Lexical Cast

```
#include "boost\lexical_cast.hpp"
#include <vector>
#include <iostream>
#include <array>

using namespace std;
using boost::lexical_cast;
using boost::bad_lexical_cast;

int main()
{
/*  Convert from string
atof     Convert string to double (function )
atoi     Convert string to integer (function )
atol     Convert string to long integer (function )
atoll    Convert string to long long integer (function )
strtod   Convert string to double (function )
strtof   Convert string to float (function )
strtol   Convert string to long integer (function )
strtold  Convert string to long double (function )
strtoll  Convert string to long long integer (function )
strtoul  Convert string to unsigned long integer (function )
strtoull Convert string to unsigned long long integer (function )
sscanf()

   Convert to string
stringstream strm;
strm << int_val;
string s = strm.str();
sprintf()
itoa  // non-standard
*/

    try
    {
        int s = 345;
        string str = lexical_cast<string>(s);
        str = "Message: " + lexical_cast<string>('A') + "==" +
lexical_cast<string>(34.5);
        cout << str << endl;
        array<char, 64> msg = lexical_cast< array<char, 64> >(23456);

        s = lexical_cast<int>("5678");
        //s = lexical_cast<int>("56.78"); // bad_lexical_cast
        //s = lexical_cast<int>("3456yut");  // bad_lexical_cast
        s = lexical_cast<int>("3456yut", 4);
        cout << s << endl;
    }
    catch(bad_lexical_cast & e)
    {
        cout << "Exception caught:" << e.what() << endl;
    }

}
```

Variant

```
#include "boost\variant.hpp"
#include <vector>
#include <iostream>
#include <array>
#include <string>

using namespace std;

class DoubleVisitor : public boost::static_visitor<> {
public:
   void operator() (int& i) const {
       i += i;
   }
   void operator() (string& str) const {
       str += str;
   }
};

void Double( boost::variant<int, string> u) {
}

int main()
{
//   C union:
   union {int i; float f;} u;   // u contains either int or float
   u.i = 34;
   cout << u.i << endl;
   u.f = 2.3;  // u.i is overwritten
   cout << u.i << endl;  // output garbages
// Problem: it can only work with POD (Plain Old Data)

   //union {int i; string s;} u;   //  Won't compile


// variant is an union for C++
   boost::variant<int, string> u1, u2;
   u1 = 2;
   u2 = "Hello";
   cout << u1 <<  endl;
   cout << u2 << endl;
   //u1 = u1 * 2;  // * is not overloaded for variant
   u1 = boost::get<int>(u1) * 2;    // get() return a reference of the int
                                    // if variant<int*, string>, get() returns
pointer of int

   cout << boost::get<int>(u1) <<  endl;  // output: 64
   //cout << boost::get<string>(u1) << endl;  // crash.  variant is discriminated union
container
   //  if retrieval failed, get() returns a null pointer or throws an exception: bad_get
   u1 = "good";  // u1 becomes a string
   u1 = 32;   // u1 becomes an int again


       // A variant is never empty
   boost::variant<int, string> u3;
   cout << boost::get<int>(u3) <<  endl;
```

```
   // Problem with boost::get(): we don't always know what type is saved in the variant

   //Using visitor
   boost::apply_visitor( DoubleVisitor(), u1 );
   cout << boost::get<int>(u1) <<  endl;  // output: 128

   boost::apply_visitor( DoubleVisitor(), u2 );
   cout << boost::get<string>(u2) << endl; // output: HelloHello

   std::vector< boost::variant<int, string> > arr;
   arr.push_back("good");
   arr.push_back(25);
   arr.push_back("bad");
   for (auto x : arr) {
       boost::apply_visitor( DoubleVisitor(), x);
   }
}
```

