Overture Developers Guide
Version 1.0 William D. Henshaw Centre for Applied Scientific Computing

Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture September 26, 2012 UCRL-MA-134300

Abstract: This document provides information for developers of Overture software.

Contents

1 Naming Conventions

general identifiers : are written without underscores, using embedded captials to denote the beginning of new words. Examples **numberOfDimensions**, **getClassName**, **inverseVertexDerivative**. All identifiers other than class names are **not** capitalized.

class names : begin with an initial capital, examples: **SquareMapping**, **MappedGridOperators**, **GridCollection**.

publically visible identifiers : such as class names and public member function names should avoid introducing short forms. For example, we prefer **numberOfComponentGrids()** to **numGrids()** and **realMappedGridFunction** to **realMGF**.

macro names : should be in all capitals with words separated by underscores.

enumerators : The name of the enumerator should be capitalized (?) while the actual enumerators should follow the rules for general identifiers.

2 Overture Types

2.1 Type names

The following are the suggested type names to use, found in **Overture/include/OvertureTypes.h**

bool : is always defined even on machines that do not yet support the standard.

Real, real : is either a float or double, depending on whether Overture is compiled in single precision or double precision. Normally you should never declare a **float** or **double**, use **Real** instead so that your code is precision independent.

IntegerArray : is a serial (non-distributed array), equal to **intSerialArray** in P++ (the type **intSerialArray** does not exist in A++ ?).

IntegerDistributedArray, intArray : is a distributed array.

RealArray : is a serial (non-distributed array), equal to **floatSerialArray** or **doubleSerialArray**.

RealDistributedArray, realArray : is a distributed array.

2.2 Enums and constants

Here are some useful enum's and constants some of which are found in **Overture/include/wdhdefs.h**

REAL_EPSILON : use instead of **FLT_EPSILON** or **DBL_EPSILON** to have precision independent code. The other machine constants, such as **REAL_RADIX**, **REAL_ROUNDS**, **REAL_MANT_DIG**, **REAL_MAX**, **REAL_MIN** , and **REAL_MIN_EXP** , are defined in a similar way

axis1, axis2, axis3 : are just equal to 0,1,2 and can be used to name the coordinates axes.

Start, End are just equal to 0,1 and can be used to name the start and end of a coordinate axis.

Pi, twoPi : **Pi**$= \pi$ and **twoPi**$= 2\pi$.

getCPU() : returns the current cpu count in seconds.

char* sPrintF(char *s, const char *format, ...) : like **sprintf** but returns the character string.

int sScanF(const String & s, const char *format, ...) like **sscanf** but removes commas from the input String 's' (so that the 'format' string can assume there are no commas separating input fields)and converts '

2.3 Global objects

Here are some useful global variables some of which are found in **Overture/include/wdhdefs.h** These can be used as a default arguments, for example.

Index nullIndex : a null Index.

Range nullRange : a null Range.

String nullString

String blankString

IntegerArray nullIntArray

RealArray nullRealArray

MappingParameters nullMappingParameters

GraphicsParameters defaultGraphicsParameters

PlotStuffParameters defaultPlotStuffParameters

BoundaryConditionParameters defaultBoundaryConditionParameters

3 Common pitfalls and useful tricks

avoid scalar indexing of arrays where possible. In particular avoid scalar indexing of distributed arrays as this will be slow in the parallel environment. Be aware of the **seqAdd**, **where** and **sum** A++ operations.

evaluate dangling A++ expressions since an expression is a temporary until it hits an equals operator, copy constructor or is evaluated.

```
floatArray a,b;
....
myFunction( a(I1,I2)+b(I1,I2) );             // ***** WRONG WAY ******
myFunction( evaluate(a(I1,I2)+b(I1,I2)) );   // the right way
realArray & c = evaluate(a(I1,I2)+b(I1,I2));
realArray d = a(I1,I2)+b(I1,I2);             // this is ok since the copy constr
```

watch out for referencing a view In the example below the array **c** will be a reference to the view **a(I)**. It will **NOT** be a copy.

```
floatArray a;
....
realArray c = a(I);     // *AVOID* This is a reference to a view on most compile
realArray c(a(I));      // This is a reference to a view
realArray & d = a(I);   // This is a reference to a view
realArray d; d = a(I);  // This is a copy
```

cast MappedGridFunctions to arrays : when writing long expressions that use array indexing, so the compiler can avoid an extra level of function calls in the parenthesis operator ().

```
realMappedGridFunction velocity(...);
...
realArray & u = velocity;
...
u(I1,I2,I3,0)=u(I1+1,I2,I3,0)*u(I1+1,I2,I3,1)+...;
```

Reference views for readability : when a view appears many times in expressions.

```
const realArray & a1 = b(I1+1,I2,I3);
const realArray & a2 = b(I1+1,I2+1,I3);
const realArray & a3 = b(I1+1,I2+1,I3+1);

c = a1*a2+a3;
d = a2+a3*a1;
```

Reuse a where mask for efficiency : An A++ logical expression just builds an intArray holding ones and zeros. Instead of computing the expression twice as in

```
where( a(I)>0. && a(I-1)>7. )
  b(I)=c(I)+3;
...
where( a(I)>0. && a(I-1)>7. )
  c(I)=7;
```

you can just build the expression first and then reuse it as in

```
intArray mask = a(I)>0. && a(I-1)>7.;
where( mask )
 b(I)=C(I)+3;
...
where( mask )
 c(I)=7.;
```

Avoid derivation from most Overture Classes : avoid the temptation to derive a new class from most Overture classes such as A++ array's, MappedGridFunction's, MappedGrid's, GridCollections's etc. Derived classes are usually hard to support and experience shows that many levels of derivation is evil. Of course some classes such as Mapping's or the ReferenceCounting class are explicitly meant to be derived from; this is clear from the documentation.

4 Parallel

When compiling with P++, one cannot mix serial and distributed array operations. The basic rule of thumb is that small arrays are serial and big arrays (e.g. those that live on a MappedGrid) are distributed.