```
#include <cxxabi.h>
#include <dlfcn.h>
#include <execinfo.h>

#include <string>
#include<sstream>
#include <thread>
#include <memory>
#include <mutex>

#ifndef  CPP_TOOLS_BACKTRACER_H_
#define CPP_TOOLS_BACKTRACER_H_
/*
 LDFLAGS += -rdynamic  should be used to have symbol info
 */
namespace cpp_tools
{


class BackTracer{

  enum { MAX_DEPTH = 256 };
public:
          BackTracer(int maxDeps = MAX_DEPTH ):m_maxDeps(maxDeps){}

  std::string GetTrace()
  {
     void *trace[MAX_DEPTH] = {NULL};
     std::string out("Call stack. Thread id=");
     std::lock_guard<std::mutex> l(m_mutex);

      std::thread::id id = std::this_thread::get_id();
      std::ostringstream os;
      os<< std::hex<<id<<std::flush;
      out += os.str();  out += "\n";

     int trace_size = backtrace(trace, MAX_DEPTH);

      using namespace abi;

     for (int i=0; i<trace_size; ++i) {

        Dl_info dlinfo = {NULL, NULL, NULL};
        if(!dladdr(trace[i], &dlinfo)){
           continue;
        }
        const char *symname = dlinfo.dli_sname;
        if(!symname){
            continue;
        }
        int status = -2;
        char *demangled = __cxa_demangle(symname, NULL, 0, &status);
        if( (status == 0) && demangled)
```

```
        {
                symname = demangled;
                out += dlinfo.dli_fname;  out += ", ";
                out += symname;  out += "\n";
        }

        if (demangled)
           free(demangled);
     }
     return out;
   }
private:
   int      m_maxDeps;
   mutable std::mutex m_mutex;

};
using BackTracerPtr = std::shared_ptr<BackTracer>;

}
#endif  /* BACKTRACER_H_ */

#include <type_traits>
#include <string>
#include <exception>
#include <boost/lexical_cast.hpp>


#ifndef CPP_TOOLS_CAST_H_
#define CPP_TOOLS_CAST_H_

namespace cpp_tools
{
   enum class CastStatus {
      OK,
      UNNOWN_ERROR
   };

  enum class OnCast {
     DO_CAST,
     DONT_DO_CAST,
     SET_DEFAULT,
     THROW
    };
    /*
     using ReturnT = std::pair<To, std::exception_ptr>
     template <typename To,  typename From> ReturnT Cast( From from )

     cast
     1. from arithmetic to arithmetic,
     2. from arithmetic to std::string,
     3. from string (cold be const char*, char* or std::string) to arithmetic

     ReturnT rt;
```

```
 if valid -   rt.first valid value and rt.second is nullptr
 if invalid - rt.first equals To() and  rt.second pointer to the cast exception

 examples
std::string s1("25.3");   std::string s2("25.x");
// std::pair<To, std::exception_ptr>
auto u1 = Cast<double, std::string>(s1);
auto u2 = Cast<double, std::string>(s2);
auto u3 = Cast<std::string, int>(222);
auto u4 = Cast<int, double>(5.2);
auto u5 = Cast<unsigned int, double>(-5.2);
 */

   // arithmetic to arithmetic
template <typename To,  typename From> inline
    typename std::enable_if<
    std::is_arithmetic<To>::value &&
    std::is_arithmetic<From>::value,
    std::pair<To, std::exception_ptr> >::type
Cast( From from )
{
   try{
      return { boost::numeric_cast<To>(from),  nullptr };
   }catch(...)
   {
      return{To(), std::current_exception()} ;
   }
}

// arithmetic to std::string
template <typename To, typename From> inline
typename std::enable_if<
   std::is_same<To,std::string>::value &&
   std::is_arithmetic<From>::value,
   std::pair<To, std::exception_ptr> >::type
Cast( const From &from )
{
   return { boost::lexical_cast<To>(from),  nullptr };
}

// string (cold be const char*, char* or std::string) to arithmetic
template <typename To, typename From> inline
typename std::enable_if<
   std::is_arithmetic<To>::value &&
   std::is_convertible<From,std::string>::value,
   std::pair<To, std::exception_ptr> >::type
Cast( const From &from )
{
   try{
      return { boost::lexical_cast<To>(from), nullptr };
   }catch(...)
