Chroma tutorial
Part 2

Christian Ehmann

Universität Regensburg

21 December 2007

Outline

1. Reminder
2. Speaking XML
3. The world beyond XML
4. Summary

Outline

1 Reminder

2 Speaking XML

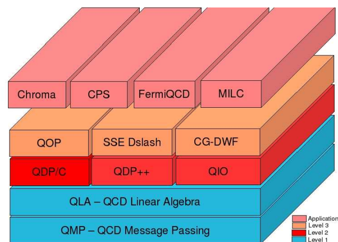
3 The world beyond XML

4 Summary

The Chroma Software System

- computational toolbox for Lattice QCD applications
- aim: flexible, portable, efficient
- development started at JLab (R.G. Edwards), later on joined by UKQCD collaboration (B. Joo)
- Chroma relies on several layers of the SciDAC software hierachy
- effort to establish a common basis for Lattice (QCD) computations
- arXiv:hep-lat/0409003, http://www.ph.ed.ac.uk/ paboyle/bagel/Bagel.html

The SciDAC project



- QMP : QCD Message Passing
- QLA : QCD Linear Algebra
- QDP/QDP++ : QCD Data Parallel
- QI/O : QCD I/O
- Level 3 : special optimized software

Application areas

- measurements: (sequential) sources, quark-/link- smearing, propagators, 2pt-/3pt-functions, wilson loops, spectroscopy, eigenvalues, ...
- fermion actions: (un-)preconditioned (Clover-)Wilson, twisted mass, overlap, domain wall, staggered, ...
- gauge actions: Wilson, plaquette/rectangle/parallelogram, tree level and 1-loop Luscher-Weisz, RG, ...
- I/O formats: NERSC, CPPACS, UKY, SciDAC, ILDG, ...
- separate software systems delivered with chroma to generate gauge configurations: HMC (Hybrid Monte Carlo) and purgauge (Pure Gauge Heatbath)

Ups & Downs

- highly portable, can be compiled on various architectures (Intel x86, PowerPC etc.) and topologies (workstation, cluster, QCDOC crosscompilaton, BG/L)
- straigthforward compilation on scalar machines
- convenient use with XML I/O files, in principle no programming knowledge neccessary
- separate programs to analyse XML output available (ADAT)
- all parallelization behind the scenes (done by Level 1 layer)

Ups & Downs

- cumbersome XML output
- compilation elaborate for QCDOC, BG/L, ...
- almost no documentation, only ChangeLog/Doxygen (however, good documentation for QDP++)
- hard C++ stuff when one wants to implement own routines
- overhead, both in time and space

Outline

1 Reminder

2 Speaking XML

3 The world beyond XML

4 Summary

Using Chroma in the traditional way

- first step: compile chroma yourself or use precompiled versions on /psi_devel
- if you have no idea how to write an XML input file for a specific measurement look at the chroma webpage
- no documentation but example input files under http://usqcd.jlab.org/usqcd-software/chroma/chroma/tests/
- tinker your own input files based on the examples
- tag names often self-explanatory, otherwise one has to look in the code
- execute /psi_devel/chroma-scalar/bin/chroma -i run.ini.xml -o run.out.xml

Explicit example

Goal: Calculate the pion and rho correlator with local source and smeared sink
Tasks to do (so called Inline Measurements):

- read in the configuration
- construct the source
- invert on the source
- prepare the sink
- compute the correlation function

Outline

1 Reminder

2 Speaking XML

3 The world beyond XML

4 Summary

Hacking Chroma

- broad spectrum of applications covered, but not all
- what to do if you want capacity not implemented yet?
- write an email to Edwards and ask him to implement it for you
- alternatively: hack chroma

Object-oriented programming

- chroma is written in C++ and is (therefore ?!) a fully object-orientated library
- to hack chroma one has to know at least the main ideas of OOP
- goal: discrete units of programming logic and re-usability
- “quarks” of OOP: encapsulation, inheritance, polymorphism

My pet

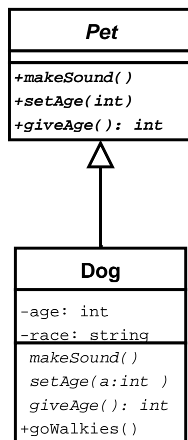
20.12.2007 1

```
class Dog {                             // class declaration
   public:
          int age;                      // member variable
          string race;
          void makeSound(){cout << "Wau";};
};

int main(){
 Dog Waldi;
 Waldi.race = "shepherd";               // instantiation
 Waldi.age = 8;
 Waldi.makeSound();
}
```

