BankApp

1.0.0

Generated by Doxygen 1.9.6

1 Real-time Vamp plugin SDK for C++20 **1**

2 Class Index **5**

2.1 Class List 5

3 File Index **7**

3.1 File List 7

4 Class Documentation **9**

4.1 Bank Class Reference 9

4.2 BankAccount Class Reference 10

4.2.1 Member Function Documentation 10

4.2.1.1 deposit() 11

4.2.1.2 displayAccountInfo() 12

4.2.1.3 getAccountBalance() 12

4.2.1.4 getAccountHolder() 12

4.2.1.5 getAccountNumber() 13

4.2.1.6 getAccountType() 13

4.2.1.7 withdraw() 13

5 File Documentation **15**

5.1 Bank.hpp 15

5.2 BankAccount.hpp 15

Chapter 1

Real-time Vamp plugin SDK for C++20

Vamp is an C/C++ plugin API for audio analysis / feature extraction plugins: `https://www.vamp-plugins.org`

This SDK for plugins and hosts targets performance-critical applications by:

- reducing memory allocations, **no memory allocation** during processing
- simplifying and restricting the plugin API
- `constexpr` evaluation for compile time errors instead of runtime errors

The SDK aims to be **well tested**, **cross-platform** and use **modern C++**. The plugin SDK is available as a `single-header library` (download as asset from `latest release` page).

Compiler support: `GCC >= 10`, `Clang >= 11`, `MSVC >= 19.30`

> **Note**: Python bindings for the hostsdk are available via `PyPI`. Please check out the `documentation`.

Links

- `API documentation`
- `Examples`
- Vamp ecosystem:
  - `Great collection of plugins`
  - `Sonic Visualiser`: Open-source software to visualize, analyze and annotate audio
  - `Sonic Annotator`: Batch tool for feature extraction
  - `Audacity supports Vamp plugins`

Performance

Following benchmarks compare the performance/overhead of the plugin SDKs based on a simple `RMS plugin`. The performance is measured as throughput (number of processed samples per second).

Results with an i7-9850H CPU (12 cores):

Throughput vs. block size	Multithreading

Results with an ARMv7 CPU: Throughput vs block size, Multithreading

Why another SDK?

The official SDK offers a convenient C++ plugin interface. But there are some drawbacks for real-time processing:

- Huge amount of memory allocations due to the use of C++ containers like vectors and lists **passed by value**.

  Let's have a look at the `process` method of the `Vamp::Plugin` class which does the main work:

  `FeatureSet process(const float *const *inputBuffers, RealTime timestamp)`

  `FeatureSet` is returned by value and is a `std::map<int, FeatureList>`. `FeatureList` is a `std::vector<Feature>` and `Feature` is `struct` containing the actual feature values as a `std::vector<float>`. So in total, those are three nested containers, which are all heap allocated.

- The C++ API is a wrapper of the C API:

  On the plugin side, the `PluginAdapter` class converts the C++ containers to C level (code). Therefore the C++ containers are temporary objects and will be deallocated shortly after creation.

  On the host side, the `PluginHostAdapter` converts again from the C to the C++ representation (code).

Solution approach

The `rt-vamp-plugin-sdk` aims to to keep the overhead minimal but still provide an easy and safe to use API:

1. Static plugin informations are provided as `static constexpr` variables to generate the C plugin descriptor at compile time.
2. The computed features are returned by reference (as a `std::span`) to prevent heap allocations during processing.
3. The input buffer is provided either as a `TimeDomainBuffer` (`std::span<const float>`) or a `FrequencyDomainBuffer` (`std::span<const std::complex<float>>`). The process method takes a `std::variant<TimeDomainBuffer, FrequencyDomainBuffer>`. A wrong input buffer type will result in an exception. The sized spans enable easy iteration over the input buffer data.

Plugin restrictions

Following features of the Vamp API `Vamp::Plugin` are restricted within the `rt-vamp-plugin-sdk`:

- `OutputDescriptor::hasFixedBinCount == true` for every output. The number of values is constant for each feature during processing. This has the advantage, that memory for the feature vector can be preallocated.
- `OutputDescriptor::SampleType == OneSamplePerStep` for every output. The plugin will generate one feature set for each input block.