   {
      return { To(), std::current_exception(), } ;
```

```
    }
}

//////

template <typename To> inline  std::exception_ptr Cast(const std::string &from,  To
&to )
{
   static_assert(std::is_arithmetic<To>::value, "Wrong cast. Can be casted only to an
   arithmetic type. ");
   try{
      to = boost::lexical_cast<To>(from);
   }catch(...)
   {
      return std::current_exception();
   }
   return nullptr;
}

 template<typename To>
inline std::exception_ptr  StrToArithmetic(const std::string &from, To &to, OnCast
onEmpty = OnCast:: SET_DEFAULT, OnCast onError = OnCast:: SET_DEFAULT)
{
    if(from.empty() ) // not an error
    {
         switch(onError)
         {
            case cpp_tools::OnCast::SET_DEFAULT:
            {
               to = To();
               break;
            }
         };
         return nullptr;
    }

   std::exception_ptr err = cpp_tools::Cast(from,to);
   if(err)
   {
    switch(onError)
    {
       case cpp_tools::OnCast::SET_DEFAULT:
       {
          to = To();
          break;
       }
       case cpp_tools::OnCast::THROW:
       {
          std::rethrow_exception (err);
       }
    };
    }
```

```
      return err;
   }


}


#endif  /* CPP_TOOLS_CAST_H_*/

#include <memory>

#ifndef CPP_TOOLS_CIRCULARCONDITIONCOUNTER_H_
#define CPP_TOOLS_CIRCULARCONDITIONCOUNTER_H_

namespace cpp_tools
{

   struct CircularConditionCounter
{
   CircularConditionCounter(int period, int  remainderOn ):
    m_period(period), m_remainderOn(remainderOn), m_maxPeriod(3600), m_current(0)
   {
      Reset(period, remainderOn);
   }

   CircularConditionCounter( ):CircularConditionCounter(1, 0)
   {}

   bool IsValid() const
   {
      return ( (m_period >= 1) &&(m_remainderOn >= 0) && ( m_period > m_remainderOn)  );
   }

   bool checkAndIncrement()
   {
     const bool isActive =  IsActive();
     if(isActive){
        m_activeNum++;
     }
     m_current++;
     return isActive;
   }

   void Reset(int period = 1, int  remainderOn = 0 )
   {
      m_current = 0;
      m_period =  (period < m_maxPeriod) ? period : m_maxPeriod;
      m_remainderOn = remainderOn;
      m_activeNum = 0;
   }

  long GetCurrent() const { return m_current; }
  int GetPeriod() const { return m_period; }
```

```
  int GetRemainderOn() const { return m_remainderOn; }
  long GetActiveNum() const { return m_activeNum; }

private:
   bool IsActive()
   {
      if( m_period == 1) return true; //
      int  remainder =  m_current % m_period;
      bool out = (remainder == m_remainderOn );
      return out;
   }

   private:
    int                m_period;
    int                m_remainderOn;
    int                m_maxPeriod;
    long               m_current;
    long               m_activeNum;
};

using CircularConditionCounterPtr = std::shared_ptr<CircularConditionCounter>;

}
#endif  /* CIRCULARCONDITIONCOUNTER_H_ */

#ifndef CPP_TOOLS_EXCEPTION_H_
#define CPP_TOOLS_EXCEPTION_H_

#include <exception>
#include <string>
#include <memory>
#include <thread>
#include <mutex>
#include <functional>
#include "CPPTools/BackTracer.h"

namespace cpp_tools
{
    enum class TraceLevel{
         NO_TRACE,
         EXCEPTION_TRACE,
         BACK_TRACE,
         MAX_TRACE
       };

       enum class PostAction{
          NO_ACTION,
          STD_DEFAULT
       };

       using Logger =  std::function<void(std::string)>;
```

```
namespace exception_impl
{
   void DummyLogger (std::string ){}// does nothing

   struct ExceptionHandlerInfo
   {

