Project : Programming Language C++
Reference Documents : X3J11.1/92-037
X3J11.1/92-008
Reply To : Bill Homer
homer@cray.com

To: Members of X3J11, WG14, X3J16, WG21, and X3H5 **X3J11.1/92-037**

From: X3J11.1

Topic: Comments on Aliasing Proposal by the Aliasing Subgroup

Date: May 11, 1992

The X3J11.1 committee is chartered to explore additional features that will make C a better language for numeric and scientific programming. We are not producing a standard, but rather a technical report that can be used by X3J11 and WG14 as guidance for a future standard, and by vendors that wish to keep pace with the state of the art in C language development. Finding an acceptable solution to the aliasing issue in C was identified by X3J11.1 as being the most important issue to address. Recently a letter ballot was sent out to X3J11.1 voting members to determine if they wanted to send a proposal titled, ''***Restricted Pointers in C***'', to a wider audience for additional comment. This ballot passed by vote of: 12 Yes, 3 Yes with comment, 1 No with comment, and 4 did not vote. We are sending this proposal to each of your committees because we would like to receive your input on the technical merits of this proposal. Please send your comments to the author, Bill Homer, whose name and Email address appear on the proposal. Response by Email is preferred. We need to receive your comments by **September 11, 1991.** All of your comments on this proposal will be collated, and the results will appear in future revisions of the proposal. There is an existing implementation by Cray Research, Inc. that supports this proposal. Please feel free to comment on all parts of the proposal. Specifically, we would appreciate receiving comments about the following three key points:

- The typing system of C has been enhanced to permit a declaration to specify that a pointer provides exclusive access to an object. Specifically, a pointer can be *restrict-qualified*. This indicates to the translator that it can assume that this pointer points to a unique object at the time the pointer is declared. Is this use of the type system an appropriate way to deal with aliasing issues?

- Even though the assumption is that a *restrict-qualified* pointer is pointing into a unique object at the point of declaration, this unique access can be relinquished by the programmer. For instance, an unrestricted pointer can be assigned the value of a restricted pointer, and now both pointers point into the same object. By contrast, two *restrict-qualified* pointers must point into different objects. Is the level of alias-free assertion adequate, too much, or too little? This proposal falls in between the *noalias* proposal explored by X3J11, and the current C Standard. The *noalias* proposal did not allow any aliasing of a *noalias-qualified* object, while the C Standard provides no mechanism for saying that two pointers to the same type point into different objects. Does this proposal provide the right level of alias-free semantics?

- A new keyword spelled **`restrict`** has been added. Is adding a new keyword acceptable? It follows from the formal definition of **`restrict`** that a correct program will not change its behavior if every occurrence of the keyword **`restrict`** is either deleted from the program or ignored by the translator. However, this does take one more name away from the user's name space.

All of the comments that accompanied the letter ballots are attached to the proposal. Most of the comments are related to the three points listed above. Proposed responses from the subgroup have been attached. The comments and the proposed responses may help you understand the issues. Thank you for your time and effort.

Restricted Pointers in C (revision 5)
X3J11.1 92-008

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
homer@cray.com or uunet!cray!homer

February 25, 1992

This document is a revision of the proposal in NCEG 91-054 (see also 91-024, 90-043, 90-029, 90-003 and 89-027).. The alternative phrasings of the formal definition of restrict presented in 91-054 have been replaced by a single phrasing. Relevant portions of 90-043 have been merged into this document (rather than being incorporated by reference, as they were in 91-054). For the sake of brevity, only key examples are incorporated here.

1 Rationale

1.1 Aliasing

For many compiler optimizations, ranging from simply holding a value in a register to the parallel execution of a loop, it is necessary to determine whether two distinct lvalues designate distinct objects. If the objects are not distinct, the lvalues are said to be *aliases*. It is aliasing through pointers that presents the greatest difficulty, because there is often not enough information available within a single function, or even within a single compilation unit, to determine whether two pointers can point to the same object. Even when enough information is available, this analysis can require substantial time and space. For example, it could require an analysis of a whole program to determine the possible values of a pointer that is a function parameter.

