VisualAge® C++ Professional for AIX®

Incremental Compilation Options Reference

Version 5.0

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Notices

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

References to VisualAge or VisualAge C++ in this book should be interpreted as VisualAge C++, Version 5.0.

Chapter 1. Options for Incremental Compilation

Build Options

Build options control the VisualAge C++ incremental C++ compiler, C compiler, and linker.

These options follow the same rules of syntax as all options in the configuration file. An option directive that specifies one option can take the following form:

option *name*(*value_list*)

The name of the option can be one of the compile or link options listed in the categories below. What is allowed in the list of values depends upon the specific option. If more than one value is allowed, values are separated by commas. Some options take yes or no values. In these cases, yes is assumed if the second value is omitted. For example, the following 2 lines are equivalent:

```
option link(exportAll, yes)
option link(exportAll)
```

Option names are not case sensitive, but literal data elements are case sensitive. For example, the following two lines are also equivalent.

```
option opt(isolatedCall, "MyFunc")
option OPT(isolatedcall, "MyFunc")
```

Options are processed in the order that they appear in option directives. If they conflict, the last option specified prevails. Some combinations of options accumulate.

Not all options apply to all platforms on which VisualAge C++ runs. If you set an option that does not apply to your platform, it is ignored by the build process. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.

Not all options apply to both C and C++ source files. If you specify an option for one or more source files, and none of those source files are of a type that uses that option, you receive a warning message.

Build options that apply to C and C++ are grouped into the following categories:

C “Language Options for C” on page 29

C++ “Language Options for C++” on page 3

AIX “Miscellaneous Build Options (AIX)” on page 41

AIX “Code Generation Options (AIX)” on page 59

AIX “Optimization Options (AIX)” on page 89

AIX “Link Options (AIX)” on page 107

C

The C compiler also supports options in the following categories:

- “C Preprocessor Options” on page 143
- “C Listing Options” on page 147
- “C File Options” on page 155

In many cases, you can use #pragma preprocessor directives in your source to achieve the same effect as specifying build options. In general, a pragma overrides the corresponding build option. There is one exception to the rule that pragmas override build options: to give you fine control over what messages are produced by a build, the report pragmas and report options in the miscellaneous options category can override #pragma info.

Some build options are used by build extensions. These options are described in discussions of using the individual build extensions.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files
How Configuration Files Are Processed
Build Extensions and Extension Specification Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
Configuration File Syntax
Build Extensions Supplied with VisualAge C++

Chapter 2. Language Options for C++

Language Options for C++

The options listed in the following table control how VisualAge C++ compiles C++ source files during builds.

Option names are not case-sensitive. If you omit both “yes” and “no” from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying lang(AllowDollarInNames) is equivalent to specifying lang(allowdollarinnames, yes).

Option	**Default**

lang(allowDollarInNames[, yes])
lang(allowDollarInNames, no)
	lang(allowDollarInNames, no)

lang(allowTypedefAsClassName[, yes])
lang(allowTypedefAsClassName, no)
	lang(allowTypedefAsClassName, no)

lang(anonymousStructs[, yes])
lang(anonymousStructs, no)
	lang(anonymousStructs[, yes])

lang(ansiForStatementScopes[, yes])
lang(ansiForStatementScopes, no)
	lang(ansiForStatementScopes[, yes])

lang(compatFriendDeclarations[, yes])
lang(compatFriendDeclarations, no)
	lang(compatFriendDeclarations[, yes])

lang(compatMath[, yes])
lang(compatMath, no)
	lang(compatMath, no)

lang(compatTempAccessChecking[, yes])
lang(compatTempAccessChecking, no)
	lang(compatTempAccessChecking[, yes])

lang(compatTemplateSpecialization[, yes])
lang(compatTemplateSpecialization, no)
	lang(compatTemplateSpecialization[, yes])

lang(digraphs[, yes])
lang(digraphs, no)
	lang(digraphs[, yes])

lang(extendedAnonymousUnions[, yes])
lang(extendedAnonymousUnions, no)
	lang(extendedAnonymousUnions[, yes])

lang(illformedPointerToMember[, yes])
lang(illformedPointerToMember, no)
	lang(illformedPointerToMember[, yes])

lang(implicitInt[, yes])
lang(implicitInt, no)
	lang(implicitInt[, yes])

lang(keyword, *string*)	none

lang(localTypedefs[, yes])
land(localTypedefs, no)
	lang(localTypedefs, no)

lang(localUnnamedEnums[, yes])
lang(localUnnamedEnums, no)
	lang(localUnnamedEnums, no)

lang(localUsingDirectives[, yes])
land(localUsingDirectives, no)
	lang(localUsingDirectives, no)

lang(longLong[, yes])
lang(longLong, no)
	lang(longLong[, yes])

WIN OS/2 lang(memoryFiles[, yes]) lang(memoryFiles, no)	lang(memoryFiles, no)

Option	Default

lang(newthrowsexception[, yes])
lang(newthrowsexception, no)
	lang(newthrowsexception, no)

lang(noKeyword, *string*)	recognize all C++ keywords

lang(offsetOfNonPODClasses[, yes])
lang(offsetOfNonPODClasses, no)
	lang(offsetOfNonPODClasses[, yes])

lang(oldDigraphs[, yes])
lang(oldDigraphs, no)
	lang(oldDigraphs, no)

lang(orderedNameLookup[, yes])
lang(orderedNameLookup, no)
	lang(orderedNameLookup, no)

lang(signedBitfields[, yes])
lang(signedBitfields, no)
	lang(signedBitfields, no)

lang(signedChars[, yes])
lang(signedChars, no)
	lang(signedChars, no)

lang(staticConstLinkage[, yes])
lang(staticConstLinkage, no)
	lang(staticConstLinkage, no)

lang(staticInlineLinkage[, yes])
lang(staticInlineLinkage, no)
	lang(staticInlineLinkage, no)

lang(templateDefImpls,dontParse)
lang(templateDefImpls, parseWithWarnings)
lang(templateDefImpls, parseWithErrors)
	lang(templateDefImpls, parseWithWarnings)

lang(trailingEnumCommas[, yes])
lang(trailingEnumCommas, no)
	lang(trailingEnumCommas[, yes])

lang(universalCharacterNames[, yes])
lang(universalCharacterNames, no)
	lang(universalCharacterNames, no)

lang(zeroExtentArrays[, yes])
lang(zeroExtentArrays, no)
	lang(zeroExtentArrays[, yes])

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1
Style Checker Options
C "Language Options for C" on page 29

allowDollarInNames Language Option

C

OS/2 WIN This option is not supported for C code on Intel platforms.

Syntax:	**Default:**

lang(allowDollarInNames[, yes])
lang(allowDollarInNames, no)
	lang(allowDollarInNames, no)

This option controls whether the dollar-sign character ($) is allowed in identifiers.

By default, dollar sign characters in identifiers are treated as syntax errors. Build with the default setting of this option to conform to the C++ and C standards.

RELATED CONCEPTS

C Compilation
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29

C++ "Build Options for Compliance With the C++ Standard" on page 180

C++ "Build Options for Compatibility with Old C++ Compilers" on page 182

allowTypedefAsClassName Language Option (C++)

Syntax:	**Default:**

lang(allowTypedefAsClassName[, yes])
lang(allowTypedefAsClassName, no)
	lang(allowTypedefAsClassName, no)

This option provides backwards compatibility with previous versions of VisualAge C++ and predecessor products.

The current C++ standard does not allow a typedef name to be specified where a class name is expected. This option relaxes that restriction. Set the allowTypedefAsClassName option to yes to allow the use of typedef names in base specifiers and constructor initializer lists.

By default, a typedef name cannot be specified where a class name is expected.

RELATED CONCEPTS

C Compilation
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29

C++ "Build Options for Compliance With the C++ Standard" on page 180

C++ "Build Options for Compatibility with Old C++ Compilers" on page 182

anonymousStructs Language Option (C++)

Syntax:	**Default:**

lang(anonymousStructs[, yes])
lang(anonymousStructs, no)
	lang(anonymousStructs[, yes])

This option controls whether anonymous structs and anonymous classes are allowed in your C++ source.

By default, VisualAge C++ allows anonymous structs. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by Microsoft® Visual C++.

Anonymous structs typically are used in unions, as in the following code fragment:

```
union U {
   struct {
      int i:16;
      int j:16;
   };
   int k;
} u;
// ...
u.j=3;
```

When this option is set to yes, you receive a warning if your code declares an anonymous struct. You can suppress the warning with "report message Miscellaneous Option (C++)" on page 55. When you build with lang(anonymousStructs, no) an anonymous struct is flagged as an error.

Set anonymousStructs to no for compliance with standard C++.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3

“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182

ansiForStatementScopes Language Option (C++)

Syntax:	**Default:**

lang(ansiForStatementScopes[, yes])
lang(ansiForStatementScopes, no)
	lang(ansiForStatementScopes[, yes])

This option controls whether scope rules defined in the C++ standard apply to names declared in for-init statements.

By default, standard C++ rules are used. For example the following code causes a name lookup error:

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
}
```

The reason for the error is that i, or any name declared within a for-init-statement, is visible only within the for statement. To correct the error, either declare i outside the loop or set ansiForStatementScopes to no.

Set ansiForStatementScopes to no to allow old language behavior. You may need to do this for code that was developed with other products, such as the compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182

compatFriendDeclarations Language Option (C++)

Syntax:	**Default:**

lang(compatFriendDeclarations[, yes])
lang(compatFriendDeclarations, no)
	lang(compatFriendDeclarations[, yes])

This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors.

By default, VisualAge C++ lets you declare a friend class without elaborating the name of the class with the keyword class. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the statement below declares the class IFont to be a friend class and is valid when compatFriendDeclarations is set to yes.

```
friend IFont;
```

Set compatFriendDeclarations to no for compliance with standard C++. The example declaration above causes a warning unless you modify it to the statement below, or suppress the message with "report message Miscellaneous Option (C++)" on page 55.

```
friend class IFont;
```

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

compatMath Language Option (C++)

Syntax:	Default:

lang(compatMath[, yes])
lang(compatMath, no)
	lang(compatMath, no)

This option controls which versions of math function declarations in <math.h> are included when you specify math.h as an included or primary source file.

By default, the new standard math functions are used. Build with lang(compatMath, no) for strict compliance with the C++ standard.

For compatibility with modules that were built with earlier versions of VisualAge C++ and predecessor products you may need to build with lang(compatMath, yes).

RELATED CONCEPTS

Incremental C++ Build
Configuration Files
C++ Source Files and Regions

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

compatTempAccessChecking Language Option (C++)

Syntax:	**Default:**
lang(compatTempAccessChecking[, yes])	lang(compatTempAccessChecking[, yes])
lang(compatTempAccessChecking, no)	

This option controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided.

By default, VisualAge C++ suppresses the access checking. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2® 3.0, VisualAge for C++ for Windows®, Version 3.5, and Microsoft Visual C++.

When this option is set to yes, you receive a warning if your code uses the extension, unless you disable the message. Disable the message by building with "report message Miscellaneous Option (C++)" on page 55 when the copy constructor is a private member, and report(disable, "CPPC0307") when the copy constructor is a protected member.

Set compatTempAccessChecking to no for compliance with standard C++. For example, the throw statement in the following code causes an error because the copy constructor is a protected member of class C:

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};
C foo() {return C("test");}  // returns a copy of a C object

void f()
{
// catch and throw both make implicit copies of the thrown object
   throw C("error");         // throws a copy of a C object
   const C& r = foo();       // uses the copy of a C object created by foo()
}
```

The example code above contains three ill formed uses of the copy constructor C(const C&).

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

compatTemplateSpecialization Language Option (C++)

Syntax:	**Default:**
lang(compatTemplateSpecialization[, yes])	lang(compatTemplateSpecialization[, yes])
lang(compatTemplateSpecialization, no)	

This option controls whether template specializations that do not conform to the C++ standard are allowed.

By default, VisualAge C++ allows these specializations. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When this option is set to yes, you receive a warning if your code uses the extension, unless you suppress the message with "report message Miscellaneous Option (C++)" on page 55.

For example, you can explicitly specialize the template class ribbon for type char with the following lines:

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

Set compatTemplateSpecialization to no for compliance with standard C++. In the example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options” on page 1
“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182

digraphs Language Option (C++)

Syntax:	**Default:**

lang(digraphs[, yes])
lang(digraphs, no)
	lang(digraphs[, yes])

This option controls whether standard digraphs and keyword operators are allowed in your C++ source.

By default, all standard forms are supported. The digraph, “<:”, which represents “[” can result in syntax errors when the first characters of a template argument are interpreted as the digraph. The following code shows how an unexpected error can occur:

```
template <int T> class C { };
const int i =1;
void f( ) {
   int i;
   C<::i> j    // error: this line is read as "C[:i. j"
   C< ::i > k  // no error due to the space beween "<" and ":"
}
```

If your code uses old-style digraphs, set“oldDigraphs Language Option (C++)” on page 20also to yes.

C

For C, whether digraphs are recognized is controlled by “cDigraphs Language Option (C)” on page 31.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182
“Language Options for C++” on page 3
“Build Options” on page 1
Digraph, Trigraph, and Escape Sequences

extendedAnonymousUnions Language Option (C++)

Syntax:	**Default:**

lang(extendedAnonymousUnions[, yes])
lang(extendedAnonymousUnions, no)
	lang(extendedAnonymousUnions[, yes])

This option controls what members are allowed in anonymous unions.

When this option is set to yes, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, typedefs, and enumerations are allowed.

Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option.

By default, VisualAge C++ allows non-data members in anonymous unions. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by previous versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this option is set to yes, you receive a warning if your code uses the extension, unless you suppress the message with “report message Miscellaneous Option (C++)” on page 55.

Set extendedAnonymousUnions to no for compliance with standard C++.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options” on page 1
“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182

illformedPointerToMember Language Option (C++)

Syntax:	**Default:**

lang(illformedPointerToMember[, yes])
lang(illformedPointerToMember, no)
	lang(illformedPointerToMember[, yes])

This option controls what expressions can be used to form pointers to members. VisualAge C++ can accept some forms that are in common use, but do not conform to the C++ standard.

By default, VisualAge C++ allows these forms. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this option is set to yes, you receive warnings if your code uses the extension, unless you suppress the messages with"report message Miscellaneous Option (C++)" on page 55 and "report message Miscellaneous Option (C++)" on page 55.

For example, the following code defines a pointer to a function member, p, and initializes it to the address of C::foo, in the old style:

```
struct C {
void foo(int);
};
void (C::*p) (int) = C::foo;
```

Set illformedPointerToMember to no for compliance with the C++ standard. The example code above must be modified to use the & operator.

```
struct C {
void foo(int);
};
void (C::*p) (int) = &C::foo;
```

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

implicitInt Language Option (C++)

Syntax:	**Default:**

lang(implicitInt[, yes])
lang(implicitInt, no)
	lang(implicitInt[, yes])

This option controls whether VisualAge C++ will accept missing or partially specified types as implicitly specifying int. This is no longer accepted in the standard but may exist in legacy code.

With the option set to no, all types must be fully specified.

With the option set to yes, a function declaration at namespace scope or in a member list will implicitly be declared to return int. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. Note that the effect is as if the int specifier were present. This means that the specifier const, by itself, would specify a constant integer.

The following specifiers do not completely specify a type.

- auto
- const
- extern
- extern "<literal>"
- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- platform specific types (for example, _cdecl, __declspec)

Note that any situation where a type is specified is affected by this option. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, dynamic_cast, new), and types for conversion functions.

By default, VisualAge C++ sets implicitInt to yes. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the return type of function MyFunction is int because it was omitted in the following code:

```
MyFunction()
{
   return 0;
}
```

Set implicitInt to no for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
   return 0;
}
```

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

keyword Language Option (C++)

Syntax:	**Default:**
lang(keyword, *string*)	recognize all C++ keywords

This option controls whether the specified string is treated as a keyword or an identifier whenever it appears in your C++ source.

By default all the built-in keywords defined in the C++ standard are reserved as keywords.

You cannot add keywords to the C++ language with this option. However, you can use it to enable built-in keywords that have been disabled using lang(noKeyword, *string*). For example, you can reinstate bool with the following option:

```
lang(keyword, "bool")
```

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

localTypedefs Language Option (C++)

Syntax:	**Default:**
lang(localTypedefs[, yes])	lang(localTypedefs, no)
lang(localTypedefs, no)	

This option controls whether types defined by typedef statements are globally visible or are local to the C++ source file that contains the definition of the type.

By default, VisualAge C++ treats typedefs as global declarations. Usually, the default setting is appropriate for incremental compiling.

The default does not conform to ANSI C++, but facilitates treating header files as sources for builds. If you convert an included header file to a primary file, make sure you build with the default. Otherwise other sources cannot use types defined in typedef statements in the converted headers.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
One-Definition Rule

RELATED TASKS

Convert Included Source Files to Primary Source Files
Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options for Compliance With the C++ Standard” on page 180

localUnnamedEnums Language Option (C++)

Syntax:	**Default:**
lang(localUnnamedEnums[, yes]) lang(localUnnamedEnums, no)	lang(localUnnamedEnums, no)

This option restricts the visibility of enumerators to the primary source in which they are declared. This supports the standards definition of these enumerators having no linkage, allowing enumerators in unnamed enumerations to have different values in different source files. Setting lang(localUnnamedEnums) prevents the use of enumerators in other source files when an unnamed enumeration in a header file has been promoted to the configuration file.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
One-Definition Rule

RELATED TASKS

Convert Included Source Files to Primary Source Files
Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options for Compliance With the C++ Standard” on page 180

localUsingDirectives Language Option (C++)

Syntax:	Default:
lang(localUsingDirectives[, yes]) lang(localUsingDirectives, no)	lang(localUsingDirectives, no)

This option controls whether directives are visible only to the file in which they are specified, or globally to all files listed in the configuration file.

The default setting, no, treats directives as being globally visible. The default does not conform to ANSI C++.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
One-Definition Rule

RELATED TASKS

Convert Included Source Files to Primary Source Files
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180

newThrowsException Language Option (C++)

Syntax:	Default:
lang(newThrowsException[, yes]) lang(newThrowsException, no)	lang(newThrowsException, no)

This option determines whether or not the C++ operator new throws an exception. The standard exception std::bad_alloc can be thrown when the requested memory allocation fails. This option does not apply to the nothrow versions of the new operator.

The standard implementation of the new operators fully support exceptions. For compatibility with previous versions of VisualAge C++, these operators return 0 by default.

RELATED CONCEPTS

Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C" on page 29
"Language Options for C++" on page 3
"Build Options" on page 1
Standard Library <new>

noKeyword Language Option (C++)

Syntax:	**Default:**
lang(noKeyword, *string*)	

This option controls whether the specified string is treated as a keyword or an identifier whenever it appears in your C++ source.

By default, all the keywords defined by the C++ standard are reserved names.

If you have code that has identifier names that have been adopted by the C++ standard as keywords, use this option to treat a keyword as an ordinary identifier. Apply the option repeatedly to treat more than one keyword as an ordinary identifier.

If you define a macro that is a standard digraph, you need to use lang(digraph,no) or else the compiler will treat it as a digraph.

For compatibility with earlier versions of VisualAge C++ and predecessor products, and with Microsoft Visual C++, build with the following options:

```
lang(noKeyword,"asm")
lang(noKeyword,"bool")
lang(noKeyword,"const_cast")
lang(noKeyword,"explicit")
lang(noKeyword,"export")
lang(noKeyword,"false")
lang(noKeyword,"mutable")
lang(noKeyword,"namespace")
lang(noKeyword,"reinterpret_cast")
lang(noKeyword,"static_cast")
lang(noKeyword,"true")
lang(noKeyword,"typename")
lang(noKeyword,"using")
```

You can reinstate a keyword using lang(keyword, *string*).

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3

"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182
Digraph, Trigraph, and Escape Sequences

offsetOfNonPODClasses Language Option (C++)

Syntax:	**Default:**
lang(offsetOfNonPODClasses[, yes])	lang(offsetOfNonPODClasses[, yes])
lang(offsetOfNonPODClasses, no)	

This option controls whether the offsetof macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes "Plain Old Data" (POD) classes.

By default, VisualAge C++ allows offsetof to be used with nonPOD classes. This is an extension to the C++ standard, and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++

When this option is set to yes, you receive a warning if your code uses the extension, unless you suppress the message with "report message Miscellaneous Option (C++)" on page 55.

Set offsetOfNonPODClasses to no for compliance with standard C++.

Set offsetOfNonPODClasses to yes if your code applies offsetof to a class that contains one of the following:
- user-declared constructors or destructors
- user-declared assignment operators
- private or protected non-static data members
- base classes
- virtual functions
- non-static data members of type pointer to member
- a struct or union that has non-data members
- references

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1

"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

oldDigraphs Language Option (C++)

Syntax:	**Default:**

lang(oldDigraphs[, yes])
lang(oldDigraphs, no)
	lang(oldDigraphs, no)

This option controls whether old-style digraphs are allowed in your C++ source. It applies only when "digraphs Language Option (C++)" on page 11 is also set to yes.

By default, VisualAge C++ supports only the digraphs specified in the C++ standard.

Set this option to yes if your code contains at least one of following digraphs:

Digraph	Resulting Character
%%	# (pound sign)
%%%%	## (double pound sign, used as the preprocessor macro concatenation operator)

Set this option to no for compatibility with standard C++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

orderedNameLookup Language Option (C++)

Syntax:	**Default:**

lang(orderedNameLookup[, yes])
lang(orderedNameLookup, no)
	lang(orderedNameLookup, no)

This option allows you to exploit all the advantages of the VisualAge C++ incremental environment while maintaining source compatibility with other traditional C++ compilers. This option makes the compiler try to resolve all names to the same declarations that a traditional compiler would. In other words, the option helps you write portable code by maintaining all the #include directives and forward declarations necessary to compile your source with a compiler other than VisualAge C++.

For each name referenced in a *nonincluded source* file (t.cpp) the lang(orderedNameLookup, yes) option ensures that only declarations occurring before the referencing location are found by name lookup. The option checks that the declaring location is either in t.cpp and before the referencing location or that the declaring location is in a source file which is #included (directly or indirectly) before the referencing location.

For each name referenced in an *included source* file (t.hpp), the list of nonincluded source files which directly or indirectly include t.hpp is determined. Ordered name lookup ensures that for each nonincluded source only declarations which are included or declared before t.hpp is included are considered. An error is issued if the set of visible declarations is not the same for each nonincluded source.

The term included source is used to refer to files actually included into another source as well as source files named on a #include directive but not included because the file is listed in the configuration file. A nonincluded source is a file listed in the configuration file but not mentioned on a #include directive.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

C++ “Language Options for C++” on page 3
“Build Options” on page 1

signedBitfields Language Option

Syntax:	Default:

lang(signedBitfields[, yes])
lang(signedBitfields, no)
	lang(signedBitfields, no)

This option determines whether plain non-char bitfields are signed or unsigned.

By default, plain non-char bitfields are treated as unsigned.

This option affects plain long long, plain long, plain int and plain short bitfields, but not plain char bitfields. The “signedChars Language Option” on page 22 option

controls whether plain chars are signed or unsigned.
For example, this option affects only j in the declaration below because j is not implicitly signed or unsigned:

```
struct S {
   signed int i:1;
   int j:1;
   unsigned int k:1;
};
```

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29
"Build Options" on page 1

signedChars Language Option

Syntax:	**Default:**

lang(signedChars[, yes])
lang(signedChars, no)
	lang(signedChars, no)

This option determines whether char variables in your C or C++ source are treated as signed or unsigned char in arithmetic and compare operations. It does not convert the type of a char to a signed char or an unsigned char.

By default, char variables are treated as unsigned char.

Set signedChars to yes to treat a char as a signed char.

The "signedBitfields Language Option" on page 21 option controls whether non-char bitfields types are treated as signed or unsigned.

You can also specify whether chars are signed or unsigned by using #pragma chars in your source. The pragma overrides this option.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

▶ C++ “Language Options for C++” on page 3

▶ C “Language Options for C” on page 29
“Build Options” on page 1

staticConstLinkage Language Option (C++)

Syntax:	**Default:**
lang(staticConstLinkage[, yes])	lang(staticConstLinkage, no)
lang(staticConstLinkage, no)	

This option controls whether const variables are treated as static or extern.

By default, VisualAge C++ treats const variables as extern. Usually, the default setting is appropriate for incremental compile.

The default does not conform to the C++ standard, but facilitates treating header files as sources for builds. If you convert an included file to a primary file, make sure you build with the default. Otherwise variables and functions declared const but not declared external may acquire internal linkage.

Set this option to yes for compatibility with standard C++ or the ANSI C++ language level supported by previous versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Convert Included Source Files to Primary Source Files
Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C++” on page 3
“Build Options for Compliance With the C++ Standard” on page 180
“Build Options for Compatibility with Old C++ Compilers” on page 182

staticInlineLinkage Language Option (C++)

Syntax:	**Default:**
lang(staticInlineLinkage[, yes])	lang(staticInlineLinkage, no)
lang(staticInlineLinkage, no)	

This option controls whether inline functions are treated as static or extern.

By default, VisualAge C++ treats inline functions as extern. Usually, the default setting is appropriate for incremental compile.

The default facilitates treating header files as sources for builds. If you convert an included file to a primary file, make sure you build with the default. Otherwise, variables and functions declared const but not declared external may acquire internal linkage.

For example, using the default lang(staticInlineLinkage, yes) setting causes function f in the following declaration to be treated as static, even though it is not explicitly declared as such.

