

[Help](#)

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <algorithm>

using namespace std;

#include "math/ImportanceSampling_jl/src/parser.hpp"

template <typename T>
std::ostream& operator<< (std::ostream& out, const std::vector<T>& v)
{
    if ( !v.empty() )
    {
        out << '[';
        std::copy (v.begin(), v.end(), std::ostream_iterator<T>(out, ", "));
        out << "\ b\ b]";
    }
    return out;
}

void TypeVal::print(const std::string &key) const
{
    cout << key << ": ";
    switch (type)
    {
        case T_INT:
            cout << boost::get<int>(Val);
            cout << endl;
            break;
        case T_LONG:
            cout << boost::get<size_t>(Val);
            cout << endl;
            break;
        case T_DOUBLE:
            cout << boost::get<double>(Val);
            cout << endl;
            break;
        case T_STRING:
```

```

        cout << boost::get<string>(Val);
        cout << endl;
        break;
    case T_VECTOR:
        {
            vector<double> v = boost::get< vector<double> >(Val);
            cout << v;
            cout << endl;
        }
        break;
    default:
        break;
    }
}

```

```

TypeVal::TypeVal() : type(T_NULL) { }

```

```

TypeVal::TypeVal(const TypeVal &v)
{
    *this = v;
    // type = v.type;
    // switch(type)
    // {
    //     case T_VECTOR:
    //         Val = boost::get<vector<double> >(v.Val);
    //         break;
    //     case T_STRING:
    //         Val = boost::get<string>(v.Val);
    //         break;
    //     case T_INT:
    //         Val = boost::get<int>(v.Val);
    //         break;
    //     case T_LONG:
    //         Val = boost::get<size_t>(v.Val);
    //         break;
    //     case T_DOUBLE:
    //         Val = boost::get<double>(v.Val);
    //         break;
    //     default:
    //         break;
    // }
}

```

```
}
```

```
TypeVal::~~TypeVal() { }
```

```
TypeVal& TypeVal::operator= (const TypeVal &v)
{
    // Copy data
    type = v.type;
    switch(type)
    {
        case T_VECTOR:
            Val = boost::get<vector<double> >(v.Val);
            break;
        case T_STRING:
            Val = boost::get<string>(v.Val);
            break;
        case T_INT:
            Val = boost::get<int>(v.Val);
            break;
        case T_LONG:
            Val = boost::get<size_t>(v.Val);
            break;
        case T_DOUBLE:
            Val = boost::get<double>(v.Val);
            break;
        case T_PTR:
            Val = boost::get<void *>(v.Val);
            break;
        default:
            cout << "Unknown type " << type << endl;
            break;
    }
    return *this;
}
```

```
Param::Param(const Param &P)
{
    Hash::const_iterator it;
    for (it = P.M.begin() ; it != P.M.end() ; it++)
    {
        M[it->first] = it->second;
    }
}
```

```

        // M.insert(pair<std::string, TypeVal>(it->first, TypeVal(it->second)));
    }
}

Param& Param::operator=(const Param &P)
{
    M.clear();
    M = P.M;
    return *this;
}

Parser::Parser() : Param(), type_ldelim('<'), type_rdelim('>') { }

Parser::Parser(const char *InputFile)
: Param()
{
    type_ldelim = '<';
    type_rdelim = '>';
    ReadInputFile(InputFile);
}

/* reads the file up to the end. Leading blanks are stripped
 * and lines beginning with # are ignored */
void Parser::ReadInputFile(const char *InputFile)
{
    FILE *fic;
    int j;
    char RedLine[MAX_CHAR_LINE];

    for (j = 0 ; j < MAX_CHAR_LINE ; j++) RedLine[j] = '\0';

    if ((fic = fopen(InputFile, "r")) == NULL)
    {
        printf("Unable to open Input File %s\ n", InputFile);
        exit(1);
    }

    j = 0;
    int last_non_blank = 0;

```

```

while (true)
{
    char c;
    if ((c = fgetc(fic)) == EOF) break;
    switch (c)
    {
        case '#':
            while ((c = fgetc(fic)) != '\ n');
            j = 0;
            last_non_blank = 0;
            break;
        case '\ n':
            if (j > 0)
            {
                // Trim ending whitespaces
                RedLine[last_non_blank + 1] = '\ 0';
                add(RedLine);
            }
            j = 0;
            last_non_blank = 0;
            break;
            // Trim leading whitespaces
        case ' ':
            if (j > 0 && RedLine[j - 1] != ' ')
            {
                RedLine[j] = c;
                j++;
            }
            break;
        default:
            RedLine[j] = c;
            last_non_blank = j;
            j++;
            break;
    }
}
fclose(fic);
}

```

