2.1 Type Traits

Summary

Most of the exercises in this section are straightforward examples of the functions in the C++ *type_traits* library. A good application is the compile-time *Bridge* that allows us to perform operations depending on the type we use. You may find and need other applications of *type_traits* and you are then essentially in the realm of *Template Metaprogramming (TMP)*, a subject that is outside the current scope. See "C++ Template Metaprogramming" by D. Abrahams and A. Gurtovoy, Addison Wesley 2005 for more information. Furthermore, The Boost MPL Library (see www.boost.org) is also very useful.
In general, you are expected to write functions to test *type_traits*. You can be as creative as you like so long as you satisfy the requirements. You should test your functions by calling them for a range of C++ types.

1. (Pointers and non-Pointers)

This exercise consists of calling some functions from *Primary type categories*.

Answer the following questions:

a) Write a function to determine if a type is a pointer, null pointer, lvalue reference or rvalue reference.
b) Determine if a type is a member function pointer or if it is a pointer to a non-static member object.
c) Is a shared pointer a pointer type? Is it a pointer type when converted to a raw pointer?

Typical code is:

```
template <typename T>
    void IsPointer(const T& t)
{ // First example of type_traits;check if t is a pointer

    // Return type is std::true_type or std::false_type
    if (std::is_pointer<T>::value)
    {
          std::cout << "This is a pointer type argument\n";
    }
    else
    {
          std::cout << "_not_ a pointer type argument\n";
    }
}
```

2. (Simple switchable *Bridge* Functionality)

We create a template function that supports both pointers and reference types. If it is a pointer it is dereferenced and then printed while if it is not a pointer type and it is a scalar reference type then it is printed directly. Use the `is_pointer()` function in conjunction with `std::true_type` and `std::false_type` to determine which implementation should be called.

3. (Type Properties and Type Relationships)

Consider the following (simple) class hierarchy:

```
class Shape
{
public:
          virtual void draw() = 0;
};

class Base
{
private:
    int y;
public:
```

```
    Base() {}
    void draw() {}
};


class Point : public Shape
{
public:
    Point() {}
    virtual void draw() override {}
};
```

Answer the following questions:
a) Which classes/types are empty, polymorphic or abstract?
b) Which classes are the same?
c) Which classes have a gen/spec (base/derived) relationship?
d) Which types can be converted to each other?

To answer these questions, you need to program/use the appropriate type traits functions applied to the above classes.

4. (Supported Operations)
Consider the following class:

```
class Point : public Shape
{
public:
    Point() {}
    virtual void draw() override {}
};
```

Answer the following questions:
a) Programmatically determine if type `Point` has a virtual destructor.
b) Programmatically determine if type `Point` has the following constructors: default, copy, move.
c) Programmatically determine if type `Point` is copy assignable/move assignable.
d) Set the copy constructor and assignment operator as 'deleted functions' and perform steps b)-c) again; do you get the same result?
e) Add move semantics code (explicit move constructor and move assignment operator) and perform steps b)-c) again; do you get the same result?

2.2 Advanced Type Traits

1. (A Numeric Application: the Beginnings of a Vector Space)

Let's pretend that we wish to create a mathematical library to model vector spaces. In particular, we define the following operations:

- Add two vectors.
- The additive inverse of a vector.
- Scalar multiplicity (premultiply the vector by a scalar).

In concrete terms, we wish to define the above operations for scalar and array/vector numeric types. You will need to draw on what you did in the previous exercises on type traits.

Answer the following questions:

a) Define addition of two vectors as follows:

```
template <typename T>
    T Addition(const T& t1, const T& t2)
{ // Vector space addition

    // Best(?) approximation to testing for an array
    return Addition_impl(t1, t2, std::is_compound<T>());
}
```

b) Define additive inverse of a vector:

```
template <typename T>
    void AdditiveInverse(T& val)
{ // val -> -val for a scalar or a vector

    // TBD
}
```

c) Define a function for scalar multiplication of a scalar and a vector, producing a new vector. Determine the signature of this function.

d) Test these three functions for double, std::array<double, N> and std::vector<double>.

e) Test the functions on containers whose underlying type is `std::complex<double>`.

