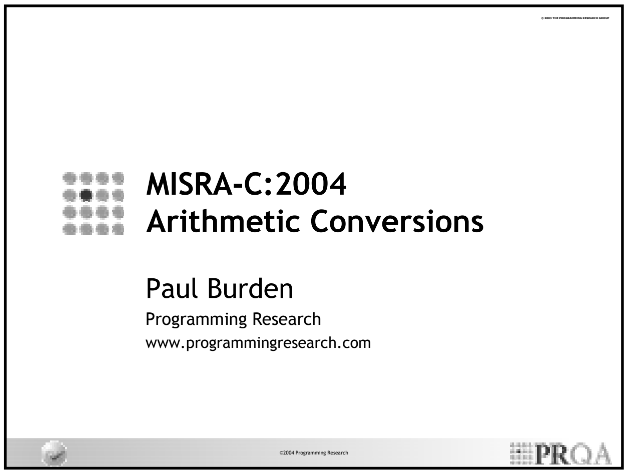
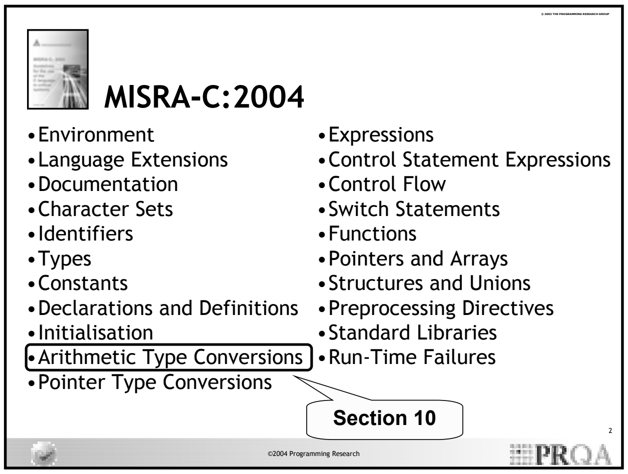


MISRA-C:2004 Arithmetic Conversions

Paul Burden
Programming Research
www.programmingresearch.com

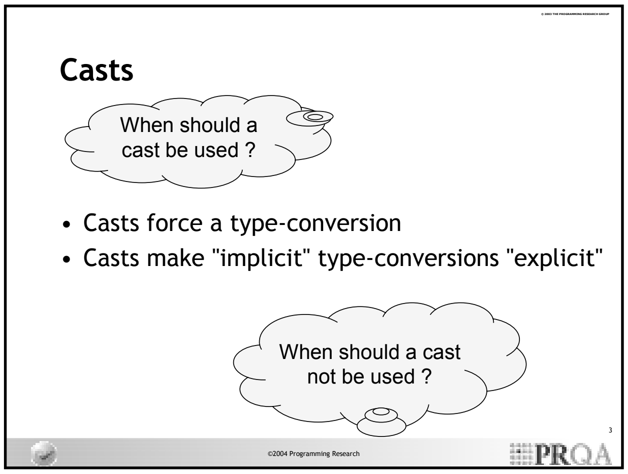


MISRA-C:2004

- Environment
- Language Extensions
- Documentation
- Character Sets
- Identifiers
- Types
- Constants
- Declarations and Definitions
- Initialisation
- Arithmetic Type Conversions
- Pointer Type Conversions
- Expressions
- Control Statement Expressions
- Control Flow
- Switch Statements
- Functions
- Pointers and Arrays
- Structures and Unions
- Preprocessing Directives
- Standard Libraries
- Run-Time Failures

Section 10

2



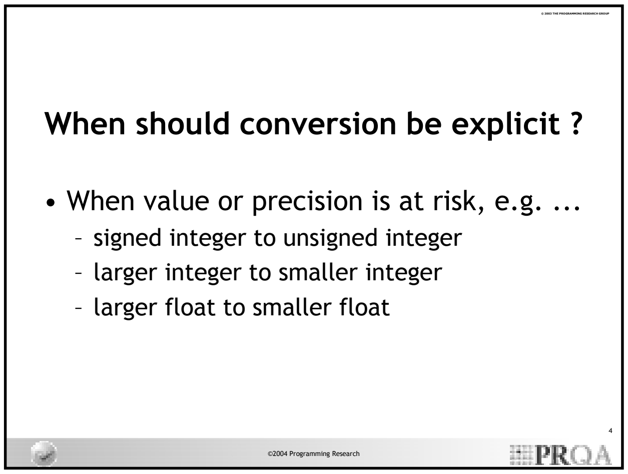
Casts

When should a cast be used ?

