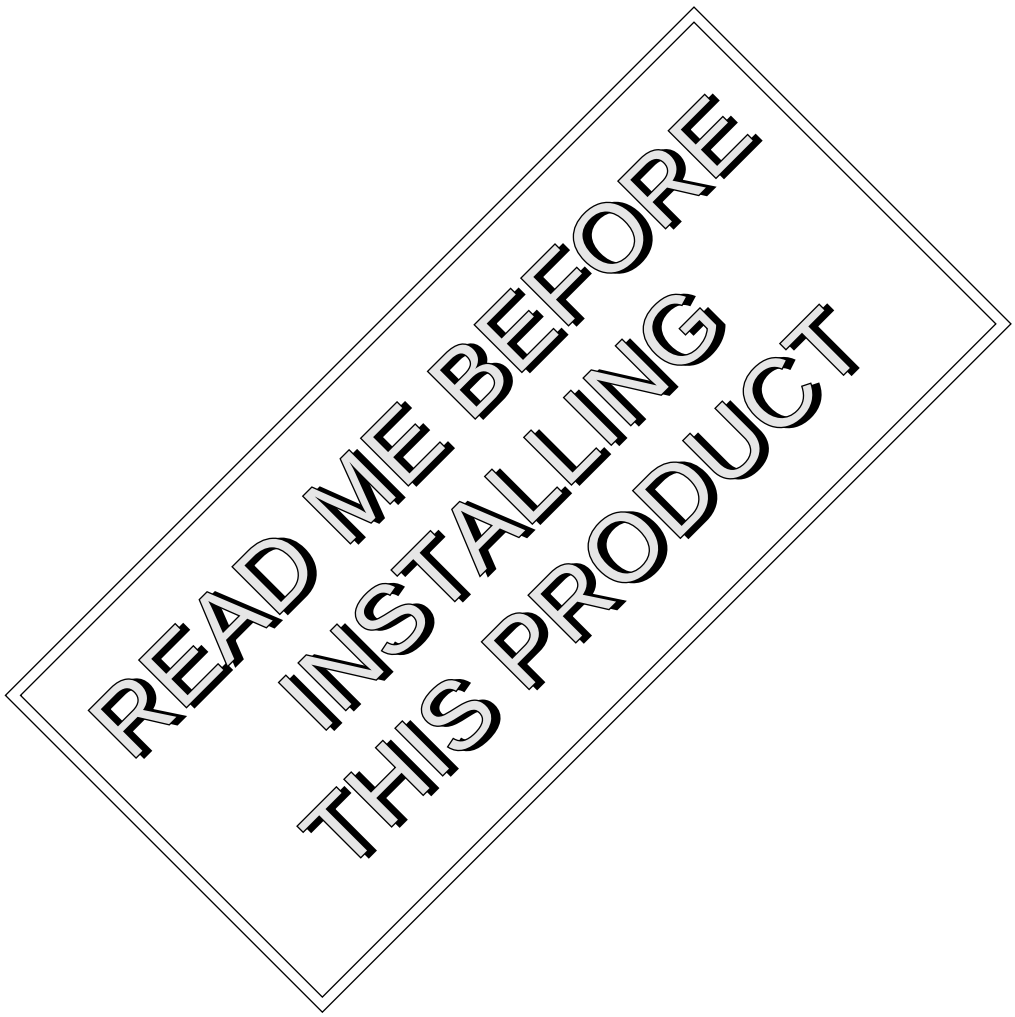
Concurrent C/C++

Version 6.1 Release Notes (Linux)

July 2004

0898497-6.1



CONCURRENT
COMPUTER
CORPORATION

Copyright

Disclaimer

The information contained in this document is subject to change without notice. Concurrent Computer Corporation has taken efforts to remove errors from this document, however, Concurrent Computer Corporation's only liability regarding errors that may still exist is to correct said errors upon their being made known to Concurrent Computer Corporation.

License

Trademark Acknowledgments

Contents

1.0. Introduction

Concurrent C/C++ is part of the PowerWorks™ Linux Development Environment (PLDE) and utilizes Edison Design Group's C++ front end and Concurrent's Common Code Generator (CCG) technology to produce highly optimized object code tailored to Concurrent systems running PowerMAX OS™.

There are several compiler switches that provide a degree of compatibility with previous drafts of the ANSI C++ Standard, USL versions 2.1 and 3.0 C++ compilers (also known as **`cfront`**), Kernighan and Ritchie C and SVR4 C, in addition to compatibility with the ANSI C++ and C languages.

