The Compositional C++ Language Definition

Peter Carlin   Mani Chandy   Carl Kesselman

March 12, 1993

Revision 0.95 3/12/93, Comments welcome.

Abstract

This document gives a concise definition of the syntax and semantics of CC++. Knowledge of the C++ language definition and the C++ language reference manual is assumed.

1 Introduction

Compositional C++, (CC++) is a parallel programming language based on the sequential language C++. The extensions in CC++ enable one to construct reliable parallel libraries that integrate a range of parallel programming paradigms, such as task parallel and data parallel programming styles. The parallel programming extensions of CC++ incorporate the fundamental ideas from compositional programming: synchronization variables and parallel composition. Our goal in designing CC++ was to introduce the concepts of compositional programming into C++, while maintaining the ability to execute existing C and C++ programs without modification.

An overview of CC++ can be found in a companion report [2]. In this document, we define the syntax and semantics of CC++ by identifying where it deviates from C++. Our starting point is C++ 3.0 as defined in the Annotated C++ Reference Manual [1]. The structure of this document follows that of the ARM; each section in this document corresponds to a chapter in the ARM.

1.1 Notation

CC++ programs and CC++ keywords will be indicated by the use of a typewriter font: `keyword`. *Italics* will be used to indicate nonterminal symbols in the CC++ grammar.

When the CC++ specification supplements or modifies the C++ language definition, a cross reference to the relevant section in the C++ reference manual is given using the notation §N. References in the form Section N are used to refer to sections of this document.

2 Lexical Conventions (§2)

2.1 Keywords (§2.4)

The following are keywords in CC++ which are not keywords in C++:

```
par     parfor
sync    atomic
global  spawn
```

3 Basic Concepts (§3)

3.1 Scopes (§3.2)

There are five kinds of scope in a CC++ program. In addition to local, function, file and class, a name in a CC++ program can have *processor object* scope. A processor object is defined by one or more translation units and a processor object declaration (Section 9). A name at file scope can also be available at processor object scope if it is explicitly named as a member of the processor object. A processor object member can only be used by a member of that processor object, or after the `->` operator applied to a pointer to a processor object.

3.2 Program and Linkage (§3.3)

A CC++ program consists of one or more processor objects, each of which consists of one or more files and a processor object declaration (Section 9). All names within in a processor object are said to have *processor object linkage*. Internally linked C++ names refer to unique objects, externally linked C++ names within an instance of a processor object refer to the same object. Objects are not shared between different instances of a processor object.

■ The name of a data object refers to different objects in different processor objects, regardless of whether the name is internally or externally linked. Thus a name with internal linkage is local to both a translation unit and a processor object. Names with external linkage refer to the same object between translation units of a processor object, but different objects in different processor objects, even if they are different instances of the same processor object. □

3.3 Start and Termination (§3.4)

A CC++ program is initiated by specifying the initial processor object. This processor object must contain a public member called `main`. A CC++ program terminates when the main program has terminated.

■ The status of any computation started by a `spawn` statement (Section 6.4) does not effect the termination of `main`. □

All nonlocal static objects in a translation unit are initialized in every processor object in which the translation unit is a member before the first use of any function of the object in that translation unit and processor object.

The library function

```
void exit(int);
```

can be called to terminate the execution of a CC++ program. The behavior of this function is to:

1. Call the processor object destructor for the initial processor object,
2. Terminate the execution of all procedures in every processor object created by the program. This includes any procedures whose execution was started via a `spawn` statement.
3. Terminate program execution.

The function

```
void abort();
```

will stop the execution of all executing CC++ procedures in every processor object prior to terminating the program.

> ■ On termination, all processor objects are destroyed, without running their destructor. It is the responsibility of the program to explicitly delete processor objects created by a program if a clean shutdown is desired. □

3.4 Lvalues (§3.7)

In CC++, an lvalue is *modifiable* if it is not a function name, an array name, a `const`, or an initialized `sync` object (Section 7.0.2).

4 Standard Conversions (§4)

4.1 Pointer Conversions (§4.6)

Exceptions to the pointer conversion rules that apply to objects of type `volatile T*` and `const T*` apply to objects of type `sync T*` as well.

There is no implicit conversion between local and global pointers (Section 8.0.3).

4.2 Pointers to Processor Objects

The conversion of a constant expression that evaluates to zero to a pointer to a processor object is allowed. This pointer is guaranteed to be distinguishable from any other pointer to a processor object.

A pointer to a processor object can be implicitly converted to a `void *` only when constructing the return value for the function `new`. No other implicit conversion of a pointer to a processor object are allowed.

5 Expressions (§5)

The rules for use of `const` types in expressions apply to `sync` types as well.