- Casts force a type-conversion
- Casts make "implicit" type-conversions "explicit"

When should a cast not be used ?

3

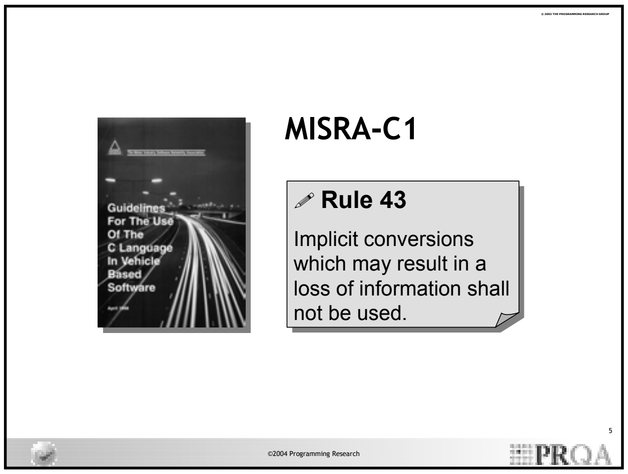


When should conversion be explicit ?

- When value or precision is at risk, e.g. ...
  - signed integer to unsigned integer
  - larger integer to smaller integer
  - larger float to smaller float

4

©2004 Programming Research



Guidelines For The Use Of The C Language In Vehicle Based Software

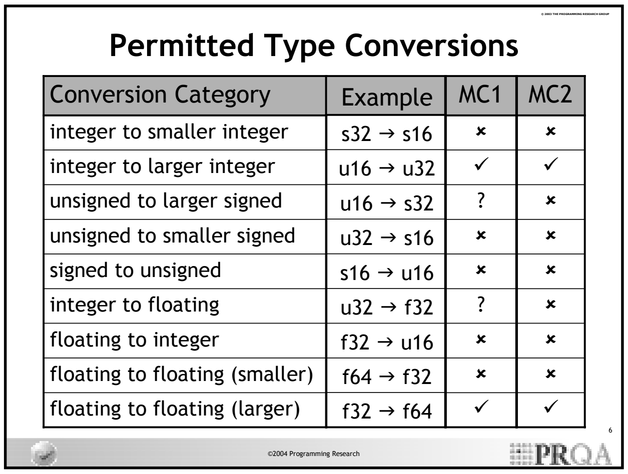
MISRA-C1

Rule 43

Implicit conversions which may result in a loss of information shall not be used.

5

©2004 Programming Research

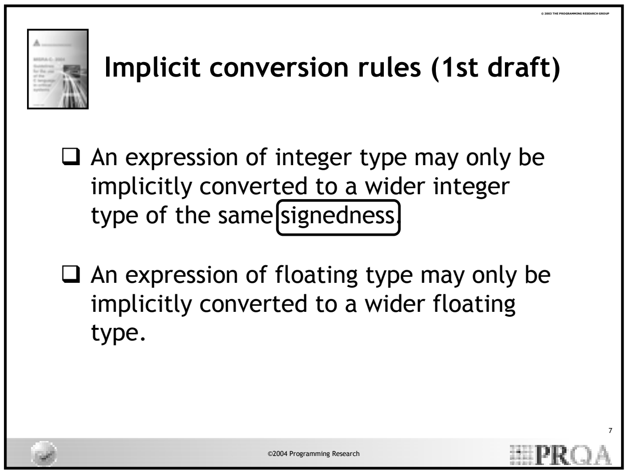


Permitted Type Conversions

Conversion Category	Example	MC1	MC2
integer to smaller integer	s32 → s16	×	×
integer to larger integer	u16 → u32	✓	✓
unsigned to larger signed	u16 → s32	?	×
unsigned to smaller signed	u32 → s16	×	×
signed to unsigned	s16 → u16	×	×
integer to floating	u32 → f32	?	×
floating to integer	f32 → u16	×	×
floating to floating (smaller)	f64 → f32	×	×
floating to floating (larger)	f32 → f64	✓	✓

