", "key\_binary"}, {"some value", std::string("z") + '\x00' + "\x1f\xff"});

// Decimals

Decimal128 decimal128\_zero{};

Decimal128 decimal128\_pos{"98765432109876543210987654321098765432"};

Decimal128 decimal128\_neg{"-98765432109876543210987654321098765432"};

BasicDecimal128 basic\_decimal128\_zero{};

BasicDecimal128 basic\_decimal128\_pos{decimal128\_pos.native\_endian\_array()};

BasicDecimal128 basic\_decimal128\_neg{decimal128\_neg.native\_endian\_array()};

Decimal256 decimal256\_zero{};

Decimal256 decimal256\_pos{

"9876543210987654321098765432109876543210987654321098765432109876543210987654"};

Decimal256 decimal256\_neg{

"-9876543210987654321098765432109876543210987654321098765432109876543210987654"};

BasicDecimal256 basic\_decimal256\_zero{};

BasicDecimal256 basic\_decimal256\_pos{decimal256\_pos.native\_endian\_array()};

BasicDecimal256 basic\_decimal256\_neg{decimal256\_neg.native\_endian\_array()};

// Data types

NullType null\_type;

auto heap\_null\_type = null();

BooleanType bool\_type;

auto heap\_bool\_type = boolean();

Date32Type date32\_type;

Date64Type date64\_type;

Time32Type time\_type\_s(TimeUnit::SECOND);

Time32Type time\_type\_ms(TimeUnit::MILLI);

Time64Type time\_type\_us(TimeUnit::MICRO);

Time64Type time\_type\_ns(TimeUnit::NANO);

auto heap\_time\_type\_ns = time64(TimeUnit::NANO);

TimestampType timestamp\_type\_s(TimeUnit::SECOND);

TimestampType timestamp\_type\_ms\_timezone(TimeUnit::MILLI, "Europe/Paris");

TimestampType timestamp\_type\_us(TimeUnit::MICRO);

TimestampType timestamp\_type\_ns\_timezone(TimeUnit::NANO, "Europe/Paris");

auto heap\_timestamp\_type\_ns\_timezone = timestamp(TimeUnit::NANO, "Europe/Paris");

DayTimeIntervalType day\_time\_interval\_type;

MonthIntervalType month\_interval\_type;

MonthDayNanoIntervalType month\_day\_nano\_interval\_type;

DurationType duration\_type\_s(TimeUnit::SECOND);

DurationType duration\_type\_ns(TimeUnit::NANO);

BinaryType binary\_type;

StringType string\_type;

LargeBinaryType large\_binary\_type;

LargeStringType large\_string\_type;

FixedSizeBinaryType fixed\_size\_binary\_type(10);

auto heap\_fixed\_size\_binary\_type = fixed\_size\_binary(10);

Decimal128Type decimal128\_type(16, 5);

Decimal256Type decimal256\_type(42, 12);

auto heap\_decimal128\_type = decimal128(16, 5);

ListType list\_type(uint8());

LargeListType large\_list\_type(large\_utf8());

auto heap\_list\_type = list(uint8());

auto heap\_large\_list\_type = large\_list(large\_utf8());

FixedSizeListType fixed\_size\_list\_type(float64(), 3);

auto heap\_fixed\_size\_list\_type = fixed\_size\_list(float64(), 3);

DictionaryType dict\_type\_unordered(int16(), utf8());

DictionaryType dict\_type\_ordered(int16(), utf8(), /\*ordered=\*/true);

auto heap\_dict\_type = dictionary(int16(), utf8());

MapType map\_type\_unsorted(utf8(), binary());

MapType map\_type\_sorted(utf8(), binary(), /\*keys\_sorted=\*/true);

auto heap\_map\_type = map(utf8(), binary());

StructType struct\_type\_empty({});