2. (Arrays Categories)

We discuss some functions that work with array types. The *rank* of an array type is equal to the number of dimensions of the array. The *extent* of an array type is the number of elements along the Nth dimension of the array if N is in the closed interval `[0, std::rank<T>::value]`. For any other type, the value is `0`.

Answer the following questions:

a) Test `std::is_array()` on a range of fundamental, scalar, object, arithmetic and compound types.

b) Create an array `int [][3][4][5]`. Find its rank and extent.

c) Call `std::remove_extent()` and `std::remove_all_extent()` on the array in question b). What is happening?

3. (Conversions)

Sometimes you may wish to structurally change fundamental properties of types in an application. Examples are:

- Mapping integers to unsigned integers and vice versa.
- Add/remove the const specifier to or from a type.
- Add/remove a pointer to or from a type.
- Add/remove the `volatile` specifier to or from a type.
- Discuss the difference between *direct list initialisation* and *copy list initialisation* in C++17/20 compared to C++11. In which case does `auto` deduction reduce to `std::initializer_list`? Give a code example. Under which circumstances can we get ill-formed expressions?
- Discuss how *Class Template Argument Deduction* (CTAD)) reduces code verbosity. Give a non-trivial example using `std::tuple` and a user-defined class.
- Investigate the applicability of decltype and std::is_same when comparing the type of a result with a "target" type.

Answer the following questions:

a) Write a separate function for each of the above requirements.
b) Test the functions on a range of fundamental, scalar, object, arithmetic and compound types.

2.3 Advanced Lambda Programming

1. (User-defined deprecated Entities)

Since C++14 it is possible to mark C++ entities as being *deprecated/obsolete* using the C++14 `deprecated` attribute, for example:

```
[[deprecated("old stuff")]] void Func()
{
    // whatever
}
```

Calling this function will result in your program failing to compile. The objective of this exercise is to mark a range of entities as being deprecated to (try to) use them in code. Answer the following by marking the following entities as being deprecated and testing you code:

a) Free/global functions.
b) Deprecated class and non-deprecated class with deprecated member function.
c) Deprecated global variable.
d) Deprecated `enum` and deprecated `enum class`.
e) Deprecated template class specialisation.
f) Deprecated lambda function.

(Remark: support for this C++14 feature may vary from compiler to compiler at the moment of writing).

2. (C++14: Using *init Captures* to move Objects into Closures)

This exercise is concerned with the issue of captured variables when working with lambda functions. In particular, copying objects into the closure can lead to performance hits and capture-by-reference can lead to dangling pointers. Furthermore, these two capture modes do not work with *move-only objects* (for example, `std::unique_ptr` or `std::future`). For this reason, we discuss the *init capture* syntax.

Answer the following questions:

a) Consider the C++11 code:

```
int size = 4; double val = 2.71;
std::vector<double> data(size, val);

// Default capture mode (not preferred)
auto fun = [&data]() mutable
{
    for (std::size_t i = 0; i < data.size(); ++i)
    {
          data[i] = 3.14; std::cout << data[i] << ",";
    }

    std::cout << '\n';
};

fun();

// 'data' still exists and has been modified
for (std::size_t i = 0; i < data.size(); ++i)
    { data[i] = 3.14; std::cout << data[i] << ":"; }
```

Modify this code using move semantics, that is the vector `data` is moved into the lambda function. Test the new code, including the values in `data` after the code has run.