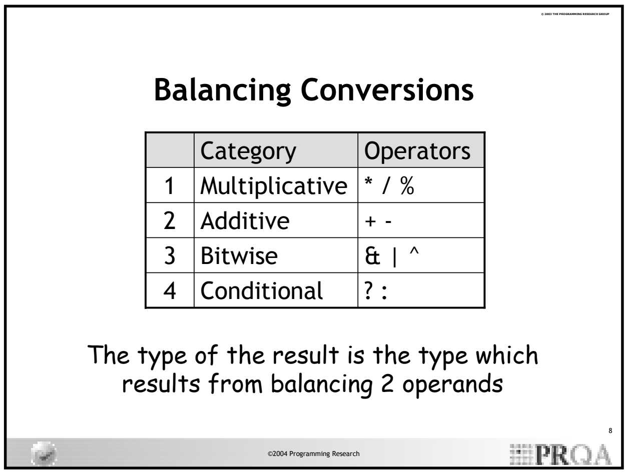
6

©2004 Programming Research



Implicit conversion rules (1st draft)

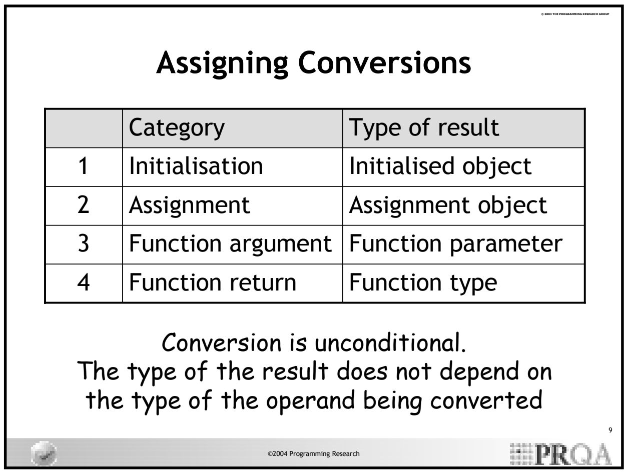
- ❑ An expression of integer type may only be implicitly converted to a wider integer type of the same $\boxed{\text{signedness}}$.
- ❑ An expression of floating type may only be implicitly converted to a wider floating type.



Balancing Conversions

	Category	Operators
1	Multiplicative	* / %
2	Additive	+ -
3	Bitwise	& | ^
4	Conditional	? :

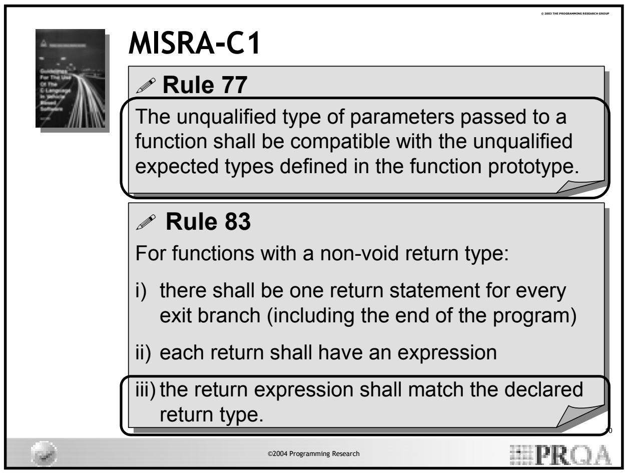
The type of the result is the type which results from balancing 2 operands



Assigning Conversions

	Category	Type of result
1	Initialisation	Initialised object
2	Assignment	Assignment object
3	Function argument	Function parameter
4	Function return	Function type

Conversion is unconditional.
The type of the result does not depend on the type of the operand being converted



MISRA-C1

Rule 77

The unqualified type of parameters passed to a function shall be compatible with the unqualified expected types defined in the function prototype.