       ExceptionHandlerInfo(  ): m_logger (DummyLogger),
       m_traceLevel(TraceLevel::NO_TRACE),  m_postAction(PostAction::STD_DEFAULT)  {}

      TraceLevel GetTraceLevel() const
      {
           std::lock_guard<std::mutex> l(m_mutex);
           return m_traceLevel;
      }

      Logger GetLogger() const
      {
           std::lock_guard<std::mutex> l(m_mutex);
           return m_logger;
      }
      PostAction GetPostAction() const
      {
           std::lock_guard<std::mutex> l(m_mutex);
           return m_postAction;
      }

       void  SetTraceLevel(TraceLevel   traceLevel)
      {
           std::lock_guard<std::mutex> l(m_mutex);
           m_traceLevel = traceLevel;
      }
        void  SetLogger(Logger  logger)  ,
      {
           std::lock_guard<std::mutex> l(m_mutex);
           m_logger = logger;
      }
         void  SetPostAction(PostAction   postAction)
      {
           std::lock_guard<std::mutex> l(m_mutex);
           m_postAction = postAction;
      }

    private:
      Logger                    m_logger;
      TraceLevel                m_traceLevel;
      PostAction                m_postAction;
      mutable                   std::mutex m_mutex;

    };

  namespace _private {
     static ExceptionHandlerInfo   m_handlerInfo;
```

```
inline std::string GetCurrentExceptionStr()
{
 std::exception_ptr pException = std::current_exception();
 if(!pException)
 {
    return "No current exception";
 }

 try{
     std::rethrow_exception (pException);
 }catch(std::exception ex)
 {
    return ex.what();
 }
 catch(...)
 {
     return "Uknown exception";
 }
 return "Coding error in GetCurrentExceptionStr";
}

inline std::string GetBackTraceStr()
{
   cpp_tools::BackTracer tracer;
   std::string trace = tracer.GetTrace();
   return trace;
}

inline void ExceptionHandler(const ExceptionHandlerInfo& info)
{
   std::string msg;
   const TraceLevel  traceLevel = info.GetTraceLevel();
   if( (traceLevel == TraceLevel::EXCEPTION_TRACE) )
   {
      msg += "Exception trace \n";  msg += GetCurrentExceptionStr(); msg +="\n";
   }
   else  if( (traceLevel == TraceLevel::BACK_TRACE) )
   {
      msg += "Stack trace:";  msg += GetBackTraceStr(); msg +="\n";
   }
   else  if( (traceLevel == TraceLevel::MAX_TRACE) )
   {
      msg += "Exception trace \n";  msg += GetCurrentExceptionStr(); msg +="\n";
      msg += "Stack trace:";  msg += GetBackTraceStr(); msg +="\n";
   }

   if(!msg.empty())
   {
      Logger loger = info.GetLogger();
      loger(msg);
      std::this_thread::sleep_for(std::chrono::milliseconds(1000));
```

```
        }

        if(info.GetPostAction() == PostAction::STD_DEFAULT )
        {
           std::this_thread::sleep_for(std::chrono::milliseconds(1000));
           std::terminate();
        }
     }

    }
     static ExceptionHandlerInfo& ExceptionHandlerInfoInstance() {return
     _private::m_handlerInfo ;}

     inline void TerminateExceptionHandler()
     {
        const ExceptionHandlerInfo& info =
        exception_impl::ExceptionHandlerInfoInstance();
        _private::ExceptionHandler(info);

     }

     inline void UnexpectedExceptionHandler()
     {
        const ExceptionHandlerInfo& info =
        exception_impl::ExceptionHandlerInfoInstance();
        _private::ExceptionHandler(info);

     }
  }

  void SetExceptionHandlers(Logger  logger, TraceLevel traceLevel =
  TraceLevel::MAX_TRACE ,  PostAction postAction = PostAction::STD_DEFAULT  )
  {
     exception_impl::ExceptionHandlerInfo& info =