This release combines into a single compiler what were, previous to release 5.1, separate C and C++ compilers. Release 5.1 was the "next" release after Concurrent C Compiler 4.3 and Concurrent C++ Compiler 3.1. This release does not require un-installing previous versions of the C and C++ compilers. It installs in its own unique location and provides a mechanism for supporting the use of multiple releases.

This release provides a command-line-based program development environment for building complex projects. Alternatively, the user may use NightBench™ to interface with this program development environment.

As of release 5.1, C/C++ no longer ships with USL iostream and complex libraries. They are available separately.

2.0. Documentation

Table 2-1 lists the Concurrent C/C++ 6.1 documentation available from Concurrent.

Table 2-1. Concurrent C/C++ Version 6.1 Documentation

Manual Name	Pub. Number
Concurrent C/C++ Reference Manual	0890497-040
Concurrent C/C++ Version 6.1 Release Notes (Linux)	0898497-6.1

Copies of the Concurrent documentation can be ordered by contacting the Concurrent Software Support Center. The toll-free number for calls within the continental United States is 1-800-245-6453. For calls outside the continental United States, the number is 1-954-283-1822 or 1-305-931-2408.

Additionally, the manuals listed above are available:

- online using the PowerWorks Linux Development Environment utility, **`nhelp`**
- in PDF format in the **`documentation`** directory of the PLDE Installation CD
- on the Concurrent Computer Corporation web site at www.ccur.com

3.0. Prerequisites

Prerequisites for Concurrent C/C++ Version 6.1 for both the host system and target system are as follows:

3.1. Host System

3.1.1. Software

- Red Hat® or RedHawk™ Linux
- Cross development packages for each PowerMAX OS system to be targeted (see "Cross-Development Libraries" on page 7)

3.1.2. Hardware

- an Intel®-based PC - 300Mhz or higher (recommended minimum configuration)
- 64MB physical memory (recommended minimum configuration)

3.2. Target System

3.2.1. Software

- PowerMAX OS 4.3 or later

3.2.2. Hardware

- Computer Systems:

  Power Hawk™ 620 and 640

  Power Hawk 710, 720 and 740

  PowerStack™ II and III

  Night Hawk® Series 6000

  TurboHawk™

  PowerMAXION™

- Board-Level Products:

  Motorola® MVME2604

  Motorola MVME4604

4.0. System Installation

Installation of the host portion of Concurrent C/C++ is normally done as part of the general installation of the PowerWorks Linux Development Environment software suite. A single command installs (or uninstalls) all software components of the PLDE, as described in the *PowerWorks Linux Development Environment Release Notes* (0898000).

The following section describes how to install (or uninstall) Concurrent C/C++ separately from the PLDE suite for those rare cases when this is necessary.

In addition, it is necessary to install the PowerMAX OS cross-development libraries on your Linux system in order to cross-compile and cross-link on that system.

4.1. Separate Host Installation

In rare cases, it may be necessary to install (or uninstall) Concurrent C/C++ independent of the installation of the PowerWorks Linux Development Environment software suite. This may be done using the standard Linux product installation mechanism, **`rpm`** (see **`rpm(8)`**).

The names of the RPMs associated with Concurrent C/C++ 6.1 are:

```
ccur-invoker
ccur-common-tools-6.1
plde-c++-6.1
plde-c++help-6.1
```

and the files associated with these RPMs, respectively, are:

```
ccur-invoker-000-1.i386.rpm
ccur-common-tools-6.1-000-1.i386.rpm
plde-c++-6.1-000.i386.rpm
plde-c++help-6.1-000-1.i386.rpm
```

which can be found in the **`linux-i386`** directory on the PowerWorks Linux Development Environment Installation CD.

NOTE

In releases prior to 6.1, **`ccur-invoker`** was named **`ccur-c++invoker`**.

NOTE

The package **`plde-c++help-6.1`** contains the online manuals for the C/C++ compiler. The installation of **`plde-c++help-6.1`** is not necessary for the proper operation of the compiler.

NOTE

The user must be root in order to use the **rpm** product installation mechanism on the Linux system.

To install the Concurrent C/C++ RPMs, issue the following commands on your Linux system:

1. Insert the PowerWorks Linux Development Environment Installation CD in the CD-ROM drive
2. Mount the CD-ROM drive (assuming the standard mount entry for the CD-ROM device exists in **/etc/fstab**)

   ```
   mount /mnt/cdrom
   ```

3. Change the current working directory to the directory containing the Concurrent C/C++ RPMs

   ```
   cd /mnt/cdrom/linux-i386
   ```

4. Install the RPMs

   ```
   rpm -i ccur-invoker-000-1.i386.rpm \
          ccur-common-tools-6.1-000-1.i386.rpm \
          plde-c++-6.1-000-1.i386.rpm \
          plde-c++help-6.1-000-1.i386.rpm
   ```

   By default, the product is installed in **/usr/opt**.