b) Create the following code snippet:

```
// C++14 init capture with local variables
int x = 1'000'000; // This is a digit separator
```

```
auto fun4 = [&r = x]()
{
    r *= 2;
    std::cout << "\nr: " << r;
};
fun4();
```

c) Migrate the following C++11 code that uses `std::bind` to emulate C++14 code that uses init capture:

```
// Emulating generalized lambda capture with C++11
int size2 = 3; double val2 = 1.41;
std::vector<double> data2(size2, val2);
std::array<double, 3> data3 = { 1.2, 2.4, 4.7 };
auto fun3 = std::bind( [] (std::vector<double>& array,
                           std::array<double, 3> array2)
{
    for (std::size_t i = 0; i < array.size(); ++i)
    {
          array[i] = 3.14; std::cout << array[i] << "/";
    }
    std::cout << '\n';
    for (std::size_t i = 0; i < array2.size(); ++i)
    {
          array2[i] = 2.71; std::cout << array2[i] << "/";
    }

}, std::move(data2), std::move(data3));

fun3();

if (0 == data2.size() || 0 == data3.size())
{
    std::cout << "\nDouble arrays have no elements, OK\n";
}
else
{
    std::cout << "\n Ouch!\n";
    for (std::size_t i=0;i<data2.size(); ++i)
                {std::cout<<data2[i]<< "^"; }
    for (std::size_t i = 0; i < data3.size(); ++i)
                {std::cout<<data3[i]<< "^"; }
    }
}
```

It is clear that C++14 syntax is more elegant and user-friendly.

3. (Doing Mathematics in C++)

In this exercise you build a simple *vector space of functions*. The basic objective is to show how to create *higher-order functions* in C++. It shows what is possible with lambda functions and the possible applications to mathematics, engineering and numeric computation in general.
We scope the problem drastically; we consider only functions of a single variable returning a scalar value. All underlying types are `double`.

First, create function classes as follows:

```
// Function maps Domain to Range
template <typename R, typename D>
    using FunctionType = std::function<R (const D x)>;

// Working function type
using ScalarFunction = FunctionType<double, double>;
```

Answer the following questions:

a) Create functions to add, multiply and subtract two functions. Create unary additive inverse of a function and a function representing scalar multiplication. A typical example is:

```
template <typename R, typename D>
    FunctionType<R,D> operator + (const FunctionType<R,D>& f,
const FunctionType<R,D>& g)
{ // Addition of functions

    return [=] (const D& x)
    {

          return f(x) + g(x);
    };
}
```

Typical functionality that you need to create is:

```
// Scalar functions: double to double
ScalarFunction f = [](double x) {return x*x; };
ScalarFunction g = [=](double x) { return x; };
std::cout << g(2) << ", " << g(8) << "*";

// Set I: Arithmetic functions
double scale = 2.0;
auto fA = 2.0*(f + g);
auto fM = f - g;
auto fD = f / g;
auto fMul = f * g;
auto fC = g << g << 4.0*g;     // Compose
auto fMinus = -(f + g);

double x = 5.0;
std::cout<< fA(x) << ", " << fM(x) << ", " << fD(x) << ", "
    << fMul(x)<<", compose: "<< fC(x) <<", " << fMinus(x);
```

b) Create trigonometric and other functions such as those in <cmath>. For example:

```
template <typename R, typename D>
    FunctionType<R,D> exp(const FunctionType<R,D>& f)
{ // The exponential function

    return [=](const D& x)
    {
          return std::exp(f(x));
    };
}
```

Implement the following functions: cos, sin, tan, exp, log, abs and sqrt.