     exception_impl::ExceptionHandlerInfoInstance();
     info.SetLogger(logger);
     info.SetTraceLevel(traceLevel);
     info.SetPostAction(postAction);

     std::set_terminate(exception_impl::TerminateExceptionHandler);
     std::set_unexpected(exception_impl::UnexpectedExceptionHandler);

   }



}
#endif  /* EXCEPTION_H_ */
```

```
#include <random>
#include <map>
#include <memory>
#include <type_traits>


#ifndef CPP_TOOLS_RANDOM_H_
#define CPP_TOOLS_RANDOM_H_

namespace cpp_tools
{

template< typename EngineT> using RandomEnginePtr = std::shared_ptr<EngineT>;

 template< typename EngineT>
 inline  RandomEnginePtr<EngineT> NewRandomEngine()
{
    return std::make_shared<EngineT>();
}


inline  std::shared_ptr<std::default_random_engine> NewRandomEngine()
{
    return NewRandomEngine<std::default_random_engine> ();
}



 template< typename EngineT = std::default_random_engine>
 inline  EngineT & RandomEngineInstance()
{
   static EngineT engine;
   return engine;
}

 enum class DistributionType
 {
    uniform_int_distribution,
    uniform_real_distribution
 };


 template <DistributionType dt, class T, class Enable = void > struct DistributionTrait
 {
    static_assert(true,"This distribution has not been implemented");
 };

 template <class T> struct DistributionTrait<DistributionType::uniform_int_distribution,
 T,  typename std::enable_if<std::is_integral<T>::value >::type >
 {
    using RandomGenerator  = std::uniform_int_distribution<T>;
 };
```

```
template <class T> struct DistributionTrait<
DistributionType::uniform_real_distribution, T, typename
std::enable_if<std::is_floating_point<T>::value >::type >
{
   using RandomGenerator  = std::uniform_real_distribution<T>;
};


template<  DistributionType dt, typename Type, typename EngineT =
std::default_random_engine >
class Distributor
{
  using RandomGenerator  = typename DistributionTrait<dt,Type > :: RandomGenerator;
  using RandomGeneratorPtr = std::shared_ptr<RandomGenerator>;
  using RangeT = std::pair<Type,Type> ;
  using Generators    =    std::map<RangeT , RandomGeneratorPtr>;
  friend bool operator < (RangeT lr, RangeT rr)
  {
     if(lr.second <  rr.second) return true;
     if(lr.second >  rr.second) return true;
     if(lr.first <  rr.first) return true;
     return false;
  }

public:
   Distributor(RandomEnginePtr<EngineT>  pEngine):m_pEngine(pEngine)
   {}
   using ResultT = std::pair<Type,bool>;
   ResultT generate(Type max)
   {
      return generate(0, max );
   }
    ResultT generate(Type min, Type max)
   {
       if(min > max)
       {
          return ResultT(0, false);
       }
      if( min == max)
      {
         return ResultT(max, true);
      }

      const RangeT key(min, max);
      auto iter = m_generators.find(key);
      if(iter != m_generators.end() ) //found
      {
         const Type value = (*(iter->second)) (*m_pEngine);
         return  ResultT(value, true);
      }else
      {
          RandomGeneratorPtr generator= std::make_shared<RandomGenerator>(key.first,
          key.second);
```

```
            m_generators.insert(typename Generators::value_type(key, generator) );
            const Type value =(*generator)(*m_pEngine);
            return  ResultT(value, true);;
        }

    }
private:
  RandomEnginePtr<EngineT>          m_pEngine ;
  Generators                        m_generators;

};

using UniformIntDistributor = Distributor<DistributionType::uniform_int_distribution, int>;