StructType struct\_type(

{field("ints", int8()), field("strs", utf8(), /\*nullable=\*/false)});

auto heap\_struct\_type =

struct\_({field("ints", int8()), field("strs", utf8(), /\*nullable=\*/false)});

std::vector<int8\_t> union\_type\_codes({7, 42});

FieldVector union\_fields(

{field("ints", int8()), field("strs", utf8(), /\*nullable=\*/false)});

SparseUnionType sparse\_union\_type(union\_fields, union\_type\_codes);

DenseUnionType dense\_union\_type(union\_fields, union\_type\_codes);

UuidType uuid\_type{};

std::shared\_ptr<DataType> heap\_uuid\_type = std::make\_shared<UuidType>();

// Schema

auto schema\_empty = schema({});

auto schema\_non\_empty = schema({field("ints", int8()), field("strs", utf8())});

auto schema\_with\_metadata = schema\_non\_empty->WithMetadata(

key\_value\_metadata({"key1", "key2"}, {"value1", "value2"}));

// Fields

Field int\_field("ints", int64());

Field float\_field("floats", float32(), /\*nullable=\*/false);

auto heap\_int\_field = field("ints", int64());

// Scalars

NullScalar null\_scalar;

auto heap\_null\_scalar = MakeNullScalar(null());

BooleanScalar bool\_scalar\_null{};

BooleanScalar bool\_scalar{true};

auto heap\_bool\_scalar = \*MakeScalar(boolean(), true);

Int8Scalar int8\_scalar\_null{};

UInt8Scalar uint8\_scalar\_null{};

Int64Scalar int64\_scalar\_null{};

UInt64Scalar uint64\_scalar\_null{};

Int8Scalar int8\_scalar{-42};

UInt8Scalar uint8\_scalar{234};

Int64Scalar int64\_scalar{-9223372036854775807LL - 1};

UInt64Scalar uint64\_scalar{18446744073709551615ULL};

HalfFloatScalar half\_float\_scalar{48640}; // -1.5

FloatScalar float\_scalar{1.25f};

DoubleScalar double\_scalar{2.5};

Time32Scalar time\_scalar\_s{100, TimeUnit::SECOND};

Time32Scalar time\_scalar\_ms{1000, TimeUnit::MILLI};

Time64Scalar time\_scalar\_us{10000, TimeUnit::MICRO};

Time64Scalar time\_scalar\_ns{100000, TimeUnit::NANO};

Time64Scalar time\_scalar\_null{time64(TimeUnit::NANO)};

DurationScalar duration\_scalar\_s{-100, TimeUnit::SECOND};

DurationScalar duration\_scalar\_ms{-1000, TimeUnit::MILLI};

DurationScalar duration\_scalar\_us{-10000, TimeUnit::MICRO};

DurationScalar duration\_scalar\_ns{-100000, TimeUnit::NANO};

DurationScalar duration\_scalar\_null{duration(TimeUnit::NANO)};

TimestampScalar timestamp\_scalar\_s{12345, timestamp(TimeUnit::SECOND)};

TimestampScalar timestamp\_scalar\_ms{-123456, timestamp(TimeUnit::MILLI)};

TimestampScalar timestamp\_scalar\_us{1234567, timestamp(TimeUnit::MICRO)};

TimestampScalar timestamp\_scalar\_ns{-12345678, timestamp(TimeUnit::NANO)};

TimestampScalar timestamp\_scalar\_null{timestamp(TimeUnit::NANO)};

TimestampScalar timestamp\_scalar\_s\_tz{12345,

timestamp(TimeUnit::SECOND, "Europe/Paris")};

TimestampScalar timestamp\_scalar\_ms\_tz{-123456,

timestamp(TimeUnit::MILLI, "Europe/Paris")};

TimestampScalar timestamp\_scalar\_us\_tz{1234567,

timestamp(TimeUnit::MICRO, "Europe/Paris")};

TimestampScalar timestamp\_scalar\_ns\_tz{-12345678,

timestamp(TimeUnit::NANO, "Europe/Paris")};