```
inline void f() {/*...*/};
```

Using the default lang(staticInlineLinkage, no) setting gives f external linkage.

Set staticInlineLinkage to yes for compatibility with C++ compilers provided by previous versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Convert Included Source Files to Primary Source Files
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

templateDefImpls Language Option (C++)

Syntax:	Default:

lang(templateDefImpls,dontParse)
lang(templateDefImpls, parseWithWarnings)
lang(templateDefImpls, parseWithErrors)
	lang(templateDefImpls, parseWithWarnings)

This option controls whether parsing and semantic checking are applied to template definition implementations (function bodies and static data member initializers) or only to template instantiations. VisualAge C++ can check function bodies and variable initializers in template definitions and produce error or warning messages.

By default, VisualAge C++ parses template implementations and issues warning messages for semantic errors. Error messages are issued for errors found while parsing.

This option applies to template definitions, not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside implementations. For example, errors found during the parsing or semantic checking of constructs such as the following, always cause error messages:

- return type of a function template
- parameter list of a function template
- member list of a class template
- base specifier of a class template

Set templateDefImpls to dontParse to reduce the number of errors issued in code written for previous versions of VisualAge C++ and predecessor products.

Set templateDefImpls to parseWithErrors to treat problems in template implementations as errors, even if the template is not instantiated.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

trailingEnumCommas Language Option (C++)

Syntax:	**Default:**

lang(trailingEnumCommas[, yes])
lang(trailingEnumCommas, no)
	lang(trailingEnumCommas[, yes])

This option controls whether trailing commas are allowed in enum declarations.

By default, VisualAge C++ allows one or more trailing commas at the end of the enumerator list. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The following enum declaration uses this extension:

```
enum grain { wheat, barley, rye,, };
```

Set trailingEnumCommas to no for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

universalCharacterNames Language Option

Syntax:	**Default:**

lang(universalCharacterNames[, yes])
lang(universalCharacterNames, no)
	lang(universalCharacterNames, no)

This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in C and C++ sources.

The Unicode character set is supported by the ISO standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set.

When this option is set to yes, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are \u*hhhh* for 16-bit characters, or \U*hhhhhhhh* for 32-bit characters, where *h* represents a hexadecimal digit.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1

zeroExtentArrays Language Option (C++)

Syntax:	**Default:**

lang(zeroExtentArrays[, yes])
lang(zeroExtentArrays, no)
	lang(zeroExtentArrays[, yes])

This option controls whether zero-extent arrays are allowed as the last non-static data member in a class definition.

By default, VisualAge C++ allows arrays with zero elements. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The example declarations below define dimensionless arrays a and b.

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

Set zeroExtentArrays to no for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

When this option is set to yes, you receive warnings about zero-extent arrays in your code, unless you suppress the message with "report message Miscellaneous Option (C++)" on page 55.

RELATED CONCEPTS

Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C++" on page 3
"Build Options" on page 1
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

Chapter 3. Language Options for C

Language Options for C

The options listed in the following table control how VisualAge C++ compiles C source files during builds.

- If an option does not apply to all platforms, the middle column shows the platforms on which it is used.
- If you specify options that are not recognized on your platform, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case-sensitive. If you omit both “yes” and “no” from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying lang(AllowDollarInNames) is equivalent to specifying lang(allowdollarinnames, yes).

Option	Applies only to	Default

lang(allowDollarInNames[, yes])
lang(allowDollarInNames, no)
	AIX	lang(allowDollarInNames, no)

lang(cDigraphs[, yes])
lang(cDigraphs, no)
		lang(cDigraphs,no)

lang(checkNonProto[, yes])
lang(checkNonProto, no)
	AIX	lang(checkNonProto, no)

lang(dbcs[, yes])
lang(dbcs, no)
		lang(dbcs, no)

lang(ddNames[, yes])
lang(ddNames, no)
	OS/2 WIN	lang(ddNames, no)

lang(ignObsoleteKeywords[, yes])
lang(ignObsoleteKeywords, no)
	OS/2 WIN	lang(ignObsoleteKeywords, no)

lang(level, ansi)
lang(level, extended)
lang(level, saal2)
AIX lang(level, saa)
AIX lang(level, classic)
		AIX lang(level, ansi)
		OS/2 WIN lang(level, extended)

lang(longLong[, yes])
lang(longLong, no)
		lang(longLong[, yes])

lang(macpstr[, yes])
lang(macpstr, no)
	AIX	lang(macpstr, no)

lang(memoryFiles[, yes])
lang(memoryFiles, no)
	OS/2 WIN	lang(memoryFiles, no)

lang(pascal[, yes])
lang(pascal, no)
	AIX	lang(pascal, no)

lang(preserveUnsignedPromotion[, yes])
lang(preserveUnsignedPromotion, no)
		lang(preserveUnsignedPromotion, no)

lang(signedBitfields[, yes])
lang(signedBitfields, no)
		lang(signedBitfields, no)

lang(signedChars[, yes])
lang(signedChars, no)
		lang(signedChars, no)

Option	Applies only to	Default

lang(slashSlashComments[, yes])
lang(slashSlashComments, no)
		AIX lang(slashSlashComments, no)
		OS/2 WIN lang(slashSlashComments[, yes])

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3
Style Checker Options
"Build Options" on page 1

allowDollarInNames Language Option

C

OS/2 WIN This option is not supported for C code on Intel platforms.

Syntax:	Default:

lang(allowDollarInNames[, yes])
lang(allowDollarInNames, no)
	lang(allowDollarInNames, no)

This option controls whether the dollar-sign character ($) is allowed in identifiers.

By default, dollar sign characters in identifiers are treated as syntax errors. Build with the default setting of this option to conform to the C++ and C standards.

RELATED CONCEPTS

C Compilation
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29

C++ "Build Options for Compliance With the C++ Standard" on page 180
C++ "Build Options for Compatibility with Old C++ Compilers" on page 182

cDigraphs Language Option (C)

Syntax:	**Default:**

lang(cDigraphs[, yes])
lang(cDigraphs, no)
	lang(cDigraphs, no)

This option controls whether digraphs defined in the C standard are allowed in your C source.

By default, C digraphs are not recognized.

C++
For C++, whether the standard digraphs are recognized is controlled by "digraphs Language Option (C++)" on page 11.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C" on page 29
"Build Options" on page 1

checkNonProto Language Option (C, AIX)

Syntax:	**Default:**

lang(checkNonProto[, yes])
lang(checkNonProto, no)
	lang(checkNonProto, no)

This option asserts that procedure call points agree with their declaration even if the procedure has not been prototyped. It allows the caller to pass floating point arguments in the floating point registers only, and not in the General-Purpose Registers (GPRs). The compiler assumes that the arguments on procedure calls are the same types as the corresponding parameters of the procedure definition.

RELATED CONCEPTS

C Compilation: Comparing C and C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1
"Language Options for C" on page 29

dbcs Language Option (C)

Syntax:	**Default:**

lang(dbcs[, yes])
lang(dbcs, no)
	lang(dbcs, no)

This option controls whether the C compiler supports multi-byte characters or allows only single-byte characters in the source.

By default, VisualAge C++ supports only single-byte characters in C source. This improves performance of character handling during compilation.

Specify lang(dbcs) if you have any multi-byte characters in your C source.

C++
In C++ source single- and multi-byte characters are always supported.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C" on page 29
"Build Options" on page 1

level Language Option (C)

Syntax:	**Default:**

lang(level, ansi)
lang(level, extended)
lang(level, saal2)
AIX lang(level, saa)
AIX lang(level, classic)
	AIX lang(level, ansi)
	OS/2 WIN lang(level, extended)

This option determines the language level applied to C source. You can set this option to conform to the C standard, to allow all extended constructs provided by VisualAge C++, or to comply to IBM SAA Level 2 standards.

▶ AIX

By default on AIX, C code must comply to the standard.

▶ OS/2

▶ WIN The default language level is extended on OS/2 and Windows NT® platforms. This allows extended constructs for C that are supported by VisualAge C++.

To ensure complete comformance to the C standard, you must ensure the following options and defines are set:

▶ AIX	lang(level, ansi) “longLong Language Option” “define (Preprocessor Macros) Miscellaneous Build Option” on page 45

▶ OS/2 ▶ WIN	lang(level, ansi)
	“longLong Language Option”
	“slashSlashComments Language Option (C)” on page 40
	opt(integer, no)
	opt(float, no)

You can also set the language level from within your source with #pragma langlvl. The pragma overrides this configuration file option.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Language Options for C” on page 29
“Build Options” on page 1

longLong Language Option

Syntax:	Default:

lang(longLong[, yes])
lang(longLong, no)
	lang(longLong[, yes])

This option controls whether long long int objects are allowed in your C or C++ source.

By default, VisualAge C++ supports type long long as an 8-byte integer. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by VisualAge for C++ for Windows, Version 3.5 and Microsoft Visual C++.

C++

When this option is set to no, you receive an error if your C++ code uses type long long.

Set longLong to no for compliance with standard C or C++ or with the ANSI language level supported by earlier versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29

C++ "Build Options for Compliance With the C++ Standard" on page 180

C++ "Build Options for Compatibility with Old C++ Compilers" on page 182

macpstr Language Option (C, AIX)

Syntax:	**Default:**

lang(macpstr[, yes])
lang(macpstr, no)
	lang(macpstr, no)

This option converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.

A Pascal string literal contains the characters "`\p`. The characters `\p` in the middle of a string do not form a Pascal string literal. The characters must be immediately preceded by the " (double quote) character.

The final length of the Pascal string literal can be no longer than 255 bytes.

For example, `lang(macpstr, yes)` converts:

```
"/pABC
```

to

```
'\03' , 'A' , 'B' , 'C' , '\0'
```

The compiler ignores the macpstr option when the lang(dbcs, yes) option is specified because Pascal-string-literal processing is only valid for single-byte characters.

The #pragma options keyword macpstr is only valid at the top of a source file before any C or C++ source statements. If you attempt to use it in the middle of a source file, it is ignored and the compiler issues an error message.

Examples of Pascal String Literals

The compiler replaces trigraph sequences by the corresponding single-character representation. For example:

```
"??/p pascal string"
```

becomes:

```
"\p pascal string"
```

The following are examples of valid Pascal string literals:

ANSI Mode

```
"\p pascal string"
```

Each instance of a new-line character and an immediately preceding backslash (\) character is deleted, splicing the physical source lines into logical ones. For example:

```
"\p pascal \
 string"
```

Two Pascal string literals are concatenated to form one Pascal string literal. For example:

```
"\p ABC" "\p DEF"
```

or

```
"\p ABC" "DEF"
```

becomes:

```
"\06ABCDEF"
```

For the macro ADDQUOTES:

```
#define ADDQUOTES (x) #x
```

where x is:

```
\p pascal string
```

or

```
\p pascal \
 string
```

becomes:

```
"\p pascal string"
```

Note however that:

```
ADDQUOTES(This is not a "\p pascal string")
```

becomes:

```
"This is not a \"\\p pascal string\""
```

Extended Mode

This mode is the same as ANSI mode, except the macro definition would be:

```
#define ADDQUOTES_Ext (x) "x"
```

Where x is the same as in the ANSI example:

```
\p pascal string
\p pascal \
 string
```

String Literal Processing

The following describes how Pascal string literals are processed.

- Concatenating a Pascal string literal to a normal string gives a non-Pascal string. For example:

  ```
  "ABC" "\pDEF"
  ```

  gives:

  ```
  "ABCpDEF"
  ```

- A Pascal string literal cannot be concatenated with a **wide** string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.
- If `lang(dbcs, yes)` is used, the compiler ignores `lang(macpstr, yes)` and issues a warning message.
- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence **\p** in a wide string literal with `lang(macpstr, yes)`, generates a warning message and the escape sequence is ignored.
- The Pascal string literal is *not* a basic type different from other C or C++ string literals. After the processing of the Pascal string literal is complete, the resulting string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.
- Concatenating two Pascal string literals, for example, `strcat()`, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.
- Entering the following characters into a character array does not form a Pascal string literal:

  ```
  '\p' , 'A' , 'B' , 'C' , '\0'
  ```

RELATED CONCEPTS

C Compilation: Comparing C and C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1
"Language Options for C" on page 29

pascal Language Option (C, AIX)

Syntax:	Default:

lang(pascal[, yes])
lang(pascal, no)
	lang(pascal, no)

This option accepts, with `lang(pascal, yes)`, or ignores, with `lang(pascal, no)`, the word `pascal` in type specifiers and function declarations.

RELATED CONCEPTS

C Compilation: Comparing C and C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1
"Language Options for C" on page 29

preserveUnsignedPromotion Language Option (C)

Syntax:	Default:

lang(preserveUnsignedPromotion[, yes])
lang(preserveUnsignedPromotion, no)
	lang(preserveUnsignedPromotion, no)

This option controls whether unsignedness is preserved when an integer value is promoted in C code. This option does not apply when explicit type casts are used, and affects only integral promotions. It controls the signedness of the type of the resulting value, not its sign.

By default, integer promotions conform to the C standard. Such type conversions preserve values but not signedness.

If you set preserveUnsignedPromotion to yes, type conversions preserve signedness. That is, unsigned types will remain unsigned when they are promoted. This does not conform to the C standard.

If an expression mixes integer types, VisualAge C++ applies any implicit conversions required to make the types the same. This option controls whether the conversions applied are "unsigned preserving" or "value preserving". In unsigned preserving conversions, an unsigned short int or an unsigned char would be promoted to an unsigned int. In value preserving conversions, they would be promoted to a signed int, except in implementations where the size of a short is the same size as an int.

For example, the sum of a signed int plus an unsigned char would be treated in the following ways, as determined by the setting of this option:

Setting	**Effect**
lang(preserveUnsignedPromotion, no)	Convert the unsigned char to a positive, signed int, then add two signed ints, giving a signed int. This is the default.
lang(preserveUnsignedPromotion, yes)	Convert the unsigned char to an unsigned int, convert the signed int to an unsigned int, then add the two unsigned ints, giving a result of type unsigned int.

The bit pattern remains the same in either case, but its type and meaning will be different, which could affect subsequent operations.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

C “Language Options for C” on page 29
“Build Options” on page 1

signedBitfields Language Option

Syntax:	**Default:**
lang(signedBitfields[, yes]) lang(signedBitfields, no)	lang(signedBitfields, no)

This option determines whether plain non-char bitfields are signed or unsigned.

By default, plain non-char bitfields are treated as unsigned.

This option affects plain long long, plain long, plain int and plain short bitfields, but not plain char bitfields. The “signedChars Language Option” on page 22 option controls whether plain chars are signed or unsigned.
For example, this option affects only j in the declaration below because j is not implicitly signed or unsigned:

```
struct S {
   signed int i:1;
   int j:1;
   unsigned int k:1;
};
```

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29
"Build Options" on page 1

signedChars Language Option

Syntax:	**Default:**

lang(signedChars[, yes])
lang(signedChars, no)
	lang(signedChars, no)

This option determines whether char variables in your C or C++ source are treated as signed or unsigned char in arithmetic and compare operations. It does not convert the type of a char to a signed char or an unsigned char.

By default, char variables are treated as unsigned char.

Set signedChars to yes to treat a char as a signed char.

The lang(signedBitfields) option controls whether non-char bitfields types are treated as signed or unsigned.

You can also specify whether chars are signed or unsigned by using #pragma chars in your source. The pragma overrides this option.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

C++ "Language Options for C++" on page 3

C "Language Options for C" on page 29
"Build Options" on page 1

slashSlashComments Language Option (C)

Syntax:	**Default:**
lang(slashSlashComments[, yes]) lang(slashSlashComments, no)	AIX lang(slashSlashComments, no) OS/2 WIN lang(slashSlashComments[, yes])

This option controls whether the C compiler accepts C++-style comments that begin with a double-slash (//) and end at the end of the line of source.

For example, the following line is valid when slashSlashComments is set to yes:

```
i += 2; // increment i by 2
```

Set slashSlashComments to no to allow only C-style (/* ... */) comments, so that the code above must be rewritten into something like the following line.

```
i += 2; /* increment i by 2 */
```

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Language Options for C" on page 29
"Build Options" on page 1

Chapter 4. Miscellaneous Build Options

Miscellaneous Build Options (AIX)

The options listed in the following table control builds for AIX platforms in ways that are not specific to the source language, code generation, optimization, or linking.

- If an option does not apply to both C and C++, the middle column shows which type of source that it affects.
- If you specify options that do not apply to AIX, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case sensitive. If you omit both “yes” and “no” from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying incl(SearchPathFirst) is equivalent to specifying incl(searchPathfirst, yes).

Option	**Applies only to**	**Default**

alloc(debug[, yes])
alloc(debug, no)
		alloc(debug, no)

debug(maxErrors, *number*)	▶ C	none
defaults(*string*)		none
define(*macro_name*[, *string*])		none
incl(externC, *prefix*)	▶ C++	none
incl(noExternC, *prefix*)	▶ C++	none
incl(searchPath, *path*)		none

incl(searchPathFirst[, yes])
incl(searchPathFirst, no)
		incl(searchPathFirst, no)

incl(standardInclude[, yes])
incl(standardInclude, no)
		incl(standardInclude, yes)

incl(userStrategy, including)
incl(userStrategy, primary)
	▶ C++	incl(userStrategy, including)

info(*group*, yes)
info(*group*, no)
info(all, yes)
info(all, no)
	▶ C	info(gen[, yes])
		info(pro[, yes])
		info(cnd[, yes])

macros(global[, yes])
macros(global, no)
	▶ C++	macros(global, no)

macros(redefine[, yes])
macros(redefine, no)
	▶ C++	macros(redefine, no)

misc(optionString, *string*)	▶ C	none

report(level, E)
report(level, W)
report(level, I)
		report(level, I)

report(enable, *message*)
report(disable, *message*)
	▶ C++	none

Option	**Applies only to**	**Default**
report(treatAsError, *message*)	▶ C++	none

undefine(*macro_name*)
undefine(“*”)
		retain all macros

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Build Options” on page 1

alloc Miscellaneous Option

Syntax:	Default:

alloc(debug[, yes])
alloc(debug, no)
	alloc(debug, no)

This option controls whether the debug versions of memory management functions, including heap management functions, are used to build the target. When this option is set to yes, the debug versions of the memory management functions replace the regular memory management functions. For example, _debug_malloc is used when the source calls malloc, _debug_heapmin replaces _heapmin, and so on.

▶ C++

For C++ code, this option links debug versions of new and delete.

When this option is set to yes, code is inserted at the beginning of every function to initialize the bytes of all local variables with the value 0xAA. This makes it easier to find uninitialized local variables, and is equivalent to building with “initAuto Code Generation Option” on page 76.

The debug functions record the line number and file name of each call from your source to a memory management function. You can use this information as an aid for debugging. Setting this option to yes also defines the macro __DEBUG_ALLOC__.

By default VisualAge C++ uses the regular memory management functions. The regular allocation functions do not initialize the storage.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

debug maxErrors Miscellaneous Option (C)

Syntax:	**Default:**
debug(maxErrors, *number*)	none

This option sets the maximum number of errors allowed before the C compiler terminates.

By default, there is no limit on the error count. However, C compilation stops on the first unrecoverable error.

RELATED CONCEPTS

C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

defaults Miscellaneous Build Option (AIX)

Syntax:	**Default:**
defaults(*string*)	None

This option allows you to specify a predefined option group (or stanza), to use during compilation.

For example, if you specify `defaults(xlC)`, source files are compiled as C++ language source code.

The predefined option groups, represented by *string*, are listed in the following table.

xlC vacbld vacide	xlC128 vacbld128 vacide128	xlC128_r vacbld128_r vacide128_r	xlC128_r7 vacbld128_r7 vacide128_r7	xlC_r vacbld_r vacide_r	xlC_r4 vacbld_r4 vacide_r4	xlC_r7 vacbld_r7 vacide_r7
xlc	xlc128	-	-	xlc_r	xlc_r4	xlc_r7
cc	cc128	-	-	cc_r	cc_r4	cc_r7
CC_r4	-	-	-	-	-	-

c89	-	-	-	-	-	-

Note that vacbld and vacide are equivalent to xlC, and vacbld and vacide variations are equivalent to xlC variations.

The basic compiler option groups invoke the compiler as indicated below:

xlC Invokes the compiler so that source files are compiled as C++ language source code. Files with .c file extensions are compiled as C language source code.

xlc Invokes the compiler for C or C++ source files with a default language level of ansi, and specifies option opt(alias, ansi, yes) to allow type-based aliasing. Use this option group for new C programs.

cc Invokes the compiler for C source files with a default language level of extended and specifies options gen(readOnly, no) and gen(readOnlyConst, no), to provide compatibility with the RT compiler and placement of string literals or constant values in read/write storage. Use this option group for legacy C code that does not require compliance with ANSI C.

CC_r4 This option group is the equivalent of the xlC_r4 option group.

c89 Invokes the compiler for C source files with a default language level of ansi, and specifies options opt(alias, ansi, yes) to allow type-based aliasing, define("_ANSI_C_SOURCE',1) for ANSI-conformant headers, and lang(longLong, no) to disable use of longlong. Use this option group for strict conformance to the ANSI standard.

Variations of the basic compiler option groups are defined as follows:

Option groups with 128 suffix
These are functionally similar to their corresponding base option group. They specify gen(longDouble, yes) to increase the length of long double types in your program from 64 to 128 bits.

Option groups with _r suffix
In addition to the options set in the base option group, option groups with this suffix set macro names with define("_THREAD_SAFE", 1), define("_AIX32_THREADS",1), and define("_AES_SOURCE",1). They also add the libraries libpthreads.a, libc_r.a, and /usr/lib/libc.a. Use these option group variations if you want to create POSIX-threaded applications.

Option groups with _r4 suffix
In addition to the options set in the base option group, option groups with this suffix set macro names with define("_THREAD_SAFE", 1), define("_AIX32_THREADS",1), and define("_AES_SOURCE",1). They also add the libraries libdcelibc_r.a, libdcepthreads, libpthreads.a, libc_r.a, and libc.a. Use these option group variations if you want to create DCE-threaded applications for AIX.

Options groups with _r7 suffix
In addition to the options set in the base option group, options groups with this suffix set macro names with define("_THREAD_SAFE", 1), and define(_AIX_PTHREADS_D7, 1). For xlc_7 and cc_7, the libraries libpthreads_compat.a, libpthreads.a, libc_r.a, and libc.a are added. For xlC_7, the libraries libC.a, libpthreads_compat.a, libpthreads.a, libc_r.a, libm.a, and /usr/lib/libc.a are added.

Use these option group variations in AIX Version 4.3, for multithread compatibility with AIX Version 4.2.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Add or Change Source Files in the IDE
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

define (Preprocessor Macros) Miscellaneous Build Option

Syntax:	**Default:**
define(*macro_name*[, *string*])	none

This option defines preprocessor macros that can be used in the associated source files or source groups. By default, only the VisualAge C++ predefined macros and ANSI/ISO predefined macros are defined.

Use this option to define macros without having to modify sources, or to provide macros that apply in some circumstances but should not be defined at other times.

The first argument of the define option names the macro. The second argument is the replacement list of preprocessor tokens to be assigned to the macro. If you do not supply a string, the macro has the default definition, "1". You can specify "" to define a macro for the null string. If the replacement string is more than one word, delimit it with double quotes, as in the following example:

```
define(MYMACRO, "a b c")
```

Use the \ character as an escape character if the replacement string contains string literal or a reserved character, such as \. For example, \" represents a " (double quote) and \\ represents a \ (backslash). In the example below, the MY_DIR macro is give the value of `"c:\mydir"`. Note that in this example the replacement string includes the double quotes on each end of the string.

```
define(MY_DIR, "\"C:\\MYDIR\"")
```

If you specify the define option more than once with the same macro name, the last definition overrides previous replacement strings for the macro.

The macro is defined for all source files to which this option applies. Macros defined with this option are local to the source files, even if "macros global Miscellaneous Build Option (C++)" on page 52 also applies to the sources. For example, the following lines define ERA to be the constant 100 in a.cpp and c.cpp, and 1000 in b.cpp:

```
option macros(global), define(ERA, 100) {
   source "a.cpp"
   option define(ERA, 1000) {
      source "b.cpp"
   }
   source "c.cpp"
}
```

A macro named by the define option takes effect at the beginning of the source, and remains in effect until the end of the source or until the macro is undefined or redefined within the source. While it is in effect, the definition from the define option hides other visible definitions of the same macro. At the end of the source, the previous definition is restored. However, if the macro is undefined or redefined in the source, the change in definition remains in effect.

You can undefine a macro with *“undefine Miscellaneous Option” on page 57*).

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

C++ Macros in C++ Source Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Miscellaneous Build Options (AIX)” on page 41

incl externC Miscellaneous Build Option (C++, AIX)

Syntax:	**Default:**
incl(externC, *prefix*)	None

Use this option to specify include files whose names are prefixed by *prefix*, to automatically insert `extern "C" {` at the start of the included files, and a right brace (}) at the end of the files. Specify the option once for each *prefix* desired.

For example, if you specify:

```
incl(externC, /usr/include)
```

all include files that are prefixed by `/usr/include` have `extern "C" {` inserted at the start of each include file, and a right brace (}) inserted at the end of each file.

If both incl(externC) and “incl noExternC Miscellaneous Build Option (C++, AIX)” on page 47 are specified with the same prefix, incl(noExternC) takes precedence and the prefix is not inserted in the include files.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

AIX Search Paths for Include Files

RELATED TASKS

Add or Change Source Files
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"incl noExternC Miscellaneous Build Option (C++, AIX)"

AIX "Miscellaneous Build Options (AIX)" on page 41

incl noExternC Miscellaneous Build Option (C++, AIX)

Syntax:	**Default:**
incl(noExternC, *prefix*)	None

Use this option to specify which include files, identified by *prefix*, are not to have the implicit `extern "C"` inserted in each file. Specify the option once for each *prefix* desired. The insertion of the implicit `extern "C"` is specified by the "incl externC Miscellaneous Build Option (C++, AIX)" on page 46 option.

For example, if you specify:

```
incl(externC, /usr/include)
incl(noExternC, /usr/include/cppfiles)
```

all include files prefixed by `/usr/include`, except for files prefixed by `/usr/include/cppfiles`, have
`extern "C" {` inserted at the start of each include file, and a right brace (}) inserted at the end of each file.

If both incl(externC) and incl(noExternC) are specified with the same prefix, incl(noExternC) takes precedence and the prefix is not inserted in the include files.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

AIX Search Paths for Included Source Files

RELATED TASKS

Add or Change Source File in the IDE
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"incl externC Miscellaneous Build Option (C++, AIX)" on page 46

AIX "Miscellaneous Build Options (AIX)" on page 41

incl searchPath Miscellaneous Build Option

Syntax:	**Default:**
incl(searchPath, *path*)	none

This option determines which directories are searched to satisfy #include directives in the C and C++ source files. It also establishes the paths for source files that are specified with relative paths.

If you use this option more than once with the same option directive, the paths are searched in the order that the search paths appear in the configuration file. If you nest option directives, include search paths specified in inner directives are searched before include search paths specified in outer directives. For example, the following lines set up a search that looks in the subdirectories of directory root in the order c, d, a, b:

```
option incl(searchPath, '\root\a'), incl(searchPath, '\root\b') {
   option incl(searchPath, '\root\c'), incl(searchPath, '\root\d') {
      source 'x.cpp'
   }
}
```

For files that are included by directives that have the form #include "*filename.h*", the directory containing the including source file is searched before the include search path, by default. To search directories in the include search paths before the directory that contains the including file, also apply option"incl searchPathFirst Miscellaneous Build Option" on page 49.

C++

For C++ user included files, set "incl userStrategy Miscellaneous Build Option (C++)" on page 50 or incl(userStrategy, including) to determine which file is considered to be the including file.

C

OS/2 WIN For C source files on OS/2 and Windows NT platforms, you also can tell the compiler to ignore the paths specified with incl(searchPath, *path*) by building with file(ignoreUserIncPaths).

RELATED CONCEPTS

Incremental C++ Build
C Compilation

AIX Search Paths for Included Files
C++ Source Files and Source Regions

RELATED TASKS

Add or Change Source Files
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

incl searchPathFirst Miscellaneous Build Option

C

OS/2 WIN This option is not supported for C source files on OS/2 and Windows NT platforms.

Syntax:	Default:

incl(searchPathFirst[, yes])
incl(searchPathFirst, no)
	incl(searchPathFirst, no)

This option sets the search order for user include files. It does not affect the search order for system include files.

You can use this option in conjunction with incl(searchPath, *path*). However, the incl(searchPathFirst) option has no effect if no paths are specified with the incl(searchPath, *path*) option.

If the default, incl(searchPathFirst,no), is specified, the current directory is searched first, followed by the incl(searchPath) directories, and then finally the INCLUDE or default system directories.

If incl(searchPathFirst,yes) is specified, the incl(searchPath) directories are searched first, followed by the current directory, and then finally the INCLUDE or default system directories.

C++

For C++ user included files, set "incl userStrategy Miscellaneous Build Option (C++)" on page 50 or incl(userStrategy, including) to determine what file is considered to be the including file.

RELATED CONCEPTS

Incremental C++ Build
C Compilation

AIX Search Paths for Include Files

RELATED TASKS

Add or Change Source Files in the IDE
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

incl standardInclude Miscellaneous Build Option (AIX)

Syntax:	**Default:**

incl(standardInclude[,yes])
incl(standardInclude, no)
	incl(standardInclude[,yes])

This option determines whether standard include directories are searched for included source files.

If you build with the default setting of incl(standardInclude, yes), the following directories are appended, in order, to the list of paths VisualAge C++ searches to find included source files:

1. C++ ./usr/vacpp/include
2. /usr/include

You can set incl(standardInclude, no) to search only the paths specified with *"incl searchPath Miscellaneous Build Option" on page 48*).

RELATED CONCEPTS

Incremental C++ Build
C Compilation

AIX Search Paths for Include Files

RELATED TASKS

Add or Change Source Files
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

incl userStrategy Miscellaneous Build Option (C++)

Syntax:	**Default:**

incl(userStrategy, including)
incl(userStrategy, primary)
	AIX incl(userStrategy, including)
	OS/2 WIN incl(userStrategy, primary)

Use this option to define the search strategy for C++ user-included files. Included files are source files that are included in a build because they are listed in #include preprocessor directives in other source files. If a file is named as a primary C++ source in your configuration file, as well as being included with a preprocessor directive, VisualAge C++ treats it as a primary source and not as an included source.

Specify primary to search the directory where the primary including source is located. A primary source may include other sources directly or indirectly. For example, if primary source a.cpp includes b.hpp which in turns includes x.h, VisualAge C++ looks in the directory that contains a.cpp to satisfy the #include directives in both a.cpp and b.hpp, regardlesss of whether they are in the same directory.

Specify including to search the directory where the directly including file is located, even if it is not a primary source. For example, if primary source a.cpp includes b.hpp which in turns includes x.h, VisualAge C++ looks in the directory that contains a.cpp to satisfy the #include directives in a.cpp and in the directory that contains b.hpp to satisfy the #include directive in b.hpp.

This option tells VisualAge C++ where to look for user-included files, as well as in the directory specified with *"incl searchPath Miscellaneous Build Option" on page 48*). To search directories in the include search paths before the directory that contains the including file, also apply option "incl searchPathFirst Miscellaneous Build Option" on page 49.

The default setting of this option provides the same behavior as in earlier versions of VisualAge C++ and predecessor products.

RELATED CONCEPTS

Incremental C++ Build

AIX Search Paths for Include Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

info Miscellaneous Build Option (C)

Syntax:	**Default:**
info(*group*, yes)	info(gen[, yes])
info(*group*, no)	info(pro[, yes])
info(all, yes)	info(cnd[, yes])
info(all, no)	

This option determines which information messages are output during C compilation.

By default, the C compiler outputs messages that belong in the following groupings:
- general diagnostics
- missing function prototypes
- possible redundancies or problems in conditional expressions

Information messages report potential coding errors, poor programming style, and information about the structure of your program. They are categorized into "Information Message Groups" on page 185. Specify info(grp) for each category of messages you want to see.

To see all messages, specify info(all). You can combine info(all) with info(grp, no) to eliminate some specific groups of messages and reduce the volume of information output.

To suppress all information messages specify info(all, no). Combine info(all, no) with info(grp) to eliminate all but certain groups of messages.

You can also control messages from your C and C++ source code, by using #pragma info.

If you want to control the output of messages from compilation based on severity level, use "report level Miscellaneous Build Option" on page 54.

RELATED CONCEPTS

C Compilation
Messages Produced by Incremental Builds

RELATED TASKS

View Build Messages
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41
"Information Message Groups" on page 185

macros global Miscellaneous Build Option (C++)

Syntax:	Default:

macros(global[, yes])
macros(global, no)
	macros(global, no)

This option controls whether macros in the associated source files have local scope or global scope.

Set macros global to yes if the source contains macros that apply to the whole project.

Set macros global to no if all macros in the source are local to the file that contains them. This is the default setting.

This option is usually applied to files that are named as sources in the configuration file, and would be included by other sources on a traditional compiler. If such files define macros, you should apply macros(global) to them.

Source files to which macros(global) applies are called macro sources. Macro sources are effectively ordered by their appearance in the configuration file.

VisualAge C++ does not follow the standard C++ rules of scope for preprocessor macros that are declared in macro sources. These macros are visible in all other macro sources that are listed subsequently in the configuration file. Macros that are defined in macro sources are visible in all source files that are not macro sources, regardless of where such source files are listed in the configuration file.

C

The C compiler follows standard C rules for scope and the processing of #include preprocessor directives.

RELATED CONCEPTS

Incremental C++ Builds
C++ Source Files and Source Regions
Macros in C++ Source Files

RELATED TASKS

Set Build Options
Build
Convert Included Source Files to Primary Source Files

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

macros redefine Miscellaneous Option (C++)

Syntax:	**Default:**

macros(redefinel[, yes])
macros(redefine, no)
	macros(redefine, no)

This option controls whether macros in the associated C++ source files can be redefined without using the #undef preprocessor directive first.

A macro defined using the configuration file option define(*macro_name*[,*string*]) can be redefined without using the #undef option, regardless of the setting of the macros(redefine) option.

Set macros redefine to yes to allow a #define preprocessor directive for a macro that is currently defined.

By default VisualAge C++ enforces the rules of the C++ standard, and does not let you redefine a macro in your source without undefining it first.

RELATED CONCEPTS

Incremental C++ Build
Macros in C++ Source Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41
"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182

optionString Miscellaneous Option (C)

Syntax:	**Default:**
misc(optionString, *string*)	none

This option specifies a string of command-line compile options that are passed directly to the C compiler.

Use this option to specify C compile options in the command line format used by earlier versions of VisualAge C++ and predecessor products.

OS/2

WIN You can specify any of the compile options that were recognized by the old **icc** command except /B, which was used to pass link options on to the linker. Linker options will be ignored.

AIX

You can specify linker options to be passed to the system linker using the link(system) and link(systemOption) options.

All old options that are still supported have equivalent build options, and using this option gives you no advantage over using the build options. C compile options specified with misc(optionString, *string*) are appended to build options specified in the usual way.

RELATED CONCEPTS

C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

AIX "Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

report level Miscellaneous Build Option

Syntax:	**Default:**

report(level, E)
report(level, W)
report(level, I)
	report(level, I)

The report level determines which messages are produced during builds.

Set the report level to E, W or I to set the minimum severity level to error, warning, or information, respectively. By default all messages are output, and you can set the level to W or E to reduce the volume of compile and link messages.

▶ C++

For C++ you can also enable or disable specific error messages with report(enable, *number*) or report(disable, *number*). For a message to appear, it must be enabled and match at least the minimum severity level set with this option. You can set a message to have error-level severity by specifying report(treatAsError, *number*)

▶ C

You can also control which type of information messages are output during C compilation using the info(*group*) option.

You can also specify whether information messages are output by using #pragma info in your source. Although pragmas usually override configuration file options, the report options and pragmas override #pragma info so that you have greater control of information message output.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Messages Produced by Incremental Builds

RELATED TASKS

View Build Messages
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Miscellaneous Build Options (AIX)" on page 41

report message Miscellaneous Option (C++)

Syntax:	**Default:**

report(enable, *message*)
report(disable, *message*)
	none

This option controls whether specific messages are produced during C++ builds.

To output a message specify the message identifier with report(enable, *message*). To suppress the message use report(disable, *message*). VisualAge C++ messages are identified by alphanumeric strings that start with a prefix of one or more letters, and end with one or more digits.

By default all information, warning, and error messages are displayed.

For a message to appear, it must be enabled and match at least the minimum severity level set by *"report level Miscellaneous Build Option" on page 54*). However, you can tell the compiler to handle an information or warning message as though it had error severity by specifying *"report treatAsError Miscellaneous Option (C++)"*).

You can also control whether specific messages are output by using #pragma report in your source.

▶ C

For C code, use*"info Miscellaneous Build Option (C)" on page 51*)to control which messages are produced during compilation.

RELATED CONCEPTS

Incremental C++ Build
Messages Produced by Incremental Builds

RELATED TASKS

View Build Messages
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Miscellaneous Build Options (AIX)" on page 41

report treatAsError Miscellaneous Option (C++)

Syntax:	**Default:**
report(treatAsError, *message*)	none

Using this option with a message identifier causes that message be treated as though it had error severity level.

By default, message severity is not changed.

To treat an information or warning message as an error, use this option with the identifier for that message. VisualAge C++ messages are identified by alphanumeric strings that start with a prefix of one or more letters, and end with one or more digits. If you specify the identifier of an error message, this option has no effect.

To control whether information, warning, or only error messages are produced during build, use *"report level Miscellaneous Build Option" on page 54*). You can also suppress or output specific error messages with the *"report message Miscellaneous Option (C++)" on page 55*) or report(disable, *message*) option. For a message to actually appear, this option must be enabled and match at least the minimum severity level.

▶ C

For C code, use *"info Miscellaneous Build Option (C)" on page 51*) to control which messages are produced during compilation.

RELATED CONCEPTS

Incremental C++ Build
Messages Produced by Incremental Builds

RELATED TASKS

View Build Messages
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

undefine Miscellaneous Option

Syntax:	Default:

undefine(*macro_name*)
undefine("*")
	retain all macros

This option undefines the preprocessor macro with the given name for each source file to which the option applies. This option can hide macros defined with the define(*macro_name*[, *string*]) option in the configuration file but not macros defined in the source files. The macro becomes visible again if it is redefined in the source.

This option is ignored if no macro of the given name is defined in the configuration file in the scope where the undefine appears.

Specify undefine("*") to undefine all macros defined in the configuration file.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

C++ Macros in C++ Source Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

Chapter 5. Code Generation Options

Code Generation Options (AIX)

The options listed in the following table control code generation for AIX platforms during builds.

- If an option does not apply to both C and C++, the middle column shows the type of source that it affects.
- If you specify options that do not apply to AIX, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case sensitive. If you omit both "yes" and "no" from an option that has a yes or no form, yes is implied regardless of whether the default setting for that option is yes or no. For example specifying gen(DebugUnreferenced) is equivalent to specifying gen(debugunreferenced, yes).

Option	Applies Only to	Default
gen(arch, *CPUtype)*		gen(arch, common)
gen(align, *option*)		gen(align, power)

gen(check, *option*[, yes])
gen(check, *optio*n, no)
		gen(check, bounds, no)
		gen(check, nullpointer, no)
		gen(check, zerodivide, no)

gen(compatNestedTemplateAlignmentRule[, yes])
gen(compatNestedTemplateAlignmentRule,no)
	C++	gen(compatNestedTemplateAlignmentRule, no)

gen(debug[, yes])
gen(debug, no)
	C	gen(debug, no)

gen(debugUnreferenced[, yes])
gen(debugUnreferenced, no)
		gen(debugUnreferenced, no)

gen(eh[, yes])
gen(eh, no)
	C++	gen(eh[, yes])

gen(enumSize, int)
gen(enumSize, small)
gen(enumSize, *bytes*)
		gen(enumSize, int)

gen(externStaticLinkage[,yes])
gen(externStaticLinkage,no)
		gen(externStaticLinkage, no)

gen(float, *option*[, yes])
gen(float, *option*, no)
		gen(float, emulate, no)
		gen(float, fltint, no)
		gen(float, fold, yes)
		gen(float, hsflt, no)
		gen(float, hssngl, no)
		gen(float, maf, yes)
		gen(float, nans, no)
		gen(float, rndsngl, no)
		gen(float, rrm, no)
		gen(float, rsqrt, no)
		gen(float, spnans, no)

Option	Applies Only to	Default

gen(floatTrap, *option*[, yes])
gen(floatTrap, *option*, no)
		gen(floatTrap, overflow, no)
		gen(floatTrap, underflow, no)
		gen(floatTrap, zerodivide, no)
		gen(floatTrap, invalidOperation, no)
		gen(floatTrap, inexact, no)
		gen(floatTrap, enable, no)
		gen(floatTrap, imprecise, no)

gen(funcSect[, yes])
gen(funcSect, no)
	C	gen(funcSect, no)

gen(initAuto[, yes])
gen(initAuto, *value*)
gen(initAuto, no)
		gen(initAuto, no)

gen(inlinePointerGlue[, yes])
gen(inlinePointerGlue, no)
		gen(inlinePointerGlue, no)

gen(libANSI[, yes])
gen(libANSI, no)
		gen(libANSI, no)

gen(lineOnlyDebug[, yes])
gen(lineOnlyDebug, no)
	C	gen(lineOnlyDebug, no)

gen(longDouble[, yes])
gen(longDouble, no)
		gen(longDouble, no)

gen(objectmode, 32)
gen(objectmode, 64)
		gen(objectmode, 32)

gen(objectModel, compat)
gen(objectModel, ibm)
	C++	gen(objectModel, compat)

gen(profile[, yes])
gen(profile, no)
gen(profile, bsd)
gen(profile, ibm)
		gen(profile, no)

gen(readOnly[, yes])
gen(readOnly, no)
		gen(readOnly[, yes])

gen(readOnlyConst[, yes])
gen(readOnlyConst, no)
		gen(readOnlyConst, yes)

gen(roundConstFp, *option*)		gen(roundConstFp, nearest)

gen(rtti[, yes])
gen(rtti, all)
gen(rtti, typeInfo)
gen(rtti, dynamicCast)
gen(rtti, no)
	C++	gen(rtti, no)

gen(strictinduction[, yes])
gen(strictinduction, no)
		gen(strictinduction, no)

gen(vft[, yes])
gen(vft, no)
	C++	gen(vft, yes)

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

arch Code Generation Option

Syntax:
gen(arch, *CPUtype)*

Default:
AIX gen(arch, common)
OS/2 WIN gen(arch, 486)

This option picks an instruction set or a common subset of instruction sets for a processor or a family of processors. You can specify a string representing one of the processor types for your platform as the value of *CPUtype*.

AIX

Processor Types (AIX)

This option determines what instruction set is used for the generated code but does not cause optimization for the target architecture. To optimize for a processor, use opt(tune, *CPUtype*), which optimizes the scheduling of the generated instructions for a particular processor.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
AIX "Code Generation Options (AIX)" on page 59

align Code Generation Option (AIX)

Syntax:
gen(align, *option*)

Default:
gen(align, power)

This option specifies which aggregate alignment rules VisualAge C++ applies to a build target. Use this option to specify the maximum alignment to be used when mapping a class-type object, either for the whole source program or for specific parts.

The default is gen(align, power).

The align suboptions are described in the following table.

power	Use the RISC System/6000® alignment rules.
full	Use the RISC System/6000 alignment rules. The power option is the same as full.
mac68k	Use the Macintosh** alignment rules.
twobyte	Use the Macintosh alignment rules. The mac68k option is the same as twobyte.
packed	Use the packed alignment rules.
bit_packed	The compiler uses the **bit_packed** alignment rules. Alignment rules for **bit_packed** are the same as that for **packed** alignment except that bitfield data is packed on a bit-wise basis without respect to byte boundaries.
natural	Map structure members to their natural boundaries. This has the same effect as the power suboption, except that it also applies alignment rules to doubles and long doubles that are not the first member of a structure or union.

If you use the gen(align) option more than once, the last alignment rule specified applies.

You can also specify alignment options by using #pragma options align=*option*. The pragma overrides the configuration file option.

Within your source file, you can also use #pragma options align=reset to revert to a previous alignment rule. VisualAge C++ stacks other alignment directives, so you can go back to using the previous alignment directive, without knowing what it is, by specifying the #pragma align=reset directive. If no previous alignment rule appears in the file, the alignment rule specified in the invocation command is used. For example, you can use this option if you have a class declaration within an include file and you do not want the alignment rule specified for the class to apply to the file in which the class is included.

OS/2

WIN On the OS/2 and Windows NT platforms, use the gen(pack) option to align data items within structures.

Example 1 - Embedded #pragmas

Applying the gen(align, mac68k) option, so that the Macintosh alignment rules are used by default, to a target with the following source code:

```
struct A {
  int a;
  struct B {
    char c;
    double d;
#pragma options align=power /* <— B will be unaffected by this       */
                            /*     #pragma, unlike previous behavior; */
```

```
                         /*     Macintosh alignment rules still    */
                         /*     in effect                          */
  } BB;
#pragma options align=reset /* <— A unaffected by this #pragma;      */
} AA;                    /*     Macintosh alignment rules still    */
                         /*     in effect                          */
```

Example 2 - Affecting Only Aggregate Definition

Allowing the default gen(align, power) option to apply, so that the RISC System/6000 alignment rules are used, to a target with the following source code:

```
extern struct A A1;
typedef struct A A2;
#pragma options align=packed /* <— use packed alignment rules        */
struct A {
int a;
char c;
};
#pragma options align=reset /* <— Go back to default alignment rules */
struct A A1; /* <— aligned using packed alignment rules since       */
A2 A3;       /*    this rule applied when struct A was defined     */
```

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

AIX RISC System/6000 Alignment Rules

AIX Macintosh and Twobyte Alignment Rules

AIX Packed Alignment Rules

AIX Alignment Rules for Nested Aggregates

check Code Generation Option (AIX)

Syntax:	Default:
gen(check,*option*[, yes])	gen(check,bounds, no)
gen(check,*option*, no)	gen(check,nullpointer, no)
	gen(check,zerodivide, no)

This option generates code that performs certain types of run-time checking. If a violation is encountered, a run-time exception is raised by sending a SIGTRAP signal to the process.

The following can be specified for *option*:

`bounds`	Performs run-time checking of addresses when subscripting within an object of known size. The index is checked to ensure that it will result in an address that lies within the bounds of the object's storage. A trap will occur if the address does not lie within the bounds of the object's storage.
`nullpointer`	Performs run-time checking of addresses that are contained in pointer variables used to reference storage. The address is checked at the point of use; a trap will occur if the value is less than 512.
`zerodivide`	Performs run-time checking of integer division. A trap will occur if an attempt is made to divide by zero.

You can specify gen(check) more than once. The *option* settings are accumulated, but the later *option* settings override the earlier ones. For example, if you specify:
`gen(check,bounds,yes)`
`gen(check,nullpointer,yes)`
and then specify
`gen(check,bounds,no)`
the options that take effect are `gen(check,nullpointer, yes)` and `gen(check,bounds,no)`.

C

The #pragma options directive must be specified before the first statement in the compilation unit.

C++ The #pragma options directive can be specified anywhere in the source.

The gen(check) option affects the run-time performance of the applications. When checking is enabled, run-time checks are inserted into the application, which may result in slower execution.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

compatNestedTemplateAlignmentRule Code Generation Option (C++, AIX)

Syntax:	Default:
gen(compatNestedTemplateAlignmentRule [, yes]) gen(compatNestedTemplateAlignmentRule, no)	gen(compatNestedTemplateAlignmentRule, no)

The `gen(compatNestedTemplateAlignmentRule)` option determines when the compiler decides which set of alignment rules to use for nested classes of class templates. The xlC compilers prior to Version 5.0 would pick the alignment rules to use at the time the template is instaniated. By default, the VisualAge C++ 4.0 compiler, and subsequent versions, will pick the alignment rules at the time the nested class definition is seen. For example, given the following template, the size of `A<char>::B` will be 5 with `gen(compatNestedTemplateAlignmentRule,no)` and 8 with `gen(compatNestedTemplateAlignmentRule,yes)`.