Rule 83

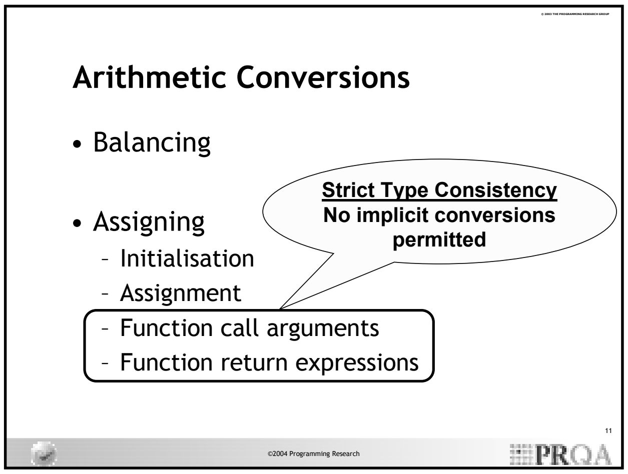
For functions with a non-void return type:

i) there shall be one return statement for every exit branch (including the end of the program)

ii) each return shall have an expression

iii) the return expression shall match the declared return type.

©2004 Programming Research



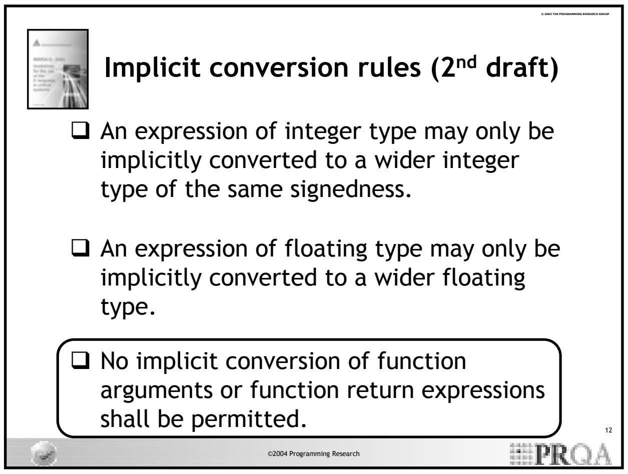
Arithmetic Conversions

- Balancing
- Assigning
  - Initialisation
  - Assignment
  - Function call arguments
  - Function return expressions

Strict Type Consistency
No implicit conversions permitted

11

©2004 Programming Research



Implicit conversion rules (2nd draft)

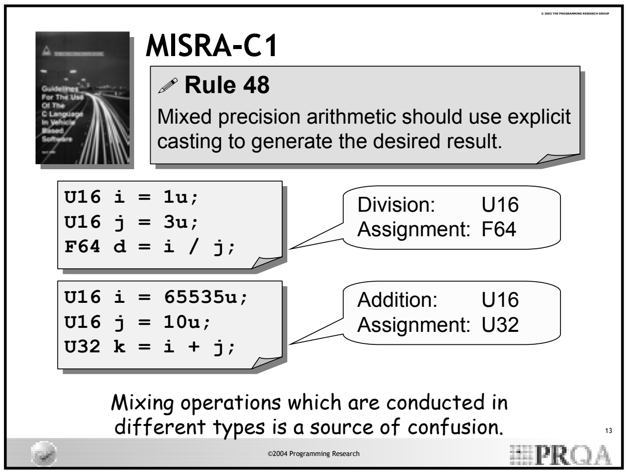
❑ An expression of integer type may only be implicitly converted to a wider integer type of the same signedness.

❑ An expression of floating type may only be implicitly converted to a wider floating type.

❑ No implicit conversion of function arguments or function return expressions shall be permitted.

12

©2004 Programming Research



MISRA-C1

Rule 48
Mixed precision arithmetic should use explicit casting to generate the desired result.

```
U16 i = 1u;
U16 j = 3u;
F64 d = i / j;
```

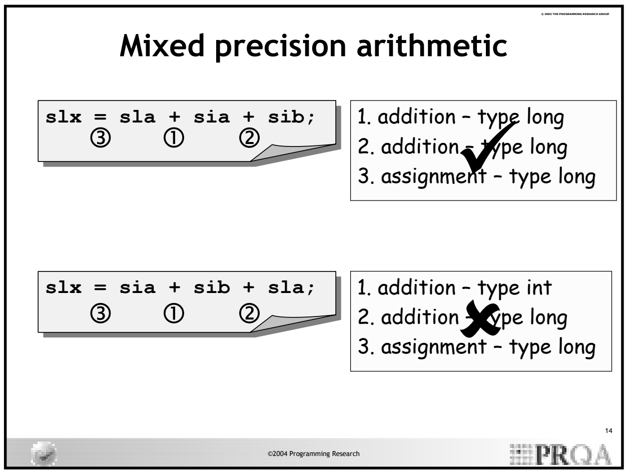
Division: U16
Assignment: F64

```
U16 i = 65535u;
U16 j = 10u;
U32 k = i + j;
```