TimestampScalar timestamp\_scalar\_null\_tz{timestamp(TimeUnit::NANO, "Europe/Paris")};

MonthIntervalScalar month\_interval\_scalar{23};

MonthIntervalScalar month\_interval\_scalar\_null{};

DayTimeIntervalScalar day\_time\_interval\_scalar{{23, -456}};

DayTimeIntervalScalar day\_time\_interval\_scalar\_null{};

MonthDayNanoIntervalScalar month\_day\_nano\_interval\_scalar{{1, 23, -456}};

MonthDayNanoIntervalScalar month\_day\_nano\_interval\_scalar\_null{};

Date32Scalar date32\_scalar{23};

Date32Scalar date32\_scalar\_null{};

Date64Scalar date64\_scalar{45 \* 86400000LL};

Date64Scalar date64\_scalar\_null{};

Decimal128Scalar decimal128\_scalar\_pos\_scale\_pos{Decimal128("1234567"),

decimal128(10, 4)};

Decimal128Scalar decimal128\_scalar\_pos\_scale\_neg{Decimal128("-1234567"),

decimal128(10, 4)};

Decimal128Scalar decimal128\_scalar\_neg\_scale\_pos{Decimal128("1234567"),

decimal128(10, -4)};

Decimal128Scalar decimal128\_scalar\_neg\_scale\_neg{Decimal128("-1234567"),

decimal128(10, -4)};

Decimal128Scalar decimal128\_scalar\_null{decimal128(10, 4)};

auto heap\_decimal128\_scalar = \*MakeScalar(decimal128(10, 4), Decimal128("1234567"));

Decimal256Scalar decimal256\_scalar\_pos\_scale\_pos{

Decimal256("1234567890123456789012345678901234567890123456"), decimal256(50, 4)};

Decimal256Scalar decimal256\_scalar\_pos\_scale\_neg{

Decimal256("-1234567890123456789012345678901234567890123456"), decimal256(50, 4)};

Decimal256Scalar decimal256\_scalar\_neg\_scale\_pos{

Decimal256("1234567890123456789012345678901234567890123456"), decimal256(50, -4)};

Decimal256Scalar decimal256\_scalar\_neg\_scale\_neg{

Decimal256("-1234567890123456789012345678901234567890123456"), decimal256(50, -4)};

Decimal256Scalar decimal256\_scalar\_null{decimal256(50, 4)};

auto heap\_decimal256\_scalar = \*MakeScalar(

decimal256(50, 4), Decimal256("1234567890123456789012345678901234567890123456"));

BinaryScalar binary\_scalar\_null{};

BinaryScalar binary\_scalar\_unallocated{std::shared\_ptr<Buffer>{nullptr}};

BinaryScalar binary\_scalar\_empty{Buffer::FromString("")};

BinaryScalar binary\_scalar\_abc{Buffer::FromString("abc")};

BinaryScalar binary\_scalar\_bytes{

Buffer::FromString(std::string() + '\x00' + "\x1f\xff")};

StringScalar string\_scalar\_null{};

StringScalar string\_scalar\_unallocated{std::shared\_ptr<Buffer>{nullptr}};

StringScalar string\_scalar\_empty{Buffer::FromString("")};

StringScalar string\_scalar\_hehe{Buffer::FromString("héhé")};

StringScalar string\_scalar\_invalid\_chars{

Buffer::FromString(std::string("abc") + '\x00' + "def\xffghi")};

LargeBinaryScalar large\_binary\_scalar\_abc{Buffer::FromString("abc")};

LargeStringScalar large\_string\_scalar\_hehe{Buffer::FromString("héhé")};

FixedSizeBinaryScalar fixed\_size\_binary\_scalar{Buffer::FromString("abc"),

fixed\_size\_binary(3)};

FixedSizeBinaryScalar fixed\_size\_binary\_scalar\_null{

Buffer::FromString(" "), fixed\_size\_binary(3), /\*is\_valid=\*/false};

std::shared\_ptr<Array> dict\_array;

dict\_array = \*ArrayFromJSON(utf8(), R"(["foo", "bar", "quux"])");