5.1 Primary Expressions (§5.1)

primary-expression:
  literal
  `this`
  `::this`
  `::` *identifier*
  `::` *operator-function-name*
  `::` *qualified-nameidentifier*
  `(` *expression* `)` *name*

The `::this` expression returns a `const` pointer to the current processor object.

5.1.1 Increment and Decrement (§5.2.5)

The increment and decrement operators are not defined for variables of type `sync T`.

5.2 Unary Operators (§5.3)

The `*` operator may not be applied to a processor object pointer (Section 9).

When the `&` operator is applied to an object of type `sync T`, the result is of type `sync T*`.

When the `&` operator is applied to an object of type `T & sync` the result is of type `T*`. When the `&` operator is applied to an object of type `T *sync` the result is of type `T *sync *`.

When the `&` operator is applied to an object of type `T & global` the result is of type `T *global`. When the `&` operator is applied to an object of type `T *global` the result is of type `T *global *`.

5.2.1 Increment and Decrement (§5.3.1)

Use of the prefix `++` or `--` operators on an lvalue of type `sync T` is not allowed.

5.3 Explicit Type Conversion (§5.4)

A pointer to a `sync` object may not be converted to an integral type. Nor can a value of integral type be converted to a pointer to a `sync` object.

It is illegal to cast a pointer to a `sync` object into a pointer to a non-`sync` object.

A pointer to a `sync` object of a fundamental type cannot be cast to any other type.

A local pointer can be cast to a global pointer. A global pointer can be cast to a local pointer. If the local pointer does not point to an object in the processor object in which the cast takes place, the cast may cause an exception.

> ■ Unlike the `const` type modifier, one is not allowed to cast away the "`sync`ness" of an object. □

Other than explicit conversion of zero, it is illegal to perform any cast operation on a pointer to a processor object.

5.4 Additive Operators (§5.7)

Adding an integral type to a variable of type `T *sync` results in a pointer to `T`.

Adding an integral type to a variable of type `T *global` results in a global pointer to `T`.

A processor object pointer is not allowed as an operand to an additive operator.

5.5 Relational Operators (§5.9)

The only relational operators that can be applied to global pointers are `==` and `!=`.

The only relational operators that can be applied to a pointer to a processor object are `==` and `!=`.

6 Statements (§6)

All of the statement types available in C++ are available in CC++. Within the lexical scope of a sequential block, statement use is identical to that of C++. Additional restrictions apply to the use of some statements in a parallel composition.

The statement syntax for CC++ is:

statement:
  labeled-statement
  expression-statement
  compound-statement
  parallel-block
  iteration-statement
  spawn-statement
  jump-statement
  declaration-statement
  try-block

6.1 Labeled Statement (§6.7)

A *labeled-statement* is not allowed in the *statement-list* of a `par` block (Section 6.2) or as the *statement* in a `parfor` statement.

6.2 Parallel Block

The `par` block is used to initiate parallel execution a fixed set of statements in a CC++ program. The syntax for a `par` block is:

parallel-block:
  `par` { *statement-list*$_{\text{opt}}$ }

The statements in *statement-list* execute in an arbitrary but fair and interleaved manner. A `par` block terminates when all statements in *statement-list* have terminated.

A *statement* in *statement-list* may not be a *declaration-statement* or a *labeled-statement*.

The flow of control cannot be transferred:

- to any statement in a `par` block from outside the `par` block,
- from any statement in a `par` block to outside the `par` block,
- between statements in a `par` block.

Consequently, the *statement-list* of a `par` block may not contain a `break` or a `continue` statement. A `return` statement may not lexically appear anywhere within a `par` block.

6.3 Iteration Statements (§6.5)

CC++ adds a parallel loop construct to C++. The syntax for an iteration statement in CC++ is:

iteration-statement:
  `while (` *expression* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  `for (` *for-init-statement expression*$_{\mathrm{opt}}$*; expression*$_{\mathrm{opt}}$`)` *statement*
  `parfor (` *for-init-statement expression*$_{\mathrm{opt}}$*; expression*$_{\mathrm{opt}}$`)` *statement*

6.3.1 The `parfor` Statement

The `parfor` statement initiates the parallel execution of a variable number of statements.

The execution of a `parfor` proceeds as a `for` statement except that the next iteration of the loop can start before the execution of the current iteration is complete. The iterations of the `parfor` body are fairly interleaved.

The `parfor` terminates when all of the iterations terminate.

If the *for-init-statement* is a *declaration-statement* then the variables declared are called loop control variables. Within an iteration, a reference to a loop control variable evaluates to the value of that variable at the start of the iteration.

Instances of a loop control variable within an iteration are considered to be `const`.

The scope of a loop control variable is limited to the `parfor` statement.

The flow of control cannot be transferred into a `parfor` block from outside the `parfor` block. The flow of control cannot be transferred from any statement inside the `parfor` statement to outside the `parfor` statement.