using UniformIntDistributorPtr = std::shared_ptr<UniformIntDistributor>;
using UniformDoubleDistributor = Distributor<DistributionType::uniform_real_distribution,
double>;
using UniformDoubleDistributorPtr = std::shared_ptr<UniformDoubleDistributor>;

/* usage

 UniformIntDistributor<> ds(NewRandomEngine<>());

int main()
{
 for(int i = 0; i < 10; ++i)
 {
    std::cout <<ds.generate(10).first<<std::endl;

 }

 for(int i = 0; i < 20; ++i)
 {
    std::cout <<ds.generate(40).first<<std::endl;

 }

    return 0;
}



 */


}
#endif  /* RANDOM_H_ */


#include <string>
#include <signal.h>
#include <exception>
```

```
#ifndef CPP_TOOLS_SIGNAL_H_
#define CPP_TOOLS_SIGNAL_H_

namespace cpp_tools
{

   using OnSignal =  std::function<void(int)>;



   namespace signal_impl
   {

   enum class SynchronizationType{UKNOWN, SYNC, ASYNC };

   template <sig_atomic_t signal> struct SignalType
   {
      enum {m_signal = signal };
      static std::string ToString(){ return std::to_string(m_signal); }
      static SynchronizationType SyncType(){ return SynchronizationType::UKNOWN; }
    };

      template <> std::string  SignalType<SIGFPE>::ToString(){ return "SIGFPE";  }
      template <> std::string  SignalType<SIGSEGV>::ToString(){ return "SIGSEGV";}

      template <> SynchronizationType  SignalType<SIGFPE>::SyncType(){ return
      SynchronizationType::SYNC;  }
      template <> SynchronizationType  SignalType<SIGSEGV>::SyncType(){ return
      SynchronizationType::SYNC;  }


      template <sig_atomic_t signal> struct SignalException : public std::exception{
         SignalException():std::exception( ){}//
         const char* what() const noexcept {return SignalType<signal>
         ::ToString().c_str();}

      };

   }

      void dummyAction(int){}
      void OnSIGFPE(int)  { throw signal_impl::SignalException<SIGFPE>() ;}
      void OnSIGSEGV(int) { throw signal_impl::SignalException<SIGSEGV>() ;}
}

#endif  /* SIGNAL_H_ */


#include <string>
#include <boost/current_function.hpp>


#ifndef CPP_TOOLS_TOOLS_H_
```

```
#define CPP_TOOLS_TOOLS_H_


#define CONCATENATE_TOKENS(tkn1, tkn2) tkn1##tkn2
#define TOKEN_TO_STR(tkn) #tkn


const std::string  LOCATION_SEP(":");

#define SRC_LOCATION ( std::string(__FILE__) +  LOCATION_SEP + \
    std::to_string(__LINE__) + LOCATION_SEP +  \
    std::string(BOOST_CURRENT_FUNCTION) )


namespace cpp_tools
{
template<class T, int N>
constexpr int array_size(const T (&a)[N])
{
    return N;
}

const double DAYS_IN_YEAR = 365.25;
const double WEEKS_IN_YEAR = DAYS_IN_YEAR/7.0;
const double BIZ_DAYS_IN_YEAR = 252.0;
const double ONE_DAY_TIME_MKT = 1.0 / BIZ_DAYS_IN_YEAR;
const double ONE_S_TIME_MKT = ONE_DAY_TIME_MKT / (24.0 * 3600.0);
const double ONE_MS_TIME_MKT = ONE_S_TIME_MKT / (1000.0);

}

#endif /* CPP_TOOLS_TOOLS_H_ */


#include <string>

#ifndef CPP_TOOLS_VERSIONINFO_H_
#define CPP_TOOLS_VERSIONINFO_H_

namespace cpp_tools
{

      template <int ver> struct VersionInfo{
            enum{VERSION = ver};
            enum{MAJOR = VERSION/10000 };
            enum{MINOR =  (VERSION - MAJOR *10000 )/ 100} ;
            enum{REVISION = VERSION%100 };

            const static std::string VERSION_STR;
            const static std::string MAJOR_STR;
            const static std::string MINOR_STR;
            const static std::string REVISION_STR;
            const static std::string VERSION_FORMATED_STR;
```

```
        };

        template <int ver>
        const std::string
        VersionInfo<ver>::VERSION_STR(boost::lexical_cast<std::string>(VersionInfo<ver>::VE