DictionaryScalar dict\_scalar{{std::make\_shared<Int8Scalar>(42), dict\_array},

dictionary(int8(), utf8())};

DictionaryScalar dict\_scalar\_null{dictionary(int8(), utf8())};

std::shared\_ptr<Array> list\_value\_array = \*ArrayFromJSON(int32(), R"([4, 5, 6])");

std::shared\_ptr<Array> list\_zero\_length = \*ArrayFromJSON(int32(), R"([])");

ListScalar list\_scalar{list\_value\_array};

ListScalar list\_scalar\_null{list\_zero\_length, list(int32()), /\*is\_valid=\*/false};

LargeListScalar large\_list\_scalar{list\_value\_array};

LargeListScalar large\_list\_scalar\_null{list\_zero\_length, large\_list(int32()),

/\*is\_valid=\*/false};

FixedSizeListScalar fixed\_size\_list\_scalar{list\_value\_array};

FixedSizeListScalar fixed\_size\_list\_scalar\_null{

list\_value\_array, fixed\_size\_list(int32(), 3), /\*is\_valid=\*/false};

auto struct\_scalar\_type = struct\_({field("ints", int32()), field("strs", utf8())});

StructScalar struct\_scalar{

ScalarVector{MakeScalar(int32\_t(42)), MakeScalar("some text")}, struct\_scalar\_type};

StructScalar struct\_scalar\_null{struct\_scalar.value, struct\_scalar\_type,

/\*is\_valid=\*/false};

auto sparse\_union\_scalar\_type =

sparse\_union(FieldVector{field("ints", int32()), field("strs", utf8())}, {7, 42});

auto dense\_union\_scalar\_type =

dense\_union(FieldVector{field("ints", int32()), field("strs", utf8())}, {7, 42});

std::vector<std::shared\_ptr<Scalar>> union\_values = {MakeScalar(int32\_t(43)),

MakeNullScalar(utf8())};

SparseUnionScalar sparse\_union\_scalar{union\_values, 7, sparse\_union\_scalar\_type};

DenseUnionScalar dense\_union\_scalar{union\_values[0], 7, dense\_union\_scalar\_type};

union\_values[0] = MakeNullScalar(int32());

SparseUnionScalar sparse\_union\_scalar\_null{union\_values, 7, sparse\_union\_scalar\_type};

DenseUnionScalar dense\_union\_scalar\_null{union\_values[0], 7, dense\_union\_scalar\_type};

auto extension\_scalar\_type = std::make\_shared<UuidType>();

ExtensionScalar extension\_scalar{

std::make\_shared<FixedSizeBinaryScalar>(Buffer::FromString("0123456789abcdef"),

extension\_scalar\_type->storage\_type()),

extension\_scalar\_type};

ExtensionScalar extension\_scalar\_null{extension\_scalar.value, extension\_scalar\_type,

/\*is\_valid=\*/false};

std::shared\_ptr<Scalar> heap\_map\_scalar;

ARROW\_CHECK\_OK(

ScalarFromJSON(map(utf8(), int32()), R"([["a", 5], ["b", 6]])", &heap\_map\_scalar));

auto heap\_map\_scalar\_null = MakeNullScalar(heap\_map\_scalar->type);

// Array and ArrayData

auto heap\_null\_array = SliceArrayFromJSON(null(), "[null, null]");

auto heap\_int32\_array = SliceArrayFromJSON(int32(), "[-5, 6, null, 42]");

ArrayData int32\_array\_data{\*heap\_int32\_array->data()};