```

bool Param::check_if_key(Hash::const_iterator &it, const std::string &key) const
{

```

```

    // it = M.find(const_cast<char *>(key));
    it = M.find(key);
    if (it == M.end())
    {
        // cout << key << " entry is missing" << endl;
        return false;
    }
    return true;
}

/**
 * Set out according to P
 *
 * @param key the key to be looked for in the map
 * @param out (output) set to the value associated to key in the map
 * @param size size of the vector to be stored
 * @param go_on a boolean, if false and key is not found in the map, the
 * abort function is called
 */
bool Param::extract(const std::string &key, PnlVect *&out, int size, bool go_on)
{
    Hash::const_iterator it;
    out = NULL;

    if (check_if_key(it, key) == false)
    {
        if (!go_on)
        {
            std::cout << "Key " << key << " not found." << std::endl;
            abort();
        }
        return false;
    }
    vector<double> v;
    try
    {
        v = boost::get<vector<double> >(it->second.Val);
    }
    catch (boost::bad_get e)
    {
        std::cout << "Boost bad get for " << key << std::endl;
    }
}

```

```
        abort();
    }

    if (v.size() == 1)
    {
        out = pnl_vect_create_from_double(size, v[0]);
    }
    else
    {
        if (v.size() != (unsigned long) size)
        {
            cout << key << " size mismatch for vector " << key << endl;
            abort();
        }
        out = pnl_vect_create_from_ptr(size, v.data());
    }
    return true;
}
```

```
static vector<double> charPtrTovector(const char *s)
{
    const char *p = s;
    char *q;
    int len = 0;

    while (true)
    {
        strtod(p, &q);
        if (p == q) break;
        len ++;
        p = q;
    }
    vector<double> v(len);

    p = s;
    for (int i = 0 ; i < len ; i++)
    {
        v[i] = strtod(p, &q);
        p = q;
    }
}
```

```
    }

    return v;
}

Param::~Param() { M.clear(); }

void Parser::add(char RedLine[])
{
    char *type_str, *key_str, *val_str;
    int type_len = 0, key_len = 0;

    key_str = RedLine;
    while (*key_str == ' ') key_str++; // Trim leading spaces for the key
    while (key_str[key_len] != type_ldelim)
    {
        key_len++; // Find left delimiter for the type
    }
    type_str = key_str + key_len + 1; // Start of the type string
    key_len--;
    while (key_str[key_len] == ' ')
    {
        key_len--; // Trim ending spaces for the key
    }

    while (*type_str == ' ') key_str++; // Trim leading spaces for the type
    while (type_str[type_len] != type_rdelim)
    {
        type_len++; // Find left delimiter for the type
    }
    val_str = type_str + type_len + 1; // Start of the val string
    type_len--;
    while (type_str[type_len] == ' ')
    {
        type_len--; // Trim leading spaces for the type
    }
    while (*val_str == ' ')
    {
        val_str++; // Trim leading spaces for the value
    }
}
```



```
type_str[type_len + 1] = '\\ 0';
key_str[key_len + 1] = '\\ 0';
TypeVal maturity;
if (strcmp(type_str, "int") == 0)
{
    int x;
    sscanf(val_str, "%d", &x);
    maturity.type = T_INT;
    maturity.Val = x;
}
else if (strcmp(type_str, "long") == 0)
{
    size_t x;
    sscanf(val_str, "%zd", &x);
    maturity.type = T_LONG;
    maturity.Val = x;
}
else if (strcmp(type_str, "float") == 0)
{
    double x;
    sscanf(val_str, "%lf", &x);
    maturity.type = T_DOUBLE;
    maturity.Val = x;
}
else if (strcmp(type_str, "string") == 0)
{
    maturity.type = T_STRING;
    maturity.Val = string(val_str);
}
else if (strcmp(type_str, "vector") == 0)
{
    maturity.type = T_VECTOR;
    maturity.Val = charPtrTovector(val_str);
}
else
{
    fprintf(stderr, "Unknown type: %s", type_str);
    abort();
}
std::string key_string(key_str);
// std::transform(key_string.begin(), key_string.end(), key_string.begin(), ::
```

```
    M[key_string] = maturity;  
}
```

```
Parser::~~Parser() { }
```