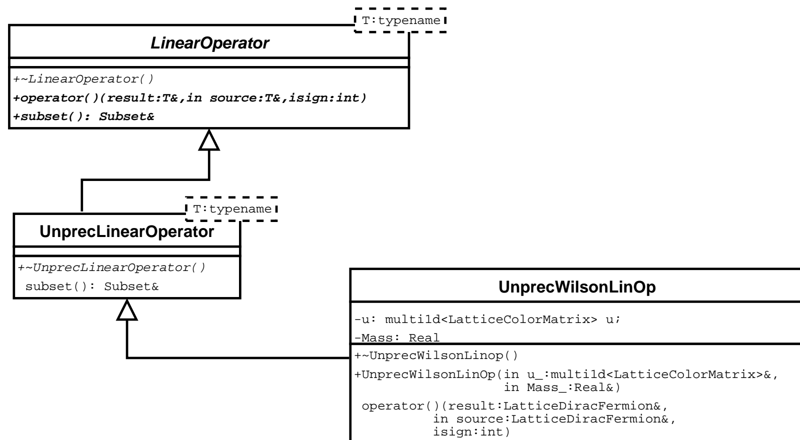
My pet family

```
21.12.2007                                                                          1
class Pet {                                         // abstract class
  public:
       virtual void makeSound() = 0;                // abstract functions
       virtual void setAge(int) = 0;
       virtual int giveAge() = 0;
};

class Dog : public Pet {                             // "Dog" inherits from "Pet"
 public:
      Dog() {};                                      // default constructor
      Dog(int a, string r){ age=a; race=r; };        // constructor
      ~Dog() {};                                     // destructor
      virtual void makeSound() { cout << "Wau"; };   // concret implementation
      virtual void setAge(int a) { age=a; };
      virtual int giveAge() { return age; };
      virtual goWalkies(){...};
 private:
      int age;                                       // member variables
      string race;
};

int main()
{
 Dog Waldi(4,"Pudel");
 Waldi.setAge(5);                                    // not allowed: Waldi.age=5; !
 cout << Waldi.giveAge() << endl;
}
```

UML diagramm



UML diagramm



My action

20.12.2007 1

```
template<typename T>                                          // T is fermiontype
class LinearOperator {
public:
virtual ~LinearOperator() {};
virtual void operator()
        (T& result, const T& source, int isign) const = 0;  // renders class a functor
virtual Subset& subset() const = 0;                           // define a subset
};

template<typename T>
class UnprecLinearOperator : public LinearOperator<T> {
public:
virtual ~UnprecLinearOperator() {};
virtual void operator()
        (T& result, const T& source, int isign) const = 0;
Subset& subset() const {return all};                          // implementation
};
```

My action

```
20.12.2007                                                                  1
template<typename T>
class UnprecWilsonLinOp : public UnprecLinearOperator<LatticeDiracFermion> {
public:
~UnprecWilsonLinOp() {};
UnprecWilsonLinop(const multi1d<LatticeColorMatrix>& u_, const Real& Mass) :
                  u(u_), Mass(Mass_) {};
void operator() (LatticeDiracFermion& result,
                 const LatticeDiracFermion& source, int isign) const;
private:
 multi1d<LatticeColorMatrix> u;
 Real Mass;
};

UnprecWilsonLinOp M(u,Mass);            // create an instance
M(solution, source, isign);
```

My Inverter

```
21.12.2007                                                              1
void InvMR(const LinearOperator<LatticeDiracFermion>& M,   //pass reference to base class
const LatticeDiracFermion& source,
LatticeDiracFermion& target,
const Real& MRovpar,
const Real& RsdMR,
int MaxMR,
int isign,
int& n_count,
Double& resid);

InvMR(M, source, ...);
```

Design patterns

- general repeatable solution to commonly occuring problems
- find more in “Modern C++ Design, Generic Programming and Design Patterns Applied” by Andrei Alexandrescu
- based on the LOKI library
- widly used in chroma: factory, functor, handle, singleton

Handles (Smart Pointers)

- reference counting
- assignment/copy of handle increases ref. count
- destruction of handle decreases ref. count
- when ref. count reaches zero destructor is called
- templated to be able to wrap any pointer
- example:

```
20.12.2007                                                                1

{                                                          // count = 0;
 Handle <LinearOperator <LatticeDiracFermion> > M2;        // count = 0;
 {
  Handle <LinearOperator <LatticeDiracFermion> > M1
                  = new UnprecWilsonLinOp(u,Mass);        // count = 1;
  M2=M1;                                                  // count = 2;
 }                                                        // count = 1;
}
```

Singletons

- An entitiy of which there is only one within a program
- kind of “virtuous global object”
- realized through static methods
- used for e.g. factories, qdp++ memory allocator, ...
- example:

21.12.2007 1

```
typedef SingletonHolder< MyClass, ... > TheMySingleton;   // common: The/the...
TheMySingleton::Instance().memberFunction();
```

Singleton implementation

20.12.2007 1

```
class Singleton
 {
 public:
     static Singleton* Instance();
 protected:
     Singleton();
     Singleton(const Singleton&);
     Singleton& operator= (const Singleton&);
 private:
     static Singleton* pinstance;
 };

 Singleton* Singleton::pinstance = 0;// initialize pointer
 Singleton* Singleton::Instance ()
 {
   if (pinstance == 0)  // is it the first call?
   {
     pinstance = new Singleton; // create sole instance
   }
   return pinstance; // address of sole instance
 }
 Singleton::Singleton()
 {    //... perform necessary instance initializations  }
 };
```