Typical functionality that you need to create is:

```
auto ftmp = log(g); auto ftmp2 = cos(f);
auto f2 = (abs(exp(g))*log(g) + sqrt(ftmp)) / ftmp2;
std::cout << f2(x) << std::endl;
```

c) Finally, produce code that allows you to do the following:

```
auto h1 = min(f, g);
auto h2 = max(f, g);
std::cout << "min(f,g): " << h1(2.0) << '\n';
std::cout << "max(f,g): " << h2(2.0) << '\n';
```

```
auto h3 = min(h1, max(2 * f, 3 * g));
auto h4 = max(h2, max(-2 * f, 3 * g));
```

4. (Emulating deprecated C++ 98 STL Bind Code)

As an exercise in showing your agility with lambda code, implement the following functions whose interfaces are specified as:

```
template <typename T>
    std::function<T (const T&)> bind1st
    (const std::function<T (const T& x, const T& y)>& f,const T& x_)
{ // Bind to the first param x of f(x,y)

    // TBD

}


template <typename T>
    std::function<T(const T&)> bind2nd
    (const std::function<T(const T& x, const T& y)>& f, const T& y_)
{ // Bind to the second param y of f(x,y)

    // TBD
}
```

An example of use is:

```
double yval = 10.0;
ScalarFunction fx = bind2nd<double>(fxy, yval);
std::cout << fx(3.0) << std::endl;

// Test new bind code on STL algorithms
std::vector<double> vec(10, 2.0);
std::transform(vec.begin(), vec.end(), vec.begin(), fx);
for (auto i = 0; i < vec.size(); ++i)
{
    std::cout << vec[i] << ",";
}
```

Experiment with some of your examples as well as implementing all of the above.

2.4 Smart Pointers in Boost and C++

1. (First Encounters with Smart Pointers: *Unique Pointers*)

Consider the following code that uses raw pointers:

```
{ // Block with raw pointer lifecycle

    double* d = new double (1.0);
    Point* pt = new Point(1.0, 2.0); // Two-d Point class

    // Dereference and call member functions
    *d = 2.0;
    (*pt).X(3.0);       // Modify x coordinate
    (*pt).Y(3.0);       // Modify y coordinate

    // Can use operators for pointer operations
    pt->X(3.0);         // Modify x coordinate
    pt->Y(3.0);         // Modify y coordinate

    // Explicitly clean up memory (if you have not forgotten)
    delete d;
    delete pt;
}
```

Answer the following questions:

a) Type, run and test the above code. Introduce a *try-catch* block in which memory is allocated and directly afterwards an exception is thrown. Since the code is not re-entrant, the memory is not reclaimed and hence introduces a *memory leak* (in more general cases it would be a *resource leak*).
b) Now port the above code by replacing raw pointers by `std::unique_ptr`. Run the code. Are there memory leaks now?
c) Experiment with the following: initialising two unique pointers to the same pointer, assigning a unique pointer to another unique pointers and *transferring ownership* from one unique pointer to another unique pointer.
d) Use *alias template* (template typedef) to make the code more readable.

2. (Shared Pointers)

The objective of this exercise is to show *shared ownership* using smart pointers in C++.
Create two classes `C1` and `C2` that share a common heap-based object as data member:

```
std::shared_ptr<double> d;
```

Answer the following questions:

a) Create the code for the classes `C1` and `C2` each of which contains the shared object from the value `d` above, for example:

```
class C1
{
private:
    //double* d; OLD WAY
    std::shared_ptr<double> d;
public:
    C1(std::shared_ptr<double> value) : d(value) {}
    virtual ~C1() { cout << "\nC1 destructor\n";}
    void print() const { cout << "Value " << *d; }
};
```

b) Create instances of these classes in scopes so that you can see that resources are automatically released when no longer needed. To this end, employ the member function `use_count()` to keep track of the number of shared owners. This number should be equal to 0 when the program exits.

c) Carry out the same exercise as in steps a) and b) but with a user-defined type as shared data:

```
std::shared_ptr<Point> p;
```

d) Now extend the code in parts a) to c) to include the following operations on shared pointers: assigning, copy and compare two shared pointers `sp1` and `sp2`. Furthermore, test the following features (some research needed here):
- Transfer ownership from `sp1` to `sp2`.
- Determine if a shared pointer is the only owner of a resource.
- Swap `sp1` and `sp2`.
- Give up ownership and reinitialise the shared pointer as being empty.