5. Change the current working directory outside the **/mnt/cdrom** hierarchy

   ```
   cd /
   ```

6. Unmount the CD-ROM drive (otherwise, you will be unable to remove the PowerWorks Linux Development Environment Installation CD from the CD-ROM drive)

   ```
   umount /mnt/cdrom
   ```

NOTE

The optional **plde-c++cfront** package containing USL iostream and complex libraries should be installed after the **plde-c++** package.

To uninstall the Concurrent C/C++ RPMs, use the following command:

```
rpm -e plde-c++help-6.1 \
       ccur-common-tools-6.1 \
       ccur-invoker \
       plde-c++-6.1
```

4.2. Cross-Development Libraries

In order to cross-compile and cross-link on the Linux system, it is necessary to have certain PowerMAX OS libraries installed on that system.

The names of the RPMs containing the PowerMAX OS libraries minimally required for cross-linking are:

`plde-pmax-crossdev-`*version*

where *version* is the version of PowerMAX OS target.

NOTE

The version number is part of the name of the RPM. Because of that, it is possible to install multiple RPMs on the Linux system at the same time. This allows the user to generate executables for multiple PowerMAX OS versions from the same Linux system.

User applications built with Concurrent C/C++ may require other capabilities which are provided by additional RPMs included on the PLDE Installation CD.

Refer to the section titled “Cross-Development Libraries and Headers” in the *PowerWorks Linux Development Environment Release Notes* (0898000) for more information.

5.0. Using Concurrent C/C++ with the PLDE

The following should be taken into consideration in order to use Concurrent C/C++ with the PowerWorks Linux Development Environment.

5.1. PATH Considerations

On Linux systems, the **cc**, **c++**, **gcc**, and **g++** commands invoke the native Linux compilers, which are completely unrelated (and incompatible at the object level) with PowerMAX OS and the Concurrent C/C++ cross-compiler.

To utilize the Concurrent C/C++ compiler, specify the following in your `PATH` environment variable:

```
PATH=$PATH:/usr/ccs/bin
```

The compiler should then be invoked with either **ec** or **ec++**.

However, if you wish to be able to invoke the Concurrent C/C++ compiler as **cc** or **c++**, insert the following at the head of your `PATH` environment variable:

```
PATH=/usr/ccs/crossbin:$PATH
```

The **/usr/ccs/crossbin** directory contains commands named **cc** and **c++** which invoke the Concurrent C/C++ compiler as opposed to the Linux compilers.

See “Makefile Considerations” on page 10 below for more information.

5.2. Include Files and Libraries

By default, the Concurrent C/C++ compiler automatically looks for PowerMAX OS include files and libraries in the tree rooted as:

/pmax/os/*version*/*arch*

where *version* and *arch* indicate the PowerMAX OS version and target architecture of your choice (see “OS Versions and Target Architectures” on page 9 for more details).

Files located under **/usr/include** and **/usr/lib** are native Linux files and are unrelated and incompatible with the corresponding files for PowerMAX OS. Do not attempt to utilize files from those directories when building PowerMAX OS programs.

Remove any explicit references to these directories in:

- source files (e.g. `#include "/usr/include/unistd.h"`)
- Makefiles (e.g. **cc -I/usr/include**)
- build scripts

Include file references of the form:

```
#include <unistd.h>
```

or

```
#include "unistd.h"
```

need not be changed. These forms are supported, as the appropriate **/pmax/os/***version***/***arch* trees are searched.

5.3. OS Versions and Target Architectures

The PowerWorks Linux Development Environment supports building PowerMAX OS programs for various versions of PowerMAX OS and various systems.

The current versions of PowerMAX OS (**osversion**) that are supported are 4.3 and later.

The current architectures (**arch**) that are supported are:

- **nh**
- **moto**
- **synergy**

which correspond to the following systems:

System type	*architecture*
PowerMAXION-4	**nh**
PowerMAXION	**nh**
Night Hawk 6800	**nh**
Night Hawk 6800 Plus	**nh**
TurboHawk	**nh**
Power Hawk 610	**moto**
Power Hawk 620	**moto**
Power Hawk 640	**moto**
PowerStack	**moto**
PowerStack II	**moto**
Power Hawk 710	**synergy**
Power Hawk 720	**synergy**
Power Hawk 740	**synergy**

NOTE

The default OS version is currently **4.3** and the default target architecture is **nh**.