■ The semantics of loop control variables are such that common errors in converting a `for` statement to a `parfor` statement can be detected at compile time. Such errors include loop carried dependencies introduced by loop control variables and failure to create a local version of a loop control variable by reusing a loop control variable from a preceding `parfor` statement.

Return statements cannot appear lexically anywhere in a `parfor` statement.

The *statement* component of a `parfor` statement cannot be labeled. □

6.4 The `spawn` Statement

The `spawn` statement is used to create a new thread of control without imposing a parent/child relationship. The syntax of a `spawn` statement is:

spawn-statement:
  `spawn` *postfix-expression* `(` *expression-list*$_{\text{opt}}$ `)` `;`

Execution of statement following `spawn` may or may not begin before the function is evaluated, however, execution of the statement following the `spawn` does not start until the *postfix-expression* and *expression-list* have been evaluated. Any return value from the function is discarded.

A spawned function executes in a fair and interleaved manner with the thread that executes the `spawn`.

■ While the default call-by-value argument passing semantics ensure that spawned functions will not reference variables that have gone out of scope, the language makes no provision for preserving non-value passed arguments. □

6.5 Jump Statements (§6.6)

Jump statements are not allowed to appear directly in the *statement-list* of a `par` block. Additional restrictions apply to the `return` and `goto` statement.

6.5.1 The `return` Statement (§6.6.3)

A `return` statement is not allowed to appear within the lexical scope of a `par` block or a `parfor` statement.

6.5.2 The `goto` Statement (§6.6.4)

A `goto` statement is not allowed to transfer control into or out of a `par` block or `parfor` statement.

6.6 Declaration Statement (§6.7)

A declaration statement is not allowed to appear within the statement list of a `par` block.

7 Declarations (§7)

7.0.1 Function Specifiers (§7.1.2)

A new function specifier, `atomic` is present in CC++. Thus the grammar rule for function specifiers is:

fct-specifier:
```
inline
virtual
atomic
```

The `atomic` modifier is used to control the granularity of interleaving in parallel execution. The execution of the statements in an atomic function is not interleaved with statements that are not part of the atomic function.

■ Atomic functions can be nested.

Atomic functions can contain `par` block, `parfor` and `spawn` statements, which are interleaved. □

The body of an atomic function is restricted as follows:

- Atomic functions must terminate.
- Atomic functions cannot read `sync` values (Section 7.0.2)
- Atomic functions cannot make references through a global pointer (Section 8.0.3).

7.0.2 Type Specifiers (§7.1.6)

CC++ introduces an additional type modifier: `sync`. Thus the grammar rule for type-specifiers becomes:

type-specifier:
  simple-type-name
  class-specifier
  enum-specifier
  elaborated-type-specifier
  `::` *class-name*
  `const`
  `volatile`
  `sync`

A declaration with a `sync` specifier declares a single assignment, or synchronization object. Single assignment objects are used to synchronize activities between statements executing in parallel. All `sync` objects are created in an uninitialized state. A `sync` object is initialized by one of the existing C++ initialization mechanisms (§8.4.1, §12.1, §12.6) or by being used as the left hand operand of an assignment operator. Once initialized, a `sync` object is no longer a modifiable lvalue and it behaves as if it had been declared `const`. Multiple assignment to a `sync` objects may result in an exception.

The evaluation of expression requiring the value of an uninitialized `sync` object will not complete until some finite time after the `sync` object has been initialized.

The following restrictions apply to the use of `sync`:

- The use of `sync` unions is not allowed.
- No member of a union be `sync`.

- A bit field is not allowed to be `sync`.

Except where noted, all of the rules that apply to `const` variables also apply to variables of type `sync T`.

Unlike `const` variables, `sync` variables do not require initialization.

Each element of a `sync` array is a `sync`. Each nonfunction, nonstatic member of a `sync` class object is `sync`. Only `sync` member functions of a `sync` object can be used.

The result of all arithmetic operations on fundamental sync types result in a non-sync type.

8 Declarators (§8)

cv-qualifier:

```
const
volatile
sync
global
```

8.0.3 Pointers (§8.2.1)

When applied to a pointer, the *cv-qualifier* `sync` produces a `sync` pointer. The rules for this pointer are the same as for any other `sync` object (Section 7.0.2).

■ A sync reference is the same as a `const` reference. □

All references and pointers between processor objects (Section 9) must be explicitly declared global by use of the `global` keyword. Intra-processor object references and pointers may be `global`.

A local pointer cannot be accessed through a global pointer.

The only relational operators that can be applied to global pointers are `==` and `!=`.

There is no implicit conversion between local and global pointers. Explicit conversion between global and local pointers is allowed (Section 5.3).