1.2 Library examples

Consider how potential aliasing enters into implementations in C of two Standard C library functions. There are no restrictions on the use of **memmove**, and the implementation shown below follows the model described in the Standard by copying through a temporary array. (Other approaches are possible, but this one is straightforward and strictly-conforming.) Since **memcpy** cannot be used for copying between overlapping arrays, its implementation can be a direct copy.

Figure 1 *Sample implementation of memmove.*

```
void *memmove(void *s1, const void *s2, size_t n) {
        char * t1 = s1;
        const char * t2 = s2;
        char * t3 = malloc(n);
        size_t i;
        for(i=0; i<n; i++) t3[i] = t2[i];
        for(i=0; i<n; i++) t1[i] = t3[i];
free(t3);
        return s1;
}
```

Figure 2 *Sample implementation of memcpy.*

```
void *memcpy(void *s1, const void *s2, size_t n);
        char * t1 = s1;
        const char * t2 = s2;
        while(n-- > 0) *t1++ = *t2++;
        return s1;
}
```

Note that the restriction on **memcpy** is expressed only in its *Description* in the Standard, and cannot be expressed directly in its implementation in C. While this does allow the source-level optimization of eliminating the temporary used in **memmove**, it does not provide for compiler optimization of the resulting single loop. In many architectures, it is faster to copy bytes in blocks, rather than one at a time. The implementation of **memmove** uses **malloc** to obtain the temporary array, and this guarantees that the temporary is disjoint from the source and target arrays. From this a compiler can deduce that block copies may safely be used for both loops. The implementation of **memcpy**, on the other hand, provides the compiler no basis for ruling out the possibility that, for example, **s1** and **s2** point to successive bytes. Therefore unconditional use of block copies does not appear to be safe, and the code generated for the single loop in **memcpy** may not be as fast as the code for each loop in **memmove**.

1.3 Overlapping objects

The restriction in the *Description* of **memcpy** which prohibits copying between overlapping objects does not make it clear whether these are "objects as declared" or "objects as referenced." For example, is the behavior of the following call of **memcpy** defined, because the arguments point into the disjoint array objects, a[1] and a[0]? Or is the behavior undefined because the arguments both point into the same array object, a?

Figure 3 *Memcpy between rows of a matrix.*

```
void f1(void) {
        extern char a[2][N];
        memcpy(a[1], a[0], N);  /* Defined? */
}
```

In the following variation, can each of the first and last half of array b be regarded as an object in its own right?

Figure 4 *Memcpy between halves of an array.*

```
void f2(void) {
        extern char b[2*N];
        memcpy(b+N, b, N);  /* Defined? */
}
```

These questions have been submitted in a Request for X3J11 Interpretation, document number 92-001, entitled "Questions about overlapping objects." For the sake of exposition, it is assumed in what follows that the calls of **memcpy** in Figure 3 and Figure 4 do give defined behavior.

1.4 Restricted pointers

If an aliasing restriction like the one for **memcpy** could be expressed in a function definition, then it would be available to a compiler to facilitate effective pointer alias analysis. The preceding discussion suggests the form that the restriction should take. It should be possible to specify in the declaration of a pointer that it provides "exclusive initial access" to the object to which it points, as though the pointer were initialized with a call to `malloc`. Because it is the unqualified versions of function parameter types that are compared for type compatibility, it is convenient to specify the restriction with a type qualifier, spelled `restrict`, on the pointer type. The following prototype for `memcpy` thus both expresses the desired restriction and is compatible with the current prototype. Note that although both pointer parameters in the new prototype for `memcpy` are restrict-qualified, it would also be sufficient to qualify only one of the two.

Figure 5 *Restricted pointer prototype for memcpy.*

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

1.5 Design choices

Given this general concept of restricted pointers, there remain a number of design choices. Most may be characterized as a choice between simplicity and expressive power. From one perspective, it seems better to simplify the analysis of restricted pointers by confining their use to the simplest paradigms. For example, restricted pointers would be quite useful even if they could be declared only as function parameters and if the restricted pointers themselves could not be modified. This would support what is clearly the most important paradigm, an alias-free function call interface in C, analogous to that in Fortran. From another perspective, it seems better to support the expression of aliasing restrictions in as many paradigms as feasible. This would be helpful in converting existing programs to use restricted pointers, and would allow more freedom of style in new programs.