```
template <class T>
struct A
   {
   #pragma options align=packed
   struct B
       {
       T m;
       int m2;
       };
   #pragma options align=reset
};
```

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

debug Code Generation Option (C)

Syntax:	Default:
gen(debug[, yes]) gen(debug, no)	gen(debug, no)

This option determines whether the C compiler generates information for debugging. To debug code generated from C source, you must use a traditional debugger, which requires that line number and other information be included in the executable image.

By default, the C compiler does not generate debug information.

This option is implictly set to no by file(genObject, no).

To produce an executable, DLL or shared library with debug information, you also need to set the link(debug[, yes]) option for the target.

To make full use of the debugger, do not use this option together with *"Optimization Levels (AIX)" on page 90*) when *level* has a non-zero value. The compiler produces debugging information as if the code is not optimized, and may produce symbol and type information that is not accurate.

The VisualAge C++ debugger or the debugger from the IBM C and C++ Compilers Version 3.6 (or 3.6.5) can be used to debug these applications.

For limited debugging, gen(lineOnlyDebug) encodes line numbers in the code but not does not provide full debug information.

C++

Code generated from C++ source is always enabled for debugging.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Debugging with VisualAge C++

RELATED TASKS

Set Build Options
Build
Debug Optimized Code

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

debugUnreferenced Code Generation Option

Syntax:	Default:

gen(debugUnreferenced[, yes])
gen(debugUnreferenced, no)
	gen(debugUnreferenced, no)

This option determines whether information about unreferenced variables is included in the symbol table.

By default, unreferenced variables are omitted from the symbol table.

▶ C

If you build C code with "debug Code Generation Option (C)" on page 65, the target module is smaller than if you accept the default.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Code Generation Options (AIX)" on page 59
Option Configuration File Directive

eh Code Generation Option (C++)

Syntax:	Default:
gen(eh[, yes])	gen(eh[, yes])
gen(eh, no)	

This option controls whether C++ exception handling is enabled in the module being built.

By default, eh is set to yes.

If your program does not use C++ exception handling, build with gen(eh,no) to prevent generation of code that is not required for your application.

This is a global codestore option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options for the Project
Build

RELATED REFERENCES

▶ AIX "Code Generation Options (AIX)" on page 59
option Configuration File Directive

enumSize Code Generation Option

Syntax:
gen(enumSize, int)
gen(enumSize, small)
gen(enumSize, *bytes*)

Default:
AIX gen(enumSize, int)
OS/2 WIN gen(enumSize, small)

This option sets the storage space allocated to enum variables, as described in the table below:

int	Enumerations occupy 4 bytes of storage, using only signed integer as the underlying data type.	AIX Default.
small	Enumerations occupy a minimal amount of storage: either 1, 2, or 4 bytes of storage, depending on the range of the enum constants.	OS/2 WIN Default.
bytes = 1, 2, or 4	Enumerations occupy 1, 2, or 4 bytes of storage.	

When gen(enumSize, small) is specified, the underlying data type used for enum variables is selected according to the following priority: unsigned char, signed char, unsigned short, signed short, unsigned int, and signed int.

If the specified storage size is smaller than that required by the range of enum variables, an (S) error message is issued.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
AIX “Code Generation Options (AIX)” on page 59

externStaticLinkage Code Generation Option (AIX)

Syntax:
gen(externStaticLinkage[,yes])
gen(externStaticLinkage, no)

Default:
gen(externStaticLinkage, no)

This option determines whether calls to static routines within a compilation unit are generated as if they were calls to external routines.

By default, they are not.

Code generated with gen(externStaticLinkage, yes) is slower than code generated with the default setting.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

float Code Generation Option (AIX)

Syntax:	**Default:**
gen(float, *option*[, yes])	gen(float, emulate, no)
gen(float, *option*, no)	gen(float, fltint, no)
	gen(float, fold, yes)
	gen(float, hsflt, no)
	gen(float, hssngl, no)
	gen(float, maf, yes)
	gen(float, nans, no)
	gen(float, rndsngl, no) *See the description below.*
	gen(float, rrm, no)
	gen(float, rsqrt, no)
	gen(float, spnans, no)

This option specifies various floating-point options. These options provide different strategies for speeding up or improving the accuracy of floating-point calculations. They affect the generated code, but not operations performed in library functions. The following float options can be specified:

emulate	hssngl	rrm
fltint	maf	rsqrt
fold	nans	spnans
hsflt	rndsngl	

Each option can be turned on and off independently. The purpose of each option is described in the table below.

Using the gen(float) options may produce results that are not precisely the same as the default. Incorrect results may be produced if not all required conditions are met. For these reasons, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly assess the possibility of introducing errors in your program.

You can also specify floating-point options by using #pragma options float=*option*. The pragma overrides the configuration file option.

emulate	Emulates the floating-point instructions omitted by the PowerPC 403™ processor. The default is gen(float, emulate, no). If gen(float, emulate, yes) is set, function calls are emitted in place of PowerPC floating-point instructions. Use this option only in a single-threaded, stand-alone environment targeting the PowerPC 403 processor. **Note**: Currently you must also manually link in the /usr/lib/libfpCS.a library to use this option. You can add a `source "libfpCS.a"` directive to your configuration file. Do not specify gen(float, emulate, yes) with any of the following options: • "arch Code Generation Option" on page 61, gen(arch, pwr2), gen(arch,pwr2s), gen(arch, pwrx), gen(arch,p2sc) • "longDouble Code Generation Option (AIX)" on page 79, this is the default • *"defaults Miscellaneous Build Option (AIX)" on page 43*), where *string* names any compiler stanza that includes any one of the options listed above, such as defaults(xlC128)
fltint	Speeds up floating-point-to-integer conversions by using faster inline code that does not check for overflows. The default is gen(float, fltint, no), which checks floating-point-to-integer conversions for out-of-range values. The default depends on the optimization level and strict options. • For "Optimization Levels (AIX)" on page 90, the default is gen(float, fltint, no). • For opt(level, 3), the default is gen(float, fltint, yes). To include range checking in floating-point-to-integer conversions with the opt(level, 3) option, specify gen(float, fltint, no). • opt(strict, no) sets gen(float, fltint, yes). Changing the optimization level will not change the setting of the fltint option if gen(float, fltint, yes) has already been specified. If the opt(strict) and gen(float) options conflict, the last setting is used.

fold	Controls whether constant floating-point expressions are to be evaluated at compile time or at run time. The default is gen(float, fold, yes), which enables compile-time evaluation of constant floating-point expressions.
hsflt	Specifies that: • double precision values are truncated instead of rounded when converted to single precision for storing in a variable • conversion from double precision to single precision can occur before expression evaluation • either the full double precision value or the truncated single precision value of a variable may be used in subsequent expressions; not all usage need be of the same value • floating-point-to-integer conversions are not checked for out-of-range values The default is gen(float, hsflt, no), which specifies that single-precision expressions are rounded after expression evaluation and floating-point-to-integer conversions are checked for out-of-range values. Specifying gen(float, hsflt, yes) speeds up calculations, but should be used only for specific applications in which floating-point computations have known characteristics. Using this option when you are compiling other application programs can produce incorrect results without warning. This suboption overrides the nans, rndsngl, and spnans suboptions.
hssngl	Specifies that single-precision expressions are rounded only when the results are stored into `float` memory locations, and subsequent calculations using values of variables which are still in registers may use their unrounded double-precision values. The default is gen(float, hssngl, no), which specifies that single-precision expressions are rounded after expression evaluation. Using gen(float, hssngl) can improve run-time performance, but it is safer than using gen(float, hsflt).

maf	Controls whether floating-point multiply-add instructions are used where appropriate. The default is gen(float, maf, yes). Specifying gen(float, maf, yes) makes floating-point calculations faster and, usually, more accurate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. This option may affect the precision of floating-point intermediate results.
nans	Controls whether extra instructions are generated to detect signaling NaN (Not-a-Number) when converting from single precision to double precision at run time. The default is gen(float, nans, no), which specifies that this conversion is not detected. The default also permits more folding of floating-point expressions; for example, the expression "x*1.0" could be folded to "x". You must specify gen(float, nans, yes) for full compliance to the IEEE 754 standard. When used with the "floatTrap Code Generation Option (AIX)" on page 74 option, the compiler detects invalid operation exceptions in comparison operations that occur when one of the operands is a signaling NaN. If your program contains signaling NaNs, you should use the gen(floatTrap, invalid) option along with the gen(float, nans) option to trap any exceptions. The hsflt suboption overrides the nans suboption.
rndsngl	Controls whether the result of each single-precision (`float`) operation is rounded to single precision. When gen(arch) is used to specify any PowerPC architecture, this option is always on. Specifying gen(float, rndsngl, yes) may sacrifice speed for consistency with results from similar calculations on other types of computers. The hsflt option overrides the rndsngl option.

rrm	Prevents floating-point optimizations that are incompatible with run-time rounding to plus or minus infinity or zero modes. Informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not *round to nearest* at run time. The default is gen(float, rrm, no). You must specify gen(float, rrm) if the Floating Point Status and Control register is changed at run time.
rsqrt	Specifies whether a sequence of code that involves division by the result of a square root can be replaced by calculating the reciprocal of the square root and multiplying. Allowing this replacement produces code that runs faster. The default is gen(float, rsqrt, no). The following notes apply to this option: • For “Optimization Levels (AIX)” on page 90, the default is gen(float, rsqrt, no). • For opt(level, 3), the default is gen(float, rsqrt, yes). Specify gen(float, rsqrt, no) to override this default. • opt(strict, no) sets gen(float, rsqrt, yes). (Note that this means that `errno` will *not* be set for any `sqrt` function calls.) • Specifying gen(float, rsqrt, yes) has no effect when “arch Code Generation Option” on page 61 is also specified. • Specifying gen(float, rsqrt, yes) has no effect unless “ignErrno Optimization Option” on page 96 is also specified. Changing the optimization level will not change the setting of the rsqrt option if gen(float, rsqrt, yes) has already been specified. If the opt(strict) and gen(float) options conflict, the last setting is used.
spnans	Controls whether extra instructions are generated to detect signaling NaN on conversion from single precision to double precision. The default is gen(float, spnans, no), which specifies that this conversion need not be detected. This option is obsolete, and is replaced by the equivalent gen(float, nans) option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

floatTrap Code Generation Option (AIX)

Syntax:	**Default:**

gen(floatTrap, *option*[, yes])
gen(floatTtrap, *option*, no)
	gen(floatTrap, overflow, no)
	gen(floatTrap, underflow, no)
	gen(floatTrap, zerodivide, no)
	gen(floatTrap, invalidOperation, no)
	gen(floatTrap, inexact, no)
	gen(floatTrap, enable, no)
	gen(floatTrap, imprecise, no)

This option specifies whether to generate instructions to detect and trap floating-point exceptions.

By default, floating-point exceptions are not detected and trapped.

The following table describes the effect of specifying gen(floatTrap, *option*) for each of the floatTrap suboptions.

Option	**Effect**
overflow	Generates code to detect and trap floating-point overflow exceptions.
underflow	Generates code to detect and trap floating-point underflow exceptions.
zerodivide	Generates code to detect and trap floating-point division by zero exceptions.
invalidOperation	Generates code to detect and trap floating-point invalid operation exceptions.
inexact	Generates code to detect and trap floating-point inexact exceptions.
imprecise	Generates code for imprecise detection of the specified exceptions. If an exception occurs, it is detected, but the exact location of the exception is not determined.
enable	Enables the specified exceptions in the prologue of the main program. This suboption is required if you want to turn on exception trapping without modifying the source code.

If your program contains signaling NaNs, you should use the "float Code Generation Option (AIX)" on page 69 option along with the gen floatTrap options to trap any exceptions.

You can also specify floating-point exception trapping options from your C and C++ source code by using #pragma options flttrap=*option*. The pragma overrides the configuration file option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

funcSect Code Generation Option (C, AIX)

Syntax:	**Default:**

gen(funcSect[, yes])
gen(funcSect, no)
	gen(funcSect, no)

Use this option to place instructions for each function in a separate object file control section, or csect.

By default each object file will consist of a single control section combining all functions defined in the corresponding source file.

Using multiple csects increases the size of the object file, but often reduces the size of the final executable by allowing the linkage editor to remove functions that are not called or that have been inlined by the optimizer at all places they are called. If the file contains initialized static data or `#pragma comment copyright`, some functions will be one machine word larger.

The #pragma options directive must be specified before the first statement in the compilation unit.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

initAuto Code Generation Option

Syntax:	Default:

gen(initAuto[, yes])
gen(initAuto, *value*)
gen(initAuto, no)
	gen(initAuto, no)

This option controls whether automatic storage is initialized, and optionally specifies the initial value.

AIX

Automatic storage allocated using alloca is currently not initialized by gen(initAuto).

By default automatic storage is not initialized.

Specifying gen(initAuto) or gen(initAuto,yes) is equivalent to specifying gen(initAuto, 170). This initializes each byte of automatic storage to 0xAA, which is an easily recognized non-zero value.

You can specify any value in the range 0 to 255. The value specified is placed in each byte of an automatic variable. For example, when gen(initAuto,170) is specified, an automatic int variable would be set to 0xAAAAAAAA, rather than 0xAA or 0x000000AA.

You can use this option as an aid to debugging when it is helpful to know initial values. It is also helpful for detecting uninitialized local variables. However, because setting these values generates extra code, you should initialize automatic storage only for problem-solving.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

inlinePointerGlue Code Generation Option (AIX)

Syntax:	Default:

gen(inlinePointerGlue[, yes])
gen(inlinePointerGlue, no)
	gen(inlinePointerGlue, no)

This option controls whether the pointer glue code necessary to make a call to an external function or call through a function pointer is inlined. External linkage is faster when the pointer glue code is inlined.

By default, inlined pointer glue code is not generated.

Glue code is used for passing control between two external functions, or when you call functions through a pointer. Therefore, the inlinePointerGlue option only affects functions called through pointers or calls to an external compilation unit. For calls to an external function, you should specify that the function is imported. Using the procImported link option is one way to do this.

The inlining of glue code can cause the size of code to grow. The “size Optimization Option” on page 104 optimization option overrides the gen(inlinePointerGlue) option, and disables the inlining of pointer glue code.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Code Generation Options (AIX)” on page 59

libANSI Code Generation Option

Syntax:	**Default:**
gen(libANSI[, yes])	gen(libANSI, no)
gen(libANSI, no)	

This option controls whether functions with same name as a standard C library function are system functions or user-defined functions.

By default, VisualAge C++ does not assume that functions with standard C library names are system functions.

C

For this option to take effect, you must include the headers for the VisualAge C++ run-time library in C sources that call standard C library functions.

Setting libANSI to yes may help the optimizer generate code that performs better because the behavior of a system function, such as whether it causes side effects, is known.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

lineOnlyDebug Code Generation Option (C)

Syntax:	**Default:**
gen(lineOnlyDebug[, yes])	gen(lineOnlyDebug, no)
gen(lineOnlyDebug, no)	

This option determines whether the C compiler generates line number information for limited debugging, rather than full debugging information.

By default, the C compiler generates no debug information.

To produce an executable, DLL or shared library with line number information, you also need to set the "debug Link Option" on page 115 option for the target.

You can use this option when you want only to single-step through the source view of the files, and not to view variables. For full debugging, use "debug Code Generation Option (C)" on page 65 to include symbol, type and other required information in the generated code.

C++

Code generated from C++ source is always fully enabled for debugging.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Debugging with VisualAge C++

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

longDouble Code Generation Option (AIX)

Syntax:	Default:

gen(longDouble[, yes])
gen(longDouble, no)
	AIX gen(longDouble, no)

This option controls whether long doubles are represented as 128-bit values or 64-bit values.

By default, gen(longDouble, no) is set, and long doubles are represented by 64-bit values.

Separate libraries are provided that support 128-bit long double types. These libraries will be automatically linked if you specify the defaults(*string*) option using any *string* with the **128** suffix, such as xlC128, xlc128, or cc128. You can also explicitly link to the 128-bit versions of the libraries by specifying them as sources in your configuration file. The 128-bit versions of the libraries that correspond to the 64-bit libraries are shown in the following table:

Default (64-bit) long double library	128-bit long double library
libC.a	libC128.a
libCns.a	libC128ns.a
libC_r.a	libC128_r.a
libCns_r.a	libC128ns_r.a
libc.a	libc128.a
libc_r.a	libc128_r.a

Linking without the 128-bit versions of the libraries when your program uses 128-bit long doubles, for example, if you specify gen(longDouble) alone, may produce unpredictable results.

The gen(longDouble) option defines __LONGDOUBLE128.

This is a global codestore option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options for the Project
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

objectmode Code Generation Option (AIX)

Syntax:	**Default:**

gen(objectmode, 32)
gen(objectmode, 64)
	AIX gen(objectmode, 32)

This setting determines the type of object code generated, either 32-bit or 64-bit. Note that the default setting is overridden by any environment variable setting.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options for the Project
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Code Generation Options (AIX)” on page 59

objectModel Code Generation Option (C++)

Syntax:	**Default:**

gen(objectModel, compat)
gen(objectModel, model)
	gen(objectModel, compat)

This option determines which predefined object model is used for code generation.

The following object models can be specified:

- ibm
- compat

By default, code is generated for the compat object model, which means that all C++ objects conform to the compat layout of the virtual function table, virtual base class support, and name mangling scheme. All shipped libraries are generated with compat. In addition, all library header files should have #pragma object_model(compat) and #pragma object_model(pop) wrapping them to make sure the correct object model is used at compile time.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Object Models

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Code Generation Options (AIX)" on page 59

profile Code Generation Option

Syntax:	Default:
gen(profile[, yes])	gen(profile, no)
gen(profile, no)	
gen(profile, ibm)	
AIX gen(profile, bsd)	

This option controls whether the code is enabled for profiling or performance analysis.

By default, gen(profile) is set to no.

You must set gen(profile) to yes to run the Performance Analyzer.

OS/2

WIN To enable code for performance analysis on OS/2 and Windows NT platforms, "debug Link Option" on page 115 must be set in addition to gen(profile). Setting gen(profile) causes link(debug, yes) to be set automatically. You may also need to explicitly link with the object module that supports trace information in your executable target. The performance analyzer object cpppan40.obj is linked automatically when you select gen(profile). Note that gen(profile, ibm) is equivilent to gen(profile, yes).

C

OS/2 WIN To use the Performance Analyzer for a C program on OS/2 and Windows NT platforms, you must set "debug Code Generation Option (C)" on page 65.

OS/2

You must specify link(extDictionary, no) with gen(profile).

AIX

On AIX, specifying gen(profile, yes) generates profiling support code for use with the **prof** profiler. Specifying gen(profile, bsd) generates profiling support code for BSD profiling using **gprof**. Also on AIX, gen(profile, ibm) inserts profiling hooks for IBM's Performance Analyzer. The link(debug,yes) option must be set and you must also link to libpahooks.a or libpahooks_r.a (multi-threaded version).

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Build With Existing Object Files and Libraries
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

readOnly Code Generation Option

Syntax:	**Default:**

gen(readOnly[, yes])
gen(readOnly, no)
	gen(readOnly[, yes])

This option controls whether string literals are placed in read-only or read-write storage.

By default, string literals are placed in read-only storage.

If readOnly is set to no, string literals are placed in the default data section.

OS/2

WIN You can specify names of the the default constant segment and the default data segment with the gen(constSeg, *name*) and gen(dataSeg, *name*) options, respectively.

You can also control how string literals are stored by using #pragma strings(readonly) or #pragma strings(writeable) in your source. The pragma overrides the configuration file option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

readOnlyConst Code Generation Option (AIX)

Syntax:	**Default:**

gen(readOnlyConst[, yes])
gen(readOnlyConst, no)
	gen(readOnlyConst, no)

This option controls whether const variables are placed in read-only or read-write storage.

If readOnlyConst is set to no, const variables are placed in the default data section, which is in read-write storage.

If readOnlyConst is set to yes, const variables are placed in read-only storage if possible.

Placing const variables in read-only memory can improve run-time performance, save storage, and provide shared access. Code that attempts to modify a read-only const variable generates a memory error.

This option does not apply to the following:
- variables qualified with the volatile attribute and aggregates (such as a struct or a union) that contain volatile variables
- aggregates that contain mutable data members
- pointers and aggregates that contain pointers
- automatic variables
- static variables with block scope
- uninitialized variables
- non-const aggregates with all data members qualified by const
- variables initialized with addresses, or addresses cast to non-address values
- aggregates initialized with incomplete initializers
- variables of a class or struct which includes a virtual function

The gen(readOnlyConst) option does not imply the "readOnly Code Generation Option" on page 82 option. Both options must be used if you wish to specify the storage characteristics of both string literals (gen(readOnly)) and const variables (gen(readOnlyConst)).

You can also control how const variables are stored by using #pragma options [no]roconst. The pragma overrides the configuration file option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

roundConstFp Code Generation Option (AIX)

Syntax:	**Default:**

gen(roundConstFp, nearest)
gen(roundConstFp, minusinf)
gen(roundConstFp, plusinf)
gen(roundConstFp, zero)
	gen(roundConstFp, nearest)

This option controls the compile-time rounding mode of constant floating point expressions. It does not affect run-time rounding.

By default, constant floating-point expressions are rounded toward the nearest representable number at compile time.

The following table describes the effect of specifying gen(roundConstFp, *mode*) for each of the roundConstFp suboptions.

Mode	**Effect**
nearest	Round to nearest representable number. This is the default.
minusinf	Round toward minus infinity.
plusinf	Round toward plus infinity.
zero	Round toward zero.

Compile-time floating-point arithmetic can have two effects on program results:

- In specific cases, the result of a computation at compile time might differ slightly from the result that would have been calculated at run time. The reason is that more rounding operations occur at compile time. For example, where a multiply-add floating point operation might be used at run time, separate multiply and add operations might be used at compile time, producing a slightly different result.
- Computations that produce exceptions can be folded to the IEEE result that would have been produced by default in a run-time operation. This would prevent an exception from occuring at run time. The gen(floatTrap) option can be used to generate instructions that detect and trap floating-point exceptions.

In general, code that affects the rounding mode at run time should be compiled with the roundConstFp option that matches that rounding mode. For example, when the following program is compiled, the expression `1.0/3.0` is folded at compile time into a double-precision result:

```
main()
{
float x, y;
int i;
x = 1.0/3.0;
i = *(int *)&x;
printf("1/3 = %.8x\n", i);
x = 1.0;
y = 3.0;
x = x/y;
i = *(int *)&x;
printf("1/3 = %.8x\n", i);
}
```

This result is then converted to single precision and stored in `float x`.

The gen(float, fold, no) option can be specified to suppress all compile-time folding of floating-point computations. For example, the following code fragment may be evaluated either at compile time or at run time, depending on the setting of opt(level), gen(float, fold), and other options:

```
x = 1.0;
y = 3.0;
x = x/y;
```

The gen(roundConstFp) option only affects compile-time rounding of floating-point computations; if this code is evaluated at run time, the default run-time rounding of "round to nearest" is still in effect and takes precedence over the compile-time rounding mode specified by gen(roundConstFp).

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

rtti Code Generation Option (C++)

Syntax:	**Default:**
gen(rtti[, yes])	gen(rtti, no)
gen(rtti, all)	
gen(rtti, typeInfo)	
gen(rtti, dynamicCast)	
gen(rtti, no)	

This option controls what run-time type information (RTTI) is generated.

By default, RTTI is not supported by the generated code. The default setting has a performance advantage. However, some libraries, like the IBM Open Class™ Library, require RTTI.

The options gen(rtti, all) and gen(rtti) or gen(rtti,yes) have the same effect. They generate code that supports both the typeid and dynamic_cast operators.

To request that only the typeid operator is supported, use gen(rtti, typeInfo).

To request that only the dynamic_cast operator is supported, use gen(rtti, dynamicCast).

This is a global codestore option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options for the Project
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Code Generation Options (AIX)" on page 59

strictinduction Code Generation Option (AIX)

Syntax:	**Default:**
gen(strictinduction[, yes]) gen(strictinduction, no)	gen(strictinduction, no)

This option sets the loop induction variable optimizations on or off. These optimizations have the potential to alter the semantics of the program.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Code Generation Options (AIX)" on page 59

vft Code Generation Option (C++)

Syntax:	**Default:**
gen(vft[, yes]) gen(vft, no)	gen(vft[, yes])

This option controls whether the virtual function table is included in the module that is the target of the build. It has no effect if the target is an executable file.

By default, virtual function tables are forced in the target module. If you build with gen(vft, no) the virtual function tables must be included in other modules when your application is linked.

▶ OS/2

WIN On OS/2 and Windows NT platforms, you can specify whether virtual function tables are treated as read-only or read-write storage by building with gen(vftReadOnly).

OS/2

WIN On OS/2 and Windows NT platforms, you can specify the name for the section in which virtual function tables are stored by building with gen(vftSeg, *name*).

AIX

On AIX you can also control whether virtual function tables are generated by using #pragma options [no]vftable. The pragma overrides the configuration file option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Code Generation Options (AIX)” on page 59

Chapter 6. Optimization Options

Optimization Options (AIX)

The options listed in the following table control how code is optimized for AIX platforms during builds.

- If an option does not apply to both C and C++, the middle column shows which type of source that it affects.
- If you specify options that are not recognized on AIX, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case sensitive. If you omit both “yes” and “no” from an option that has a yes or no form, yes is implied regardless of whether the default setting for that option is yes or no. For example specifying opt(IgnErrno) is equivalent to specifying opt(ignerrno, yes).

Option	**Applies only to**	**Default**

opt(alias, *rule*[, yes])
opt(alias, *rule*, no)
		opt(alias, *rule*, no)

opt(aliasAssert,typ[, yes])
opt(aliasAssert,typ, no)
opt(aliasAssert,allp[, yes]
opt(aliasAssert,allp, no)
opt(aliasAssert,addr[, yes]
opt(aliasAssert,addr, no)
	C	opt(aliasAssert,typ, no)
		opt(aliasAssert,allp, no)
		opt(aliasAssert,addr, no)

opt(autoInline[, yes])
opt(autoInline, no)
		opt(autoInline, no)

opt(autoInlineLimit, *number*)		opt(autoInlineLimit, 20)
opt(basePriority, *priority*)	C++	opt(basePriority,0)

opt(ignErrno[, yes])
opt(ignErrno, no)
		opt(ignErrno, no)

opt(inline[, yes])
opt(inline, no)
		opt(inline[, no]) with opt(level, 0)
		opt(inline[, yes]) otherwise

opt(inlineFunc, *name*)	C	none
opt(isolatedCall, *function*)		none
opt(level, *level*)		opt(level, 0)

opt(loopUnroll[, yes])
opt(loopUnroll, no)
opt(loopUnroll, *number*)
		opt(loopUnroll, no)

opt(maxmem, *number*)		opt(maxmem, 2048)
opt(noInlineFunc, *name*)	C	none

opt(pragmaDisjoint[, yes])
opt(pragmaDisjoint, no)
		opt(pragmaDisjoint[, yes])

opt(pragmaIsolatedCall[, yes])
opt(pragmaIsolatedCall, no)
		opt(pragmaIsolatedCall[, yes])

Option	Applies only to	Default

gen(profileDirectedFeedback, collect)
gen(profileDirectedFeedback, apply)
gen(profileDirectedFeedback, no)
		gen(profileDirectedFeedback, no)

opt(registerSpillSize, *number*)		opt(registerSpillSize, 512)

opt(size[, yes])
opt(size, no)
		opt(size, no)

opt(strict[, yes])
opt(strict, no)
		opt(strict, no)

opt(tune, *CPUtype*)		gen(tune, pwr) *If gen(arch,CPUtype) is not specified; it can vary according to the arch option setting.*

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

Optimization Levels (AIX)

Syntax:	Default:
opt(level, *level*)	opt(level, 0)

Whether the code generated during a build is optimized, and what level of optimization is applied is set by the build option opt(level, *level*). The default level is 0, or no optimization.

You can specify level 0, 2, 3, or 4. These levels are described below. The optimizations may change from one product release to the next. If you need a specific level of optimization, specify the appropriate numeric value.

Level 0
This is the default level. It performs only quick local optimizations, such as constant folding and elimination of local common subexpressions.

Level 2
Performs optimizations that the compiler developers consider to be the best combination for compilation speed and runtime performance.

Setting an opt level 2 or greater enables or implies the following options:

- implies “inline Optimization Option” on page 97
- enables “size Optimization Option” on page 104

Level 3
Performs additional optimizations that are memory intensive, compile-time intensive, or both. These optimizations are performed in addition to those performed with opt(level, 2). They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

This level is the compiler’s highest and most aggressive level of optimization. By default opt(level, 3) performs optimizations that have the potential to slightly alter the semantics of your program. It also changes the default maxmem to -1, meaning optimizations may use unbounded memory and may also use more time.

You can use the strict optimizing option with opt(level, 3) to turn off the aggressive optimizations that might change the semantics of a program. The strict option, combined with opt(level, 3), invokes all the optimizations performed at opt(level, 2), as well as further loop optimizations. Note that the strict option must appear after the opt(level, 3) option, otherwise it is ignored.

The aggressive optimizations performed when you specify opt(level, 3) are:

1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

   Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

   For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at opt(level, 2) because the computation may cause an exception. At opt(level, 3), the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at opt(level, 3). Loads in general are not considered to be absolutely safe at opt(level, 2) because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation.

   The same concepts apply to scheduling.

   Example: In the following example, at opt(level, 2), the computation of `b+c` is not moved out of the loop for two reasons:

   a. it is considered dangerous because it is a floating-point operation that could cause a trap
   b. it does not occur on every path through the loop

   At opt(level, 3), the code is moved.

   ```
     ...
   int i ;
   float a[100], b, c ;
   for (i = 0 ; i < 100 ; i++)
    {
    if (a[i] < a[i+1])
     a[i] = b + c ;
    }
     ...
   ```

2. Conformance to IEEE rules are relaxed.

With opt(strict,yes) (implied by opt(level,2)), certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.

For example, (X + 0.0) is not folded to X because, under IEEE rules, (-0.0 + 0.0) = 0.0, which is -X. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, if X is 0.0, replacing (X-Y*Z) with (-Y*Z) may result in a -0.0 where the original computation would result in 0.0.

In most cases the difference in the results is not important to an application and opt(level, 3) allows these optimizations.

3. Floating-point expressions may be rewritten.

   Computations such as (a*b*c) may be rewritten as (a*c*b) if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

Setting an opt level 3 or greater enables or implies the following options:

- "float Code Generation Option (AIX)" on page 69
- "ignErrno Optimization Option" on page 96
- "maxmem Optimization Option (AIX)" on page 100

The following notes also apply when setting opt(level, 3):

- Some built-in functions (e.g., sqrt) do not change errno.
- Aggressive optimizations do **not** include the following floating-point suboptions, or anything else that affects the precision mode of a program:
  - gen(float, hsflt)
  - gen(float, hssngl)
  - gen(float, rndsngl)
- If opt(level, 0) is specified after opt(level, 3) has been specified, opt(ignErrno, yes) remains set.

C Level 4
This option is the same as opt(level, 3), except that it also sets the arch and tune optimizing options to the architecture of the compiling machine.

Subsequent settings of level, arch, and tune options override the settings implied by the opt(level, 4) option.

C Level 5
This option is the same as opt(level, 3), except that it also sets the arch and tune optimizing options to the architecture of the compiling machine. Also, interprocedural calls are optimized at this level.

Subsequent settings of level, arch, and tune options override the settings implied by the opt(level, 5) option.

RELATED CONCEPTS

Overview of Optimization

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Optimization Options (AIX)” on page 89

aliasAssert Optimization Option (C, AIX)

Syntax:	**Default:**
opt(aliasAssert,typ[, yes])	opt(aliasAssert,typ, no)
opt(aliasAssert,typ, no)	opt(aliasAssert,allp, no)
opt(aliasAssert,allp[, yes]	opt(aliasAssert,addr, no)
opt(aliasAssert,allp, no)	
opt(aliasAssert,addr[, yes]	
opt(aliasAssert,addr, no)	

This option specifies the aliasing assertion to apply to your compilation unit. The available options are:

typ Ensures that pointers to different types are never aliased. In the compilation unit, no two pointers of different types will point to the same storage location.

allp Ensures that pointers are never aliased. In the compilation unit, no two pointers will point to the same storage location. This option implies `opt(aliasAssert,typ)`.

addr Ensures that variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has not been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
AIX “Optimization Options (AIX)” on page 89

autoInline Optimization Option

Syntax:	**Default:**
opt(autoInline[, yes])	opt(autoInline, no)
opt(autoInline, no)	

This option determines whether functions in your source code are candidates for inlining.

C++

It does not apply to functions qualified with inline in C++ source code, which are enabled for inlining by building with the “inline Optimization Option” on page 97 option.

C For C source code, setting opt(autoInline) implicitly sets opt(inline), so this option applies to functions qualified with _Inline in C source code.

By default, VisualAge C++ does no inlining.. This can reduce program size, usually at the expense of performance. You must explicitly set autoinlining on to allow the compiler to use heuristics to determine possible functions for inlining.

You can also specify a threshold for automatic inlining of functions with the following options:

AIX

opt(autoInlineLimit, *number*) on AIX platforms

OS/2 WIN opt(autoInlineThreshold, *number*) on OS/2 and Windows NT platforms

Even when inlining is turned on, some candidate functions may not actually be inlined due to internal criteria applied during the compile stage.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files
Inlining

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

autoInlineLimit Optimization Option (AIX)

Syntax:	**Default:**
opt(autoInlineLimit, *number*)	opt(autoInlineLimit, 20)

This option allows you to control the amount of automatic inlining that occurs when opt(autoInline) is enabled. The source statements are inlined if the number of statements is less than *number*.

This option, with the default limit of 20, only takes effect when “autoInline Optimization Option” on page 93 is set.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build
Debug Optimized Code

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

basePriority Optimization Option (C++)

Syntax:	**Default:**
opt(basePriority, *priority*)	opt(basePriority, 0)

This option determines the order in which static objects are initialized at run time. Use it to set a priority for initializations in individual C++ source files, relative to other sources.

By default, the base priority is set to 0 for all source files. You can specify any integer literal in the range INT_MIN+1024 to INT_MAX. The constants INT_MIN and INT_MAX are defined in <limits.h>. Static objects in sources with negative priorities or lower numeric values are initialized before objects in sources with positive or higher numeric values.

Static objects are initialized in the order specified by the C++ standard, regardless of whether you use this option. This option does not affect the order of initializations within a source file. Static objects are initialized as follows:

1. Explicit initializations are done first, at load time.
2. Static initializations are done in reference order. For example, if the constructor for A references B, B is initialized before A.
3. Within a source file, objects are initialized in the order in which they are defined.

VisualAge C++ effectively compiles all your sources at once and this option can alter the order of initialization of static objects declared in different source files. For example, if you want to make sure the constructor for a static object does not run until other static objects have been initialized, you can give the source file that declares the object a high base priority value.

Applying this option to a primary source file has the same effect as inserting a #pragma priority preprocessor directive before the first line in that source file. You can include #pragma priority directives in your sources in addition to using this option, provided you do not specify a lower priority value with the pragma than is set with this option.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
Configuration Files
Inlining

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

ignErrno Optimization Option

Syntax:
opt(ignErrno[, yes])
opt(ignErrno, no)

Default:

OS/2 WIN opt(ignErrno,no)

AIX At optimization level 0 to 2, opt(ignErrno, no) and at optimization level 3 or 4, opt(ignErrno, yes)

This option controls whether the compiler can assume your code makes no reference to errno. This assumption creates more opportunities for optimization.

OS/2

WIN By default, the compiler performs no optimizations that might affect use of errno.

AIX

The default setting depends on the optimization level you have chosen.

RELATED CONCEPTS

Overview of Optimization
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

inline Optimization Option

Syntax:	**Default:**
opt(inline[, yes])	opt(inline, no) with opt(level,0)
opt(inline, no)	opt(inline[, yes]) otherwise

This option determines whether the following functions in your source code are candidates for inlining.

C Functions qualified with _Inline in C source code

C++ Functions qualified with inline in C++ source code.

To enable inlining for other functions in your source files, use the “autoInline Optimization Option” on page 93 option.

C For C source code, setting opt(autoInline) implicitly sets opt(inline).

By default, VisualAge C++ inlines functions when optimizing, and does no inlining otherwise. When you build with an optimization level greater than 0, set opt(inline, no) to suppress inlining. This can reduce program size, usually at the expense of performance.

This option is set to yes implicitly when you build with opt(level,*level*) and *level* has a value greater than 0.

Even when inlining is turned on, some candidate functions may not actually be inlined due to criteria applied during the compile stage.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files
Inlining

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

inlineFunc Optimization Option (C, AIX)

Syntax:	**Default:**
opt(inlineFunc, *name*)	none

The compiler attempts to inline the functions identified in *name* and any other appropriate functions. To specify a list of functions to be inlined, use multiple opt(inlineFunc) options to specify one function at a time. For example, the compiler attempts to inline `func1` and `func2` specified by:

```
opt(inlineFunc, "func1")
opt(inlineFunc, "func2")
```

Because inlining does not always improve run-time performance, you should test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

isolatedCall Optimization Option

Syntax:	**Default:**
opt(isolatedCall, *function*)	none

This option states that the named function has no side effects and does not depend on any side effects. Since the entities in the calling modules are not affected by the function call and the function call depends only on its parameters, there are more opportunities for optimization.

By default, VisualAge C++ makes no assumptions about side effects.

Use this option to name functions that do not have side effects and do not rely on functions or processes that have side effects. Side effects are any changes in the state of the run-time environment, such as accessing a volatile object, modifying a file, or calling another function that does so. Functions with no side effects cause no changes to external and static variables.

You can supply this option many times, each time adding a function to the list of those that do not have side effects.

You can also specify that a function has no side effects by using #pragma isolated_call in your source, provided that "pragmaIsolatedCall Optimization Option" on page 102 is in effect.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build
Optimize your Application

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

loopUnroll Optimization Option (AIX)

Syntax:	**Default:**
opt(loopUnroll[, yes])	opt(loopUnroll, no)
opt(loopUnroll, *number*)	
opt(loopUnroll, no)	

This option controls whether inner loops in the program are unrolled.

By default, inner loops in the code are not unrolled.

When opt(loopUnroll, *number*) is specified, inner loops are unrolled by a factor of *number*. The bodies of inner loops are duplicated *number*-1 times, creating a loop with *number* original bodies. The loop control may be modified in some cases to avoid unnecessary branching. The maximum value for *number* is 8.

VisualAge C++ may perform automatic unrolling of inner loops by a factor of up to 16 at an optimization level of 2 or higher (for example, when you specify opt(level, 3)). This optimization will be disabled, however, if you specify opt(loopUnroll, no). The opt(size) option also suppresses loop unrolling.

Specifying opt(loopUnroll, yes) is equivalent to specifying opt(loopUnroll, 4).

C++

You can also control loop unrolling by using #pragma options [no]unroll and unroll=*number*. The pragma overrides the configuration file option.

Example

In the following example, loop control is not modified:

```
while (*s != 0)
{
*p++ = *s++;
}
```

Unrolling this by a factor of 2 results in code equivalent to the following code:

```
while (*s)
{
*p++ = *s++;
if (*s == 0) break;
*p++ = *s++;
}
```

In the following example, loop control *is* modified:

```
for (i=0; i<n; i++) {
a[i] = b[i] * c[i];
}
```

Unrolling by a factor of 3 results in code equivalent to the following code:

```
if (n>=0) {
       if (n<2) {
           a[0] = b[0] * c[0];
           i = 1;
           if (n>0) {
               a[1] = b[1] * c[1];
               i = 2;
           }
       }
       for (; i<n-2; i+=3) {
           a[i] = b[i] * c[i];
           a[i+1] = b[i+1] * c[i+1];
           a[i+2] = b[i+2] * c[i+2];
       }
}
```

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

maxmem Optimization Option (AIX)

Syntax:	**Default:**
opt(maxmem, *number*)	opt(maxmem, 8192)

This option limits the amount of memory used for local tables of specific, memory-intensive optimizations to *number* kilobytes. If that memory is insufficient for a particular optimization, the scope of the optimization is reduced.

The default is opt(maxmem, 8192). This default also applies when the *number* specified is 0.

The following points apply to the use of the maxmem optimization option:

- A *number* value of -1 permits each optimization to take as much memory as it needs without checking for limits.
- Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, the memory used when performing optimizations might exceed available system resources. This is more likely when the *number* specified is -1.
- The limit set by opt(maxmem) is the amount of memory for specific optimizations, and not for VisualAge C++ as a whole. Tables required during the entire compilation process are not affected by or included in this limit.
- Setting a large limit has no negative effect on the compilation of source files when VisualAge C++ needs less memory.
- Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that VisualAge C++ may finish before finding all opportunities to increase performance. If this occurs, a warning will be given.
- Increasing the limit does not necessarily mean that the resulting program will be faster, only that VisualAge C++ is better able to find opportunities to increase performance if they exist. If no warning is given with a specific limit, all opportunities were found.

You can also limit the amount of memory used for optimization by using #pragma options [no]maxmem=*number*. The pragma overrides the configuration file option.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

noInlineFunc Optimization Option (C, AIX)

Syntax:	**Default:**
opt(noInlineFunc, *name*)	none

The compiler does not inline functions listed by names in *name.* All other appropriate functions are inlined.

To specify a list of functions that should not be inlined, use multiple opt(noInlineFunc) options to specify one function at a time. For example, the compiler does not attempt to inline `func1` and `func2` specified by:

```
opt(noInlineFunc, "func1")
opt(noInlineFunc, "func2")
```

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

pragmaDisjoint Optimization Option

Syntax:	**Default:**

opt(pragmaDisjoint[, yes])
opt(pragmaDisjoint, no)
	opt(pragmaDisjoint[, yes])

This option determines whether references to #pragma disjoint are honored or ignored.

By default, VisualAge C++ assumes, for optimization purposes, that identifiers listed in #pragma disjoint are not aliased to each other in the scopes in which they are used.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

pragmaIsolatedCall Optimization Option

Syntax:	**Default:**

opt(pragmaIsolatedCall[, yes])
opt(pragmaIsolatedCall, no)
	opt(pragmaIsolatedCall[, yes])

This option determines whether references to #pragma isolated_call are honored or ignored.

By default, VisualAge C++ assumes, for optimization purposes, that functions specified by #pragma isolated_call do not have side effects and do not rely on functions or processes that have side effects. Side effects are any changes in the state of the runtime environment, such as accessing a volatile object, modifying a file, or calling another function that does such things. Functions with no side effects cause no changes to external and static variables.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Optimization Options (AIX)” on page 89

profileDirectedFeedback Optimization Option

Syntax:	**Default:**
gen(profileDirectedFeedback, collect)	gen(profileDirectedFeedback, no)
gen(profileDirectedFeedback, apply)	
gen(profileDirectedFeedback, no)	

Perform aggressive optimizations with profile directed feedback. option is one of the following:

- collect
- apply
- no

opt(profileDirectedFeedback,collect) - the compiler will add instructions into the object code to collect statistices for later use.

opt(profileDirectedFeedback,apply) - the compiler should read in the statistices and use them to guide the generation of better code.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Build With Existing Object Files and Libraries
Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Code Generation Options (AIX)" on page 59

registerSpillSize Optimization Option (AIX)

Syntax:	**Default:**
opt(registerSpillSize, *number*)	opt(registerSpillSize, 512)

This option specifies the register allocation spill area as being *number* bytes.

The default is opt(registerSpillSize, 512). This default also applies when the *number* specified is 0.

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area.

You can also set the size of the register allocation spill area by using #pragma options spill=*number*. The pragma overrides the configuration file option.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

size Optimization Option

Syntax:	**Default:**

opt(size[, yes])
opt(size, no)
	opt(size, no)

This option determines whether code is optimized for size in preference to speed. Use this option to reduce the amount of space occupied by your executable code. This option is enabled when opt(level,*level*) is set to a level greater than 0.

By default, no optimization is performed.

Specifying opt(size, yes) causes VisualAge C++ to select size optimizations over speed optimizations whenever they conflict.

Programs optimized for size usually run faster than if they had not been optimized, but they may not be as fast as when built with opt(size, no). However, optimizing for size can reduce the overhead of paging in larger applications, and a program may run faster when it is optimized for size than when it is optimized for speed alone. Optimizing for size can also sometimes improve cache locality.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

tune Optimization Option

Syntax:
opt(tune, *CPUtype*)

Default:

AIX opt(tune, pwr) *If gen(arch,CPUtype) is not specified; it can vary according to the arch option setting.*

OS/2 WIN opt(tune, blend)

This option specifies the CPU architecture for which the code is optimized. You can specify a string representing one of the processor types for your platform as the value of *CPUtype*:

AIX

Processor Types (AIX)

AIX

On AIX platforms, you can use this option together with the gen(arch, *CPUtype*) option. The gen(arch, *CPUtype*) option establishes what instruction set is used to generate code. The opt(tune, *CPUtype*) option determines the processor for which the code is optimized. The default and allowed settings of the tune option depend on the processor type set with gen(arch, *CPUtype*), as described in the AIX processor types reference, and the following points apply to the relationship between these two options:

- If both arch and tune options are not explicitly set, arch defaults to common and tune defaults to pwr.
- If tune is specified without arch, the compiler uses gen(arch, common) along with the specified tune setting.

- If arch is specified without tune, the compiler uses the default tune setting for the specified architecture.

  C When compiling C programs, listings will show only the following setting:

  ```
  TUNE=DEFAULT
  ```
- If you attempt to use an invalid gen(arch) and opt(tune) combination, a warning message will be issued. The gen(arch) setting is accepted, and the default opt(tune) setting corresponding to that gen(arch) setting is assumed.

OS/2

WIN The generated code can run on any x86 processor, but is specifically tuned for the target processor. The compiler also performs optimizations that help other processors if those optimizations do not in turn detract from the performance of the target processor. For example, code optimized for a 486 processor also includes any Pentium® optimizations that do not diminish 486 execution.

OS/2

WIN If you do not know what processor your application will be run on, specify opt(tune, blend). The compiler includes optimizations that help all x86 processors, but specifically improve execution on Pentium, PentiumII and PentiumPro processors.

RELATED CONCEPTS

Overview of Optimization
Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

AIX Optimize Your Application
Debug Optimized Code

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Optimization Options (AIX)" on page 89

AIX Processor Types (AIX)

Chapter 7. Link Options

Link Options (AIX)

The options listed in the following table control the link stage of builds on AIX.

- If an option does not apply to both C and C++, the middle column shows which type of source that it affects.
- If you specify options that are not recognized on AIX, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case sensitive. If you omit both "yes" and "no" from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying link(DebugFullPath) is equivalent to specifying link(debugfullpath, yes).

Option	**Applies Only to**	**Default**

link(autoExport[, yes])
link(autoExport, no)
		link(autoExport[, yes])

link(bigTOC, [yes])
link(bigTOC, no)
		link(bigTOC, no)

link(dataStart, *address*)		link(dataStart, 0x20000000)

link(dataImported, all)
link(dataImported, none)
link(dataImported, names)
		link(dataImported, all)

link(dataImportedNames, *string*)		none

link(dataLocal, all)
link(dataLocal, none)
link(dataLocal, names)
		link(dataLocal, none)

link(dataLocalNames, *string*)		none

link(debug[, yes])
link(debug, no)
		link(debug, no)

link(debugFullPath[, yes])
link(debugFullPath, no)
		link(debugFullPath, no)

link(entry[, yes])
link(entry, *string*)
link(entry, no)
		link(entry, yes)

link(export, *external_name* , [*internal_name*])		none

link(exportAll[, yes])
link(exportAll, no)
	C++	link(exportAll, no)

link(export_NameSpace, *string*)	C++	link(export_NameSpace, null)
link(export_Prefix, *string*)	C++	link(export_Prefix, null)
link(exportList, *fileName*)		none

Option	Applies Only to	Default
link(exportSymbolVisibility, *option*)		link(exportSymbolVisibility, “nosymbolic-”) For BSS symbols: link(exportSymbolVisibility, nosymbolic)

link(force[, yes])
link(force, no)
		link(force, no)

link(generatedFilesDirectory, *pathName*)		link(generatedFilesDirectory, *current target directory*)
link(heap, *size*)		link(heap, 0)
link(import, [*internal_name*], *library_name*, *export_name*)		none
link(initializationTermination, [*initf*][, [*finif*][, *priority*]]))		priority defaults to 0

link(libPathOut, *string*)
link(libPathOut, no)
		link(libPathOut, no)

link(libSearchPath, *path*)		link(libSearchPath, “/usr/vacpp/lib”)
link(libSearchPathPrefix, *string*)		none

link(linkWithMultiThreadLib[, yes])
link(linkWithMultiThreadLib, no)
		link(linkWithMultiThreadLib, no)

link(linkWithSharedLib[, yes]) link(linkWithSharedLib, no		link(linkWithSharedLib, no)

link(map[, yes])
link(map, *fileName*)
link(map, no)
		link(map, no)

link(nameMangling, ansi, *characters*)
link(nameMangling, compat, *characters*)
	▶ C++	link(nameMangling, ansi)

link(padding[, yes])
link(padding, no)
		link(padding[, yes])

link(runtimeLinking[, yes])
link(runtimeLinking, no)
		link(runtimeLinking, no)

link(runtimeLinkerStructure[, yes])
link(runtimeLinkerStructure, no)
		link(runtimeLinkerStructure, no)

link(runtimeLinkPath[, yes])
link(runtimeLinkPath, no)
		link(runtimeLinkPath, yes)

link(sharedLibPriority, *number*)	▶ C++	none

link(staticSymbols[, yes])
link(staticSymbols, no)
		link(staticSymbols, no)

link(stack, *size*)		link(stack, 0)

Option	Applies Only to	Default

link(static[, yes])
link(static, no)
		link(static, no)

link(strip[, yes])
link(strip, no)
		link(strip, no)

link(system[, yes])
link(system, no)
		link(system, no)

link(systemOption, *string*)		none
link(textStart, *address*)		link(textstart, 0x10000000)

link(traceBackTable, none)
link(traceBackTable, small)
link(traceBackTable, full)
		link(traceBackTable, small) when link(debug, no)
		link(traceBackTable, full) when link(debug)

You can set these options with the IDE or by using the option directive in your configuration file.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

autoExport Link Option (AIX)

Syntax:	Default:

link(autoExport[, yes])
link(autoExport, no)
	link(autoExport[,yes])

This option automatically exports any symbol that is referenced by a shared module from module '.' (DOT). If "runtimeLinking Link Option (AIX)" on page 134 is on, symbols referenced by shared modules that are imported from '..' (DOT-DOT), are also exported, as well as BSS symbols.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

bigTOC Link Option (AIX)

Syntax:	Default:

link(bigTOC, yes)
link(bigTOC, no)
	link(bigTOC, no)

By default, an error message is generated if the TOC (Table Of Contents) is greater than 64K.

If you specify link(bigTOC, yes), extra code is generated when the size of the table of contents (TOC) grows to a size greater than 64KB. Extra code is needed for every reference to a TOC symbol that cannot be addressed with a 16-bit offset. Because a program containing generated code may have poor performance, you should try to reduce the number of TOC entries needed by the program before using this option.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files
Codestore

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Link Options (AIX)" on page 107

dataStart Link Option (AIX)

Syntax:	Default:
link(dataStart, *address*)	link(dataStart, 0x20000000)

This option specifies *address* as the starting address for the initialized data (the .data section) of the output file.

By default, the starting address for the data section is 0x20000000.

Only positive values of *address* are permitted, and *address* must be greater than or equal to the default 0x20000000. A warning will be issued otherwise.

The system loader relocates the .data section at run time, so the specified number only affects addresses listed in address maps or printed by utilities such as the **dump** or **nm** commands.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

dataImported Link Option (AIX)

Syntax:	**Default:**

link(dataImported, all)
link(dataImported, none)
link(dataImported, names)
	link(dataImported, all)

This option controls whether extern variables should be treated as imported data by default. *Imported* variables are dynamically bound with a shared portion of a library.

The default is link(dataImported, all), which causes all variables to be treated as imported data by default.

If link(dataImported, names) is specified, use link(dataImportedNames,*string*) once for each variable to be treated as imported data, where *string* consists of the *mangled* variable name. The variables specified are always treated as imported data, even if link(dataLocal) is set. Using link(dataImported, names) and link(dataImportedNames, *string*) does not affect whether variables are treated as imported or local by default.

Performance may decrease if an imported variable is assumed to be local.

Conflicts among the dataImported, dataImportedNames, dataLocal, and dataLocalNames link options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	The last option specified is the default for variables not explicitly listed in the name-list form.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

dataImportedNames Link Option (AIX)

Syntax:	**Default:**
link(dataImportedNames, *string*)	none

This option specifies an extern variable that should be treated as imported data, where *string* is the *mangled* variable name. *Imported* variables are dynamically bound with a shared portion of a library.

By default, no variables are specified.

This option is enabled when link(dataImported, names) is set.

Use link(dataImportedNames, *string*) once for each variable to be treated as local data. The variables specified are always treated as imported data, even if link(dataLocal) is set. Variables can be marked as *local*, rather than imported, data using the link(dataLocal, names) and *"dataLocalNames Link Option (AIX)" on page 114*) options.

Using link(dataImported, names) and link(dataImportedNames, *string*) does not affect whether other variables are treated as imported or local by default. The default for these other variables is specified using the link(dataImported, all) or link(dataLocal, all) options.

Using the dataImported and dataImportedNames to declare that an extern variable is imported from another shared library when it is actually local to the current shared library may decrease performace.

Conflicts among the dataImported, dataImportedNames, dataLocal, and dataLocalNames link options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	The last option specified is the default for variables not explicitly listed in the name-list form.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

dataLocal Link Option (AIX)

Syntax:	**Default:**

link(dataLocal, all)
link(dataLocal, none)
link(dataLocal, names)
	link(dataLocal, none)

This option controls whether extern variables should be treated as local data by default. *Local* variables are statically bound with the functions that use them.

The default is link(dataLocal, none). Because link(dataImported, all) is set by default, all variables are treated as imported data by default.

Setting link(dataLocal, all) will change the default to assume that all variables are treated as local data.

If link(dataLocal, names) is specified, use link(dataLocalNames,*string*) once for each variable to be treated as local data, where *string* consists of the *mangled* variable name. The variables specified are always treated as local data, even if link(dataImported, all) is set. Using link(dataLocal, names) and link(dataLocalNames, *string*) does not affect whether variables are treated as imported or local by default.

Note that performance may decrease if an imported variable is assumed to be local.

Conflicts among the dataImported, dataImportedNames, dataLocal, and dataLocalNames link options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	The last option specified is the default for variables not explicitly listed in the name-list form.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

dataLocalNames Link Option (AIX)

Syntax:	**Default:**
link(dataLocalNames, *string*)	none

This option specifies an extern variable that should be treated as local data, where *string* is the *mangled* variable name. *Local* variables are statically bound with the functions that use them.

By default, no variables are specified.

This option is enabled when "dataLocal Link Option (AIX)" on page 113 is set.

Use link(dataLocalNames, *string*) once for each variable to be treated as local data. The variables specified are always treated as local data, even if "dataImported Link Option (AIX)" on page 111 is set. Variables can be marked as *imported*, rather than local, data using the link(dataImported, names) and *"dataImportedNames Link Option (AIX)" on page 112*) options.

Incorrectly using the dataLocal and dataLocalNames link options to declare that an extern variable is local to the current shared library when it is not will result in an error. Correctly using dataLocalNames to declare that an extern variable is local can improve performance.

Using link(dataLocal, names) and link(dataLocalNames, *string*) does not affect whether other variables are treated as imported or local by default. The default for these other variables is specified using the link(dataImported, all) or link(dataLocal, all) options.

Conflicts among the dataImported, dataImportedNames, dataLocal, and dataLocalNames link options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	The last option specified is the default for variables not explicitly listed in the name-list form.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

debug Link Option

Syntax:	**Default:**
link(debug[, yes])	link(debug, no)
link(debug, no)	

Use this option to prepare your program for analysis with a traditional debugger or performance analyzer. The IDE has the option to use traditional debugging information, which means that symbolic data and line number information is included in the target file. This option is not required for debugging C++ programs with the standard VisualAge C++ debugger, since the only information it requires for debugging is always contained in the codestore.

Neither ipmd (from VisualAge for C++ for OS/2, Version 3.0) nor idebug (from VisualAge for C++ for Windows, Version 3.5) can be used to debug C applications linked with this option.

If you build with "profile Code Generation Option" on page 81, link(debug) is set to yes implicitly.

Debug data increases the size of the executable code. Therefore, you usually rebuild the module without this option for the production release.

If you use the IDE to set build options to generate code for traditional debugging, the IDE sets link(debug) implicitly for you. If you edit your configuration file directly and set "debug Code Generation Option (C)" on page 65 or "lineOnlyDebug Code Generation Option (C)" on page 78, you must explicitly set link(debug) as well.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files
Codestore
Debugging with VisualAge C++

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

debugFullPath Link Option (AIX)

Syntax:	**Default:**
link(debugFullPath[, yes])	link(debugFullPath, no)
link(debugFullPath, no)	

This option specifies what path information is stored for source files when you use the link(debug) option.
The default is link(debugFullPath, no), which preserves the relative path names of files.

Specifying link(debugFullPath, yes) causes VisualAge C++ to preserve the absolute (full) path name of source files to which the link(debug) option applies. The full path is useful if the executable file is moved to another directory; if you specified link(debugFullPath, no), traditional debuggers such as **xldb** would be unable to find source files, but specifying link(debugFullPath, yes) would allow them to locate the files successfully.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

entry Link Option

Syntax:	**Default:**
AIX link(entry[, yes])	AIX link(entry[, yes])
AIX link(entry, no) link(entry, *string*)	OS/2 WIN none

This option sets the entry point, or starting address, of the executable output file, as shown in the following table:

link(entry, ***string*****)**	Sets the executable's entry point to the address of the symbol in *string*. The symbol must be the name of a function.
AIX link(entry[, yes])	Sets the executable's entry point to "__start" (double underscore start).
AIX link(entry, no)	Indicates that the executable output file has no entry point.

Specifying link(entry, yes) will prevent C++ initialization, correct multi-thread startup, and profiling.

AIX

The default is link(entry, yes).

OS/2

WIN No default entry point is set.

C++

You can also specify the starting address in C++ code with #pragma entry. The pragma overrides this configuration file option.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

export Link Option

Syntax:

AIX link(export, *external_name*
, [*internal_name*])

OS/2 WIN link(export, *external_name*
, [*internal_name*]
, [*ordinal*]
, [RESIDENTNAME | NONAME]
, [PARMWORDS | CONSTANT]
, [*numWords*])

Default:
none

This option makes an internal symbol visible outside the module being built. You can use it to export symbols from a DLL or shared library.

C++

For modules built from C++ source code, the options link(exportAll), link(export_NameSpace,*name*), and link(export_Prefix, *prefix*) are easier to use than link(export, ...) or link(exportList, *fileName*.), but give you less flexibility.

You can specify just the name of the symbol with this option. Optionally, you can specify some or all the parameters described in the table below. The only way to apply the optional parameters to exported symbols is to use this option or link(exportList, *fileName*).

The parameters shown enclosed in [and] are optional. When you omit them, insert a comma as a place holder if you specify any later parameters. The | character separates alternatives.

For example, the line below exports the symbol ABC. The second parameter is omitted, so the internal name of the symbol must also be ABC. The third parameter puts ABC in position 5 in the module definition table, giving it ordinal value 5. The parameter after NONAME is also omitted, but a place-holding comma is not required because no other parameters follow. The effect of NONAME depends on the operating system as explained in the table below.