You can change the **osversion** and **arch** settings in several ways:

- Specify the options on the **ec** or **ec++** command line:

  ec -o main main.c --arch=*arch* **--osversion=***os*

- Change the default for your user on a specific Linux system using the Concurrent C/C++ command line utility **c.release**:

  c.release -arch *arch* **-osversion** *os*

- When using the Concurrent C/C++ PDE utilities (**c.build**, etc.), you can:
  - Set the **arch** and **osversion** for an environment using **c.mkenv**:

    c.mkenv -arch *arch* **-osversion** *os*

  - Set the **arch** and **osversion** for a specific partition using **c.partition**:

    c.partition -oset "--arch=*arch* **--osversion=***os***" main**

5.4. Shared vs. Static Linking

By default, the Concurrent C/C++ compiler links with shared libraries.

Thus, if you attempt to execute your C++ program on a PowerMAX OS system it will require, at a minimum, the shared library **libCruntime.so**.

If your PowerMAX OS system doesn't have either the Concurrent C/C++ product or the **c++runtime** package installed, your program will fail to execute.

You can install the full PowerMAX OS version of the Concurrent C/C++ compiler, install just the **c++runtime** package, or relink your program using static libraries.

The PowerMAX OS **c++runtime** package is included on the PowerWorks Linux Development Environment Installation CD. See the section titled "Target Installation" in the *PowerWorks Linux Development Environment Release Notes* (0898000) for installation instructions.

To link your program using static libraries, append the **-Zlink=static** option to your command line:

```
ec++ -o main main.c -Zlink=static
```

5.5. Makefile Considerations

Makefiles may already contain references to **cc** or **c++** commands explicitly within them. Additionally, if default rules for compilation, such as

```
.c.o:
```

or

```
.cc.o:
```

are not explicitly mentioned, the **make** processor will also attempt to invoke **cc**, **c++**, or even **g++**.

By default, unless you have **/usr/ccs/crossbin** early in your PATH variable, these situations will result in the Linux native compilers being invoked instead of the Concurrent C/C++ compiler.

To resolve these problems you can take any of the following approaches.

5.5.1. Explicit Modification Using ec/ec++

Ensure that **/usr/ccs/bin** is in your PATH environment variable.

Modify all occurrences of **cc** and **c++** to utilize **ec** and **ec++**, respectively.

Supply default .c.o rules (and the like) to explicitly utilize the **ec** and **ec++** commands.

5.5.2. Use of /usr/ccs/crossbin in PATH Environment Variable

Put **/usr/ccs/crossbin** at the head of your PATH environment variable.

This will cause references to **cc** and **c++** to invoke the Concurrent C/C++ compiler as opposed to the Linux compilers.

5.5.3. Use of CC Environment or Make Variables

If you don't want **/usr/ccs/crossbin** early on your PATH (perhaps because you plan to build for Linux and/or PowerMAX OS at various times), then you'll want to just use the **ec** and **ec++** when you want to compile for PowerMAX OS (it is still necessary to add **/usr/ccs/bin** to your PATH).

One approach to using **ec** and **ec++** that requires minimal changes to Makefiles, etc., is to use environment variables or **make** variables to control which C/C++ compiler you're using. The following commands will all build using the PLDE cross-compilers:

Short-lived environment variables:

```
# CC=ec CXX=ec++ make arguments
```

make variables:

```
# make arguments CC=ec CXX=ec++
```

Long-lived environment variables:

```
# export CC=ec
# export CXX=ec++
# make arguments
```

You can also use the long-lived environment variable approach if you intend to always build for PowerMAX OS, by adding the following to your login script (e.g. **.profile** or **.login** depending on your shell):

```
export CC=ec
export CXX=ec++
```

Or, if you prefer finer-grained control, you can add lines like the following to the top of any Makefiles that should use the Concurrent C/C++ cross-compiler:

```
CC=ec
CXX=ec++
```

The changes will then only affect the modified Makefiles. Note that this solution only works for Makefiles that use the default .c.o and .cpp.o, etc. rules. If they contain hard-coded references to **cc** or **cc++**, then either **/usr/ccs/crossbin** must be used, or the Makefiles must be changed to use $(CC) and $(CXX) instead. If the Makefile references anything like **g++** (Linux's GNU C++ compiler), then it will need to be changed, regardless.

Here are two more complete and robust sets of variables which will work equally well with well-written Makefiles.

```
CC=/usr/ccs/crossbin/cc
CXX=/usr/ccs/crossbin/c++
```

```
AS=/usr/ccs/crossbin/as
AR=/usr/ccs/crossbin/ar
LD=/usr/ccs/crossbin/ld
```

Or, alternatively:

```
CC=/usr/ccs/bin/ec
CXX=/usr/ccs/bin/ec++
AS=/usr/ccs/bin/as.pmax
AR=/usr/ccs/bin/ar.pmax
LD=/usr/ccs/bin/ld.pmax
```

These two sets are mentioned in order to provide very easy support for those users that want to compile only for PowerMAX OS (**`/usr/ccs/crossbin`**) and for those users that may want to compile for either Linux or PowerMAX OS, depending on the application (**`/usr/ccs/bin`**).