3. (The deprecated Smart Pointer `std::auto_ptr`)

This pointer type is deprecated in C++11 and should not be used in future applications. Consider the following code:

```
using std::auto_ptr;

// Define auto_ptr pointers instead of raw pointers
std::auto_ptr <double> d(new double (1.0));

// Dereference
*d = 2.0;

// Change ownership of auto_ptr
// (after assignment, d is undefined)
auto_ptr <double> d2 = d;
*d2 = 3.0;

cout << "Auto values: " << *d.get() << ", " << *d2.get();
```

Answer the following questions:

a) Type the above code and run it. Why does it give a run-time error?

b) Port to code that uses `std::unique_ptr`. Run and test the new code.

4. (Smart Pointers and STL Algorithms)

In this case we create a simple example of STL containers whose elements are smart pointers.
To this end, consider the following class hierarchy:

```
class Base
{ // Base class
private:

public:
    Base() { }
    virtual void print() const = 0;

protected:
    virtual ~Base() { cout << "Base destructor\n\n"; }
};

class Derived : public Base
{ // Derived class
private:

public:
    Derived() : Base() { }
    ~Derived() { cout << "Derived destructor\n"; }
    void print() const { cout << "derived object\n";}
};
```

Answer the following questions:
a) Create a list of smart pointers to `Base`. In particular, test the above code with both shared and unique pointers. Which option compiles and why does unique pointer not compile?
b) Create a factory function to create instances of `Derived` and then add them to the list.
c) Test the functionality and check that there are no memory leaks.

5. (Custom Deleter)

Shared and unique pointers support deleters. A *deleter* is a callable object that executes some code before an object goes out of scope. A deleter can be seen as a kind of *callback function*. We first give a simple example to show what we mean: we create two-dimensional points as smart pointers. Just before a point goes out of scope a callback delete function will be called:

```
auto deleter = [](Point* pt)-> void
{ std::cout << "Bye:" << *pt; };
```

The corresponding code is:

```
using SmartPoint = std::shared_ptr<Point>;
    SmartPoint p1(new Point(), deleter);
```

We now discuss the exercise to be done. To this end, answer the following questions:
a) The goal of this exercise is to open a file, write some data to the file an then close it when we are finished writing. Under normal circumstances we are able to close the file. However, if an exception occurs before the file can be closed the file pointer will still be open and hence it cannot be accessed. In order to ensure *exception safety* we employ a shared pointer with a delete function in the constructor, for example by using a function object:

```
std::shared_ptr<FILE> mySharedFile(myFile, FileFinalizer());
```

where `FileFinalizer` is a function object.

b) Create a free function and a stored lambda function that also play the role of custom deleters for this problem.
c) Test the code for the three kinds of deleter functions (the delete closes the file).
d) Create a small loop in which records are added to the file; throw an exception at some stage in the loop, catch the exception and then open the file again. Does it work?

6. (Weak Pointers)

A *weak pointer* is an observer of a shared pointer. It is useful as a way to avoid dangling pointers and when we wish to use shared resources without assuming ownership.

Answer the following questions:
a) Create a shared pointer, print the use count and then create a weak pointer that observes it. Print the use count again. What are the values?
b) Assign a weak pointer to a shared pointer and check that the weak pointer is not empty.
c) Assign a weak pointer to another weak pointer; assign a weak pointer to shared pointer. What is the use count in both cases?

2.5 and 2.6 IEEE 754 and C Floating Point Classification

Summary and Goals

The main objective of the exercises in this section is to examine *floating-point arithmetic*, *round-off error* and *machine precision* issues in C++11 and Boost Math Toolkit. We recommend that you experiment with the libraries to get a good feeling for floating-point arithmetic based on IEEE 754.