```
link(export, "ABC",, 5,NONAME)
```

This table describes the parameters of this option in detail.

parameter	**meaning**
external_name	This string is the name by which the symbol is known to other modules.
internal_name	This string is the actual name of the function or data item as it appears within the DLL or shared library. By default this is the same as the external name. C++ If the name is defined in a C++ module and is not declared with extern "C" linkage, this must be the mangled name. WIN On Windows, the name must be in decorated form.
ordinal	This number specifies the function's ordinal position within the module definition table. The value of ordinal must be in the range 1 to 65,535. If you supply an ordinal number, the symbol can be referred to when it is imported by either its external name or its ordinal value. AIX *ordinal* is ignored on AIX
RESIDENTNAME or NONAME	OS/2 RESIDENTNAME tells OS/2 to keep the name of the symbol resident in memory at all times. You can specify this only if you also supply an ordinal. If you do not supply an ordinal, OS/2 keeps the name memory resident by default. WIN RESIDENTNAME is ignored on Windows NT. AIX RESIDENTNAME is ignored on AIX. OS/2 NONAME is ignored on OS/2. WIN NONAME removes the name of the function from the symbol table on Windows NT. You can specify this only if you also supply an ordinal, because the function can be located only by ordinal number. AIX NONAME is ignored on AIX.

parameter	**meaning**
PARMWORDS or CONSTANT	OS/2 Use PARMWORDS only when you are including the sixth parmeter, described below. WIN PARMWORDS is ignored on Windows NT. AIX PARMWORDS is ignored on AIX. OS/2 CONSTANT is ignored on OS/2. WIN Use CONSTANT on Windows NT when the exported symbol is the name of a data item. Do this to avoid inappropriate symbol resolution when linking to a module that does not declare the data reference as an import. AIX CONSTANT is ignored on AIX.
numWords	OS/2 This number must be preceded by PARMWORDS. It specifies the number of 2-byte words required to store the function parameters on OS/2 and used only for symbols in sections with I/O Privilege. The number and PARMWORDS are ignored in sections that are not IOPL. You can specify IOPL with link(code,*attributes*) or link(section, *name*, *attributes*). WIN *numWords* is ignored on Windows NT. AIX *numWords* is ignored on AIX.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files
Decorated and Mangled Names

RELATED TASKS

Set Build Options
Build
Get Mangled Names for C++ Symbols

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

exportAll Link Option (C++)

Syntax: **Default:**

link(exportAll[, yes]) link(exportAll, no)	link(exportAll, no)

This option determines whether all defined, external symbols in C++ source are exported from the target module. If you are building a DLL or shared library, this is the simplest way of making symbols externally visible so that other modules can access them.

By default, all C++ symbols referenced by any other DLL, shared library or executable file that is represented in the same codestore as the target DLL or shared library file are exported. If you are building a DLL or shared library to be used by modules not in the current codestore, you should consider alternative ways of exporting symbols.

The options link(exportAll), link(export_NameSpace, *name*), and link(export_Prefix, *prefix*) are easier to use than link(export, ...) or link(exportList, ...), but give you less flexibility.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files
Codestore

RELATED TASKS

Set Build Options
Build

export_NameSpace Link Option (C++)

Syntax:	**Default:**
link(export_NameSpace, *name*)	link(export_NameSpace, null)

Use the export_NameSpace link option to export all symbols in the specified namespace. If you are building a DLL or shared library, this option lets you select a subset of all symbols in a library for access by other modules.

Only one instance of this option can be specified. Also, the effect of this option is cumulative with other export options. For example:

- Specifying link(exportAll) and link(export_NameSpace, my_proj) together exports all symbols.
- Similarly, specifying link(export_Prefix, CS_) and link(export_NameSpace, my_proj) together exports all symbols found in namespace my_proj and all symbols that start with CS_.

The options link(exportAll), link(export_NameSpace, *name*), and link(export_Prefix, *prefix*) are easier to use than link(export, ...) or link(exportList, ...), but give you less flexibility.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files
Codestore

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

export_Prefix Link Option (C++)

Syntax:	**Default:**
link(export_Prefix, *prefix*)	link(export_Prefix, null)

Use the export_Prefix link option to export all symbols that start with the specified prefix. If you are building a DLL or shared library, this option lets you select a subset of all symbols in a library for access by other modules.

Only one instance of this option can be specified. Also, the effect of this option is cumulative with other export options. For example:

- Specifying link(exportAll) and link(export_Prefix, CS) together exports all symbols.
- Specifying link(export_Prefix, CS_) and link(export_NameSpace, my_proj) together exports all symbols found in namespace my_proj and all symbols that start with CS_.

The options link(exportAll), link(export_NameSpace,*name*), and link(export_Prefix, *prefix*) are easier to use than link(export, ...) or link(exportList, ...), but give you less flexibility.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files
Codestore

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

exportList Link Option

Syntax:	**Default:**
link(exportList, *fileName*)	none

This option names a file containing specifications for symbols to be exported from the target module. If you are building a DLL or shared library, you can use this option to make symbols visible to other modules that use them.

C++

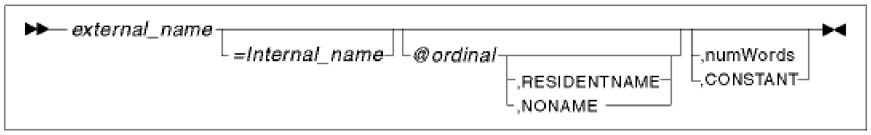
For modules build from C++ source code, the options link(exportAll), link(export_NameSpace,*name*), and link(export_Prefix, *prefix*) are easier to use than link(export, ...) or link(exportlist, ...), but give you less flexibility.

You can specify just the names of symbols to export in the export file the you use with this option. Optionally, you can specify some or all the parameters described in the table below. The only way to apply the optional parameters to exported symbols is to use this option or link(export, ...).

You must prepare the file before processing the configuration file. Therefore, you may want to consider alternative methods of exporting symbols from a DLL or shared library.

The file must be formatted as one or more lines, each of which has one of the following two forms.

1. You can put a comment line anywhere. This diagram shows the syntax of a comment line:
2. An export specification must fit on one line. This diagram shows the syntax of an export specification:



You can insert whitespace anywhere there is a blank in the text above. The parameters of an export specification are as follows:

parameter	meaning
external_name	This string is the name by which the symbol is known to other modules.
internal_name	This string is the actual name of the function or data item as it appears within the DLL or shared library. By default this is the same as the external name. C++ If the name is defined in a C++ module and is not declared with extern "C" linkage, this must be the mangled name. WIN On Windows, the name must be in decorated form.
ordinal	This number specifies the function's ordinal position within the module definition table. The value of *ordinal* must be in the range 1 to 65,535. If you supply an ordinal number, the symbol can be referred to when it is imported by either its external name or its ordinal value. AIX *ordinal* is ignored.

parameter	meaning
RESIDENTNAME or NONAME	OS/2 RESIDENTNAME tells OS/2 to keep the name of the symbol resident in memory at all times. You can specify this only if you also supply an ordinal. If you do not supply an ordinal, OS/2 keeps the name memory-resident by default. AIX WIN RESIDENTNAME is ignored . NONAME removes the name of the function from the symbol table on Windows. You can specify this only if you also supply an ordinal, because the function can be located only by ordinal number. AIX NONAME is ignored.
numWords or CONSTANT	OS/2 The number specifies the number of 2-byte words required to store the function parameters on OS/2 and is used only for symbols in sections with I/O privilege. This number is ignored in sections that are not IOPL. You can specify IOPL with link(code,*attributes*) or link(section, *name*, *attributes*). AIX WIN This number is ignored. AIX OS/2 CONSTANT is ignored. WIN Use CONSTANT on Windows when the exported symbol is the name of a data item. Do this to avoid inappropriate symbol resolution when linking to a module that does not declare the data reference as an import.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files
Decorated and Mangled Names

RELATED TASKS

Set Build Options
Build
Get Mangled Names for C++ Symbols

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
CreateExportList Command

AIX "Link Options (AIX)" on page 107

exportSymbolVisibility Link Option (AIX)

Syntax:	**Default:**

link(exportSymbolVisibility, *option*)	link(exportSymbolVisibility,“nosymbolic-”)
	For BSS symbols:
	link(exportSymbolVisibility,nosymbolic)

This option sets the visibility for exported symbols that do not have the attribute set explicitly (with an export file). The value for *option* can be one of the following:

nosymbolic
Inserts glue code for function calls to this symbol from within the same module. This option allows references to this symbol to be overridden.

symbolic
Prevents the glue code from being inserted.

nosymbolic-
Function calls within a module cannot be overridden. This behavior is equivalent to the non-run-time linking behavior. The default is `"nosymbolic-"` (with quotes), except for BSS symbols, in which case the default is `nosymbolic`.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

▶ AIX “Link Options (AIX)” on page 107

force Link Option

Syntax:	**Default:**

link(force[, yes])
link(force, no)
	link(force, no)

Use this option to force creation of a load module even if there are unresolved symbol errors from the link stage. By default a load module is not produced if errors occur during a build.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

generatedFilesDirectory Link Option

Syntax:	Default:
link(generatedFilesDirectory, *pathName*)	link(generatedFilesDirectory, *current target directory*)

Use this option to specify where extra generated files, such as the import library or .lib file for a DLL or shared library, will be placed.

If not specified otherwise, these files are placed in the current target directory.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

heap Link Option (AIX)

Syntax:	Default:
link(heap, *size*)	link(heap, 0)

This option sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap to *size* bytes. This value is used by the system loader to set the soft ulimit.

The default setting is link(heap, 0).

Valid values of *size* are 0 and multiples of 0x10000000 (0x10000000, 0x20000000, 0x30000000, ...). The maximum value allowed by the system is 0x80000000.

If the value of *size* is 0, a single 256MB (0x10000000 byte) data segment (segment 2) will be shared by the static data, the
heap, and the stack. If the value is non-zero, a data area of the specified size

(starting in segment 3) will be shared by the static data and the heap, while a separate 256 MB data segment (segment 2) will be used by the stack. So, the total data size when 0 is specified 0 is 256MB, and the total size when 0x10000000 is specified is 512MB, with 256MB for the stack and 256MB for static data and the heap.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Link Options (AIX)” on page 107

import Link Option

Syntax:	**Default:**
link(import, [*internal_name*], *library_name*, *export_name*)	none
OS/2 WIN link(import, *internal_name*, *library_name*, *ordinal*)	

Use this option to specify that a symbol is imported from a dynamic link library or shared library. Use this option if your target file uses a library that is not built with the same configuration file and when you do not have an import library.

The optional first argument specifies the name for the symbol in the importing program. If you do not specify an internal name, the name of the function or data item in the exporting library is used as the name for the symbol in the module being built. You must include a comma as a placeholder when you omit the internal name.

OS/2

WIN You must specify an internal name when you specify the exported symbol by ordinal number.

The second argument is the name of the exporting module.

OS/2

WIN In the third argument, you can supply either the ordinal or the name of the symbol, as defined in the exporting module.

AIX In the third argument, you must supply the name of the symbol, as defined in the exporting module

C++

If the exported symbol is defined in a C++ module, and is not declared with extern "C" linkage, the export name must be the mangled name of the symbol in the DLL or shared library. The mangled name is the name as it appears in the object module or link map, rather than as it appears in the source file.

WIN

On Windows, you must supply the name in decorated form.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

initializationTermination Link Option (AIX)

Syntax:	**Default:**
link(initializationTermination, [initf][, [finif][, priority]])	priority defaults to 0

This option specifies an initialization and termination function for a module, where `initf` is an initialization routine, `finif` is a termination routine, and `priority` is a signed integer with values from -2,147,483,648 to 2,147,483,647. You must specify at least one of `initf` or `finif`. If you do not specify `priority`, 0 is the default.

This option sorts routines by priority, starting with the routine with the smallest (most negative) priority. It invokes initialization routines in order, and termination routines in reverse order.

It invokes routines with the same priority in an unspecified order. However, if multiple `link(initializationTermination)` options specify the same priority and both an initialization and termination routine, it preserves the relative order of the routines.
For example, if you specify `link(initializationTermination, i1, f1)` and `link(initializationTermination, i2, f2)`, function `i1` and `i2` are invoked in an unspecified order. However if `i1` is invoked before `i2` when the module is loaded, `f2` will be invoked before `f1` when the module is unloaded.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

libPathOut Link Option

Syntax:	Default:

link(libPathOut[, *string*])
link(libPathOut, no)
	link(libPathOut, no)

This option sets the search path in the .loader section of the target to *string*. This overrides the search path specified with the "libSearchPath Link Option" option.

The default setting is link(libPathOut, no), which causes the path specified with link(libSearchPath) to be used.

If link(libPathOut, *string*) is specified with an empty or missing *string*, the path specified with link(libSearchPath) is used.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Linker Searches

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

libSearchPath Link Option

Syntax:	Default:
link(libSearchPath, *path*)	AIX link(libSearchPath, "/usr/vacpp/lib") OS/2 WIN link(libSearchPath, "")

This option specifies the search path for libraries at the link stage of a VisualAge C++ build.

AIX

On AIX, this option also specifies the search path for shared libraries to be loaded at run time. The path specified with this option is stored in the build target. If you set the LIBPATH environment variable, the variable overrides the path specified here.

When VisualAge C++ looks for libraries to link, the search has two distinct stages:

1. The path controlled by this option is searched first.

   AIX On AIX, the search starts by default in /usr/vacpp/lib.

   OS/2 WIN On OS/2 and Windows NT platforms, the initial search path is empty.
2. The default search path is searched last.

   AIX On AIX, the default search path is /usr/lib:/lib.

   OS/2 WIN On OS/2 and Windows NT platforms the default search path is stored in the LIB environment variable.

The default libSearchPath option path will be searched after any paths you specify using this option, and before the default system search path.

Multiple search paths can be specified in the following ways:

- AIX using the ":" separator within a single option string
- OS/2 WIN using the ";" separator within a single option string
- using multiple options within the same option directive; paths are appended in the order they appear
- using nested option directives; paths in inner directives are prepended to paths in outer directives

The following example illustrates the order in which multiple search paths are concatenated:

```
option link(libsearchpath,"a:b") {
option link(libsearchpath, "c:d") link(libsearchpath,"e:f") {
         option link(libsearchpath,"g:h") {
             source ...
}
}
}
```

The option directives above apply the specified paths to the source in the following order:

```
g,h,c,d,e,f,a,b
```

AIX

On AIX, you can also specify a prefix for the paths specified with this option by building with option link(libSearchPathPrefix,*path*). Usually you do this to build on one level of AIX to run on a different level, so that you can search a different path on the development platform than on the target platform. The prefix is not stored in the executable.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Linker Searches

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

libSearchPathPrefix Link Option (AIX)

Syntax:	Default:
link(libSearchPathPrefix, *string*)	none

This option prefixes the names of the library search paths with the specified *string*.

When this option is used, the standard library directories, including the ones specified by the link(libSearchPath, *string*) option, are not searched.

This option is useful when developing a new version of a library. Usually you use it to build on one level of AIX and run on a different level, so that you can search a different path on the development platform than on the target platform. This is possible because the prefix is not stored in the executable.

If you apply this option more than once in an option directive, the strings are appended to each other in the order in which they appear in your configuration file, and then they are added to the beginning of the library search paths. When the directives are nested, the paths from inner libSearchPathPrefixes precede those from outer libSearchPathPrefixes.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Linker Searches

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

linkWithMultiThreadLib Link Option

Syntax:	Default:

link(linkWithMultiThreadLib[, yes])
link(linkWithMultiThreadLib, no)
	link(linkWithMultiThreadLib, no)

This option determines whether the module being built is linked with the singlethread or multithread version of the VisualAge C++ run-time library. By default, the singlethread version is linked.

OS/2

WIN This option determines which run-time library is used when you build with link(defaultLibs), and is overridden by link(defaultLibs, no).

All multithread applications and programs that use the Open Class Library require the multithread run-time library.

Take care, when setting this option, that all modules required by applications that use the module being built are compatible. If you link one module with the multithread run-time library, you should link every module used by the same application also the multithread library, so that the application has only one run-time environment.

OS/2

WIN If you are building a subsystem with link(standAlone), link(linkWithMultiThreadLib, no) is set implicitly.

C

OS/2 If you are building a device driver with gen(ring0), link(linkWithMultiThreadLib, no) is set implicitly.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Link Options (AIX)” on page 107

linkWithSharedLib Link Option

Syntax:	**Default:**

link(linkWithSharedLib[, yes])
link(linkWithSharedLib, no)
	link(linkWithSharedLib, no)

This option determines whether the module being built is linked with the shared, or dynamically loadable, version of the VisualAge C++ run-time library or with the static version.

OS/2

WIN If you are building a subsystem with link(standAlone), linkWithSharedLib is set to no implicitly.

OS/2

WIN On OS/2 and Windows NT platforms, this option determines which run-time library is used when you build with link(defaultLibs), and is overriden by link(defaultLibs, no).

Linking statically produces a larger executable file that includes its own run-time environment, but that does not depend upon the VisualAge C++ run-time library being available on the target machine for dynamic loading at run-time.

Take care, when setting this option, that all modules required by applications which use the module being built are compatible. Generally, you want to avoid linking to more than one run-time environment by building all modules with the same version of the library.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

map Link Option

Syntax:	**Default:**

link(map[, yes])
link(map, *fileName*)
link(map, no)
	link(map, no)

This option controls whether the linker produces a map file, and optionally specifies the name of the map file. By default a map is not created.

If link(map, yes) is set, the map file generated will have the same base name as the module being created, and its extension will be .map.

C++

OS/2 WIN The map option is set implicitly to yes by link(lineNum).

C

This option is implicitly set to no when "syntaxOnly C File Option" on page 161 or "preprocessOnly C Preprocessor Option" on page 143 are in effect.

The map shows the composition of each segment and lists all the public symbols defined in the module being built. The symbols appear in two lists, one sorted by name and one sorted by address.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

nameMangling Link Option (C++)

Syntax:	**Default:**
link(nameMangling, *ansi*, *characters*)	link(nameMangling, *ansi*)
link(nameMangling, *compat*, *characters*)	

This option choses the name mangling scheme for external symbol names generated from C++ source code. On AIX the *characters* argument has no meaning and is ignored.

You can select one of two mangling schemes:

- compat
  This scheme uses the same mangling scheme as earlier versions of VisualAge C++ and predecessor products Use this scheme for compatibility with link modules build with ealier versions of VisualAge C++
- ansi
  This scheme complies with the C++ standard. Use this scheme for compatibility with standard C++.

By default VisualAge C++ uses a scheme that conforms to specifications of the C++ standard.

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

AIX "Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

"Build Options for Compliance With the C++ Standard" on page 180
"Build Options for Compatibility with Old C++ Compilers" on page 182
"Link Options (AIX)" on page 107

padding Link Option

Syntax:	Default:

link(padding[, yes])
link(padding, no)
	link(padding[, yes])

This option determines whether the module being built is to be compressed to the minimum possible size. By default, the target contains extra padding bytes for quicker linking of future changes. Usually, this saves development time.

Set padding to no when you want to produce a smaller target module. Usually you do not eliminate padding until your code is tested and stable.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
AIX "Link Options (AIX)" on page 107

runtimeLinking Link Option (AIX)

Syntax:	Default:

link(runtimeLinking[, yes])
link(runtimeLinking, no)
	link(runtimeLinking, no)

This option applies to AIX Version 4.2 only.

When `link(runtimeLinking, yes)` is specified, undefined symbols are resolved at run time. This option needs to be on if an executable is to use run-time linking successfully.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Link Options (AIX)" on page 107

runtimeLinkerStructure Link Option (AIX)

Syntax:	Default:

link(runtimeLinkerStructure[, yes])
link(runtimeLinkerStructure, no)
	link(runtimeLinkerStructure, no)

This option applies to AIX Version 4.2 only.

When `link(runtimeLinkerStructure, yes)` is specified, the run-time linker data structure is allocated in the target (and exported), and a pointer to the run-time linker is created.

When `link(runtimeLinkerStructure, no)` is specified, the run-time linker data structure is only created if "initializationTermination Link Option (AIX)" on page 127 has been specified. In this case the pointer to the run-time linker is NULL.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

▶ AIX "Link Options (AIX)" on page 107

runtimeLinkPath Link Option (AIX)

Syntax:	Default:

link(runtimeLinkPath[, yes])
link(runtimeLinkPath, no)
	link(runtimeLinkPath, yes)

This option includes the full path name for shared objects that are placed into the .loader section unless you specify link(runtimeLinkPath, no).

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

sharedLibPriority Link Option (C++, AIX)

Syntax:	**Default:**
link(sharedLibPriority, *number*)	None

This option sets the initialization order for multiple shared libraries.

You can specify any integer literal for *number* in the range INT_MIN+1024+1 (highest priority, initialized first) to INT_MAX (lowest priority, initialized last). Numbers from INT_MIN and INT_MIN+1024 are reserved for system use.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

staticSymbols Link Option (AIX)

Syntax:	**Default:**

link(staticSymbols[, yes])
link(staticSymbols, no)
	link(staticSymbols, no)

This option adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the XCOFF symbol table.

The default is to not add static variables to the symbol table. However, static functions are added to the symbol table.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

stack Link Option (AIX)

Syntax:	**Default:**
link(stack, *number*)	link(stack, 0)

This option sets the maximum size allowed for the user stack to *number* bytes.

The default is link(stack, 0), which causes the system default soft limit to be used.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

static Link Option (AIX)

Syntax:	**Default:**

link(static[, yes])
link(static, no)
	link(static, no)

This option specifies whether shared objects are statically bound into the source.

The default is link(static, no).

Symbols from any shared object file specified in the bind will be imported if they are defined by import files.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

strip Link Option (AIX)

Syntax:	Default:

link(strip[, yes])
link(strip, no)
	link(strip, no)

This option strips the symbol table, line number information, and relocation information from the output file.

The default is link(strip, no).

Specifying link(strip, yes) saves space, but impairs the usefulness of traditional debug programs when you are generating debug information using options such as "debug Code Generation Option (C)" on page 65 or "debug Link Option" on page 115. Stripping does not affect the VisualAge C++ debugger, since the only information it requires for debugging is contained in the codestore.

Using the **strip** command has the same effect.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

system Link Option (AIX)

Syntax:	Default:

link(system[, yes])
link(system, no)
	link(system, no)

The option `link(system, yes)` specifies that the system linker (ld) is invoked during a build.

The option `link(system, no)` calls the linker shipped with this product.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

systemOption Link Option (AIX)

Syntax:	Default:
link(systemOption, *string*)	None

This option allows you to specify system linker options in *string*. Multiple occurrences of this option are allowed, and the linker options specified are concatenated.

This option is used only when `"system Link Option (AIX)" on page 138` is specified.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

textStart Link Option (AIX)

Syntax:	Default:
link(textStart, *address*)	link(textStart, 0x10000000)

This option specifies *address* as the starting address for the text segment of the output file.

By default, the starting address for the text segment is 0x10000000.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165

AIX “Link Options (AIX)” on page 107

traceBackTable Link Option (AIX)

Syntax:	**Default:**

link(traceBackTable, none)
link(traceBackTable, small)
link(traceBackTable, full)
	link(traceBackTable, small) when link(debug, no)
	link(traceBackTable, full) when link(debug)

This option generates a traceback table that contains information about each function, including the type of function as well as stack frame and register information. The traceback table is placed in the text segment at the end of its code.

The available options are:

none No traceback table is generated.

small A traceback table is generated with no name or parameter information. This is the default when `"debug Link Option" on page 115` is specified.

full A full traceback table is generated. This is the default when `link(debug)` is specified.

C

The #pragma options directive `tbtable=`*option* must be specified before the first statement in the compilation unit.

C++ The #pragma options directive `tbtable=`*option* can be specified anywhere in the source.

RELATED CONCEPTS

Incremental C++ Builds
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Link Options (AIX)" on page 107

AIX #pragma options

Chapter 8. C Preprocessor Options

C Preprocessor Options

The options listed in the following table control how VisualAge C++ preprocesses C sources during builds.

- If an option does not apply to all platforms, the middle column shows the platforms on which it is used.
- If you specify options that are not recognized on your platform, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case-sensitive. If you omit both "yes" and "no" from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying pp(PreProcessOnly) is equivalent to specifying pp(preprocessonly, yes).

Option	Applies only to	Default
"preprocessOnly C Preprocessor Option"		pp(preprocessOnly, no)
"preserveComments C Preprocessor Option" on page 144		pp(preserveComments, no)
"stdout C Preprocessor Option" on page 145		pp(stdout, no)
"suppressLine C Preprocessor Option" on page 146		pp(suppressLine, no)

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

preprocessOnly C Preprocessor Option

Syntax:	Default:

pp(preprocessOnly[, yes])
pp(preprocessOnly, no)
	pp(preprocessOnly, no)

This option determines whether a full compile or just the preprocessing stage is executed for C source files. When only preprocessing is done, the output is a file that has the same name as the source file, with the extension .i.

By default, the C source is preprocessed and compiled, and no preprocessor output is produced.

You can use this option to process preprocessor directives in your source. For example, you can define and expand macros, include files, and see the effect of directives that control conditional compilation directives without performing the compilation.

Specifying pp(preprocessOnly) implicitly sets the following options:

"genAsm C File Option" on page 156
"genObject C File Option" on page 157
"map Link Option" on page 132
"listing C List Option" on page 151
OS/2 WIN list(allStructVar, no)
"expAllInc C List Option" on page 149
OS/2 WIN list(expMacros, no)
OS/2 WIN list(expUserInc, no)
OS/2 WIN list(full, no)
"fullXRef C List Option" on page 150
"incSource C List Option" on page 150
"minXRef C List Option" on page 152
OS/2 WIN list(refStructVar, no)

Specifying pp(preprocessOnly) enables the following options. Use these options to control preprocessor output:

"preserveComments C Preprocessor Option"
"stdout C Preprocessor Option" on page 145
"suppressLine C Preprocessor Option" on page 146

This option is implicity set to no by "syntaxOnly C File Option" on page 161.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C Preprocessor Options" on page 143

preserveComments C Preprocessor Option

Syntax:	**Default:**

pp(preserveComments[, yes])
pp(preserveComments, no)
	pp(preserveComments, no)

This option determines whether the C preprocessor output file includes comments from the source code.

By default, the C preprocessor does not include comments in its output.

This option is enabled only when you build with “preprocessOnly C Preprocessor Option” on page 143 and is implicitly set to no when you build with “syntaxOnly C File Option” on page 161.

You can also control preprocessor output with “stdout C Preprocessor Option” and “suppressLine C Preprocessor Option” on page 146.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Preprocessor Options” on page 143

stdout C Preprocessor Option

Syntax:	Default:

pp(stdout[, yes])
pp(stdout, no)
	pp(stdout, no)

This option determines whether the C preprocessor echoes output to stdout.

By default, C preprocessor output is written only to a file that has the same base name as the source file, with the extension .i. Set pp(stdout) to cause the preprocessor to echo this output to stdout as well.

This option is enabled only when you build with “preprocessOnly C Preprocessor Option” on page 143. You can also control preprocessor output with “preserveComments C Preprocessor Option” on page 144 and “suppressLine C Preprocessor Option” on page 146.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Preprocessor Options” on page 143

suppressLine C Preprocessor Option

Syntax:	**Default:**

pp(suppressLine[, yes])
pp(suppressLine, no)
	pp(suppressLine, no)

This option determines whether the C preprocessor generates or suppresses #line directives in the preprocessor output.

By default, the C preprocessor creates #line directives.

This option is enabled only when you build with “preprocessOnly C Preprocessor Option” on page 143 and is implicity set to no when you build with “syntaxOnly C File Option” on page 161.

You can also control preprocessor output with “preserveComments C Preprocessor Option” on page 144 and “stdout C Preprocessor Option” on page 145.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Preprocessor Options” on page 143

Chapter 9. C Listing Options

C Listing Options

The options listed in the following table control listings that the C compiler can produce during builds. The options apply only to C sources.

- If an option does not apply to all platforms, the middle column shows the platforms on which it is used.
- If you specify options that are not recognized on your platform, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case-sensitive. If you omit both "yes" and "no" from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying list(allStructVar) is equivalent to specifying list(allStructVar, yes).

Option	Applies only to	Default

list(allStructVar[, yes])
list(allStructVar, no)
	OS/2 WIN	list(allStructVar) with list(full)
		list(allStructVar, no) with list(listing)

list(attr[, yes])
list(attr, no)
list(attr, full)
	AIX	list(attr, no)

list(expAllInc[, yes])
list(expAllInc, no)
		OS/2 WIN list(expAllInc) with list(full)
		list(expAllInc, no) with list(listing) and list(incSource)

list(expMacros[, yes])
list(expMacros, no)
	OS/2 WIN	list(expMacros) with list(full)
		list(expMacros, no) with list(listing) and list(incSource)

list(expUserInc[, yes])
list(expUserInc, no)
	OS/2 WIN	list(expUserInc) with list(full)
		list(expUserInc, no) with list(listing) and list(incSource)

list(fileName, *fileName*)	OS/2 WIN	same base name as source and extension .lst

list(full[, yes])
list(full, no)
	OS/2 WIN	list(full, no)

list(fullXRef[, yes])
list(fullXRef, no)
		list(fullXRef, no)

list(incSource[, yes])
list(incSource, no)
		OS/2 WIN list(incSource) with list(full)
		list(incSource, no) with list(listing)

list(listing[, yes])
list(listing, no)
		list(listing, no)

list(minXRef[, yes])
list(minXRef, no)
		OS/2 WIN list(minXRef) with list(full)
		list(minXRef, no) with list(listing)

list(options[, yes])
list(options, no)
	AIX	list(options, no)

Option	Applies only to	Default
list(pageLen, *lines*)	OS/2 WIN	list(pageLen, 66)

list(refStructVar[, yes])
list(refStructVar, no)
	OS/2 WIN	list(refStructVar) with list(full)
		list(refStructVar, no) with list(listing)

list(reportLevel, *level*)	AIX	list(reportLevel, i)
list(subTitle, *string*)	OS/2 WIN	none
list(title, *string*)	OS/2 WIN	none

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

attr List Option (AIX Batch)

Syntax:	**Default:**

list(attr[, yes])
list(attr, no)
list(attr, full)
	list(attr, no)

This option controls whether the C or batch C++ compiler listing includes an attribute listing for all identifiers.

By default, no attribute listing is produced.

This option is available when you request a listing file with "listing C List Option" on page 151. When you build with list(listing), you must also set list(attr, yes) or list(attr, full) to add the attribute listing to your listing file. Specify list(attr, yes) to produce an attribute listing that reports only those identifiers that are used. Specify list(attr, full) to produce an attribute listing that reports all identifiers in the program.

Attribute information includes the type, storage class, and offset of the identifier. For structures, the relative displacement of each member is also given. All types of storage classes can appear in the attribute listing. In addition, the structure relationships for a structure member are given, and variables that are parameters or typedefs are marked as such.

This option does not produce a cross-reference listing unless you also set "fullXRef C List Option" on page 150 or "minXRef C List Option" on page 152.

This option is implicitly set to no when "preprocessOnly C Preprocessor Option" on page 143 is in effect.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C Listing Options" on page 147

expAllInc C List Option

Syntax:	**Default:**

list(expAllInc[, yes])
list(expAllInc, no)
	list(expAllInc, no) with "listing C List Option" on page 151 and "incSource C List Option" on page 150
	OS/2 WIN list(expAllInc) with list(full)

This option determines whether the listing file shows the contents of files that are named in #include preprocessor directives. It has an effect only when you request a listing file that includes your C source.

When you build with list(listing) and list(incSource), set list(expAllInc) to yes to insert the contents of included files into your source listing.

OS/2

WIN On OS/2 and Windows NT platforms, you can show the contents of files included with `#include "`*filename*`"` but not with `#include <`*filename*`>`, by building with list(expUserInc). When you build with list(full), specify list(expAllInc,no) to show only the #include directives in the listing.

This option is implicitly set to no when "preprocessOnly C Preprocessor Option" on page 143 is in effect.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C Listing Options" on page 147

fullXRef C List Option

Syntax:	**Default:**

list(fullXRef[, yes])
list(fullXRef, no)
	list(fullXRef, no)

This option determines whether the C compiler listing includes a full cross-reference table. This table shows all variables, structures, unions, and function names, and gives line numbers where names are declared.

This option is available when you request a listing file with list(listing). When you build with list(listing), you must also set list(fullXRef) to yes to add the cross-reference table to your listing file.

OS/2

WIN On OS/2 and Windows NT platforms, this option is implicitly set to yes when you build with list(full). You can specify list(fullXRef, no) to remove the cross-reference table from the listing.

To output a cross-reference table for only variables, structs, and function names that are referenced in your code, use “minXRef C List Option” on page 152.

This option is implicitly set to no when “preprocessOnly C Preprocessor Option” on page 143 is in effect.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Listing Options” on page 147

incSource C List Option

Syntax:	**Default:**

list(incSource[, yes])
list(incSource, no)
	OS/2 WIN list(incSource) with list(full)
	list(incSource, no) with “listing C List Option” on page 151

This option determines whether the listing file includes your C source code.

This option is available when you request a listing file with list(listing). When you build with list(listing), you must also set list(incSource) to yes to include your source code in your listing file.

OS/2

WIN On OS/2 and Windows NT platforms, this option is implicitly set to yes when you build with list(full), and you can specify list(incSource, no) to exclude the source code from the listing file.

This option is implicitly set to no when “preprocessOnly C Preprocessor Option” on page 143 is in effect.

You can control whether macros and include files are expanded in the listing with the following options:

OS/2 WIN list(expMacros)

OS/2 WIN list(expUserInc)

“expAllInc C List Option” on page 149

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Listing Options” on page 147

listing C List Option

Syntax:	**Default:**

list(listing[, yes])
list(listing, no)
	list(listing, no)

This option determines whether the C compiler generates a listing file. By default no compile listing is produced.

If you specify list(listing), a listing is produced for every source to which the option applies. By default, these files have the same name as the source file and the extension .lst.

OS/2

WIN On OS/2 and Windows NT platforms you can supply a file name by building with option list(fileName, *fileName*).

This option is implicitly set to no when any of the following options are in effect:

“preprocessOnly C Preprocessor Option” on page 143
“preserveComments C Preprocessor Option” on page 144
“stdout C Preprocessor Option” on page 145
“suppressLine C Preprocessor Option” on page 146

You can use a listing file as a tool for debugging. It contains error messages, and a prolog that gives the following information:

- The VisualAge C++ product number
- The C compiler version and release number
- The date and time compilation commenced
- A list of the compiler options in effect

You can control the appearance and contents of the listing file by specifying the options shown below. These options are enabled when you build with list(listing):

OS/2 WIN list(allStructVar)

AIX “attr List Option (AIX Batch)” on page 148
“expAllInc C List Option” on page 149

OS/2 WIN list(expMacros)

OS/2 WIN list(expUserInc)

“fullXRef C List Option” on page 150
“incSource C List Option” on page 150
“minXRef C List Option”

AIX “options C List Option (AIX)” on page 153

OS/2 WIN list(pageLen)

OS/2 WIN list(refStructVar)

AIX list(reportLevel)

OS/2 WIN list(subTitle, *string*)

AIX “tabSize C File Option (AIX)” on page 162

OS/2 WIN list(title, *string*)

OS/2 WIN On OS/2 and Windows NT platforms, building with list(full) also enables all the listing options that apply on OS/2 and Windows NT platforms, and implicitly sets them to produce a full listing.

Listing options are processed in the order they are specified. Therefore, you can combine them to include only what you want to appear in a listing file.

Example of a C Compile List File

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C Listing Options” on page 147

minXRef C List Option

Syntax: **Default:**

list(minXRef[, yes])
list(minXRef, no)
	list(minXRef, no) with "listing C List Option" on page 151
	OS/2 WIN list(minXRef) with list(full[, yes])

This option determines whether the C compiler listing includes a cross-reference table. This table shows referenced variables, structures, unions, and function names, and gives line numbers where names are declared.

To output a cross-reference table for all variables, structs, and function names, use"fullXRef C List Option" on page 150. The difference is that the full cross-reference table includes all symbols and the table produced by list(minXRef) includes only symbols that are referenced in the source.

This option is available when you request a listing file with list(listing). When you build with list(listing), you must also set list(minXRef) to yes to add the cross-reference table to your listing file.

OS/2

WIN On OS/2 and Windows NT platforms, this option also has an effect when you build with list(full). Option list(fullXRef) is implicitly set to yes when you build with list(full), and you can specify list(fullXRef, no) or list(minXRef, no) to remove cross-reference table from the listing.

This option is implicitly set to no when "preprocessOnly C Preprocessor Option" on page 143 is in effect.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C Listing Options" on page 147

options C List Option (AIX)

Syntax:	**Default:**

list(options[, yes])
list(options, no)
	list(options, no)

This option controls whether the C compiler listing includes a list of all option settings.

By default, no option setting information is produced.

This option is available when you request a listing file with "listing C List Option" on page 151. When you build with list(listing), you must also set list(options, yes) to add the list of option settings to your listing file.

This option is implicitly set to no when any of the following options are in effect:

"preprocessOnly C Preprocessor Option" on page 143
"preserveComments C Preprocessor Option" on page 144
"stdout C Preprocessor Option" on page 145

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C Listing Options" on page 147

Chapter 10. C File Options

C File Options

The options listed in the following table control how the C compiler uses files during builds. The options apply only to C sources.

- If an option does not apply to all platforms, the middle column shows the platforms on which it is used.
- If you specify options that are not recognized on your platform, they are ignored. You can specify options that are specific to other platforms so that your configuration file is portable between platforms.
- Option names are not case-sensitive. If you omit both “yes” and “no” from an option that has a yes or no form, yes is implied regardless of the default setting for that option. For example, specifying file(GenAsmWithCode) is equivalent to specifying file(genasmwithcode, yes).

Option	**Applies only to**	**Default**
file(asmFileName, *pathName*)	OS/2 WIN	same base name as source and the extension .asm

file(genAsm[, yes])
file(genAsm, no)
		file(genAsm, no)

file(genAsmWithCode[, yes])
file(genAsmWithCode, *pathName*)
file(genAsmWithCode, no)
	OS/2 WIN	file(genAsmWithCode, no)

file(genObject[, yes])
file(genObject, *pathName*)
file(genObject, no)
		file(genObject[, yes])

file(genPCH[, yes])
file(genPCH, *pathName*)
file(genPCH, no)
		file(genPCH, no)

file(genProto[, yes])
file(genProto, no)
file(genProto, withParameterNames)
	AIX	file(genProto, no)

file(ignoreEnvIncPaths[, yes])
file(ignoreEnvIncPaths, no)
	OS/2 WIN	file(ignoreEnvIncPaths, no)

file(ignoreUserIncPaths[, yes])
file(ignoreUserIncPaths, no)
	OS/2 WIN	file(ignoreUserIncPaths, no)

file(makeDep[, yes])
file(makeDep, no)
		file(makeDep, no)

file(once[, yes])
file(once, no)
	AIX	file(once, no)

file(syntaxOnly[, yes])
file(syntaxOnly, no)
		file(syntaxOnly, no)

file(tabSize, *number*)	AIX	file(tabSize, 8)

Option	Applies only to	Default

file(usePCH[, yes])
file(usePCH, *fileName*)
file(usePCH, no)
		file(usePCH, no)

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

genAsm C File Option

Syntax:
file(genAsm[, yes])
file(genAsm, no)

Default:
file(genAsm, no)

This option determines whether the C compiler generates an assembler listing file.

By default, no assembler listing is produced.

If you build with file(genAsm), an assembly listing is produced for every source to which the option applies. Each listing file has the same base name as the source file and the extension .asm (on OS/2 and Windows NT platforms) or .s (on AIX).

OS/2

WIN On OS/2 and Windows NT platforms, you can specify file names for assembly listing files by also using the option file(asmFileName, *fileName*). You can also get an assember listing that includes machine code, by building with file(genAsmWithCode).

You can use the output file as an aid for debugging. It may not be valid input to an assembler.

This option is implicitly set to no when "syntaxOnly C File Option" on page 161 or "preprocessOnly C Preprocessor Option" on page 143 are in effect.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

genObject C File Option

Syntax:	**Default:**
file(genObject[, yes])	file(genObject[, yes])
file(genObject, *pathName*)	
file(genObject, no)	

This option determines whether the C compiler generates object files.

By default, the C compiler produces an object file in the current working directory for each source file.

When file(genObject) is set to yes, an object file is produced for every source to which the option applies. By default, object files have the same name as the source file and the extension .obj (on OS/2 and Windows NT platforms) or .o (on AIX).

When file(genObject, *pathName*) is specified,*pathName* can be the relative or absolute path name of either a file or a directory. If you supply a file name, the name of the object file created will be *pathName*. In this case, make sure the option applies to only one source file. Otherwise the object code is overwritten by every file compiled after the first source because the same output file name is used for each source. If you supply a directory name, the object files created are placed into that directory.

This option is implicitly set to no when "syntaxOnly C File Option" on page 161 or "preprocessOnly C Preprocessor Option" on page 143 are in effect.

Setting this option to no implicitly sets "debug Code Generation Option (C)" on page 65.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

genPCH C File Option

Syntax:	**Default:**
file(genPCH[, yes])	file(genPCH, no)
file(genPCH, *pathName*)	
file(genPCH, no)	

This option determines whether the C compiler generates a precompiled header file, and optionally names that file.

By default no precompiled header file is produced.

When you specify file(genPCH), but not "usePCH C File Option" on page 163, a precompiled header file is always produced. When you specify both file(genPCH) and file(usePCH), the precompiled header file is produced only if does not already exist or is out of date.

If you do not supply a name, the precompiled header file is named csetc.pch.

When file(genPCH, *pathName*) is specified, *pathName* can be the relative or absolute path name of either a file or a directory. If you supply a file name, the name of the output file created will be *pathName*. In this case, supply a different file name for every source that does not share a common header file. Otherwise the output is overwritten by every file compiled after the first source because the same output file name is used for each source. If you supply a directory name, the precompiled header files created are placed into that directory.

To use the precompiled header files, use file(usePCH). Specify both file(genPCH) and file(usePCH) to keep precompiled header files up to date. If you specify different file names with the two options, the name specified last is used for both options.

You can also use #pragma hdrfile in your source, to request precompiled headers. The file name in the pragma overrides the file name specified with this option.

This option is implicitly set to no when "syntaxOnly C File Option" on page 161 or "preprocessOnly C Preprocessor Option" on page 143 are in effect.

RELATED CONCEPTS

C Compilation
Configuration Files
Precompiled C Headers
Organizing Your C Source for Precompiled Headers

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

genProto C File Option (AIX)

Syntax:	Default:

file(genProto[, yes])
file(genProto, no)
file(genProto, withParameterNames)
	file(genProto, no)

This option determines whether ANSI C prototypes are produced from K&R function definitions. The prototypes will be produced in a file with the same base name as the original source file and the file extension .proto. They will also be echoed to stdout.

By default, ANSI prototypes are not produced.

Setting file(genProto, yes) will cause prototypes to be generated without parameter names. Parameter names are included in the prototype when file(genProto, withParameterNames) is set.

For example, for the following function:

```
foo(a,b,c)
float a;
int *b;
/* ... */
```

specifying file(genProto, yes) produces the following prototype:

```
int foo(a,b,c) foo(double, int*, int);
```

The parameter names are dropped. On the other hand, specifying file(genProto, withParameterNames) produces the following prototype:

```
int foo(a,b,c) foo(double a, int* b, int c);
```

In this case the parameter names are kept.

Note that `float a` is represented as `double` or `double a` in the prototype, since ANSI states that all narrow-type arguments (such as char, short, and float) are widened before they are passed to K&R functions.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

makeDep C File Option

Syntax:	Default:

file(makeDep[, yes])
file(makeDep, no)
	file(makeDep, no)

The C compiler can generate information that is suitable for inclusion in a description file for a make utility for traditional compilers. You never need this output to build with VisualAge C++, but VisualAge C++ can produce it in case you have some other use for the information.

If you build with file(makeDep, yes), a file is output for each source file to which the option applies. The files have the same base name as the source file and the extension .u.

RELATED CONCEPTS

C Compilation
Configuration Files
When to Use Makefiles

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C File Options” on page 155

once C File Option (AIX)

Syntax:	**Default:**

file(once[, yes])
file(once, no)
	file(once, no)

This option controls whether to include a header file more than once when it is specified in several of the files you are compiling.

By default, the C compiler includes header files as many times as they are specified in the source files being compiled.

Set file(once) to cause the C compiler to include each header file once only, regardless of the number of times it is specified in the source files.

The compiler uses the full path name to determine if a file has already been included. No attempt is made to resolve . or .. in the path name. #include statements that include . or .. in the path statements may cause the same file to be included more than once.

Note: Do not use the file(once) option if both of the following conditions are true:

1. You include both stdio.h and stdarg.h (in that order) in your source files.
2. You are using the macro va_list.

va_list must be defined twice to have any effect, and file(once) defeats this purpose.

The following example shows how the C compiler resolves whether a file has already been included.