Any

```
#include <vector>
#include <iostream>
#include <array>
#include <string>
#include "boost\any.hpp"

using namespace std;

int main() {
   boost::any x, y, z;
   x = string("hello");  // string
       x = 2.34;  // double
   y = 'A';   // char
   z = vector<int>();  // dynamic storage allocation
                                                    // variant uses stack storage, more
efficient

   //cout << y << endl; // won't compile
   cout << boost::any_cast<char>(y) << endl; //  returns a copy of y's data: 'A'
                                  // boost::get() returns a reference
                                  // boost::any_cast() reuturns a copy of data
   cout << boost::any_cast<double>(x) << endl; //  returns a copy of x's data: 2.34
   int i = boost::any_cast<int>(x);  // No static check, only run-time check
                                    // Throws bad_any_cast exception
   cout << boost::any_cast<float>(x) << endl; //  bad_any_cast

   boost::any_cast<vector<int>>(z).push_back(23);
   int i = boost::any_cast<vector<int>>(z).top();  // crash, because z's vector is still
empty


   boost::any p;
   p = &i;  // p is a pointer of int, variant can also store pointers
   int* pInt = boost::any_cast<int*>(p);  // returns a pointer
```

```
   y.empty(); // return true if y is empty
   if (y.type() == typeid(char))   // run-time type check.  variant has compile-time type
check with visitor
        cout << "y is a char" << endl;

   vector<boost::any> many;
   many.push_back(2);
   many.push_back('s');
   many.push_back(x);
   many.push_back(boost::any());


   struct Property {
       string name;
       boost::any value;
   };
   vector<Property> properties;

}



/*
Boost.Variant vs. Boost.Any

As a discriminated union container, the Variant library shares many of the same
features of the Any library. However, since neither library wholly encapsulates
the features of the other, one library cannot be generally recommended for use
over the other.

That said, Boost.Variant has several advantages over Boost.Any, such as:

1. Boost.Variant guarantees the type of its content is one of a finite, user-specified
   set of types.
2. Boost.Variant provides compile-time checked visitation of its content. (By contrast,
   the current version of Boost.Any provides no visitation mechanism at all; but even if
   it did, it would need to be checked at run-time.)
3. Boost.Variant enables generic visitation of its content. (Even if Boost.Any did
provide
   a visitation mechanism, it would enable visitation only of explicitly-specified
types.)
4. Boost.Variant offers an efficient, stack-based storage scheme (avoiding the overhead
of
   dynamic allocation).

Of course, Boost.Any has several advantages over Boost.Variant, such as:
1. Boost.Any, as its name implies, allows virtually any type for its content, providing
   great flexibility.
2. Boost.Any provides the no-throw guarantee of exception safety for its swap operation.
3. Boost.Any makes little use of template metaprogramming techniques (avoiding
potentially
   hard-to-read error messages and significant compile-time processor and memory
demands).
*/
```

Optional

```cpp
#include <vector>
#include <deque>
#include <iostream>
#include <array>
#include <string>
#include "boost\optional.hpp"
#include "boost\variant.hpp"

using namespace std;

deque<char> queue;

//char get_async_data() {
//    if (!queue.empty())
//        return queue.back();
//    else
//        return '\0'; // this is a valid char
//}

boost::optional<char> get_async_data() {
    if (!queue.empty())
        return boost::optional<char>(queue.back());
    else
        return boost::optional<char>();
}

int main() {
    // What we need:
    boost::variant<nullptr_t, char> v;

    // Optional:
    boost::optional<char> op;  // op is uninitalized, no char is constructed
    //op.get();  // assertion failure
    op = 'A';

        op = get_async_data();
    if (!op)  // same as: if (op != 0)
        cout << "op is not initialized" << endl;
    else  {
        cout << "op contains " << op.get() << endl; // get() requires op to be
initialized, otherwise crash (assertion failure)
        cout << "op contains " << *op << endl; // same as get()
    }

    // Remove the if/else check
    op.reset();  // reset op to uninitialzed state
    cout << op.get_value_or('z') << endl;  // if op is unitialized, return 'z', else
return *op

    // Alternatively
    char* p = op.get_ptr();  // return a pointer to contained value, or null if not
initialized

   // optional can store any kind of data
    struct A {string name; int value;};
    A a;
    boost::optional<A> opA;  // constructor of A is not called
    boost::optional<A> opA(a);
```

```
    cout << opA->name << " " << opA->value << endl;

    // Pointer
    boost::optional<A*> opAP(&a);
    cout << (*opAP)->name << " " << (*opAP)->value << endl;

    // Reference
    boost::optional<A&> opAR(a);
    opAR->name = "Bob";  // This changes a

    // Relational Operator
    boost::optional<int> oInt1(9);
    boost::optional<int> oInt2(1);
    if (oInt1 < oInt2)
        cout << "oInt1 is bigger" << endl;  // If both are initialized, compare *oInt1
and *oInt2
                                           // Otherwise, uninitialized optional is
considered smallest

    // Important: optional is not modeled as pointer!!!
}
```