1. (Classifying Numbers)

In this exercise we use C++11 syntax to categorise `float`, `double`, `long double` and integral types as *zero*, *subnormal*, *normal*, *infinite*, *NaN* or an implementation-defined category.
To this end, employ `std::fpclassify()` as used in the template function:

```
template <typename T>
    const char* Classify(T t)
{
    switch (std::fpclassify(t))
    {
           case FP_INFINITE:  return "Inf";
           case FP_NAN:       return "NaN";
           case FP_NORMAL:    return "normal";
           case FP_SUBNORMAL: return "subnormal";
           case FP_ZERO:      return "zero";
           default:           return "unknown";
    }
}
```

Answer the following questions:

a) Test this function for the following values (in other words, determine the category):

```
double val = std::numeric_limits<double>::max();
    2.0 * val;
    3.1415 + val;
    double val2 = std::numeric_limits<double>::min() - 3.1415;
    val2/2.0;
    DBL_MIN/2.
```

b) Apply the functions `std::isfinite`, `std::isinf`, `std::isnan`, `std::isnormal` to the following values:

```
double factor = 2.0;
val = factor*std::numeric_limits<double>::max();
val - val;
std::sqrt(-1.0);
std::log(0.0);
std::exp(val);
```

2. (Machine Epsilon)

Machine epsilon gives an upper bound on the relative error due to rounding in floating point arithmetic. In C++11, epsilon is the difference between 1.0 and the next representable value of the given floating-point type.

Answer the following questions:

a) Write a small C function to compute epsilon:

```
double epsilon = 1.0;

while ((1.0 + 0.5*epsilon) != 1.0)
{
    epsilon *= 0.5;
}
```

Generalise this function for other types.

b) Compare the value from part a) with the value produced by

```
std::numeric_limits<double>::epsilon().
```

3. (Machine Precision Issues)

This exercise requires that you do some research into how C++ 11 and Boost C++ Math Toolkit supports the following functionality (x is a given value):

- The next representable value that is greater than x. An `overflow_error` is thrown if no such value exists (`float_next`).
- The next representable value that is less than x. An `overflow_error` is thrown if no such value exists (`float_prior`).
- Advance a floating-point number by a specified number of ULP (*Unit in Last Place*) (`float_advance`).
- Find the number of gaps/bits/ULP between two floating-point value (`float_distance`).
- (C++11 and Boost) Return the next representable value of x in the direction y (`std/boost::nextafter(x,y)`).

The Boost header file to include is:

```
#include <boost/math/special_functions/next.hpp>
```

Take some specific values and experiment with these functions. Some examples are:

```
// Number gaps/bits/ULP between 2 floating-point values a and b
// Returns a signed value indicating whether a < b
double a = 0.1; double b =
    a + std::numeric_limits<double>::min();
std::cout << boost::math::float_distance(a, b) << std::endl;
a = 1.0; b = 0.0;
std::cout << boost::math::float_distance(a, b) << std::endl;
```

2.7 C++ System Error

Summary and Goals

The exercises in this section are concerned with defining (platform-independent) *error conditions* and (platform-dependent) *error codes* in both Boost and C++11. These libraries are used in a range of C++ applications involving systems programming, multithreading and parallel programing and network programming. It is for these reasons that we devote some attention to the relevant functionality.
In order to execute each exercise you need to know which classes and functions to use. To this end, we give some hints in each exercise to help you to pinpoint what you need to know.

1. (Boost and C++11 Error Condition)

The objective of this exercise is to open a file by giving the location of the file as a string and using `std::ifstream`. Print each record in the file and then close it.

Answer the following questions:

a) Write the code and test it using an existing file as input. The return type is `boost::system::error condition`.
b) We now need to check if the file exists. To this end, employ the function `make_error_condition` and the *scoped enumeration* `errc`.
c) Test the code with an existing file and a non-existing file.
d) Port the Boost code to C++11 and test your program again.