Int32Array int32\_array{heap\_int32\_array->data()->Copy()};

auto heap\_int32\_array\_no\_nulls = SliceArrayFromJSON(int32(), "[-5, 6, 3, 42]");

const char\* json\_int32\_array = "[-1, 2, -3, 4, null, -5, 6, -7, 8, null, -9, -10]";

auto heap\_int32\_array\_sliced\_1\_9 = SliceArrayFromJSON(int32(), json\_int32\_array, 1, 9);

auto heap\_int32\_array\_sliced\_2\_6 = SliceArrayFromJSON(int32(), json\_int32\_array, 2, 6);

auto heap\_int32\_array\_sliced\_8\_4 = SliceArrayFromJSON(int32(), json\_int32\_array, 8, 4);

auto heap\_int32\_array\_sliced\_empty =

SliceArrayFromJSON(int32(), json\_int32\_array, 6, 0);

const char\* json\_bool\_array =

"[false, false, true, true, null, null, false, false, true, true, "

"null, null, false, false, true, true, null, null]";

auto heap\_bool\_array = SliceArrayFromJSON(boolean(), json\_bool\_array);

auto heap\_bool\_array\_sliced\_1\_9 = SliceArrayFromJSON(boolean(), json\_bool\_array, 1, 9);

auto heap\_bool\_array\_sliced\_2\_6 = SliceArrayFromJSON(boolean(), json\_bool\_array, 2, 6);

auto heap\_bool\_array\_sliced\_empty =

SliceArrayFromJSON(boolean(), json\_bool\_array, 6, 0);

auto heap\_list\_array = SliceArrayFromJSON(list(int64()), "[[1, 2], null, []]");

ListArray list\_array{heap\_list\_array->data()};

const char\* json\_double\_array = "[-1.5, null]";

auto heap\_double\_array = SliceArrayFromJSON(float64(), json\_double\_array);

const char\* json\_float16\_array = "[0, 48640]";

auto heap\_float16\_array =

\*SliceArrayFromJSON(uint16(), json\_float16\_array)->View(float16());

auto heap\_date32\_array =

SliceArrayFromJSON(date32(), "[0, null, 18336, -9004, -719162, -719163]");

auto heap\_date64\_array = SliceArrayFromJSON(

date64(), "[1584230400000, -777945600000, -62135596800000, -62135683200000, 123]");

const char\* json\_time\_array = "[null, -123, 456]";

auto heap\_time32\_array\_s =

SliceArrayFromJSON(time32(TimeUnit::SECOND), json\_time\_array);

auto heap\_time32\_array\_ms =

SliceArrayFromJSON(time32(TimeUnit::MILLI), json\_time\_array);

auto heap\_time64\_array\_us =

SliceArrayFromJSON(time64(TimeUnit::MICRO), json\_time\_array);

auto heap\_time64\_array\_ns = SliceArrayFromJSON(time64(TimeUnit::NANO), json\_time\_array);

auto heap\_month\_interval\_array =

SliceArrayFromJSON(month\_interval(), "[123, -456, null]");

auto heap\_day\_time\_interval\_array =

SliceArrayFromJSON(day\_time\_interval(), "[[1, -600], null]");

auto heap\_month\_day\_nano\_interval\_array =

SliceArrayFromJSON(month\_day\_nano\_interval(), "[[1, -600, 5000], null]");

const char\* json\_duration\_array = "[null, -1234567890123456789]";

auto heap\_duration\_array\_s =

SliceArrayFromJSON(duration(TimeUnit::SECOND), json\_duration\_array);

auto heap\_duration\_array\_ns =

SliceArrayFromJSON(duration(TimeUnit::NANO), json\_duration\_array);

auto heap\_timestamp\_array\_s = SliceArrayFromJSON(

timestamp(TimeUnit::SECOND),

R"([null, "1970-01-01 00:00:00", "1900-02-28 12:34:56", "3989-07-14 00:00:00"])");

auto heap\_timestamp\_array\_ms = SliceArrayFromJSON(

timestamp(TimeUnit::MILLI),

R"([null, "1900-02-28 12:34:56.123", "3989-07-14 00:00:00.789"])");

