Breakpad C++ Cheatsheet

1. Setting Up Breakpad in Your Application

Include Required Headers

```
#include "client/linux/handler/exception_handler.h"
```

Initialize Breakpad Exception Handler

```
bool DumpCallback(const google_breakpad::MinidumpDescriptor& descriptor,
                  void* context,
                  bool succeeded) {
    printf("Dump path: %s\n", descriptor.path());
    return succeeded;
}

int main() {
    google_breakpad::MinidumpDescriptor descriptor("/tmp/dumps");
    google_breakpad::ExceptionHandler eh(descriptor, nullptr, DumpCallback, nullptr, true, -1);

    // Application code here...

    return 0;
}
```

---

2. Generating Symbol Files

```
./dump_syms my_application_binary > my_application.sym
```

- Store symbols in a structured directory:

  ```
  mkdir -p symbols/my_application/BUILD_ID
  mv my_application.sym symbols/my_application/BUILD_ID/
  ```

---

3. Simulating a Crash

```
void causeCrash() {
    volatile int* nullPointer = nullptr;
    *nullPointer = 42; // Intentional crash
}
```

```
int main() {
    causeCrash();
    return 0;
}
```

---

4. Processing Minidumps

```
./minidump_stackwalk /tmp/dumps/minidump.dmp ./symbols > crash_report.txt
```

---

5. Using Custom Crash Reports

```
#include "client/linux/handler/exception_handler.h"
#include <fstream>

bool CustomDumpCallback(const google_breakpad::MinidumpDescriptor& descriptor,
                        void* context,
                        bool succeeded) {
    std::ofstream log("/tmp/crash_log.txt");
    log << "Crash report saved to: " << descriptor.path() << std::endl;
    return succeeded;
}

int main() {
    google_breakpad::MinidumpDescriptor descriptor("/tmp/dumps");
    google_breakpad::ExceptionHandler eh(descriptor, nullptr, CustomDumpCallback, nullptr,
true, -1);

    causeCrash();
    return 0;
}
```

---

6. **Debugging with **``

```
gdb ./my_application core
```

Breakpad C++ Enhanced Cheatsheet {#breakpad-c-enhanced-cheatsheet }

1. Setting Up Breakpad in Your Application

Include Required Headers

Ensure you include the appropriate Breakpad headers in your project. For Linux systems:

```
#include "client/linux/handler/exception_handler.h"
```

For Windows systems:

```
#include "client/windows/handler/exception_handler.h"
```

Initialize the Exception Handler

Initialize the Breakpad exception handler at the start of your `main` function to capture crashes effectively.

Linux Example:

```
#include "client/linux/handler/exception_handler.h"

bool DumpCallback(const google_breakpad::MinidumpDescriptor& descriptor,
                  void* context,
                  bool succeeded) {
    printf("Dump path: %s\n", descriptor.path());
    return succeeded;
}

int main() {
    google_breakpad::MinidumpDescriptor descriptor("/tmp");
    google_breakpad::ExceptionHandler eh(descriptor, nullptr, DumpCallback, nullptr, true, -1);
    return 0;
}
```

Windows Example:

```
#include "client/windows/handler/exception_handler.h"

bool DumpCallback(const wchar_t* dump_path,
                  const wchar_t* minidump_id,
                  void* context,
                   EXCEPTION_POINTERS* exinfo,
                   MDRawAssertionInfo* assertion,
                   bool succeeded) {
    wprintf(L"Dump path: %s\\%s.dmp\n", dump_path, minidump_id);
    return succeeded;
}

int main() {
    google_breakpad::ExceptionHandler eh(L"C:\\temp", nullptr, DumpCallback, nullptr, true);
    return 0;
}
```

2. Generating Symbol Files

Symbol files are essential for translating memory addresses in minidump files into human-readable function names and line numbers.

Using `dump_syms`:

1. Build the `dump_syms` tool from the Breakpad source.
2. Generate the symbol file:

   ```
   ./dump_syms your_application_binary > your_application.sym
   ```

3. Organize the symbol files into a structured directory:

   ```
   mkdir -p symbols/your_application/BUILD_ID
   mv your_application.sym symbols/your_application/BUILD_ID/
   ```

   Replace `BUILD_ID` with the actual build identifier of your binary.

3. Simulating a Crash

To test your Breakpad integration, you can simulate a crash in your application.

```
void CauseCrash() {
    volatile int* ptr = nullptr;
    *ptr = 42;  // This will cause a segmentation fault
}

int main() {
    CauseCrash();
    return 0;
}
```

4. Processing Minidump Files

After a crash, Breakpad generates a minidump file. To analyze this file:

1. Use the `minidump_stackwalk` tool:

```
./minidump_stackwalk /tmp/minidump.dmp ./symbols > crash_report.txt
```

Ensure that the `./symbols` directory contains the correct symbol files corresponding to your application binary.

5. Handling Unhandled Exceptions

On Windows, certain exceptions might not be caught by Breakpad due to the Visual C++ runtime library resetting custom exception handlers in specific scenarios, such as buffer overflows. To address this, you can use the following workaround:

```cpp
#include <windows.h>

LONG WINAPI UnhandledExceptionFilter(EXCEPTION_POINTERS* ExceptionInfo) {
    // Custom handling code here...
    return EXCEPTION_EXECUTE_HANDLER;
}

int main() {
    SetUnhandledExceptionFilter(UnhandledExceptionFilter);
    return 0;
}
```

This approach helps in capturing exceptions that Breakpad might miss due to the runtime library's behavior.

6. Additional Resources

- **Breakpad GitHub Repository:** github.com/google/breakpad
- **Getting Started Guide:** Breakpad Docs

By following this enhanced cheatsheet, you can effectively integrate and utilize Breakpad in your C++ applications, ensuring robust crash reporting and analysis.