6.0. Changes in This Release

6.1. Documentation Updates

The *Concurrent C/C++ Reference Manual* has been updated and is available online through the **nhelp** and **c.man** tools. The traditional man pages are abbreviated to provide only brief descriptions of the options; the user is directed to the *Reference Manual* for complete documentation.

6.2. Invoking The Compiler

To use the release 5.1 and later of the C/C++ compiler, it is necessary to add the path **/usr/ccs/bin** to your PATH environment variable. The C compiler is invoked by the **ec** command, and the C++ compiler is invoked by the **ec++** command.

If an earlier C or C++ compiler is still installed on the system, the system administrator may choose to configure the previous compiler commands (**cc**, **hc**, **cc++**, **c++**, **analyze**, and **report**) to invoke the pre-5.1 release. However, by default, they will invoke the latest release. The user may override this default behavior by setting the PDE_RELEASE environment variable to either **pre5.1**, **5.4**, or another release name, or by using the **c.release** command to select the user-specific default release.

6.3. Debug Information: Location Lists

Traditionally, debug information associates one location with a variable. However, modern optimizing compilers (such as Concurrent's) may keep a variable in a variety of locations over the variable's lifetime. For example, a variable stored in memory may be kept in a register during the execution of a loop, or a variable may be bound to several registers over the course of its lifetime. Another drawback of the traditional approach is that a location may not always contain the value of the variable. The location may contain other values at any point where it is not required for the variable, such as before the varaible is first set, or after it is last used. At such times, examination of the variable with the debugger would produce incorrect values.

This release of the compiler implements a more detailed description of variable locations called "location lists". These lists specify where a variable is located as a function of the program counter. Thus while using the debugger, if the user is stopped inside a loop, and the variable has been copied into a register, then the register will be used to examine the variable's value. When the user is stopped outside the loop, and the variable exists only in memory, then the memory location will be used to examine or modify the variable. If a variable is in a register, but the user is stopped at a location when the register contains some other value, such as after the last use of the variable, then the debugger will indicate that the variable has no valid location.

It is recommended that NightView 5.4 or later be used to debug programs compiled with location lists.

The **-g** option has been enhanced for location lists to give the user more control over the amount of debug information the compiler generates:

-g0 (*none*)	No debug information is generated. (This is the default if no **-g** option is specified.)

-g1 (*lines*)	Only line number information is generated. (In previous releases, this was the default when no **-g** option was specified.)
-g2 (*simple*)	Variables are described in only the most common location, and for the entirety of their defined scope. (In previous releases, this was the default when the **-g** option was specified.)
-g3 (*full*)	Full location lists are generated for variables. (This is the default if the **-g** option is specified without a numeric modifier.)

6.4. Debug Information: --compress_debug_info

When compiling under the PDE (using the **c.*** tools or NightBench), there is a new option available:

--compress_debug_info

This option will direct the compiler to record type definitions in the PDE's database rather than in each individual object file. The result is much smaller debug information in the final executable. Without the same types defined repeatedly from each object file, the NightView debugger will operate much faster.

6.5. Enhancements to c.script

The **c.script** tool, which is used by NightBench to save an environment to a file, no longer generates a **/bin/sh** script for recreating an environment. Instead, it generates a **/usr/ccs/bin/c.sh** script. The **/bin/sh** script was slow because each command it invoked had to open, modify, and update the database file. The **/usr/ccs/bin/c.sh** tool only needs to open the database one time, interpret all the commands, then close the database when done. This should be transparent to the user except when hand-editing the script file (the syntax and layout of the script differs from the **/bin/sh** script); in addition, the run speed of the script will be much faster.

6.6. static_cast between enum types allowed

The standards committee ruled in Core Issue 128 that `static_cast` between `enum`s is allowed. The compiler now allows.

```
enum E1 { ke1 };
enum E2 { ke2 };
int main () {
  static_cast<E2>(ke1);   // Now allowed
}
```

6.7. Operator keywords in #if directives

The compiler now correctly processes operator keywords (like "`compl`" or "`new`") in `#if` directives when alternative tokens are enabled (see the `--[no]_alternative_tokens` option). For example:

```
#if 0 or 1    // Now tests true when alternative tokens are enabled
#if delete 0  // Still invalid, but different diagnostic
#endif
#endif
```

6.8. Value-initialization

Value-initialization, a refinement of default-initialization introduced in TC1 of the C++ standard (and Core Issue 178), is now implemented. It ensures that all fields of non-POD classes without user-written constructors are properly initialized by zeroing the storage for the object before calling the constructor.