auto heap\_timestamp\_array\_us = SliceArrayFromJSON(

timestamp(TimeUnit::MICRO),

R"([null, "1900-02-28 12:34:56.654321", "3989-07-14 00:00:00.456789"])");

auto heap\_timestamp\_array\_ns = SliceArrayFromJSON(

timestamp(TimeUnit::NANO), R"([null, "1900-02-28 12:34:56.987654321"])");

auto heap\_decimal128\_array = SliceArrayFromJSON(

decimal128(30, 6),

R"([null, "-1234567890123456789.012345", "1234567890123456789.012345"])");

auto heap\_decimal256\_array = SliceArrayFromJSON(

decimal256(50, 6), R"([null, "-123456789012345678901234567890123456789.012345"])");

auto heap\_decimal128\_array\_sliced = heap\_decimal128\_array->Slice(1, 1);

auto heap\_fixed\_size\_binary\_array =

SliceArrayFromJSON(fixed\_size\_binary(3), "[null, \"abc\", \"\\u0000\\u001f\xff\"]");

auto heap\_fixed\_size\_binary\_array\_zero\_width =

SliceArrayFromJSON(fixed\_size\_binary(0), R"([null, ""])");

auto heap\_fixed\_size\_binary\_array\_sliced = heap\_fixed\_size\_binary\_array->Slice(1, 1);

const char\* json\_binary\_array = "[null, \"abcd\", \"\\u0000\\u001f\xff\"]";

auto heap\_binary\_array = SliceArrayFromJSON(binary(), json\_binary\_array);

auto heap\_large\_binary\_array = SliceArrayFromJSON(large\_binary(), json\_binary\_array);

const char\* json\_string\_array = "[null, \"héhé\", \"invalid \xff char\"]";

auto heap\_string\_array = SliceArrayFromJSON(utf8(), json\_string\_array);

auto heap\_large\_string\_array = SliceArrayFromJSON(large\_utf8(), json\_string\_array);

auto heap\_binary\_array\_sliced = heap\_binary\_array->Slice(1, 1);

// ChunkedArray

ArrayVector array\_chunks(2);

array\_chunks[0] = \*ArrayFromJSON(int32(), "[1, 2]");

array\_chunks[1] = \*ArrayFromJSON(int32(), "[3, null, 4]");

ChunkedArray chunked\_array{array\_chunks};

// RecordBatch

auto batch\_schema = schema({field("ints", int32()), field("strs", utf8())});

ArrayVector batch\_columns{2};

batch\_columns[0] = \*ArrayFromJSON(int32(), "[1, 2, 3]");

batch\_columns[1] = \*ArrayFromJSON(utf8(), R"(["abc", null, "def"])");

auto batch = RecordBatch::Make(batch\_schema, /\*num\_rows=\*/3, batch\_columns);

auto batch\_with\_metadata = batch->ReplaceSchemaMetadata(

key\_value\_metadata({"key1", "key2", "key3"}, {"value1", "value2", "value3"}));

// Table

ChunkedArrayVector table\_columns{2};

ARROW\_CHECK\_OK(

ChunkedArrayFromJSON(int32(), {"[1, 2, 3]", "[4, 5]"}, &table\_columns[0]));

ARROW\_CHECK\_OK(ChunkedArrayFromJSON(

utf8(), {R"(["abc", null])", R"(["def"])", R"(["ghi", "jkl"])"},

&table\_columns[1]));

auto table = Table::Make(batch\_schema, table\_columns);

// Datum

Datum empty\_datum{};

Datum scalar\_datum{MakeNullScalar(boolean())};

Datum array\_datum{heap\_int32\_array};

Datum chunked\_array\_datum{chunked\_array};

Datum batch\_datum{batch};

Datum table\_datum{table};

#ifdef \_\_clang\_\_

\_Pragma("clang diagnostic pop");

#elif defined(\_\_GNUC\_\_)

\_Pragma("GCC diagnostic pop");

#endif

// Hook into debugger

::arrow::internal::DebugTrap();

}

} // namespace gdb

} // namespace arrow