Addition: U16
Assignment: U32

Mixing operations which are conducted in different types is a source of confusion.

13



Mixed precision arithmetic

```
slx = sla + sia + sib;
     ③     ①     ②
```

1. addition – type long
2. addition – type long
3. assignment – type long

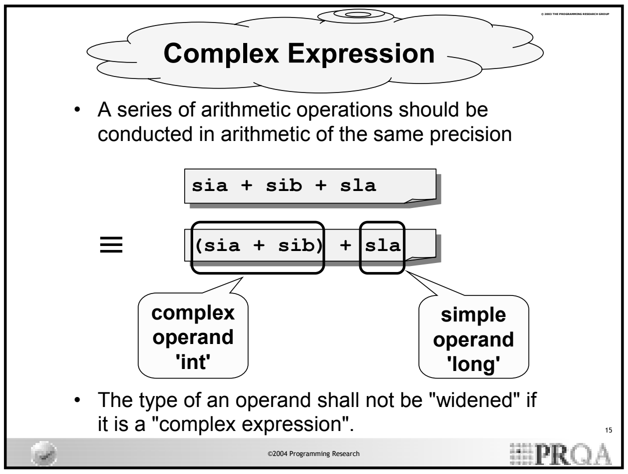
✓

```
slx = sia + sib + sla;
     ③     ①     ②
```

1. addition – type int
2. addition – type long
3. assignment – type long

✗

14



Complex Expression

- A series of arithmetic operations should be conducted in arithmetic of the same precision

```
sia + sib + sla
```

$\equiv$

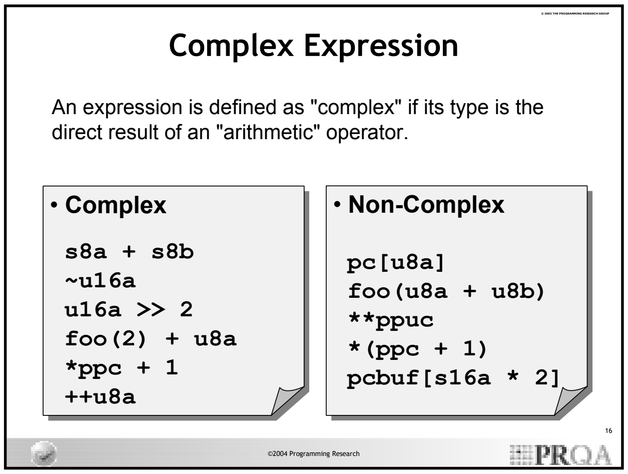
```
(sia + sib) + sla
```

complex operand 'int'

simple operand 'long'

- The type of an operand shall not be "widened" if it is a "complex expression".

15



Complex Expression

An expression is defined as "complex" if its type is the direct result of an "arithmetic" operator.

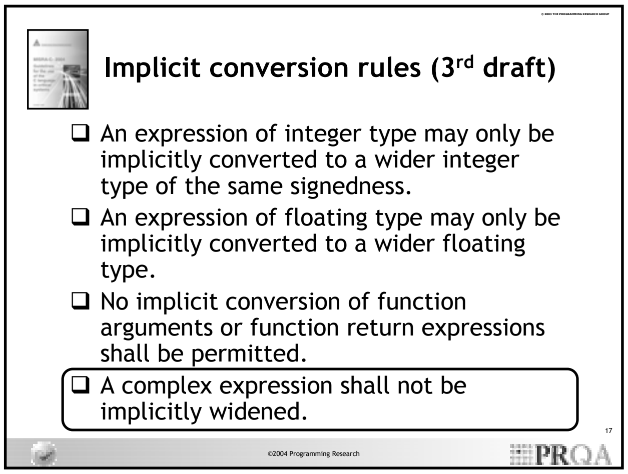
- Complex

```
s8a + s8b
~u16a
u16a >> 2
foo(2) + u8a
*ppc + 1
++u8a
```