6.9. GNU C compatibility option

The compiler now supports command-line switches **`--gcc`** and **`--no_gcc`** that provide some language compatibility with GNU C compilers. The compatibility is neither exact (i.e. some extensions may have a slightly different meaning) nor complete (some extensions are not implemented). This compatibility is only provided in C mode; in fact, the command-line switches imply C mode even when no "`--c`" option is specified.

The set of extensions enabled with the **`--gcc`** option initially includes:

- variable length arrays
- extended designated initializers
- compound literals (extended over C99 to allow more kinds of initializations)
- extended variadic macros
- identifiers with dollar signs
- C++-style comments
- digraphs
- long long (and related extensions provided by C99)
- hexadecimal floating-point constants
- nonconstant aggregate initializers for automatic variables
- the `typeof` operator
- the `__inline__` (and `inline`) and `__extension__` keywords
- `__FUNCTION__` and `__PRETTY_FUNCTION__`
- zero-sized arrays
- the `\e` escape sequence
- Empty structs and structs with flexible array members
- attributes
- `&&label`
- `goto *`*expression*
- builtin functions

- statement expressions i.e., `({...})`
- multi-line strings
- binary conditional operators `("x ?: y")`
- extern inline
- the sizeof operator is applicable to `void` and function types and evaluates to the value one
- arithmetic on pointers to `void` and function types is possible (they behave much like `char*`)
- variables can be redeclared with different top-level cv-qualifiers (the new qualification is merged into existing qualifiers)
- locally declared labels
- the (ternary) conditional operation produces an lvalue if its second and third operand are lvalues
- the comma operator produces an lvalue if its second operand is an lvalue
- certain casts preserve the lvalueness of their operands
- case ranges
- union casts
- implicit conversions between integer and pointer types, and between incompatible pointer types, with a warning
- implicit conversions between pointer types that drop cv-qualifiers
- typedef initializers
- **`--preinclude_macros`** option (like the GCC **`--imacros`** feature)
- do-nothing casts to struct or union types
- return expressions in void functions
- oversized bit fields are turned into ordinary fields
- old-style definitions with unpromoted parameter types can follow function prototypes with those same parameter types
- `_Bool` keyword
- "`?`" operators with mixed `void`/non-`void` operands

6.10. `__alignof__` equivalent to `__ALIGNOF__`

The compiler now accepts `__alignof__` as an alternative spelling of `__ALIGNOF__`. The two are exactly equivalent, but the former was added to increase compatibility with other C++ compilers. The `__ALIGNOF__` spelling remains available.

6.11. `#warning` directive

The compiler now supports a `#warning` directive that is similar to `#error`, except that it generates a warning instead of a catastrophic error.

```
#warning Issue this warning
```

This new directive is not recognized in strict ANSI mode.

6.12. Semantic analysis of friends of class templates

The standard currently specifies that semantic analysis of a friend function defined in a class template is done when the enclosing class is instantiated, even if the friend function is never used. Most implementations, however, defer such analysis until the function is used. As a result, there is a body of code that requires such deferral to compile. We believe that the standard should be revised to require such deferral and have changed the compiler accordingly pending consideration of the issue by the standard committee.

```
template <class T> struct A {
  friend void f(A a) {
    g(a);
  }
};

A<int> ai;
```

6.13. Indirect flexible array members in C99 mode

The default C99 mode of the front end now accepts structure definitions whose last field has a struct type containing a flexible array member. For example:

```
struct F { int i; int a[]; };
struct S {
  int n;
  struct F f;  /* Now accepted in default C99 mode. */
};
```

This is still an error in strict C99 mode.

6.14. Command-line option to allow `long long` in strict mode

Except in C99 mode, the use of `long long` results in a diagnostic in strict mode. The **`--long_long`** option can now be used to override this behavior so that code that uses `long long` can be compiled without diagnostics in strict mode. The **`--long_long`** option must follow the strict mode option on the command line.

6.15. Limiting the amount of instantiation context output in diagnostics

In certain cases, the context information provided for errors during template instantiations can become very lengthy. In many cases much of the context information is unnecessary. The compiler now provides a mechanism to limit the volume of context output. The "context limit" is the maximum number of context entries to be displayed as part of a diagnostic message and may be set with the **`--context_limit`** command-line option.

