C++23
スタックトレースライブラリ

高橋 晶 (Akira Takahashi)
faithandbrave@gmail.com
Preferred Networks, Inc.
2024/06/14 (金) C++ MIX #11

C++23 スタックトレースのライブラリ

- スタックトレースは、デバッグのための機能です
- エラーの場所がどういう経路を辿ってそこに至ったのかを取得できます (引数はとれない)
- スタックトレースはプラットフォームに取得できる方法が提供されていましたが、標準ライブラリに入ったことで共通化されました
- 今回は、このライブラリの使い方と定義を解説していきます

基本的な使い方

```cpp
#include <print>
#include <stacktrace>

void g() {
  std::println("{}", std::stacktrace::current());
}

void f() {
  g();
}

int main() {
  f();
}
```

```
   0#  g() at /app/example.cpp:5
   1#  f() at /app/example.cpp:9
   2# main at /app/example.cpp:13
   3#       at :0
   4# __libc_start_main at :0
   5# _start at :0
   6#
```

この1行を考える 1/4

```
std::println("{}", std::stacktrace::current());
```

- std::basic_stacktraceというクラスがスタックトレースを取得するためのクラス
- その別名としてstd::stacktraceが定義される

```
namespace std {
  template <class Allocator>
  class basic_stacktrace;

  using stacktrace = basic_stacktrace<allocator<stacktrace_entry>>;
}
```

この1行を考える 2/4

```
std::println("{}", std::stacktrace::current());
```

- std::basic_stacktraceクラスのstaticメンバ関数current()でスタックトレースを取得する
- skip / max_depthについてはあとで説明

```
static basic_stacktrace current();
static basic_stacktrace current(size_type skip);
static basic_stacktrace current(size_type skip, size_type max_depth);
```

この1行を考える 3/4

```
std::println("{}", std::stacktrace::current());
```

- std::basic_stacktraceクラスは文字列への変換機能がある
  - std::format()用のformatter特殊化
  - 出力ストリーム用のoperator<<オーバーロード
  - to_string()メンバ関数

```
cout << std::stacktrace::current() << endl;
std::string trace = std::stacktrace::current().to_string();
```

この1行を考える 4/4

```
std::println("{}", std::stacktrace::current());
```

```
0#  g() at /app/example.cpp:5
1#  f() at /app/example.cpp:9
2# main at /app/example.cpp:13
3#       at :0
4# __libc_start_main at :0
5# _start at :0
6#
```

- std::basic_stacktraceクラスはstd::stacktrace_entryクラスの配列
- operator[ ] / at()、size()メンバ関数やbegin() / end()を使って各行の情報にアクセスできる
- 取得できる情報は以下：
  - source_file()：ソースファイル名 (パス)
  - source_line()：行番号
  - description()：説明 (関数名…function_name()はない)
  - operator bool()：空でないか判定
- スタックトレースの各行の番号はstd::basic_stacktrace側でつけている

主な利用方法

```
template <class E>
void throw_exception(string_view error) {
  println(stderr, "{}", stacktrace::current(1));
  throw E{string{error}};
}

void g() {
  throw_exception<invalid_argument>("error");
}
```

- スタックトレースを出力してから例外を投げる
- current(1)とすることで自身の関数を除いたスタックトレースにできる
- エラーを出力する関数自体の情報はだいたいいらない
  エラー処理を開始する場所が最初にほしい

エラー内容のちがい

current()の場合

```
   0# void
throw_exception<invalid_argument>(str
ing_view) at /app/example.cpp:7
   1#  g() at /app/example.cpp:12
   2#  f() at /app/example.cpp:16
   3# main at /app/example.cpp:20
   4#       at :0
   5# __libc_start_main at :0
   6# _start at :0
   7#

terminate called after throwing an
instance of 'std::invalid_argument'
  what():  error
```

current(1)の場合

```
   0#  g() at /app/example.cpp:12
   1#  f() at /app/example.cpp:16
   2# main at /app/example.cpp:20
   3#       at :0
   4# __libc_start_main at :0
   5# _start at :0
   6#

terminate called after throwing an
instance of 'std::invalid_argument'
  what():  error
```

補足

- Visual Studioのようなデバッガ付き環境では、デバッグ時にスタックトレースが表示されます
- GCCやClangでデバッガをつなぐのがめんどくさい環境では、スタックトレースを出力してしまうのがラクです
- エラーログにエラー情報を出力しておくことで、問題の調査がしやすくなります