The definition given below favors the view that supporting a variety of common paradigms is worth the risk of allowing usages too complicated for compilers to analyze. It allows restricted pointers to be modifiable, to be members of structures and elements of arrays, and to be "strongly scoped," in the sense that a restricted pointer declared in a nested block makes a non-aliasing assertion only within that block.

Note that the design choices in the following two sections are not central to the proposal. They are included for reasons given, but could also easily be omitted, with corresponding omissions from the formal definition.

1.6 Aliasing of unmodified objects

Comparison with Fortran suggests an additional issue concerning aliasing unmodified objects, as illustrated in Figure 6. Knowing that parameter a provides exclusive access to the array into which it points is sufficient to allow the the loop in **f3** to be vectorized or executed in parallel. Knowing that b and c are not used to access overlapping portions of a single array allows no additional optimizations.

Figure 6 *Aliasing matters only for modified objects.*

```
void f3(int n, float * restrict a, float *b, float *c) {
    int i;
    for ( i=0; i<n; i++ )
        a[i] = b[i] + c[i];
}
```

As a practical matter, some compilers may be better able to analyze functions in which all pointer parameters are restricted. This usage is better supported by avoiding restrictions that do not afford opportunities for optimization. Therefore the formal definition given below allows aliasing of unmodified objects through distinct restricted pointers. This is closely analogous to the restriction on Fortran dummy arguments, and thus is convenient for programmers who use both languages, perhaps even in the same program. It means that b and c can be restrict-qualified in the definition of f3 without prohibiting calls for which b and c are used to access overlapping portions of a single array.

1.7 Array-like syntax for parameters

By section 3.7.1 of the Standard, a declaration of a function parameter as "array of *type*" is adjusted to "pointer to *type*." As an extension of this syntax, if the keyword **restrict** appears either preceding or in place of the array size expression, and then the declaration is adjusted to "restrict-qualified pointer to *type*."

Figure 7 *Use of array size expression*

```
void f4(int n, float a[restrict n], float b[restrict n]);

void g4(float *p) {
        f4(100, p, p+1);   /* Inconsistent with prototype. */
        /* This call is an error if the array is modified. */
}
```

There are two advantages to this syntax. The first is for multi-dimensional arrays. The form `int (* restrict p)[100]` requires parentheses, but the equivalent form `int p[restrict][100]` does not. The latter form seems less awkward and less error-prone.

The second advantage is that the array size expression may be used by a compiler for bounds checking within a function definition, and to check a function call for arguments that overlap. For these purposes, the adjusted pointer is assumed to point to the initial (lowest addressed) element of the array that is referenced through the parameter. Figure 7 also uses a proposed variable length array syntax.

2 Formal definition of restrict

What follows are proposed additions to the American National Standard X3.159-1989. The relevant sections are noted in brackets in each header.

Keywords [3.1.1]

Add a keyword: **restrict**

Type Qualifiers [3.5.3]

Syntax

Add a type-qualifier: **restrict**

Constraints

Add a constraint:

Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

Semantics

Add the following text after line 22.

Let D be a declaration of an ordinary identifier that provides a means of designating an object P as a restrict-qualified pointer.

If D appears inside a block and does not have storage-class **extern**, let B denote the block. If D appears in the list of parameter declarations of a function definition, let B denote the associated block. Otherwise, let B denote the block of **main** (or the block of whatever function is called at program startup, in a freestanding environment).

During each execution of B, P must point into a single object, O, and if O is modified, then all references to values of O must be through pointer values derived from P.

Here an execution of B means that portion of the execution of the program during which storage is guaranteed to be reserved for an instance of an object that is associated with B and has automatic storage duration. A reference to a value means either an access to or a modification of the value. During an execution of B, attention is confined to those references that are actually evaluated (this excludes references that appear in unevaluated expressions, and also excludes references that are "available," in the sense of employing visible identifiers, but do not actually appear in the text of B). Finally, the object O is the array referenced through pointer values derived from P, as in the String Function Conventions in 4.11.1.

A translator is free to ignore any or all aliasing implications of uses of `restrict`.

Declarators [3.5.4]

Syntax

Add new direct-declarator:

direct-declarator `[ restrict` *constant-expression* $_{opt}$ `]`

Array Declarators [3.5.4.2]

Constraints

Add a constraint:

The type qualifier **restrict** may appear preceding or in place of a size expression only in a declaration of a function parameter of array type, and then only in the outermost array type derivation.