  Following parameters are therefore negitable:

  - `OutputDescriptor::sampleRate`
  - `OutputDescriptor::hasDuration`
  - `Feature::hasTimestamp` & `Feature::timestamp`
  - `Feature::hasDuration` & `Feature::duration`
- Only one input channel allowed: `getMinChannelCount() == 1`

Minimal example

More examples can be found here: https://github.com/lukasberbuer/rt-vamp-plugin-sdk/tree/master/

Plugin

```
class ZeroCrossing : public rtvamp::pluginsdk::Plugin<1 /* one output */> {
public:
    using Plugin::Plugin;  // inherit constructor

    static constexpr Meta meta{
        .identifier    = "zerocrossing",
        .name          = "Zero crossings",
        .description   = "Detect and count zero crossings",
        .maker         = "LB",
        .copyright     = "MIT",
        .pluginVersion = 1,
        .inputDomain   = InputDomain::Time,
    };

    OutputList getOutputDescriptors() const override {
        return {
            OutputDescriptor{
                .identifier  = "counts",
                .name        = "Zero crossing counts",
                .description = "The number of zero crossing points per processing block",
                .unit        = "",
                .binCount    = 1,
            },
        };
    }

    bool initialise(uint32_t stepSize, uint32_t blockSize) override {
        initialiseFeatureSet();  // automatically resizes feature set to number of outputs and bins
        return true;
    }

    void reset() override {
        previousSample_ = 0.0f;
    }

    const FeatureSet& process(InputBuffer buffer, uint64_t nsec) override {
        const auto signal = std::get<TimeDomainBuffer>(buffer);

        size_t crossings   = 0;
        bool   wasPositive = (previousSample_ >= 0.0f);

        for (const auto& sample : signal) {
            const bool isPositive = (sample >= 0.0f);
            crossings += int(isPositive != wasPositive);
            wasPositive = isPositive;
        }

        previousSample_ = signal.back();

        auto& result = getFeatureSet();
        result[0][0] = crossings;  // first and only output, first and only bin
        return result;             // return span/view of the results
    }

private:
    float previousSample_ = 0.0f;
};

RTVAMP_ENTRY_POINT(ZeroCrossing)
```

Host

```
// list all plugins keys (library:plugin)
for (auto&& key : rtvamp::hostsdk::listPlugins()) {
    std::cout « key.get() « std::endl;
}

auto plugin = rtvamp::hostsdk::loadPlugin("minimal-plugin:zerocrossing", 48000 /* samplerate */);
plugin->initialise(4096 /* step size */, 4096 /* block size */);

std::vector<float> buffer(4096);
// fill buffer with data from audio file, sound card, ...

auto features = plugin->process(buffer, 0 /* timestamp nanoseconds */);
std::cout « "Zero crossings: " « features[0][0] « std::endl;
```

Chapter 2

Class Index

2.1 Class List

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

Bank 9
BankAccount 10

Chapter 3

File Index

3.1 File List

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

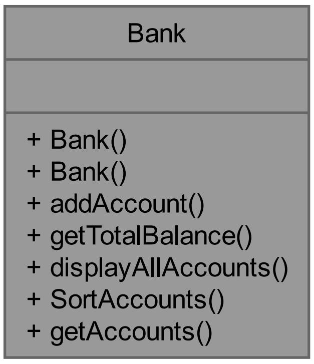
Bank.hpp 15
BankAccount.hpp 15

Chapter 4

Class Documentation

4.1 Bank Class Reference

Collaboration diagram for Bank:



Public Member Functions

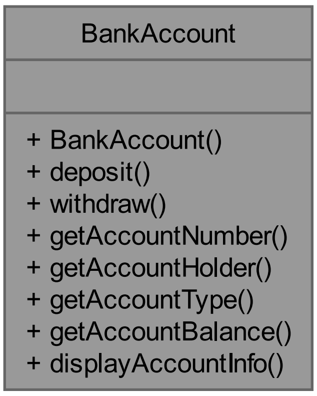
- **Bank** (Bank &&obj)
- void **addAccount** (const BankAccount &account)
- double **getTotalBalance** () const
- void **displayAllAccounts** () const
- void **SortAccounts** ()
- std::vector< BankAccount > **getAccounts** () const