        RSION) );
        template <int ver>
        const std::string
        VersionInfo<ver>::MAJOR_STR(boost::lexical_cast<std::string>(VersionInfo<ver>::MAJO
        R) );
        template <int ver>
        const std::string
        VersionInfo<ver>::MINOR_STR(boost::lexical_cast<std::string>(VersionInfo<ver>::MINO
        R) );
        template <int ver>
        const std::string
        VersionInfo<ver>::REVISION_STR(boost::lexical_cast<std::string>(VersionInfo<ver>::R
        EVISION) );
        template <int ver>
        const std::string VersionInfo<ver>::VERSION_FORMATED_STR( MAJOR_STR + "." +
        MINOR_STR + "." + REVISION_STR);


}
#endif  /* CPP_TOOLS_VERSIONINFO_H_ */
```

```
/
// trace.hpp

#include <string>
#include<iostream>
#include <thread>
#include <chrono>
#include <random>
#include <memory>
#include <signal.h>
#include <mutex>

std::mutex TrapMutex;

void TrapSignal(int signal)
{
   std::lock_guard<std::mutex> l(TrapMutex);
   switch (signal)
   {
      case SIGFPE:
      {
         std::thread::id id = std::this_thread::get_id();
         std::cout <<"SIGFPE traped in thread :" << std::hex<< id << std::endl;
         // this thread id and stack is available and printable here, could be parsed to exception;
         std::this_thread::sleep_for(std::chrono::milliseconds(10000));
         throw std::string("SIGFPE");
         break;
      }
      case SIGSEGV:
      {

         std::thread::id id = std::this_thread::get_id();
         std::cout <<"SIGSEGV traped in thread :" << std::hex<< id << std::endl;
          // this thread id and stack is available and printable here, could be parsed to exception;
         std::this_thread::sleep_for(std::chrono::milliseconds(10000));
         throw std::string("SIGSEGV");
         break;
      }
   }
}

void good()
{
   std::thread::id id = std::this_thread::get_id();
   std::cout << "good thread started:" << std::hex<< id << std::endl;
   for(int i =0; i < 100; ++i ) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }
}

void emitSIGFPE()
{
   std::thread::id id = std::this_thread::get_id();
   std::cout <<"SIGFPE thread started:" << std::hex<< id << std::endl;
   for(int i =0; i < 100 ; i++ ) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }

    int y = 1/0;
    for(int i =0; i < 100 ; i++  ) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }
}

void emitSIGSEGV()
{
```

```
   std::thread::id id = std::this_thread::get_id();
   std::cout <<"SIGSEGV thread started:" << std::hex<< id << std::endl;
   for(int i =0; i < 100 ; i++ ) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }
    int * pi = NULL;
     int y = *(pi);
    for(int i =0; i < 100 ; i++  ) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }
}


int main()
{
      signal(SIGFPE, &TrapSignal);
      signal(SIGSEGV, &TrapSignal);
      signal(SIGPIPE, SIG_IGN);

      std::thread g1( good);
      //std::thread SIGSEGV1(emitSIGSEGV);
     std::thread SIGFPEE2(emitSIGFPE);
     std::thread g2(good);
      std::thread SIGSEGV2( emitSIGSEGV);
     // std::thread SIGFPEE3(emitSIGFPE);
      std::thread g3(good);
     // std::thread SIGSEGV3(emitSIGSEGV);
      SIGSEGV2.join();
      g1.join();
      g2.join();
      g3.join();


    return 0;
}
/*
SIGFPE thread started:4332b940
good thread started:43d2c940
SIGSEGV thread started:4472d940
good thread started:4512e940
SIGFPE traped in thread :4332b940
terminate called after throwing an instance of 'std::string'


 */
```