Function Definition [3.7.1]

Semantics

Modify lines 23-24 to read:

... A declaration of a parameter as "array of *type*" shall be adjusted to "pointer to *type*", or "restrict-qualified pointer to *type*" if the outermost derivation has the form `[restrict` $expr_{opt}$ `]`, ...

3 Examples

3.1 Function parameters

One of the most useful paradigms for restricted pointers is for pointer parameters of a function. In the function `f5` in Figure 8, it is possible for a compiler to infer that there is no aliasing of modified objects, and so to optimize the loop aggressively. Upon entry to `f5`, the restricted pointers `a` and `b` must provide exclusive access to their associated arrays. Within `f5`, the file scope pointer `c` must not point into the array associated with `a` because `c` is not assigned a pointer value derived from `a`.

Figure 8 *Restricted pointer function parameters.*

```
float x[100];
float *c;

void f5(int n, float * restrict a, float * restrict b) {
    int i;
    for ( i=0; i<n; i++ )
        a[i] = b[i] + c[i];
}
void g5(void) {
    float d[100], e[100];
    c = x; f5(100,   d,    e); /* Behavior defined.   */
           f5( 50,   d, d+50); /* Behavior defined.   */
           f5( 99, d+1,    d); /* Behavior undefined. */
    c = d; f5(100,   d,    e); /* Behavior undefined. */
           f5(100,   e,    d); /* Behavior defined.   */
}
```

Two of the calls shown in `g5` result in aliasing that is inconsistent with the restrict-qualifiers, and their behavior is undefined. Note that it is permitted for `c` to point into the array associated with `b`, if that array is not modified. Note also that, for these purposes, the "array" associated with a particular pointer means only that portion of an array object which is actually referenced through that pointer.

3.2 Block scope

A block scope restricted pointer makes an aliasing assertion that is limited to its block. This seems more natural than allowing the assertion to have function scope. It allows local assertions that apply only to key loops, for example. When inlining a function by converting it into a macro, it is safe to represent restricted pointer parameters by block scope restricted pointers, as illustrated in Figure 9.

Figure 9 *Inline version of f5.*

```
float x[100];
float *c;

#define f5(N, A, B)                                  \
{   int n = (N);                                     \
    float * restrict a = (A);                        \
    float * restrict b = (B);                        \
    int i;                                           \
    for ( i=0; i<n; i++ )                            \
        a[i] = b[i] + c[i];                          \
}
```

3.3 Extern restricted pointers

An extern restricted pointer is subject to a quite strong restriction. It should point into a single array for the duration of the program, and may not alias any other extern object name or restricted pointer. Such a pointer would typically be used to point to an array allocated with **malloc**.

Figure 10 *Extern restricted pointer does not alias extern array.*

```
float a[100];
float * restrict c; /* Overlapping portions of the
                       array a may not be referenced
                       through both c and a. */
```

3.4 Members of structs

A restricted pointer member of a struct makes an aliasing assertion, and the scope of that assertion is the scope of the ordinary identifier used to access the struct. Thus although the struct type is declared at file scope in Figure 11, the assertions made by the declarations of the parameters of f6 have block (of the function) scope.

Figure 11 *Restricted pointers as members of a struct.*

```
struct t {      /* Restricted pointers assert that    */
    int n;      /* members point to disjoint storage. */
    float * restrict p;
    float * restrict q;
};

void f6(struct t *r, struct t *s) {
    /* r->p, r->q, s->p, s->q should all point to    */
    /* disjoint storage during each execution of f6. */
    /* ... */
}
```

3.5 Ineffective usages

Although it is not a constraint violation, no assertion is made by restricting the return type of a function. The reason for this is that a function call expression f() is not an lvalue, and so there is no restricted pointer object involved. Thus the presence of the restrict qualifier in the declaration in Figure 12 has no effect.

Figure 12 *Function returning restricted pointer.*

```
float * restrict f7(void);  /* No assertion about aliasing. */
```

Although it is not a constraint violation, no assertion is made by a cast to a restricted pointer type. The reason for this is that a cast expression

```
(T * restrict)P
```

is not an lvalue, and so there is no restricted pointer object involved.