```
#include <stdio.h>  /* Found in /usr/include/stdio.h  */
#include <stdio.h>  /* Already included    */
#include </usr/include/stdio.h> /* Already included */
#include <./stdio.h>  /* Resolves to /usr/include/./stdio.h */
/* which is the same file, but this */
/* file will be included again. */
```

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

syntaxOnly C File Option

Syntax:	**Default:**

file(syntaxOnly[, yes])
file(syntaxOnly, no)
	file(syntaxOnly, no)

This option determines whether the C compiler performs a full compile, or just a syntax check.

By default, a full compile is performed.

If you set syntaxOnly to yes, the C compiler outputs only error, warning, and information messages. You can use "listing C List Option" on page 151 to produce listing files as well.

This setting implicity sets the following options:

“genAsm C File Option” on page 156

OS/2

WIN file(genAsmWithCode)

“genObject C File Option” on page 157

“map Link Option” on page 132

“preprocessOnly C Preprocessor Option” on page 143

“preserveComments C Preprocessor Option” on page 144

“stdout C Preprocessor Option” on page 145

“suppressLine C Preprocessor Option” on page 146

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“C File Options” on page 155

tabSize C File Option (AIX)

Syntax:	**Default:**
file(tabSize, *number*)	file(tabSize, 8)

This option tells the C compiler to consider the tabs in your source files to be *number* character spaces in length.

By default, tabs are 8 character spaces in length.

This option affects error messages that specify the column number at which an error occurred. For example, the compiler will consider tabs as having a width of one character if you specify file(tabSize, 1). In this case, you can consider one character *position* (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character *column*.

If “listing C List Option” on page 151 and “incSource C List Option” on page 150 are specified, tab characters in the listed source lines are expanded to spaces, based on the value of the tabSize option.

RELATED CONCEPTS

C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

usePCH C File Option

Syntax:	**Default:**

file(usePCH[, yes])
file(usePCH, *fileName*)
file(usePCH, no)
	file(usePCH, no)

This option determines whether the C compiler uses a precompiled header file, if it exists and is current.

By default, the C compiler does not use precompiled header files.

If you do not supply a name, the precompiled header file is named csetc.pch. If you do not supply an absolute path, the file name is relative to the current working directory.

To create a precompiled header file, use "genPCH C File Option" on page 158. Specify both file(genPCH) and file(usePCH) to keep precompiled header files up to date and to use them whenever possible. If you specify different file names with the two options, the name specified last is used for both options.

This option is implicitly set to no when any of the following options are in effect:

"expAllInc C List Option" on page 149
WIN OS/2 list(expUserInc)
"preprocessOnly C Preprocessor Option" on page 143

You can also use #pragma hdrfile in your source, to request precompiled headers. The file name in the pragma overrides the file name specified with this option.

RELATED CONCEPTS

C Compilation
Configuration Files
Precompiled C Headers
Organizing Your C Source for Precompiled Headers

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"C File Options" on page 155

Chapter 11. Related References

Equivalent Batch Compile-Link and Incremental Build Options (AIX)

VisualAge C++ includes an incremental C++ compiler, with a C compiler extension, and traditional C and C++ compilers. For both C and C++, you can control the compile and link stages of a build by specifying options in a configuration file or in a traditional makefile.

The syntax of configuration file options indicates whether the option relates to code generation, optimization, the C or C++ language, the link stage, or other features of builds.

The tables below give the configuration file option equivalents to compile options of the traditional batch compiler. In an incremental C++ compile, not all the batch options are meaningful. Some are automatic, and the functionality of others is provided in a different way. An entry of N/A indicates that an equivalent incremental option is not available. Whether the incremental option applies only to C or only to C++ code in this version of VisualAge C++ is indicated in the description line.

In the option syntax, the following rules apply:

- The characters [and] enclose optional items.
- The characters (and) enclose a list of alternatives.
- The character | indicates alternatives.
- The character * denotes zero or more items.

For example -qkeyword=string[:string]* is equivalent to -qkeyword=string, -qkeyword=string1:string2, -qkeyword=string1:string2:string3 and so on. Another example: -qtbtable=(none|small|full) is equivalent to -qtbtable=none, or -qtbtable=small, or -qtbtable=full.

Batch Option	**Incremental Option**
-#	N/A
Display language commands but do not invoke them; output goes to stdout.	
-+	Use the c++ source type parameter on a source directive.
Treat all source files as C++ source code.	
-B*prefix*	N/A
Construct alternate program names.. The -t option determines which programs have alternate names.	

-bbigtoc
-bnobigtoc
	link(bigTOC, yes)
	link(bigTOC, no)

Generates extra code to allow the table of contents (TOC) to grow to a size greater than 64KB.	
-bD:*number* -bmaxdata:*number*	link(heap, *number*)
Set the size of the heap in bytes. The first number specifies the total virtual address space to reserve. The second number the amount of physical memory to commit initially.	

Batch Option	**Incremental Option**
-bdynamic -bstatic	N/A
Determines which types of library files are searched by the linkage editor and whether a library is linked dynamically or statically.	
-bI:*filename*	supported as a source in the configuration: source type(imp) “*filename*” or source “*filename*.imp”
Names a file containing a list of imported symbols.	
-blibpath:*string*	link(libPathOut, *string*)
Override search path used at run time.	
-bnoentry	link(entry,no)
Specify that the address of the symbol is the start address of the executable module being built.	

-bnso or -bnautoimp
-bso or -bautoimp
	link(static, yes)
	link(static, no)

Specifies if unstripped, shared objects are statically linked as ordinary object files.	
-brtl	link(runtimeLinking)
Tells the linkage editor to search for both .so and .a library file types and to resolve undefined symbols at run time.	
-c	Specify a target with type object file
Do not send object files to the linkage editor.	

-C	▶ C
	pp(preprocessOnly)
	▶ C
	pp(preserveComments)

Preserve comments in preprocessed output.	
-D*name*[=[*def*]]	define(*name*[, *def*])
Define preprocessor macro *name* as in #define directive. If *def* is not specified, 1 is assumed.	
-Wl,-D*number*,-bpD:number	link(dataStart, *number*)
Set the starting address for the data section of the output file to *number*.	
-E	▶ C pp(preprocessOnly) ▶ C pp(stdout)
Preprocess but do not compile. Output goes to stdout.	
-e*name*	link(entry, *string*)
Specify that the address of the symbol is the start address of the executable module being built.	
-F*x*[:*stanza*]	supported as a vacbld command option stanzas also supported by defaults(*stanza*)
Use alternate configuration file or stanza, or both.	

Batch Option	Incremental Option

-g	link(debug[, yes])
	link(debug, no)

Include traditional debug information in the target.	
-H*number*	link(alignAddr, *number*)
Set the address alignment for all loadable segments so that the start of each segment is aligned at a multiple of the specified number of bytes.	
-I*dir*	incl(searchpath, *path*)
Search in directory *dir* for include files that do not start with an absolute path.	
-K -z	link(alignPage, yes)
Aligns the header, text, data, and loader sections of the output file so that each section begins on a page boundary.	
-l*key*	N/A
Search the specified library file, where *key* selects the file lib*key*.a, or lib*key*.so if the -rtl option is used.	
-L*string*	link(libSearchPath,*string*)
Specify search path for library files.	
-ma	N/A
Generate inline calls to the "alloca" function as if "#pragma alloca" directives were in the source file.	
-M	▶ C file(makeDep[,yes])
Generate information to be included in a "make" description file; output goes to .u file.	
-o*name*	▶ C file(genObject[,yes]) ▶ C file(genObject,*name*) ▶ C file(genObject,no) filename specified on target directive
Name generated executable or object file.	
-O -O2 -O3	opt(level, *number*)
Optimize code.	
▶ C -O4 ▶ C -O5	▶ C opt(level, `number`)
Level 4 is equivalent to -O3 -qipa with automatic detection of architecture and tuning option. At level 5, interprocedural calls are also optimized.	
-p	gen(profile[, yes])
Enable code for performance analysis. Enables profiling with **prof**.	
-pg	gen(profile, bsd)
Generate profiling support code including BSD profiling support. Enables profiling with **gprof**.	
-P	▶ C pp(preprocessOnly[,yes])

Batch Option	Incremental Option
Preprocess but do not compile. Output goes to .i file.	

-Q
-qinline
	opt(inline[, yes])
	opt(autoInline[, yes])

Consider any function for inlining. The default is to inline only those functions declared inline.	

-Q!
-qnoinline
	opt(inline, no)
	opt(autoInline, no)

Do not inline any function.	

C -Q=*lc*
C -qinline=*lc*
	opt(inline)
	opt(autoInline)
	opt(autoInline, `lc`)

Inline if number of source statements in function is less than the number specified in *lc*.	

C -Q-*name*[:*name*]*
C -qinline-*name*[:*name*]*
	opt(inline[, yes])
	opt(noInlineFunc,name)

Do not inline function listed by names.	

C -Q+*name*[:*name*]*
C -qinline+*name*[:*name*]*
	opt(inline[, yes])
	opt(inlineFunc,name)

Attempt to inline funtions listed by names.	
-qaggrcopy=[no]overlap	N/A
Destructive aggregate copy.	
-qalias=(typ | allp | addr | ansi)	opt(alias,...)
Specifies the aliasing assertion to be applied to your compilation unit. The available options are: typ= Pointers to different types are never aliased. allp= Pointers are never aliased. addr= Variables are disjoint from pointers unless their address is taken. ansi= Pointers can only point to an object of the same type. Requires -O option.	
-qalign=*name*	gen(align, *name*)
Specify alignment of data items.	
-qansialias -qnoansialias	opt(alias, ...)
Specify which aliasing rules can be used during optimization.	
-qarch=*name*	gen(arch, *name*)
Specifies the architecture on which the executable program will be run. See the AIX processor types page for a list of valid names. The default is -qarch=com. If the -qarch option is specified without the -qtune=<option>, the compiler uses -qtune=pwr.	

C++ -qassert=*option*
C -qassert=(typ | allp | addr)
	opt(alias, *option*)
	opt(aliasAssert,typ[,yes])
	opt(aliasAssert,typ,no)
	opt(aliasAssert,allp[,yes])
	opt(aliasAssert,allp,no)
	opt(aliasAssert,addr[,yes])
	opt(aliasAssert,addr,no)

Batch Option	Incremental Option
Specifies the aliasing assertion to be applied to your compilaton unit. The available options are: typ= Pointers to different types are never aliased. allp= Pointers are never aliased. addr= Variables are disjoint from pointers unless their address is taken.	
-qattr	▶ C list(attr[,yes])
Produce an attribute listing; output goes to.lst file.	
-qattr=full	▶ C list(attr, full)
Produce an attribute listing containing all names, whether referenced or not; output goes to .lst file.	

-qbitfields=signed
-qbitfields=unsigned
	lang(signedBitfields[,yes])
	lang(signedBitfields,no)

Specify whether bitfields will be signed or unsigned.	
-qcache	
Specifies the hardware cache characteristics.	

-qchars=signed
-qchars=unsigned
	lang(signedChars[, yes])
	lang(signedChars, no)

Treat plain *char* variables as signed or unsigned.	

-qcheck
-qnocheck
-qcheck=*suboptions*
	gen(check, zeroDivide[, yes])
	gen(check, zeroDivide, no)
	gen(check, nullPointer[,yes])
	gen(check, nullPointer,no)
	gen(check, bounds[,yes])
	gen(check, bounds,no)

The suboptions are: all= Switches on all of the following suboptions. nullptr= Performs run-time checking of addresses contained in pointer variables used to reference storage. bounds= Performs run-time checking of addresses when subscipting within an object of known size. divzero= Performs run-time checking of integer division.	

▶ C++ -qcinc=*prefix*
▶ C++ -qnocinc=*prefix*
	incl(externC, *prefix*)
	incl(noExternC, *prefix*)

Includes files from the subdirectory *prefix* and inserts `extern "C" {` at the beginning of the file and } at the end.	

-qcompact
-qnocompact
	opt(size[, yes])
	opt(size, no)

Optimize code for size.	
▶ C -qcpluscmt ▶ C -qnocpluscmt	▶ C lang(slashSlashComment[,yes]) ▶ C lang(slashSlashComment,no)
Permit "//" to introduce comment that lasts until the end of the current source line, as in C++.	

Batch Option	Incremental Option

-qdataimported
-qdataimported=*name*[:*name*]*
	link(dataImported, none)
	link(dataImported, all)
	link(dataImported, names)
	link(dataImportedNames, *name*)

Specify which data items are imported. If no names are specified, all data items are assumed to be imported. This is the default.	

-qdatalocal
-qdatalocal=*name*[:*name*]*
	link(dataLocal, none)
	link(dataLocal, all)
	link(dataLocal, names)
	link(dataLocalNames, *name*)

Specify which data items are local. If no names are specified, all data items are assumed to be local.	

-qdbcs
-qnodbcs
	lang(dbcs[,yes])
	lang(dbcs,no)

Allow use of DBCS.	

C -qdbxextra
C -qnodbxextra
	gen(debugunreferenced[, yes])
	gen(debugunreferenced,no)

Produce a symbol table for unreferenced variables.	

-qdigraph
-qnodigraph
	lang(digraphs[, yes])
	lang(digraphs, no)

Permit ANSI digraph and keyword operators.	

-qdollar
-qnodollar
	lang(allowDollarInNames[, yes])
	lang(allowDollarInNames, no)

Allow the dollar sign ('$') in identifier names.	

-qdpcl
-qnodpcl
	link(dpcl[, yes])
	link(dpcl, no)

C++ -qeh C++ -qnoeh	C++ gen(eh[, yes]) C++ gen(eh, no)
Controls whether C++ exception handling is enabled in the module being built.	
-qenum=(small|int|intlong|1|2|4|8)	gen(enumSize, ...)
Small denotes that either one, two or four bytes of storage will be allocated for enum variables based on the range of the enum constants. int is the default, and causes enum variables to be treated as though they were of type signed int for C and either signed or unsigned int for C++ depending on the range of the enum constants. intlong causes enum variables to be treated as though they were of type int or type long depending on the range of enum constants. The sign of the type is determined in the same way as for int. 1 will cause enum variables to be packed into 1 byte. 2 will cause enum variables to be packed into 2 bytes. 4 will cause enum variables to be packed into 4 bytes. 8 will cause enum variables to be packed into 8 bytes.	

Batch Option	Incremental Option

-qextchk
-qnoextchk
	link(typeCheck[, yes])
	link(typeCheck, no)

Perform external name type-checking and function call checking.	
-qfdpr -qnofdpr	gen(fdpr, yes)
Collect information about programs for use with the AIX fdpr (Feedback Directed Program Restructuring) performance-tuning utility.	

-qflag=*sev1[:sev2]*	list(reportLevel, sev1)
	report(level, sev2)

Specify severity level of diagnostics to be reported in source listing *sev1*, and stderr *sev2*; the severity levels are one of i,w,e,s.	

-qfloat=*opt*[:*opt*]*	gen(float, ...)
	gen(float, *string*[,yes])
	gen(float, *string*,no)

Specifies various floating-point options. These options provide different strategies for speeding up or improving the accuracy of floating-point calculations. They affect the generated code, but not operations performed in library functions.	

-qflttrap=*opt*[:*opt*]*
-qnoflttrap
	gen(floatTrap, *option*[, yes])
	gen(floatTrap,*option*, no)

Generate calls to detect and trap floating point exceptions. The available options are: overflow, underflow, zerodivide, invalid, inexact, enable, imprecise.	

-qfold
-qnofold
	gen(float, fold[, yes])
	gen(float, fold, no)

Specifies that constant floating-point expressions are to be evaluated at compile time.	

-qfullpath
-qnofullpath
	link(debugFullPath[, yes])
	link(debugFullPath, no)

Specify path information.	

-qfuncsect
-qnofuncsect
	gen(funcSect[, yes])
	gen(funcSect, no)

Place instructions for each function in a separate csect.	
C -qgenpcomp[=filename]	N/A
Generate a precompiled header file; *name* can either be a directory containing the precompiled header (named csetc.pch) or the name of the precompiled header.	
C -qgenproto C -qnogenproto	C file(genProto[, yes]) C file(genProto, no)
Generate ANSI prototypes from K&R function definitions.	
C -qgenproto=parmnames	C file(genProto, withParameterNames)
Produces ANSI prototypes from K&R function definitions with the names of parameters included.	
-qhalt=(i|w|e|s)	N/A
Stop the compiler after the first phase if the severity of errors detected equals or exceeds <sev>; the severity level is one of i, w, e, s, which stand for information, warning, error, and severe.	
-qhaltonmsg=*message*[:*message*]*	report(treatAsError, *message*)

Batch Option	Incremental Option
Treats the specified message like an error message (preventing the creation of the target).	

-qheapdebug
-qnoheapdebug
	alloc(debug[,yes])
	alloc(debug,no)

Enables debug versions of memory management functions.	

-qidirfirst
-qnoidirfirst
	incl(searchpathfirst[, yes])
	incl(searchpathfirst, no)

Specify search order for files.	

-qignerrno
-qnoignerrno
	opt(ignErrno)
	opt(ignErrno, no)

Tell the optimizer that the program will never refer to or set errno.	

-qignprag=disjoint
-qignprag=isolated
-qignprag=all
	opt(pragmaDisjoint[, yes])
	opt(pragmaDisjoint, no)
	opt(pragmaIsolatedCall[, yes])
	opt(pragmaIsolatedCall, no)

Honor or ignore references to #pragma disjoint and/or isolated_call.	
-qinfo=*class*	info(*class*)
Produce additional lint-like messages based on *class*.	

-qinitauto=*number*	gen(initAuto[, yes])
	gen(initAuto, *number*)
	gen(initAuto, no)

Initialize automatic storage to *hh*, where *hh* is a hexadecimal value. A number is a hexadecimal number if it begins with 0x, an octal numbner if it begins with 0 or a decimal number if it begins with 1-9. This option generates extra code and should only be used for error determination	

-qinlglue
-qnoinlglue
	gen(inlinePointerGlue[, yes])
	gen(inlinePointerGlue, no)

Generate fast external linkage by inlining the code (pointer glue code) necessary for calls via a function pointer and calls to external procedures.	

-qinline
-qnoinline
	opt(autoInline[, yes])
	opt(autoInline, no)
	opt(inline[, yes])
	opt(inline, no)

Determine whether functions in your code are candidates for inlining. See the various -Q options.	
▶ C -qipa ▶ C -qnoipa ▶ C -qipa=*option*	N/A
Turn the IPA optimizer on or off.	
-qisolated_call=*name*[:*name*]*	opt(isolatedCall,...)
Specify that the calls to the function *name* listed have no side effects.	
▶ C++ -qkeyword=*string*[:*string*]* ▶ C++ -qnokeyword=*string*[:*string*]*	lang(keyword, *string*)
Controls whether the specified *string* is treated as a keyword or an identifier whenever it appears in your C++ source.	

Batch Option		Incremental Option

-qlanglvl	C++ =[no]anonstruct
	C++ =[no]anonunion
	C++ =[no]ansifor
	C++ =[no]illptom
	C++ =[no]implicitint
	C++ =[no]newexcp
	C++ =[no]offsetnonprod
	C++ =[no]olddigraphs
	C++ =[no]oldfriend
	C++ =[no]oldmath
	C++ =[no]oldtmplacc
	C++ =[no]oldtmplalign
	C++ =[no]oldtmplspec
	C++ =[no]trailenum
	C++ =[no]typedefclass
	=[no]ucs
	C++ =[no]zeroextarray
		lang(anonymousStructs[, yes])
		lang(extendedAnonymousUnions[, yes])
		lang(ansiForStatementScopes[, yes])
		lang(illformedPointerToMember[, yes])
		lang(implicitInt[, yes])
		lang(newThrowsException[,yes])
		lang(offsetOfNonPODClasses[, yes])
		lang(oldDigraphs[, yes])
		lang(compatFriendDeclarations[, yes])
		lang(compatMath[, yes])
		lang(compatTempAccessChecking[, yes])
		gen(compatNestedTemplateAlignmentRule)
		lang(compatTemplateSpecialization[, yes])
		lang(trailingEnumCommas[, yes])
		lang(allowTypedefAsClassName[, yes])
		lang(universalCharacterNames[, yes])
		lang(zeroExternArrays[, yes])

Set the language level option.		
C++ -qlanglvl=<*compat366* | *strict98* | *ansi* | *extended*>		options for compatibility with older C++ compilers options for compatility with the C++ standard
Specify the language level to be used during compilation.		
C -qlanglvl=<*ansi* | *saa12* | *saa* | *extended* | *classic*>		lang(level,...)
Specify language level to be used during compilation.		

-qldbl128
-qnoldbl128
-qlongdouble
-qnolongdouble
		gen(longDouble[, yes])
		gen(longDouble, no)

Represent long doubles as 128 bit or 64 bit values.		

-qlibansi
-qnolibansi
		gen(libansi)
		gen(libansi, no)

Process functions with names that match ANSI C library names as being the ANSI C functions. This allows the compiler to make certain assumptions about the behavior of these functions.		

-qlinedebug
-qnolinedebug
		gen(lineOnlyDebug[,yes])
		gen(lineOnlyDebug,no)

Generates abbreviated line number and source file name information for the debugger.		
-qlist -qnolist		C list(listing[,yes]) C list(listing,no)
Produce an object listing; output goes to .lst file.		

Batch Option	Incremental Option
-qlistopt -qnolistopt	▶C list(options[, yes]) ▶C list(options, no)
Display the settings of all options; output goes to .lst file.	

-qlonglong
-qnolonglong
	lang(longlong)
	lang(longlong, no)

Build process disallows long long int.	
-qlonglit -qnolonglit	N/A
Makes unsuffixed literals into the long type in 64-bit mode.	
▶C -qmacpstr ▶C -qnomacpstr	▶C lang(macPStr[, yes]) ▶C lang(macPStr, no)
Allow '\p' as the first character of a string constant.	
-qmakedep -qnomakedep	▶C file(makeDep[,yes]) ▶C file(makeDep,no)
Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.	
-qmaxerr=*number*[:*severity_level*]	▶C debug(maxErrors, *number*)
Instructs the compiler to halt compilation when a specified number of errors is reached.	
-qmaxmem=*num*	opt(maxMem, *number*)
Limit the amount of memory used by space intensive ooptimization to *num*. *num* is specified in kilobytes.	

-qmbcs
-qnombcs
	lang(dbcs[, yes])
	lang(dbcs, no)

Allow use of DBCS.	

-qnamemangling=*option*	link(nameMangling, ansi)
	link(nameMangling, compat)

Choses the name mangling scheme for external symbol names generated from C++ source code.	
-qnoprint	▶C list(listing, no)
Direct listing to /dev/null.	
▶C -qonce ▶C -qnoonce	▶C file(once[, yes]) ▶C file(once, no)
Process #include files only once.	
▶C++ -qobjmodel=(ibm | compat)	▶C++ gen(objectModel, *model*)
Select the default C++ object model for the compilation unit.	
▶C -qpascal ▶C -nopascal	▶C lang(pascal[, yes]) ▶C lang(pascal, no)

Batch Option	**Incremental Option**
Accept and ignore the keyword ’pascal’ as a type modifier.	
-qpath=*components*:*path*	N/A

Construct alternate program names. The programs in the directory *path* and specified by *components* are used instead of the regular ones. *components* can a combination of any of the following:
p= preprocessor
c= compiler frontend
b= compiler backend
i= compiler inliner
a= assembler
l= linkage editor
m= ’munch’ utility
E= create export list
I= inter procedural analysis	

-qpdf1 -qpdf2	opt(profileDirectedFeedback[, yes])
Perform aggressive optimizations with profile directed feedback.	
-qphsinfo	N/A
Produce compiler phase information	
C++ -qpriority=*num*	C++ link(sharedLibPriority, *number*)
Specifies the priority level for initialization of static constructors.	

-qproclocal
-qproclocal=*name*[:*name*]*
	link(procLocal,<*none* | *all* | *names*>)
	link(procLocalNames, *string*)

Specify which functions are local. If no filenames are specified, all invoked functions are assumed to be defined within the current file. The last explicit specification for a function takes precedence.	

-qprocimported
-qprocimported=*name*[:*name*]*
	link(procImported,<*nome* | *all* | *names*>)
	link(procImportedNames,*string*)

Specify which functions are imported. If no filenames are specified, all invoked functions are assumed to be defined outside the current file. The last explicit specification for a function takes precedence.	

-qprocunknown
-qprocunknown=*name*[:*name*]*
	link(procUnknown,<*none* | *all* | *names*>)
	link(procUnknownNames, *string*)

Specify which functions are unknown to be local or imported. If no filenames are specified, all functions called are assumed to be unknown. This is the default when no user options are specified. The last explicit specification for a function takes precedence.	
-qprofile=ibm	gen(profile, ibm)
AIX Enables profiling with IBM’s Performance Analyzer.	

C -qproto
C -qnoproto
	C
	lang(checkNonProto[,yes])
	C
	lang(checkNonProto,no)

Assert that procedure call points agree with their declarations even if the procedure has not been prototyped. This allows the caller to pass floating point arguments in floating point registers instead of in general purpose registers.	

Batch Option	Incremental Option

-qro
-qnoro
	gen(readonly[, yes])
	gen(readonly, no)

Put string literals in read only area.	

-qroconst
-qnoroconst
	gen(readonlyconst)
	gen(readonlyconst, no)

Put constant values in read only area.	
C++ -qrtti C++ -qnortti	C++ gen(rtti[, yes]) C++ gen(rtti, all) C++ gen(rtti, typeInfo) C++ gen(rtti, dynamicCast) C++ gen(rtti, no)
Generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.	
-qshowinc -qnoshowinc	C list(expAllInc[,yes]) C list(expAllInc,no)
Include the source for all included files in the source listing, if the -qsource option is in effect.	

-qsource
-qnosource
	list(incSource[,yes])
	list(incSource,no)

Produce a source listing; output goes to.lst file.	
-qspill=*number*	opt(registerSpillSize, *number*)
Specify the size of the register allocation spill area.	
C -qsrcmsg C -qnosrcmsg	C report(srcMsg[, yes]) C report(srcMsg, no)
Reconstruct source lines in error along with the diagnostic messages.	

-qstaticinline
-qnostaticinline
	lang(staticInlineLinkage[, yes])
	lang(staticInlineLinkage, no)

Controls whether inline functions are treated as static or extern.	

-qstatsym
-qnostatsym
	link(staticSymbols)
	link(staticSymbols, no)

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of xcoff objects).	

-qstdinc
-qnostdinc
	incl(standardInclude[,yes])
	incl(standardInclude, no)

If -qnostdinc is specified, the /usr/vacpp/include and /usr/include directories are not searched for include files.	

-qstrict
-qnostrict
	opt(strict)
	opt(strict, no)

Batch Option	Incremental Option
Valid only at -03. This option turns off aggresive optimizations which have the potential to alter the semantics of a user's program. This option also sets -qfloat=nofltint:norsqrt.	

-qstrict_induction
-qnostrict_induction
	gen(strictinduction[, yes])
	gen(strictinduction, no)

Disable loop induction variable optimizations that have the potential to alter the semantics of the program.	

C -qsymtab=unref
-qsymtab=static
	gen(debugUnreferenced[,yes])
	gen(debugUnreferenced,no)
	link(staticSymbols)
	link(staticSymbols,no)

Set certain back end options.	
C -qsyntaxonly C -qnosyntaxonly	C file(syntaxOnly[,yes]) C file(syntaxOnly,no)
Causes the compiler to perform syntax checking without generating an object file.	
-qtabsize=*size*	file(tabSize, *number*)
Change the length of tabs in your source file.	
-qtbtable(non | small | full)	link(traceBackTable,<*none* | *full* | *small*>)
Generate a traceback table for each function and place it in the text segment at the end of the function code. The available options are: none= No traceback table is generated. full= A full traceback table is generated. This is the default with -g specified. small= A traceback table is generated with no name or parameter information. This is the default when -g is NOT specified.	
C++ -qtempinc[=*directory*] C++ -qnotempinc	N/A
Automatically generate template functions in the specified directory. Use the "notempinc" option to suppress this behavior.	
C++ -qtempmax=*number*	N/A
Specify the maximum number of files to be created in the tempinc directory for each template file.	
-qthreaded -qnothreaded	N/A
Make sure only thread safe optimizations thread safe optimizations are performed.	

-qtmplparse= no
-qtmplparse=
-qtmplparse=
	lang(templateDefImpls, parseWithWarnings)
	lang(templateDefImpls, parseWithErrors)
	lang(templateDefImpls, dontParse)

Controls whether parsing and semantic checking are applied to template definition implementations or only to template instantiations.	

-qtune=*name*	opt(tune, *name*)
	opt(tune,...)

Specifies the system architecture for which the executable program is optimized.	

Batch Option	Incremental Option
C++ -qtwolink C++ -qnotwolink	N/A
Link twice in order to minimize the number of static constructors included from libraries.	
C++ -qunique C++ -qnounique	C++ link(uniqueNames[, yes]) C++ link(uniqueNames, no)
Generates unique names for static constructor or destructor compilation units.	

-qunroll
-qunroll=*number*
-qnounroll
	opt(loopUnroll[, yes])
	opt(loopUnroll, *number*)
	opt(loopUnroll, no)

Allow the optimizer to unroll loops.	
C -qupconv C -qnoupconv	C lang(preserveUnsignedPromotion[, yes]) C lang(preserveUnsignedPromotion, no)
Preserves the unsigned specification when performing integral promotions.	
C -qusepcomp[=*name*]	N/A
Use precompiled header files. *name* can either be a directory containing then precompiled header (named csetc.pch) or the name of the precompiled header.	

-qvftable
-qnovftable
	gen(vft[, yes])
	gen(vft, no)

Determine whether the virtual function table is included in the module that is the target of the build.	

-qxcall
-qnoxcall
	gen(externStaticLinkage[, yes])
	gen(externStaticLinkage, no)

Generate code to static routines within a compilation unit as if they were external routines.	
-qxref -noxref	C list(minXRef[, yes]) C list(minXRef, no)
Produce a cross-reference listing; output goes to .lst file.	
-qxref=full	C list(fullXRef[, yes]) C list(fullXRef, no)
Produce a cross-reference listing containing all names, whether referenced or not; output goes to .lst file.	
-r	link(force)
Permits the output file to be produced even if it contains unresolved symbols.	

-s	link(strip[, yes])
	link(strip, no)

Strip the symbol table to save space.	

Batch Option	Incremental Option
-S	C file(genAsm[,yes]) C file(genAsm, no)
Generate an assembler language (.s) file	
-Wl, -S*number*	link(stack,*number*)link(stack, *number*)
Set the total size of the program stack in bytes.	
-t(p|c|b|i|a|l|m|E|I)	N/A

Apply the prefix from the -B option to the specified program, where the program can be one or more of the following:
p= preprocessor
c= compiler frontend
b= compiler backend
i= compiler inliner
a= assembler
l= linkage editor
m= 'munch' utility
E= create export list
I= inter procedural analysis	

-T*number*	link(textStart, *number*)
Sets the start address of the text section of the output file to *number*.	
-U*name*	undefine(*name*)
Undefine name as in #undef directive.	
-v	N/A
Display language processing commands as they are invoked by xlC; output goes to stdout.	
-V	N/A
Same as -v except format is similar to shell commands.	
-w	report(level, ...)
Suppress information, language-level, and warning messages.	
C++ -W	N/A
Give specified options to specified compiler program.	
-W*components*, *option*	N/A
Give the *option* to the compiler program where *components* can be one or more of the following: p,c,b,i,a,l,I,m,E. See -t for *component* meanings.	
-y*mode*	gen(roundConstFp, *mode*)
Specify compile-time rounding of constant floating-point expressions, where *mode* can be one of the following: n= round to nearest m= round toward minus infinity p= round toward positive infinity z= round toward zero.	
-Z*string*	link(libsearchpathprefix, *string*)
Prefix the names of the library search paths with the *string*.	

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"debugUnreferenced Code Generation Option" on page 66
"Build Options" on page 1
Batch Compiler Options

C++ "Build Options for Compliance With the C++ Standard"

C++ "Build Options for Compatibility with Old C++ Compilers" on page 182

Build Options for Compliance With the C++ Standard

By default, VisualAge C++ provides several extensions to the C++ standard. To make sure your code complies strictly with the standard in all ways other than the order of declarations, build with the options listed below.

The following are default settings. To conform to standard C++, you do not have to explicitly set them.

- "macros redefine Miscellaneous Option (C++)" on page 53
- "allowDollarInNames Language Option" on page 4
- "allowTypedefAsClassName Language Option (C++)" on page 5
- "ansiForStatementScopes Language Option (C++)" on page 7
- "compatMath Language Option (C++)" on page 8
- OS/2 WIN link(nameMangling, *ansi*, *size*)

The following options do not conform to standard C++ by default. Instead, they support extensions to the standard or take advantage of features of VisualAge C++ that usually are desirable. Apply the following settings to C++ source files to make sure they comply with the C++ standard.

- "longDouble Code Generation Option (AIX)" on page 79
- "rtti Code Generation Option (C++)" on page 85
- OS/2 WIN gen(winmain, no)
- "anonymousStructs Language Option (C++)" on page 6
- "compatFriendDeclarations Language Option (C++)" on page 7
- "compatTempAccessChecking Language Option (C++)" on page 9
- "compatTemplateSpecialization Language Option (C++)" on page 10
- "extendedAnonymousUnions Language Option (C++)" on page 12
- "illformedPointerToMember Language Option (C++)" on page 12
- "implicitInt Language Option (C++)" on page 13
- "longLong Language Option" on page 33
- "offsetOfNonPODClasses Language Option (C++)" on page 19
- "orderedNameLookup Language Option (C++)" on page 20

- “trailingEnumCommas Language Option (C++)” on page 25
- “universalCharacterNames Language Option” on page 26
- “zeroExtentArrays Language Option (C++)” on page 26
- OS/2 WIN link(ansiExtensionsLibrary, no)
- OS/2 WIN opt(float, no)
- OS/2 WIN opt(integer, no)

Many of the option settings listed above override defaults that support the VisualAge C++ extensions to the standard. Other default values conform to the standard.

The default settings of three other options do not strictly conform to standard C++. However, the default setting is usually appropriate for incremental compile. The defaults facilitate treating header files as primary sources for builds. Eliminating included headers greatly increases the efficiency of C++ incremental builds. If you convert included header files to primary sources, make sure you apply these defaults to the converted sources:

- “staticInlineLinkage Language Option (C++)” on page 23
- “staticConstLinkage Language Option (C++)” on page 23
- “localTypedefs Language Option (C++)” on page 15

The following table shows which options should be used to enforce either the 1998 ANSI C++ standard or the IBM extended standard.

	strict98/ansi	**extended**
gen (winmain)	no	yes
gen (rtti)	yes	no
gen (compatNestedTemplateAlignmentRule)	no	no
lang (allowTypedefAsClassName)	no	no
lang (anonymousStructs)	no	yes
lang (ansiForStatementScopes)	yes	yes
lang (compatFriendDeclarations)	no	no
lang (compatMath)	no	no
lang (compatTempAccessChecking)	no	no
lang (compatTemplateSpecialization)	no	no
lang (extendedAnonymousUnions)	no	yes
lang (ilformedPointerToMember)	no	yes
lang (implicitInt)	no	yes
lang (longlong)	no	yes
lang (offsetOfNonPODClasses))	no	yes
lang (oldDigraphs)	yes	yes
lang (staticConstLinkage)	yes	yes
lang (templateDefImpl,)	parsewithwarnings	parsewithwarnings
lang (trailingEnumCommas)	no	yes
lang (zeroExtentArrays)	no	yes
link (ansiExtensionsLibrary)	no	yes

	strict98/ansi	extended
link (nameMangling)	ansi	ansi
macros (redefine)	no	no
lang (keyword,bool)	yes	yes
lang (keyword,explicitl)	yes	yes
lang (keyword,export)	yes	yes
lang (keyword,false)	yes	yes
lang (keyword,mutable)	yes	yes
lang (keyword,namespace)	yes	yes
lang (keyword,true)	yes	yes
lang (keyword,typenamel)	yes	yes
lang (keyword,using)	yes	yes

RELATED CONCEPTS

Incremental C++ Build
Configuration Files
C++ Source Files and Source Regions

RELATED TASKS

Set Build Options
Build
Convert Included Source Files to Primary Source Files

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165
"Build Options" on page 1

Build Options for Compatibility with Old C++ Compilers

VisualAge C++ supports the C++ standard. The standard introduces several important differences from coding C++ practices that were in common use before the C++ standard was finalized. As a result, code that was accepted by earlier versions of VisualAge C++ and predecessor products, and other compilers, may fail to compile due to syntax and semantic incompatibilities.

You may be able to use code that was written for earlier versions of VisualAge C++, if you set build options. These options let your code break some of the new standard rules and use earlier versions of iostream and other classes that have been redefined by the standard or belong to the Standard Template Library (STL).

The table below lists the options you should use to compile C++ coded to two different language levels recognized by previous versions of VisualAge C++ and predecessor products. The left column supports the effective industry standard that was based on early drafts of the ANSI C++ working paper.

OS/2

WIN Build with the option settings in the right column if your code compiled with the old option /Se on OS/2 and Windows NT platforms. The right column supports many extensions to that version of C++. Build with the option settings in the left column if your code compiled with the old option /Sa. If you did not explicitly set the language level for VisualAge C++ 3.0 or 3.5, CSet++ 2.0 or 2.1, and IBM C and C++ Compilers 3.6, use the right column because /Se was the default setting. You can omit options if your code does not use the language feature they relate to. For example, you do not need to set lang(nokeyword, "bool") unless your code uses the keyword bool as an identifier.

Previous Industry Standard C++	**Extended C++ Level**

OS/2
WIN gen(mangleLinkageKeywordOnlyFuncs,no)
"allowTypedefAsClassName Language Option (C++)" on page 5
"anonymousStructs Language Option (C++)" on page 6
"compatMath Language Option (C++)" on page 8
"longLong Language Option" on page 33
"noKeyword Language Option (C++)" on page 18
"noKeyword Language Option (C++)" on page 18
lang(noKeyword,"export")
lang(noKeyword,"false")
lang(noKeyword,"mutable")
lang(noKeyword,"namespace")
lang(noKeyword,"reinterpret_cast")
lang(noKeyword,"static_cast")
lang(noKeyword,"true")
lang(noKeyword,"typename")
lang(noKeyword,"using")
"staticConstLinkage Language Option (C++)" on page 23
"templateDefImpls Language Option (C++)" on page 24
"trailingEnumCommas Language Option (C++)" on page 25
"universalCharacterNames Language Option" on page 26
"zeroExtentArrays Language Option (C++)" on page 26
OS/2
WIN link(ansiExtensionsLibrary,no)
OS/2 WIN link(nameMangling, *compat*, *size*)
"macros redefine Miscellaneous Option (C++)" on page 53
	gen(mangleLinkageKeywordOnlyFuncs, no)
	"allowTypedefAsClassName Language Option (C++)" on page 5
	"ansiForStatementScopes Language Option (C++)" on page 7
	"compatMath Language Option (C++)" on page 8
	"noKeyword Language Option (C++)" on page 18
	lang(noKeyword,"explicit")
	lang(noKeyword,"export")
	lang(noKeyword,"false")
	lang(noKeyword,"mutable")
	lang(noKeyword,"namespace")
	lang(noKeyword,"reinterpret_cast")
	lang(noKeyword,"static_cast")
	lang(noKeyword,"true")
	lang(noKeyword,"typename")
	lang(noKeyword,"using")
	"oldDigraphs Language Option (C++)" on page 20
	"templateDefImpls Language Option (C++)" on page 24
	"universalCharacterNames Language Option" on page 26
	OS/2 WIN link(nameMangling, *compat*, *size*)
	"macros redefine Miscellaneous Option (C++)" on page 53

The following table shows which options should be used to gain the best compatibility with IBM C and C++ Compilers Version 3.6.6. These options are grouped by the batch compiler when the -qlanglvl suboption compat366.

	compat366
gen (winmain)	yes
gen (rtti)	no

	compat366
gen (compatNestedTemplateAlignmentRule)	yes
lang (allowTypedefAsClassName)	yes
lang (anonymousStructs)	no
lang (ansiForStatementScopes)	no
lang (compatFriendDeclarations)	yes
lang (compatMath)	yes
lang (compatTempAccessChecking)	yes
lang (compatTemplateSpecialization)	yes
lang (extendedAnonymousUnions)	yes
lang (ilformedPointerToMember)	yes
lang (implicitInt)	yes
lang (longlong)	yes
lang (offsetOfNonPODClasses))	yes
lang (oldDigraphs)	no
lang (staticConstLinkage)	yes
lang (templateDefImpl,)	dontparse
lang (trailingEnumCommas)	yes
lang (zeroExtentArrays)	no
link (ansiExtensionsLibrary)	no
link (nameMangling)	compat
macros (redefine)	no
lang (keyword,bool)	no
lang (keyword,explicitl)	no
lang (keyword,export)	no
lang (keyword,false)	no
lang (keyword,mutable)	no
lang (keyword,namespace)	no
lang (keyword,true)	no
lang (keyword,typenamel)	no
lang (keyword,using)	no

RELATED CONCEPTS

Incremental C++ Builds
Configuration Files

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 165
“Build Options” on page 1

"Build Options for Compliance With the C++ Standard" on page 180

AIX "Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

Information Message Groups

The compiler can output information messages. These messages tell you about poor programming style, potential coding problems, and the structure of your program. They are categorized into the groups listing in the table below. You can use info(*group*) and info(all) to control which groups of messages are produced or suppressed.

The default info groups for C++ are: por and ret.

The default info groups for C are: por, pro, and ret.

group	**Message Category**
`all`	All diagnostics except warn64.
C++ `cls`	Display information about class use.
C `cmp`	Possible redundancies in unsigned comparisons.
`cnd`	Possible redundancies or problems in conditional expressions.
C `cns`	Operations involving constants.
`cnv`	Conversions.
C `cpy`	Possible problems with generation of copy constructors and copy assignment operators
C `dcl`	Consistency of declarations.
`eff`	Statements with no effect.
`enu`	Consistency of `enum` variables.
C `ext`	Unused external definitions.
`gen`	General diagnostics.
`gnr`	Generation of temporary variables.
`got`	Usage of `goto` statements.
C `ini`	Possible problems with initialization.
C `lan`	Effects of the language level.
C `obs`	Features that are obsolete.
C `ord`	Unspecified order of evaluation.
`par`	Unused parameters.
`por`	Nonportable language constructs.
C `ppc`	Possible problems with using the preprocessor.
C `ppt`	Trace of preprocessor actions.
C `pro`	Missing function prototypes.
`rea`	Code that cannot be reached.

group	**Message Category**
`ret`	Consistency of `return` statements.
`trd`	Possible truncation or loss of data or precision.
C `tru`	Variable names truncated by the compiler.
C `uni`	Uninitialized variables.
`use`	Unused `auto` and `static` variables.
`warn64`	Enables checking for possible 32- to 64-bit migration problems.

RELATED CONCEPTS

C Compilation
Messages Produced by Incremental Builds
Messages Produced by Batch Builds

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

"Equivalent Batch Compile-Link and Incremental Build Options (AIX)" on page 165

AIX "Miscellaneous Build Options (AIX)" on page 41

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Getting Started book, or our PDF documents. Send your e-mail to: torrcf@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

http://www.ibm.com/support/ describes IBM Support Offerings on all platforms, worldwide.

http://www.ibm.com/rs6000/support/ describes support offerings on the RS/6000® platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

http://www.lotus.com/passport describes the IBM and Lotus® Passport Advantage™ contracting option.

The IBM Software Support Handbook, accessible from http://www.ibm.com/software/support, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- **United States**: 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- **Canada**: 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

- Elsewhere, please contact your local IBM office.

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred

- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere™ Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit http://www.ibm.com/software/ad/vaws-services/ or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com