Application For Google Summer Of Code 2008

A Safe Variable Template Class For The Boost Library

Youssef Oujamaa

March 30, 2008

About me

I'm a 3e year student of Information and Communication Technology living in The Netherlands, Utrecht. My primary interests are developing programs and generally exploring the field of computing. I primarily use Linux as my desktop operating system and write my programs in C++ using the GNU C++ compiler. Although I sometimes just try out new languages from all sides of the programming language spectrum like Fortran, PHP and SQL

Why Boost?

I chose Boost to be my mentoring organization because I'm really interested in developing useful C++ applications and libraries. As a Boost user myself (boost.program_options and boost.date_time to be precise) I enjoy using libraries that work well for what they are designed for and also work on multiple platforms so things can just get done without to much hassle.

I think I'm capable of working on this because I have the will for it and the experience needed to realize this project. I'm also planning to actually continue development even after the Google Summer of Code by maintaining the code and documentation.

Project proposal

The idea is to implement a safe variable template class which for example can be used to prevent integer under- and overflows, which in turn could cause bugs in code like, for example buffer overruns.
This might look like a trivial library on the surface, but the value it adds is that it can save developers time by providing a ready to use safe variable template class and give the developers another new tool to use if they ever need it.
I chose not to name the library *SafeInt* or any form that refers to a specific type. Because they would be misnomers when the template class allows for more then just the built-in integer type. Thus, the name I chose was *safe_variable*, which is more generic and so more suited for multiple types.
In principle, allowing for dynamic maximum and minimum values for the safe variable gives the user of the library more control. The problem with this is the memory overhead it can create by introducing two more types that need to be stored in the class. For this reason I suggest the best is idea is to provide both a static and dynamic version of the safe variable class. Where the static version would allow the maximum and minimum values to be passed as template arguments.

The API

An example of the API design of the template class with dynamic boundaries:

```
#include <iostream>
#include <stdexcept>

template<typename T>
class safe_variable;
```

```
template<typename T> safe_variable<T>
operator+(const T& left, const safe_variable<T>& right) throw(std::runtime_error);
template<typename T> safe_variable<T>
operator-(const T& left, const safe_variable<T>& right) throw(std::runtime_error);
template<typename T> safe_variable<T>
operator*(const T& left, const safe_variable<T>& right) throw(std::runtime_error);
template<typename T> safe_variable<T>
operator/(const T& left, const safe_variable<T>& right) throw(std::runtime_error);
template<typename T>
bool operator<(const T& left, const safe_variable<T>& right);
template<typename T>
bool operator>(const T& left, const safe_variable<T>& right);
template<typename T>
bool operator<=(const T& left, const safe_variable<T>& right);
template<typename T>
bool operator>=(const T& left, const safe_variable<T>& right);
template<typename T>
bool operator==(const T& left, const safe_variable<T>& right);
template<typename T>
bool operator!=(const T& left, const safe_variable<T>& right);

template<typename T>
class safe_variable
{
    public:

    // Default constructor sets everything safe to zero
    safe_variable() : variable_value(T()), min_value(T()), max_value(T()) {
        // ...
    }

    // With this constructor an exception can be thrown if the supplied values conflict with the type.
    safe_variable(const T& new_value, const T& new_min, const T& new_max) throw(std::runtime_error);

    // The following member functions can be used to retrieve the set maximum and minimum value.
    T max();
    T min();

    // These 'setters' will throw an exception when the supplied value is outside
    // of the real maximum range of the type.
    void max(const T&) throw(std::runtime_error);
    void min(const T&) throw(std::runtime_error);

    // Some of the operators a safe variable class needs which respectively
    // throw an exception when a certain operation can't complete.
    safe_variable<T>& operator=(const safe_variable<T>& right) throw(std::runtime_error);
    safe_variable<T>& operator=(const T& right) throw(std::runtime_error);

    safe_variable<T> operator--() const throw(std::runtime_error);
    safe_variable<T> operator--(int) const throw(std::runtime_error);
    safe_variable<T> operator++() const throw(std::runtime_error);
    safe_variable<T> operator++(int) const throw(std::runtime_error);

    friend safe_variable<T> operator+<>(const T& left, const safe_variable<T>& right)
    throw(std::runtime_error);
    friend safe_variable<T> operator-<>(const T& left, const safe_variable<T>& right)
    throw(std::runtime_error);
    friend safe_variable<T> operator*<>(const T& left, const safe_variable<T>& right)
    throw(std::runtime_error);
    friend safe_variable<T> operator/<>(const T& left, const safe_variable<T>& right)
    throw(std::runtime_error);

    safe_variable<T>& operator+=(const T&) throw(std::runtime_error);
    safe_variable<T>& operator-=(const T&) throw(std::runtime_error);
    safe_variable<T>& operator*=(const T&) throw(std::runtime_error);
    safe_variable<T>& operator/=(const T&) throw(std::runtime_error);
    friend bool operator< <>(const T& left, const safe_variable<T>& right);
    friend bool operator> <>(const T& left, const safe_variable<T>& right);
    friend bool operator<=<>(const T& left, const safe_variable<T>& right);
    friend bool operator>=<>(const T& left, const safe_variable<T>& right);
    friend bool operator==<>(const T& left, const safe_variable<T>& right);
    friend bool operator!=<>(const T& left, const safe_variable<T>& right);

    operator T() const;

    private:
    T variable_value; // Holds the value.
    T min_value; // Holds the minimum value allowed.
    T max_value; // Holds the maximum value allowed.
};

int main()
{
    // Now you can make a safe integer type and construct it with limits.
    typedef safe_variable<int> safe_integer;
    typedef safe_variable<char> safe_char;

    safe_integer bar(0, 0, 9999);
    safe_char foo(0, 0, 255);

    bar = 100;
    foo = 7;

    bar = foo;
    try {
        // With this assignment an exception will be thrown.
        foo = 300;
    } catch(std::runtime_error& e) {
        // ...
    }
    return 0;
}
```

Exceptions are a vital part of the safe variable template class, they will be used to warn for under- and overflow exceptions. The exception classes that will be used are the *std::underflow_exception* and *std::overflow_exception* which are derived from the *std::runtime_exception* class. It's also possible to allow user to supply predicates which define the behaviour of the safe_variable class.
More things that need to be considered are, a template specialization for float and double data types and keeping performance up so it doesn't drastically drag down program performance.

Timeline

This is an estimation on what can be done in 2 months worth of time. Leaving room in case of time shortage at certain points.

- Week 1: Familiarize myself with Boosts best coding practises, coding guidelines, version control system etc. Gathering input from other developers on what they would like to see in a safe variable template class. (and what not)
- Weeks 2 - 3 - 4: Discuss design with mentor, implement the class and the associated functions and discuss the progress with the mentor.
- Weeks 5 - 6: Implement template specializations for floating point types. Improve performance of the library and test for bugs.
- Week 7: Test cross platform compatibility across different compilers and operating systems. Refactor code where needed.
- Week 8: Contact with other developers through mailinglist for input and make changes if necessary.
- Weeks 9 - 10: Write documentation for the library and usage examples.

I think with this timeline this project can be fruitful and be finished well before the ’pencils down’ date, leaving room for the unexpected.