3.6 Constraint violations

It is a violation of the constraint in 3.5.3 to restrict-qualify an object type which is not a pointer type, or to restrict-qualify a pointer to a function.

Figure 13 *Restrict cannot qualify non-pointer object types.*

```
int restrict x;   /* Constraint violation. */
int restrict *p;  /* Constraint violation. */
```

Figure 14 *Restrict cannot qualify pointers to functions.*

```
float (* restrict f8)(void); /* Constraint violation. */
```

4 Comparison with noalias

Tom MacDonald provided the following overview of the differences between **restrict** and the previously proposed **noalias**.

The X3J11 committee attempted to solve the aliasing problem in C by introducing a new type qualifier **noalias**. That effort failed because of technical problems with the proposed semantics of **noalias**. This restricted pointer proposal is different in many ways.

- Only pointers can be declared to be restricted. In the **noalias** proposal, all objects were permitted to be declared noalias-qualified.
- It is the declaration of the restricted pointer that makes the aliasing assertions, while it was the noalias-qualified lvalue that made the aliasing assertions.
- A restricted pointer can be an alias with an unrestricted pointer, whereas a pointer to a noalias-qualified type was guaranteed to be completely alias free.
- The proposed semantics of **noalias** defined an alternate execution path in which virtual objects were created and later synchronized with the original object. No alternate execution path is defined for restricted pointers. The proposed semantics for restricted pointers merely permit the optimizer to statically analyze restricted pointers.
- A block scope restricted pointer only makes assertions on the containing scope. In the **noalias** proposal, a block scope noalias-qualified object made assertions that affected the entire containing function.

- A pointer to a noalias-qualified type made no aliasing assertion if it was a member of a structure (because identifiers designating members of structures were not "handles"). A restricted pointer member of a structure does make an aliasing assertion.

Comments re Restricted Pointers received with the letter ballots

> This prefix is used for responses from the subgroup.
> The comments themselves are indented.

From: Paul Kohlmiller -- CDC

There is nothing I like better than trying to improve the performance of a program. As an optimizer person I want to learn every trick in the book. The restricted pointer idea would certainly make life easier for people like me. However, I think the proposed solution is not the correct way to solve the problem.

First, I have a problem calling restrict a qualifier. It can appear in places that a qualifier cannot (like [restrict]). It can change the way a program can execute when highly optimized (although not for strictly conforming programs). It has additional constraints that other qualifiers do not share (i.e., it can only be used for pointers). This really makes restrict a new kind of type modifier and we are trying to force it to be a qualifier (not unlike making typedef a storage specifier).

> There is already one type qualifier, ''volatile,'' that is designed to
> influence optimization, and it seems more natural (and economical) to
> add another than to invent a whole new syntactic category. While it
> is unusual for a constraint to single out one instance of a category,
> it is not unprecedented. For example: ''The only storage-class
> specifier that shall occur in a parameter declaration is register.''

Second, the notion of adding a keyword to the language to improve optimization has a familiar ring to it. We already have one keyword, register, that has become a problem. Most optimizing compilers ignore it. It was created during a time when there were few compilers that did global register allocation. With modern compilers, the register keyword just doesn't add much information. The restrict keyword is doing the same thing. It is being created at a time when most compilers do not do full application optimization. These newer compilers will be able to make all the aliasing assertions that the restrict keyword can make. When they do, we will have another keyword that is essentially ignored.

Finally, when we get to the point that restrict is just as useful as register is today, we will still have all of the compiler overhead that we have for register. For example, for ANSI C we had to add code to make it illegal to take the address of a register variable. For restrict, even an implementation that has decided to ignore the optimization aspects of restricted pointers must still diagnose errors like using restrict on non-pointers.

> Since aliasing was identified as the most important issue at the first
> NCEG meeting, simply waiting for ''full application optimization'' to
> become generally available apparently did not seem realistic.

There are two other points that I would like to make. First, I know that this proposal has been very well crafted to solve a very real problem. I just think that compiler technology will catch up to the problem faster than programmers will modify their code. Second, there is another proposal on the table, the Ritchie-Prosser "fat pointer" proposal. This proposal solves a different problem that cannot be solved by the latest compiler technology (variable dimensioned arrays), does not introduce any new keywords and it doesn't create a class of implementations that ignore the feature except for constraint violation detection. Granted, it only partially solves the optimization problem (for formal parameters) but it solves the problem for a very large segment of the number of programs that might otherwise be modified to use the restrict keyword.