Factory

- we want to choose the type of fermion action, boundary condition, ... at runtime
- problem: C++ is a statically typed language
- type must be known at compile time, e.g.:
  Base* pB = new Derived;
- classes and objects are different beasts in C++: classes created by the programmer, objects created by the program
- solution: object-type-object trade

Factory

- naive implementation with switch statement:

20.12.2007 1

```
Handle<FermAct> ferm_factory(string t)
{
read(xml, "/FermAct/Type", t);
FermAct *fermact;
switch(t) {
 case WILSON:
      fermact = WilsonFermAct::create();
      break;
 case OVERLAP:
      fermact = OverlapFermAct::create();
      break;
 default:
      QDPIO::cerr << "Unknown Fermion Action" << endl;
      QDP_abort(1);
};
Handle<FermAct> fermact_handle(fermact);
return fermact_handle;
}
```

Factory

- criticism:
    - for every new type of fermion action I have to edit both the source file for the action and the factory function
    - factory function must contain all possible implementations
    - switch statement may become unmanageably long
- much better way: use a map (= associative array)
- map from a string to a factory:
  ”Wilson” $\longrightarrow$ WilsonFermAct* WilsonFermAct::create()

Maps

- details of creation localized in the map
- recommended: global map implementet as a singleton
- chroma relies on the LOKI implementation
- Example for a map:

21.12.2007 1

```
std::map<std::string, FermAct* (*) (void)> ferm_factory_map;
ferm_factory_map.insert( make_pair(''Wilson'', WilsonFermAct::create() ) );
ferm_factory_map.insert( make_pair(''Overlap'', OverlapFermAct::create() ) );
std::string fermact_type;
read(xml, ''/FermAct/Type", fermact_type);
Handle<FermAct> fermact_handle( (ferm_factory_map[fermact_type])());
```

Factory implementation

in xxx_factory.h
(e.g. chroma/lib/actions/ferm/fermacts/fermact_factory_w.h):

21.12.2007 1

```
typedef SingletonHolder<                                    // singleton pattern
 ObjectFactory<                                             // factory template
   FermionAction<LatticeFermion,                            // product type
   multi1d<LatticeColorMatrix>,multi1d<LatticeColorMatrix> >,
   std::string,                                             // product id
   TYPELIST_2(XMLReader&, const std::string&),              // parameter for creation
   FermionAction<LatticeFermion,                            // creation function type
       multi1d<LatticeColorMatrix>,
       multi1d<LatticeColorMatrix> >* (*)(XMLReader&,const std::string&),
   StringFactoryError> >                                    // error type
 TheFermionActionFactory;
```

Factory implementation

in xxx_product.h
(e.g. chroma/lib/actions/ferm/fermacts/unprec_wilson_fermact_w.h):

```
21.12.2007                                                              1

namespace UnprecWilsonFermActEnv
{
  extern const std::string name;
  bool registerAll();
}
class UnprecWilsonFermAct : public UnprecWilsonTypeFermAct<LatticeFermion,
                             multi1d<LatticeColorMatrix>, multi1d<LatticeColorMatrix> >
{
public: ...
```

Factory implementation

in xxx_product.cc
(e.g. chroma/lib/actions/ferm/fermacts/unprec_wilson_fermact_w.cc):

20.12.2007 1

```
namespace UnprecWilsonFermActEnv
 {
    WilsonTypeFermAct<LatticeFermion, multi1d<LatticeColorMatrix>,
                      multi1d<LatticeColorMatrix> >* createFermAct(XMLReader& xml_in,
                                                              const std::string& path)
    {
      return new UnprecWilsonFermAct(CreateFermStateEnv::reader(xml_in, path),
                                     WilsonFermActParams(xml_in, path));
    }
    const std::string name = "UNPRECONDITIONED_WILSON";
    static bool registered = false;
    bool registerAll()  {
      bool success = true;
      if (! registered) {
        success &=                              // reference the registered symbol
         Chroma::TheWilsonTypeFermActFactory::Instance().registerObject(name, createFermAct);
        registered = true;
      }
      return success;
    }
 }..
```

Linkage issues

- registered symbols have to be referenced, otherwise xxx_product.o is not linked
- solution (aka hack): linkageHack() function in chroma.cc
- in linkageHack() all registrered products explicitly get referenced
- yet not ideal solution, equivalent of big switch statement
- for convenience: aggregation files xxx_aggregate.h/.cc

20.12.2007 1

```
namespace InlineHadronAggregateEnv
  ...
  bool registerAll()
  {
    bool success = true;
    if (! registered)
    {
      success &= WilsonTypeFermActsEnv::registerAll();
      success &= InlineFermStateEnv::registerAll();
      success &= InlineMakeSourceEnv::registerAll();
      success &= InlinePropagatorEnv::registerAll();
      ...
```

Outline

1 Reminder

2 Speaking XML

3 The world beyond XML

4 Summary

Summary & Outlook

- covered: implemented moduls, I/O, XML, compiling, running, hacking
- not covered: generating Makefiles for chroma, dozens of qdp functions, PETE, ADAT, Bagel/ Wilson Dslah
- future: resolve linkage issue, SSE3 support, improving PETE, even more measurements,...
- you're welcome to contribute to chroma!

Merry Christmas!