C++ Signal Handling

- Signals are the interrupts which are delivered to a process by the operating system to stop its ongoing task and attend the task for which the interrupt has been generated.
- Signals can also be generated by the operating system on the basis of system or error condition.
- You can generate interrupts by pressing Ctrl+ C on Linux, UNIX, Mac OS X, or Windows system.

There are signals which cannot be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal.

These signals are defined in <csingnal> header file.

Here are the list of signals along with their description and working capability:

Signals	Description
SIGABRT	(Signal Abort) Abnormal termination of the program, such as a call to abort.
SIGFPE	(Signal floating- point exception) An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	(Signal Illegal Instruction) It is used for detecting an illegal instruction.
SIGINT	(Signal Interrupt) It is used to receipt an interactive program interrupt signal.
SIGSEGV	(Signal segmentation Violation) An invalid access to storage.
SIGTERM	(Signal Termination) A termination request sent to the program.
SIGHUP	(Signal Hang up) Hang Up (POSIX), its report that user's terminal is disconnected. It is used to report the termination of the controlling process.
SIGQUIT	Used to terminate a process and generate a core dump.
SIGTRAP	Trace trap.
SIGBUS	This is a BUS error which indicates an access to an invalid address.
SIGUSR1	User defined signal 1.
SIGUSR2	User defined signal 2.

SIGALRM	Alarm clock, which indicates an access to an invalid address.
SIGTERM	Used for termination. This signal can be blocked, handled, and ignored. Generated by kill command.
SIGCOUNT	This signal sent to process to make it continue.
SIGSTOP	Stop, unblockable. This signal is used to stop a process. This signal cannot be handled, ignored or blocked.

The signal() Function

C++ signal-handling library provides function signal to trap unexpected interrupts or events.

Syntax

```
void (*signal (int sig, void (*func)(int)))(int);
```

Parameters

This function is set to handle the signal.

It specifies a way to handle the signals number specified by ***sig***.

Parameter ***func*** specifies one of the three ways in which a signal can be handled by a program.

- **Default handling (SIG_DFL):** The signal handled by the default action for that particular signal.
- **Ignore Signal (SIG_IGN):** The signal is ignored and the code execution will continue even if not purposeful.
- **Function handler:** A particular function is defined to handle the signal.

We must keep in mind that the signal that we would like to catch must be registered using a signal function and it must be associated with a signal handling function.

> Note: The signal handling function should be of the void type.

Return value

The return type of this function is the same as the type of parameter func.

If the request of this function is successful, the function returns a pointer to the particular handler function which was in charge of handling this signal before the call, if any.

Data Races

Data race is undefined. If you call this function in a multi- threaded program then it will cause undefined behavior.

Exceptions

This function never throws exception.

Example 1

Let's see a simple example to demonstrate the use of signal() function:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t signalled = 0;

void handler(int sig)
{
   signalled = 1;
}

int main()
{
   signal(SIGINT, handler);

   raise(SIGINT);
   if (signalled)
      cout << "Signal is handled";
   else
```

```
        cout << "Signal is not handled";

    return 0;
}
```

Output:

```
Signal is handled
```

Example 2

Let's see another simple example:

```
#include <csignal>
#include <iostream>

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

Output:

```
SignalValue: 0
Sending signal 2
SignalValue: 2
```

The raise() Function

The C++ signal raise() function is used to send signals to the current executing program.

<csignal> header file declared the function raise() to handle a particular signal.

Syntax

```
int raise (int sig);
```

Parameters

sig: The signal number to be sent for handling. It can take one of the following values:

- SIGINT
- SIGABRT
- SIGFPE
- SIGILL
- SIGSEGV
- SIGTERM
- SIGHUP

Return value

On success, it returns 0 and on failure, a non-zero is returned.

Data Races

Concurrently calling this function is safe, causing no data races.

Exceptions

This function never throws exceptions, if no function handlers have been defined with signal to handle the raised signal.

Example 1

Let's see a simple example to illustrate the use of raise() function when SIGABRT is passed:

```
#include <iostream>
#include <csignal>

using namespace std;

sig_atomic_t sig_value = 0;

void handler(int sig)
{
   sig_value = sig;
}

int main()
{
   signal(SIGABRT, handler);
   cout << "Before signal handler is called" << endl;
   cout << "Signal = " << sig_value << endl;
   raise(SIGABRT);
   cout << "After signal handler is called" << endl;
   cout << "Signal = " << sig_value << endl;

   return 0;
}
```

Output:

```
Before signal handler is called
Signal = 0
After signal handler is called
Signal = 6
```

Example 2

Let's see a simple example to illustrate the use of raise() function when SIGINT is passed:

```
#include <csignal>
#include <iostream>
using namespace std;

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
{
    signal(SIGINT, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGINT);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 2
```

Example 3

Let's see a simple example to illustrate the use of raise() function when SIGTERM is passed:

```
#include <csignal>
#include <iostream>
using namespace std;

sig_atomic_t s_value = 0;
void handle(int signal_)
{
   s_value = signal_;
}

int main()
{
   signal(SIGTERM, handle);
   cout << "Before called Signal = " << s_value << endl;
   raise(SIGTERM);
   cout << "After called Signal = " << s_value << endl;
   return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 15
```

Example 4

Let's see a simple example to illustrate the use of raise() function when SIGSEGV is passed:

```
#include <csignal>
#include <iostream>
using namespace std;

sig_atomic_t s_value = 0;
void handle(int signal_)
{
   s_value = signal_;
}

int main()
{
    signal(SIGSEGV, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGSEGV);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 11
```

Example 5

Let's see a simple example to illustrate the use of raise() function when SIGFPE is passed:

```
#include <csignal>
#include <iostream>
using namespace std;

sig_atomic_t s_value = 0;
void handle(int signal_)
{
    s_value = signal_;
}

int main()
{
    signal(SIGFPE, handle);
    cout << "Before called Signal = " << s_value << endl;
    raise(SIGFPE);
    cout << "After called Signal = " << s_value << endl;
    return 0;
}
```

Output:

```
Before called Signal = 0
After called Signal = 8
```

← Prev

Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

Splunk tutorial
Splunk

SPSS tutorial
SPSS

Swagger tutorial

T-SQL tutorial
Transact-SQL