2. (std::error_condition Fundamentals)

This exercise is concerned with the interface and properties of `std::error_condition`.

Answer the following questions:

a) Create instances of `std::error_condition` based on the following requirements:
   - Default constructor.
   - Value (stored error code) and error category.
   - Based on `std::errc` code.
b) Create a function that returns the following information pertaining to instances of `std::error_condition`:
   - Its error code (an integer).
   - Its message (a string).
   - The name of its category (a string).

The return type is `std::tuple < std::string, int, std::string>`.

3. (Portable Error Conditions corresponding to POSIX Error Codes)

This exercise concerns the class `std::error_condition`, how to instantiate it and how to access some of its properties.

Answer the following questions:

a) Create error condition instances based on the following POSIX error codes (use scoped enumerator `std::errc`):
   - `io_error`.
   - `network_unreachable`.
   - `no_such_file_or_directory`.
   - `not_a_socket`.
   - `permission_denied`.
b) Create an `std::error_condition` object based on the value 128 and the generic error category.
c) Use `std::make_error_condition` (with `std::io_errc` as argument) to create an instance of `std::error_condition`.

4. (Error Code Fundamentals)

The class `std::error_code` models platform-depenendent code.

Answer the following questions:

a) Create a default error code and an error code with a platform-dependent error code value and an error category.

b) Create a function that returns the following information pertaining to instances of `std::error_code`:
   - Its error code (an integer).
   - Its message (a string).
   - The name of its category (a string).

   The return type is `std::tuple < std::string, int, std::string>`.

c) Test equality/inequality of instances of `std::error_code` and `std::error_condition` using operators == and !=.

5. (Catching Exceptions)

This exercise entails that you may need to research and investigate C++11 libraries and functionality. The objective of this exercise is to catch a (simulated) exception thrown when setting the *exception mask* of an input file stream:

```
std::ifstream file(std::string("DOESNOTEXIST.txt"));

try
{
    // Set the exception mask of the file stream
    // In this case 1) logical error on I/O operation or
    // 2) read/write error on I/O operation
    file.exceptions(file.failbit | file.badbit);
}
catch (const std::ios_base::failure& e)
{
    // TBD (see steps!)
}
catch (...)
{
    std::cout << "catch all\n";
}
```

Answer the following questions:

a) In the catch block use `e.code()` with `std::io_errc::stream`.

b) If the comparison in a) is true, then:
   - Create an error code instance `ec1` with `e.code()` as argument.
   - Create an error condition `ec2` instance with `ec1` as argument.
   - Print the properties value, message and error category of `ec2`.

c) Test your code.

2.8 STL Bitset and Boost Dynamic Bitset

Summary and Goals

The objective in this section is to introduce compact data structures to model containers whose elements are bits. To this end, we discuss C++11 *compile-time* bitsets and *dynamic bitsets* in Boost. The two libraries have the same interface for all intents and purposes. This means for example, that if you learn the C++11 bitset class then it is easy to learn Boost dynamic bitset class.

There are many applications of bitsets. A full discussion is outside the current scope but we do give some pointers that might help. A discussion and source code is in "Introduction to the Boost C++ Libraries" (Datasim Press) by Robert Demming and Daniel J. Duffy.

C++98 had support for bitsets and the corresponding functionality will not be discussed here. Note that the rightmost part of a bitset is its least significant bit (extra bits are padded from the left).

1. (STL Bitset 101)

This is a straightforward exercise on understanding and using `std::bitset<T, N>`.

Answer the following questions:

a) Create bitsets from unsigned long and unsigned long long.
b) Create bitsets from full strings (for example, "01010"). Create a bitset from parts of strings based on a start index and the number of bits to use.
c) In the case of strings (for example), use exception handling to check for *out-of-range* values and *invalid arguments* (bits that are neither 0 nor 1).
d) Set the bits in a bitset in different ways:
   - Set/reset all bits.
   - Flip the bits.
   - Test if none, all or any bits are set.
   - Access the elements.
   - Count the number of set bits.
e) Convert a bitset to `std::string`, unsigned long and unsigned long long.
f) Test if two bitsets are equal or unequal.