When the `&` operator is applied to an object of type `T & global` the result is of type `T *global`. When the `&` operator is applied to an object of type `T *global` the result is of type `T *global *`.

8.1 Functions (§8.2.5)

The *cv-qualifier* `global` cannot be applied to functions. When `sync` is applied to the declaration of a member function, it indicates that the function can be called from a `sync` instance of the class.

9 Processor Objects

In CC++, the syntax for a *class-specifier* is:

class-specifier:
  `global`$_{\text{opt}}$ *class-head* { *member-list*$_{\text{opt}}$}

The keyword `global` is used to indicate the declaration of a processor object class.

The global functions and data names that are public members of a processor object class are of processor object scope and are available from outside the processor object via the `->` operator applied to a processor object pointer. All global functions not explicitly named in the public part of a processor object declaration are assumed private members of the processor object. Any external name can be bound to the name of a member of a processor object at link time, quantification of the definition is not required.

> ■ Implicit membership in a processor object class allows an C++ object library to be reused in different processor object classes.
> □

Static member functions of a processor object class are executed in the calling processor object. All other member functions must be available in a form that is executable in the environment on which the processor object is allocated (Section 11.2).

■ Since static member functions are not associated with an instance of an object, the only place a static member function of a processor object can execute is in the current processor object. Because of this, an executable version of the member function must be available to the caller. □

Processor object classes can not be inherited, nor can they inherit from other classes. A processor object class can not have protected members, virtual functions or static data members. A processor object class can not have friends, nor can it be the friend of other classes.

The scoping operator `::` refers to a member of the current processor object. The keyword `::this` is a constant pointer to the current processor object.

Only pointers to processor object classes may be defined and this pointer can only be dereferenced by the `->` operator. It is illegal to access a processor object in any other manner. The result of referencing a processor object pointer which has been deleted is undefined.

Processor objects can be dynamically created with the `new` operator (Section 11.2).

10 Classes (§9)

10.1 Static Members (§9.4)

A static member function is always evaluated in the processor object from which the call is made.

11 Special Member Functions (§12)

11.1 Constructors (§12.1)

A ctor list is not allowed in a constructor for a processor object. The constructors for all processor object data members are called with the arguments specified in their definition. If no initial value is provided in the definition the default constructor for the object is called.

11.2 Free Store (§12.5)

The functions `new` and `delete` are used to dynamically allocate and free processor objects. The system defined `new` operator can be called with zero or one argument. In the single argument form, the argument is of an implementation defined type`proc_t`, which is defined in `stddef.h`. It is guaranteed that a `proc_t` is distinguishable from a `size_t` via the normal C++ overloading resolution mechanism.

The implementation may customize the behavior of the `new` operator based on the `proc_t` argument specified.

When overloaded, the first argument to the `new` operator for a processor object class must be of type `proc_t`. An overloaded `new` operator for a processor object type must have at least one argument in addition to the `proc_t` argument inserted by the compiler.

■ The requirement on the arguments of an overloaded `new` operator for a processor object ensure that we can distinguish between the system `new` and a user defined `new`. Without this restriction a call to

```
global class pobj;
proc_t p;
pobj * ptr = new (p) pobj;
```

could not be unambiguously resolved between the system

```
::new (proc_t)
```

and a user defined

```
new(proc_t,proc_t)
```

operator. □

The implicit conversion of a processor object pointer to a `void *` is allowed when constructing the return value in `new`.

If the processor object class declares a constructor, that constructor is called before the pointer is returned. Processor object members are initialized with the arguments specified in their definitions. The definition of a constructor for a processor object must be explicitly quantified.

The system provided `::new` operator may use the optional placement arguments to specify the mapping of a processor object to a physical processing resource and the location of the executable code for the processor object. Such use is implementation defined.

11.3 Void Functions

The functions:

voidfunction:
`void operator << ( void ,` *argument-declaration* `)`
`void operator >> ( void ,` *argument-declaration* `)`

are used by the compiler to transfer data between processor objects. Data structures are moved in two stages: the `<<` operator is used to pack the data structure in an architecture independent format and the `>>` operator is used to unpack the data structure. These functions are automatically invoked by the compiler whenever a data transfer is needed.

The compiler will generate default operators if they are not defined by the program and `T` does not contain any local pointers. The default operators perform the copy memberwise.

12 Acknowledgment

The research on CC++ object libraries for concurrent computation is funded by DARPA under grant N00014-91-J-4014. The research on compositional concurrent notations was funded by NSF under grant CCR-8809615.

References

[1] Ellis, M. A., and Stroustrup B. *The Annotated C++ Reference Manual* Addison-Wesley Publishing Company, 1990.

[2] Chandy, K. M., and Kesselman C. *A Description of CC++* Technical Report, CS-92-01, California Institute of Technology, 1992.