The time and effort in developing this proposal has been useful and enlightening. I feel that other solutions will be used in the near future. This particular solution would be useful for a while. I prefer a solution that can't be so readily obsoleted by new compiler technology.

Many thanks to Tom MacDonald for his efforts on this proposal.

From: David Prosser -- AT&T and USL

92-008 (aliasing) Yes w/comment: "It is inappropriate to use the type system to express aliasing information -- it is inherently an algorithmic property."

From: Jim Thomas -- Taligent

aliasing - Yes w/comment:

1.7 (top of p6) I don't see why there is a special issue for multi-dimensional arrays. Change ''p[restrict][100]'' to ''p[restrict 100]''. The construction of the first sentence of the second paragraph is not parallel.

> In the first paragraph referenced, p is intended to be a pointer to
> the first row of a two-dimensional array. The length of each row is
> specified to be 100, but the number of rows is not specified.
> Therefore, the suggested change is not appropriate, since it would
> change p into a pointer to the first element of an array of 100 ints.

2 (p7) Should there be a statement that behavior is undefined if a restrict assertion is not met? I don't understand the last sentence of the fourth paragraph: ''Finally, the object O ... in 4.11.1.'' Add ''of the Standard'' to the end of this sentence.

> The two instances of ''must'' in the third paragraph on page 7 should
> read ''shall,'' to indicate that if the requirements of that paragraph
> are not met, the behavior is undefined. There could also be an explicit
> statement to that effect, for emphasis.

===

3.6 (p11) After ''... the constraint in 3.5.3'' insert something like ''of the Standard modified as proposed''.

4 (p11, 4th bullet) Move ''only' to after ''... assertions''.

From: David Keaton -- Cray Computer Corp

Aliasing - Yes w/comment:

One of the stated intents was to interoperate efficiently with Fortran. Therefore, some effort should be expended to see that the rules coincide with Fortran rules so that equally efficient ''drop-in'' replacements for Fortran code can be written in C.

> The subgroup believes that C restricted pointer function parameters
> are compatible with Fortran dummy arguments in interlanguage calls
> in both directions. This should be stated more clearly in the
> proposal.

Accredited Standard Committee[1]
X3, INFORMATION PROCESSING SYSTEMS

Doc. No: X3J16/92-0056
WG21/N0133
Date: June 1, 1992
Project: Programming Language C++
Ref Doc: X3J16/91-0067
Reply To: Bill Gibbons
Taligent Inc.
10725 N. De Anza Blvd.
Cupertino, CA 95014
`bgibbons@taligent.com`

Proposal for Compile Time Constants in Class Scope

Bill Gibbons and David Goldsmith

This document is a revision of X3J16/91-0067 with additional examples, including problems caused by the absence of this feature; and additional details about semantics and implementation. The document number has been changed to accommodate WG21 numbering.

The Problem

The class scoping features of C++ are very helpful in maintaining large projects, because they allow you to keep most names out of the global namespace. Compile time constants are important for parametrizing constant values which might change, such as array bounds.

However, compile time constants are currently not allowed in class scope:

```
class Foo {
    static const int maxSize = 500;   // illegal
    char localBuffer[maxSize];        // illegal
};
```

It's possible to declare a static const data member and initialize it elsewhere. This is a compile time constant, but it cannot be used within the class declaration (because the initialization appears after the class declaration). The only alternative is to declare such a constant in global scope.

There is a workaround: a similar effect can be had by declaring an appropriate enumeration in class scope:

```
class Foo {
    enum {maxSize = 500};
    ...
};
```

but only needed to hold highest value accepted in constant expression

This works, but is highly inelegant. The constant is not an integral type, either. What's more, enumerations are not guaranteed to be able to hold the largest integral types: they may not be able to hold the same range of values as a long.[2]

1. *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001

2. §7.2, Enumeration Declarations: "The value of an enumerator must be an `int` or a value that can be promoted to an `int` by integral promotions (§4.1)."