The documentation for this class was generated from the following files:

- Bank.hpp
- Bank.cpp

4.2 BankAccount Class Reference

Collaboration diagram for BankAccount:



Public Member Functions

- **BankAccount** (int accountNumber, const std::string &accountHolder, const std::string &accountType, double accountBalance)
- void deposit (double amount)

  Deposits the specified amount into the bank account.
- bool withdraw (double amount)

  Withdraws the specified amount from the bank account.
- int getAccountNumber () const

  Get the account number.
- const std::string & getAccountHolder () const

  The std::string type represents a sequence of characters.
- const std::string & getAccountType () const

  Returns the account type.
- double getAccountBalance () const

  Get the current balance of the bank account.
- void displayAccountInfo () const

  Displays the account information.

4.2.1 Member Function Documentation

4.2.1.1 deposit()

```
void BankAccount::deposit (
             double amount )
```

Deposits the specified amount into the bank account.

This function adds the specified amount to the current account balance. If the amount is greater than zero, the deposit is considered successful. Otherwise, an error message is displayed.

Parameters

amount	The amount to be deposited.

4.2.1.2 displayAccountInfo()

```
void BankAccount::displayAccountInfo ( ) const
```

Displays the account information.

This function prints the account number, account holder, account type, and account balance to the standard output.

4.2.1.3 getAccountBalance()

```
double BankAccount::getAccountBalance ( ) const
```

Get the current balance of the bank account.

This function returns the current balance of the bank account.

Returns

The current balance of the bank account.

4.2.1.4 getAccountHolder()

```
const std::string & BankAccount::getAccountHolder ( ) const
```

The std::string type represents a sequence of characters.

It is a standard library class that provides a convenient way to manipulate strings in C++.

See also

https://en.cppreference.com/w/cpp/string/basic_string

4.2.1.5 getAccountNumber()

```
int BankAccount::getAccountNumber ( ) const
```

Get the account number.

This function returns the account number associated with the bank account.

Returns

The account number.

4.2.1.6 getAccountType()

```
const std::string & BankAccount::getAccountType ( ) const
```

Returns the account type.

This function returns the account type as a constant reference to a string.

Returns

A constant reference to the account type string.

4.2.1.7 withdraw()

```
bool BankAccount::withdraw (
             double amount )
```

Withdraws the specified amount from the bank account.

This function allows the user to withdraw a specified amount from the bank account. If the withdrawal amount is greater than the account balance, an error message is displayed.

Parameters

amount	The amount to be withdrawn from the account.

Returns

True if the withdrawal is successful, false otherwise.

The documentation for this class was generated from the following files:

- BankAccount.hpp
- BankAccount.cpp

Chapter 5

File Documentation

5.1 Bank.hpp

```
#ifndef BANK_HPP
#define BANK_HPP

#include "BankAccount.hpp"
#include <vector>
#include <algorithm>

class Bank
{
private:
    std::vector<BankAccount> accounts;

public:
    // Default constructor
    Bank();

    // Move constructor
    Bank(Bank&& obj);

    void   addAccount(const BankAccount& account);
    double getTotalBalance() const;
    void   displayAllAccounts() const;

    void SortAccounts();

    // getter for accounts
    std::vector<BankAccount> getAccounts() const;
};

#endif // BANK_HPP
```

5.2 BankAccount.hpp

```
#ifndef BANKACCOUNT_HPP
#define BANKACCOUNT_HPP

#include <string>
#include <iostream>
class BankAccount
{
private:
    int         accountNumber;
    std::string accountHolder;
    std::string accountType;
    double      accountBalance{};

public:
    // Default constructor
    BankAccount(int accountNumber, const std::string& accountHolder, const std::string& accountType,
                double accountBalance);

    void               deposit(double amount);
    bool               withdraw(double amount);
    int                getAccountNumber() const;
    const std::string& getAccountHolder() const;
    const std::string& getAccountType() const;
    double             getAccountBalance() const;
    void               displayAccountInfo() const;
};

#endif // BANKACCOUNT_HPP
```