A value of zero is used to indicate that there is no limit. If the number of context entries exceeds the limit, the first and last *N* context entries are displayed where *N* is half of the context limit. The following is an example of the output when a portion of the context is suppressed using **`--context_limit=2`**. The default limit is **`10`**.

```
detected during:
  instantiation of "void f(T) [with T=int ***]" at line 4
  [ 20 instantiation contexts not shown ]
  instantiation of "void f(T) [with T=int]"
```

6.16. __ALIGNOF__ no longer requires parentheses around argument expressions

The `__ALIGNOF__` (or `__alignof__`) operator now more closely matches the standard `sizeof` operator in that it can be followed by an expression that is not parenthesized.

For example:

```
int a = __ALIGNOF__ a;
```

This change makes the `__ALIGNOF__` extension more compatible with similar extensions in other compilers.

6.17. C99, GNU C mode: lowering of bool increment/decrement

In C99 mode and GNU C mode, when a `bool` value is incremented or decremented it should be set to 1 (`true`) or 0 (`false`), respectively. The generated code for those cases now does that.

6.18. GNU C: variable-length array fields of local structs

The GNU C compiler accepts variable-length array fields in local struct declarations, which may result in size and offset computation being delayed until run time. However, it is common for this feature to be used simply to indicate that the last field of a structure is a flexible array member. To enable the latter idiom, the compiler now accepts such declarations in GNU C mode, but it discards the given length with a warning, and sets the length of the array to zero. For example:

```
void f(int n) {
  struct S {
    int a[n]; /* Warning: equivalent to "int a[0];". */
  };
}
```

6.19. GNU C compatibility: "auto" ignored in file scope

In GNU C mode, the front end ignores the "`auto`" specifier with a warning when it appears on a file-scope declaration.

```
auto int i;  // Warning in GNU C mode; error otherwise
```

6.20. GNU C: incomplete array parameter types

In GNU C mode, the compiler now accepts the following code:

```
void f(int x[][]) {}
```

Within the definition of the function, the parameter `x` is treated as being a pointer to an incomplete type (e.g., it cannot be subscripted).

6.21. GNU C: --short_enums option

The compiler now accepts the **`--short_enums`** option in GNU C mode. It indicates that all the enumeration types should be treated as if they were declared with the "`packed`" attribute (i.e. the underlying type should be the smallest integer type that can accommodate the enumerator constants).

6.22. GNU C compatibility: parameter names in unprototyped declarations

The compiler now accepts parameter names in unprototyped function declarations that are not definitions. A warning is issued (in strict C mode an error is issued).

```
void f(i);  /* Accepted with a warning in GNU C mode. */
```

6.23. GNU C mode: typedefs with sign or size modifiers

The compiler now accepts the modifiers `short`, `long`, `signed` and `unsigned` on `typedef` types that are synonyms for builtin integer or floating-point types.

```
typedef int Int;
typedef unsigned Int UInt; /* Accept in GNU C and pcc modes; error
                              in other modes. */
```

6.24. How conversion functions compete with constructors

The C++ standards committee in closing Core Issue 243 without action affirmed a piece of the C++ standard we weren't sure was intended, that constructors win out over conversion functions in direct-initializations. Accordingly, we have changed the compiler to follow that rule, except in `cfront` mode.

```
extern "C" int printf(const char *, ...);
struct B;
struct A {
  A(B);
};
struct B {
  operator A() { printf("B::operator A()\n"); return A(*this); }
} b;
A::A(B) { printf("A(B)\n"); }
void f(A) {}
int main() {
  f(A(b));  // Prints A(B)
  f((A)b);  // Prints A(B)
  return 0;
}
```

6.25. GNU C mode: Warning only on invalid uses of `inline`

In GNU C mode, declaring a variable "`inline`" causes the front end to only issue a warning (rather than an error). The keyword is ignored after the warning is issued.

```
int inline x;  /* Warning in GNU C mode; error in other modes. */
```

The use of "`inline`" in block-extern declarations of functions is treated similarly:

```
void f() {
inline void g();  /* Warning in GNU C mode;
                     error in other modes. */
```

6.26. Template function with explicit arguments treated as simple function

In accordance with the proposed resolution to Core Issue 115, the compiler now treats a template function with explicit template arguments as a simple function if the arguments are sufficient to select a unique template function.

```
template <class T> void f(T);
int main() {
  &f<int>;  // OK now
}
```

6.27. Select C99 features

The C99 features Variable Length Arrays and Complex Arithmetic have been added.

6.28. The --[no]_vector_safe_prologs Compile Option

The **`--vector_safe_prologs`** compile option places a runtime test in function prologs to avoid executing AltiVec instructions in the prolog. This makes it possible for the user to write a single function that uses AltiVec instructions if available, or non-vector operations otherwise.

