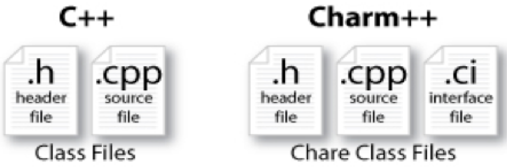
Charm++ File structure

- C++ objects (including Charm++ objects)
  - Defined in regular `.h` and `.cpp` files
- Chare objects, entry methods (asynchronous methods)
  - Defined in `.ci` file
  - Implemented in the `.cpp` file



Charm Interface: Modules

- Charm++ programs are organized as a collection of modules
- Each module has one or more chares
- The module that contains the *mainchare*, is declared as the `mainmodule`
- Each module, when compiled, generates two files: `<modulename>.decl.h` and `<modulename>.def.h`

```
[main]module <modulename> {
  //... chare definitions ...
};
```

Charm Interface: Chares

- Chares are parallel objects that are managed by the RTS
- Each chare has a set *entry methods*, which are asynchronous methods that may be invoked remotely
- The following code, when compiled, generates a C++ class `CBase_<charename>` that encapsulates the RTS object
- This generated class is extended and implemented in the `.cpp` file

```
[main]chare <charename> {
  //... entry method definitions ...
};

class MyChare : CBase_<charename> {
  //... entry method implementations ...
};
```

Charm Interface: Entry Methods

- Entry methods are C++ methods that can be remotely and asynchronously invoked by another chare

```
entry <charename>(); /* constructor entry method */
entry void foo();
entry void bar(int param);

<charename>::<charename>() { /*... constructor code ...*/ }

<charename>::foo() { /*... code to execute ...*/ }

<charename>::bar(int param) { /*... code to execute ...*/ }
```

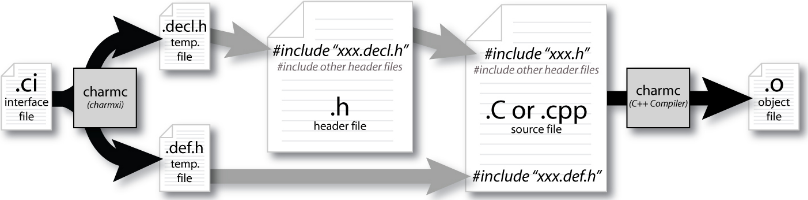
Charm Interface: `mainchare`

- Execution begins with the `mainchare`'s constructor
- The `mainchare`'s constructor takes a pointer to system-defined class `CkArgMsg`
- `CkArgMsg` contains `argv` and `argc`
- The `mainchare` will often construct other parallel objects and then wait for them to finish

Charm Termination

- There is a special system call `CkExit()` that terminates the parallel execution on all processors (but it is called on one processor) and performs the requisite cleanup
- The traditional `exit()` is insufficient because it only terminates one process, not the entire parallel job (and will cause a hang)
- `CkExit()` should be called when you can safely terminate the application (you may want to synchronize before calling this)

Compiling a Charm++ Program



Hello World Example

- `hello.ci` file

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
};
```

- `hello.cpp` file

```
#include <stdio.h>
#include "hello.decl.h"

struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};

#include "hello.def.h"
```

Hello World Example

- Compiling
  - `charmc hello.ci`
  - `charmc -c hello.cpp`
  - `charmc -o hello hello.cpp`
- Running
  - `./charmrun +p7 ./hello`
  - The +p7 tells the system to use seven cores

Creating a Chare

- A chare declared as `chare <charename> {...};` can be instantiated by the following call:

```
CProxy_<charename>::ckNew(... constructor arguments ...);
```

- To communicate with this class in the future, a *proxy* to it must be retained

```
CProxy_<charename> proxy =
  CProxy_<charename>::ckNew(... constructor arguments ...);
```

Chare Creation Example: `.ci` file

```
mainmodule MyModule {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };

  chare Simple {
    entry Simple(int x, double y);
  };
};
```

Chare Creation Example: .cpp file

```
#include <stdio.h>
#include "MyModule.decl.h"

struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    ckout << "Hello World!" << endl;
    if (m->argc > 1) ckout << " Hello " << m->argv[1] << "!!!" << endl;
    double pi = 3.1415;
    CProxy_Simple::ckNew(12, pi);
  };
};
struct Simple : public CBase_Simple {
 Simple(int x, double y) {
   ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
   ckout << "Area of a circle of radius" << x << " is " << y*x*x << endl;
   CkExit();
 }
};

#include "MyModule.def.h"
```

Asynchronous Methods

- Entry methods are invoked by performing a C++ method call on a chare's proxy

```
CProxy_<charename> proxy =
  CProxy_<charename>::ckNew(... constructor arguments ...);

proxy.foo();
proxy.bar(5);
```

- The `foo` and `bar` methods will then be executed with the arguments, wherever `<charename>` happens to live
- The policy is one-at-a-time scheduling (that is, one entry method on one chare executes on a processor at a time)

Asynchronous Methods

- Method invocation is not ordered (between chares, entry methods on one chare, etc.)!
- For example, if a chare executes this code:

```
CProxy_<charename> proxy = CProxy_<charename>::ckNew();
proxy.foo();
proxy.bar(5);
```

- These prints may occur in **any** order

```
<charename>::foo() {
  ckout << "foo executes" << endl;
}

<charename>::bar(int param) {
  ckout << "bar executes with " << param << endl;
}
```

Asynchronous Methods

- For example, if a chare invokes the same entry method twice:

```
proxy.bar(7);
proxy.bar(5);
```

- These may be delivered in **any** order

```
<charename>::bar(int param) {
  ckout << "bar executes with " << param << endl;
}
```

- Output

```
bar executes with 5
bar executes with 7
```

OR

```
bar executes with 7
bar executes with 5
```

Asynchronous Example: .ci file

```
mainmodule MyModule {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
  chare Simple {
    entry Simple(double y);
    entry void findArea(int radius, bool done);
  };
};
```

Asynchronous Example: .cpp file

- Does this program execute correctly?

```
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    double pi = 3.1415;
    CProxy_Simple sim = CProxy_Simple::ckNew(pi);
    for (int i = 1; i< 10; i++) sim.findArea(i, false);
    sim.findArea(10, true);
  };
};

struct Simple : public CBase_Simple {
 float y;
 Simple(double pi) {
   y = pi;
   ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
 }
 void findArea(int r, bool done) {
   ckout << "Area of a circle of radius" << r << " is " << y*r*r << endl;
   if (done) CkExit();
 }
};
```

Data types and entry methods

- You can pass basic C++ types to entry methods (`int`, `char`, `bool`, etc.)
- C++ STL data structures can be passed by including `pup_stl.h`
- Arrays of basic data types can also be passed like this:

```
entry void foobar(int length, int data[length]);

<charename>::foobar(int length, int* data) {
 // ... foobar code ...
}
```

Chare Proxies

- A chare's own proxy can be obtained through a special variable `thisProxy`
- Chare proxies can also be passed so chares can learn about others
- In this snippet, `<charename>` learns about a chare instance `main`, and then invokes a method on it:

```
entry void foobar2(CProxy_Main main);

<charename>::foobar2(CProxy_Main main) {
  main.foo();
}
```

Chare Proxy Example

```
mainchare Main {
  entry Main(CkArgMsg *m);
  entry void finished();
};
chare Simple {
  entry Simple(CProxy_Main mainProxy);
};


struct Main : public CBase_Main {
  Main(CkArgMsg *m) {
    CProxy_Simple::ckNew(thisProxy);
  }
  void finished() { CkExit(); }
};
struct Simple : public CBase_Simple {
  Simple(CProxy_Main mainProxy) {
    ckout << "Hello from Simple" << endl;
    mainProxy.finished();
  }
};
```

Readonlys

- A *readonly* is a global (within a module) read-only variable that can only be written to in the `mainchare`'s constructor
- Can then be read (**not written!**) by any chare in the module
- It is declared in the `.ci` file:

```
readonly <type> <name>;
readonly CProxy_Main mainProxy;
readonly int numChares;
```

- And defined the the `.cpp` file:

```
<type> <name>;
CProxy_Main mainProxy;
int numChares;
```

- And set in the `mainchare`'s constructor

```
<charename>::<charename>(CkArgMsg *m) {
  mainProxy = thisProxy;
  numChares = 10;
}
```

PI Example

```
mainmodule MyModule {

  readonly CProxy_Master mainProxy;

  mainchare Master {
    entry Master(CkArgMsg *m);
    entry void addContribution(int numIn, int numTrials);
  };

  chare Worker {
    entry Worker(int numTrials);
  };

};
```

PI Example

```
CProxy_Master mainProxy; // readonly

struct Master: public CBase_Master {
  int count, totalInsideCircle, totalNumTrials;
  Master(CkArgMsg* m) {
    int numTrials = atoi(m->argv[1]), numChares = atoi(m->argv[2]);
    if (numTrials % numChares) {
      ckout << "Need numTrials to be a divisible by numChares.. Sorry" << endl;
      CkExit();
    }
    for (int i = 0; i < numChares; i++)
      CProxy_Worker::ckNew(numTrials/numChares);
    count = numChares; // wait for count responses.
    mainProxy= thisProxy;
    totalInsideCircle = totalNumTrials = 0;
  };

  void addContribution(int numIn, int numTrials) {
    totalInsideCircle += numIn;
    totalNumTrials += numTrials;
    count--;
    if (count == 0) {
      double myPi = 4.0* ((double) (totalInsideCircle))
        / ((double) (totalNumTrials));
      ckout << "Approximated value of pi is:" << myPi << endl;
      CkExit();
    }
  }
};
```

PI Example

```
struct Worker : public CBase_Worker {
  float y;
  Worker( int numTrials) {
    int inTheCircle = 0;
    double x, y;
    ckout << "Hello from a simple chare running on " << CkMyPe() <<
         endl;

    for (int i=0; i< numTrials; i++) {
      x = drand48();
      y = drand48();
      if ((x*x + y*y) < 1.0)
        inTheCircle++;
    }
    mainProxy.addContribution(inTheCircle, numTrials);
  }
};
```