2. (Boolean Operations on Bitsets)

It is possible to perform bitwise binary and unary operations on bitsets. In the former case we can choose to modify the left-hand operand or we can create a new bitset from the left and right-hand operands.

Answer the following questions:

a) Create two bitsets bs1 and bs2 (they must have the same number of bits, that is the same size).
b) Toggle all the bits of bs1 and bs2;
c) Perform bitwise XOR, OR and AND on bs1 and bs2.
d) Perform right and left shift operations on bs1 and bs2.
e) Use `std::hash` to create hashed values for bs1 and bs2.
f) Investigate how to create *binary literals* in C++ (since C++14) and their relationship with bitsets, for example:

```
auto blit = 0b0011;

std::byte b{ 0b0011 }; // Create from a binary literal
std::bitset<4> bs(blit);
boost::dynamic_bitset<unsigned int> dbs(4); // all 0 by default
dbs[1] = dbs[0] = 1;

std::cout << "\nbyte: " << std::to_integer<int>(b) << '\n';
std::cout << "bitset: " << bs << '\n';
std::cout << "dynamic bitset: " << dbs << '\n';
```

g) Create a byte in different ways: from a binary literal, from an integer, as binary logical operators on existing bytes.
h) Perform bitwise operations on bytes (AND, OR).
i) Perform left and right shift operations on bytes. Consider “extreme cases” of shifting.
j) Can you think of examples/applications where bytes and bitsets can be used, for example raw memory access and savings and performance improvements?

3. (Boost Dynamic Bitset 101)

It is possible to define the length of a dynamic bitset at run-time.

Answer the following questions:

a) Port the code that you created in questions 1 and 2 to `boost::dynamic_bitset<T>`. Test your code.
b) (Changing the size of a bitset at run-time). Apply `resize()`, `clear()`, `push_back()`, `pop_back()` and `append()` to bitset instances.
c) (Set operations). Test the functionality to test if a bitset is a subset (or proper subset) of another bitset.
d) (Searching in bitsets). Use `find_first()` and `find_next()` to search for bits in bitsets.

4. (`std::vector<bool>` versus Bitset)

An alternative to bitsets is to employ the class `std::vector<bool>`. There has been much discussion about the shortcomings of this class (for example, its iterators and it does not necessarily store its elements as a contiguous array).

Answer the following questions:

a) Determine which functionality it supports compared to the two bitset classes discussed here.
b) Create a function to compute the intersection of two instances `std::vector<bool>`.

Having completed the exercise will probably convince you that it is better to use bitset classes instead of `std::vector<bool>`?

5. (Creating Object Adapters for Bitset, Compile-Time)

In this exercise we create a compile-time matrix (call it `BitMatrix<N, M>`) consisting of `N` rows and `M` columns whose elements are bits. Some requirements are:

- The chosen data structure must be efficient (for example, accessing the elements).
- Its interface must have the same look and feel as that of `std::bitset`.
- It must be generic enough to support a range of applications in different domains (for example, Computer Graphics and its many applications).

Answer the following questions:

a) Determine which data structure to use in order to implement `BitMatrix<N, M>`, for example as a nested array `std::array<std::bitset<M>, N>` or a one-dimensional array `std::bitset<N*M>`. Which choice is "optimal" is for you to decide.
b) Constructors need to be created. Use the same defaults as with `std::bitset<M>`.
c) Implement the following for all rows in the matrix and for a given row in the matrix:
   - Set/reset all bits.
   - Flip the bits.
   - Test if none, all or any bits are set.
   - Access the elements.
   - Count the number of set bits.
d) Create member functions for OR, XOR and AND Boolean operations on bit matrices.