6.29. New Edison Front End

Version 3.0 of the Edison front end has been incorporated.

7.0. Cautions

7.1. va_list Structure Format

When using PowerMAX OS 5.1 headers, **varargs** and **stdarg** use a different format for the `va_list` structure. Because of this, the user should avoid passing `va_list` structures to routines that are built with previous versions of the PowerMAX OS headers.

Code built with PowerMAX OS 5.1 headers can handle both the old and the new format of `va_list`. There should therefore be no problem with old code, compiled with PowerMAX OS 5.0 and earlier headers, passing `va_list` to code compiled with PowerMAX OS 5.1 headers.

For example, a dynamically-linked program that calls `vfprintf` and that is compiled under PowerMAX OS 4.3 will have no problem running on a PowerMAX OS 5.1 system, calling the `vfprintf` in the 5.1 version of **libc.so**. That same program compiled under PowerMAX OS 5.1 will not work if run on a PowerMAX OS 4.3 system. The `vfprintf` in the 4.3 version of **libc.so** will not know how to handle the new `va_list` format.

7.2. Retain Source

Users are encouraged to retain the source for their applications. Major releases may have changes in the object-file format which will require the recompilation of their programs. This release is one such release. In the process of implementing additional ANSI/ISO C++ features, some changes to "name mangling" and interfaces to runtime routines were necessary.

7.3. curses.h and bool

The **<curses.h>** header file contains a **bool** type definition. In compilation modes where **bool** is a keyword, this results in a compilation error. There are two workarounds. The user may turn off the **bool** keyword with the **--no_bool** option, or he may use the following sequence when using the header file:

```
#define bool _curses_bool
#include <curses.h>
#undef bool
```

When using this second workaround, if the user makes reference to curses' `bool` type definition, the user must use the name **_curses_bool** instead.

7.4. Structure Compares

The C 4.3 and earlier releases supported an enhancement that allowed the user to implicitly compare a structure to zero (meaning all bytes of the structure are zero) in an if or while statement:

```
struct S {int a,b,c,d,e,f;} s;
...
if (s) {
...
```

As this enhancement is not present in any other major C compiler and is undocumented in the Concurrent C compiler, it is no longer present in C/C++ 5.1. The user can easily write a small function to test a structure for being zero should any code actually use this feature.

7.5. Known Problem with C/C++ Standard Library

The Concurrent software workaround provided below addresses a problem detected by the C++ standards committee Library Working Group. Concurrent will continue to monitor the situation and evaluate the group's final recommendation for any further action that might be necessary.

If a function is explicitly instantiated and its type is supplied, the compiler looks up all functions of that name and tries to instantiate them in an attempt to determine what the closest match is. If one of these cannot be instantiated for the given type, the compiler will then issue an error.

The workaround is to omit the explicit type, but let the compiler determine it from context. An example is provided below:

```
namespace std {
 template <class Iterator> struct iterator_traits {
   typedef typename Iterator::difference_type difference_type;
 };

 template <class T> class reverse_iterator;

 template <class T>
   void operator+(typename iterator_traits<T>::difference_type,
            const reverse_iterator<T>&);

 template <class T> struct complex;

 template <class T> void operator+(const T& lhs, const complex<T>& rhs){;}

}

// Explicit instantiation which gives the error:
template void std::operator+<float>(const float&, const std::complex<float>&);

// Workaround which does not:
template void std::operator+<>(const float&, const std::complex<float>&);
```

7.6. Pragma min_align

Using a pragma `min_align` of the form:

```
#pragma min_align [struct | union | class] name align
[struct | union | class] name {
  declaration
};
```

will result in a front end assertion violation.

The workaround is to declare the `[struct | union | class]` *name* previous to the `#pragma` statement.

Specifically:

```
struct foo_tag;
#pragma min_align foo_tag 16
struct foo_tag {
   double b1;
   double b2;
   double b3;
} foo;
```

will work while:

```
#pragma min_align foo_tag 16
struct foo_tag {
   double b1;
   double b2;
   double b3;
} foo;
```

will give an assertion violation.

8.0. Direct Software Support

Software support is available from a central source. If you need assistance or information about your system, please contact the Concurrent Software Support Center at 1-800-245-6453. Our customers outside the continental United States can contact us directly at 1-954-283-1822 or 1-305-931-2408. The Software Support Center operates Monday through Friday from 8 a.m. to 7 p.m., Eastern Standard time.

Calling the Software Support Center gives you immediate access to a broad range of skilled personnel and guarantees you a prompt response from the person most qualified to assist you. If you have a question requiring on-site assistance or consultation, the Software Support Center staff will arrange for a field analyst to return your call and schedule a visit.