- Non-Complex

```
pc[u8a]
foo(u8a + u8b)
**ppuc
*(ppc + 1)
pcbuf[s16a * 2]
```

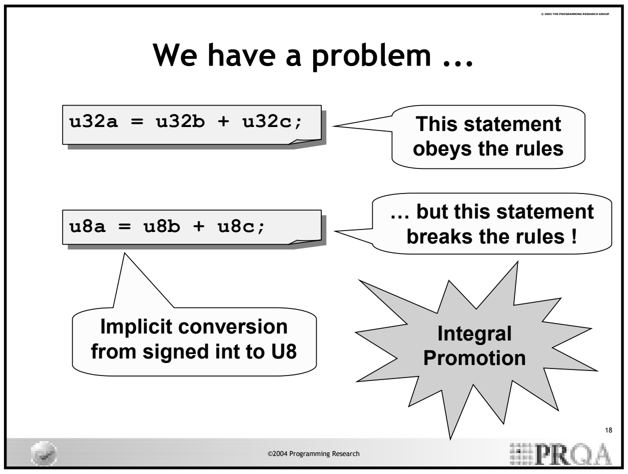
16



Implicit conversion rules (3rd draft)

- ❑ An expression of integer type may only be implicitly converted to a wider integer type of the same signedness.
- ❑ An expression of floating type may only be implicitly converted to a wider floating type.
- ❑ No implicit conversion of function arguments or function return expressions shall be permitted.
- ❑ A complex expression shall not be implicitly widened.

17



We have a problem ...

```
u32a = u32b + u32c;
```

This statement obeys the rules

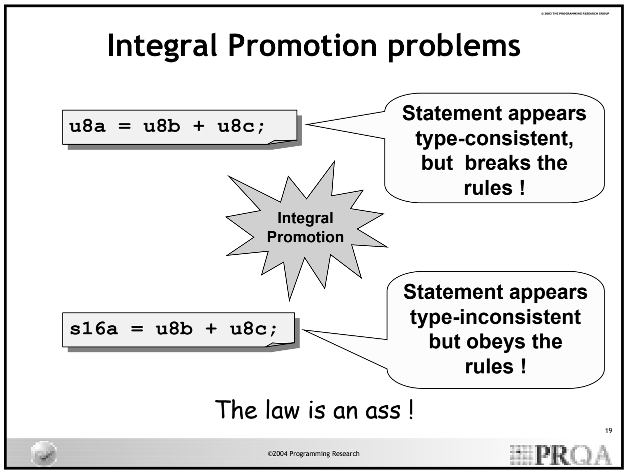
```
u8a = u8b + u8c;
```

... but this statement breaks the rules !

Implicit conversion from signed int to U8

Integral Promotion

18



Integral Promotion problems

```
u8a = u8b + u8c;
```

Statement appears type-consistent, but breaks the rules !

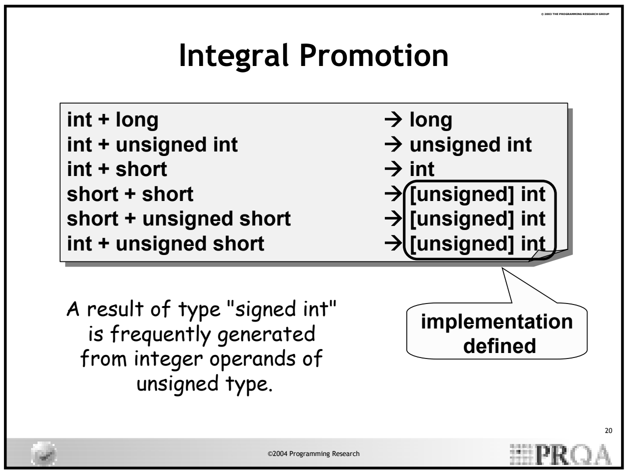
Integral Promotion

```
s16a = u8b + u8c;
```

Statement appears type-inconsistent but obeys the rules !

The law is an ass !

©2004 Programming Research 19



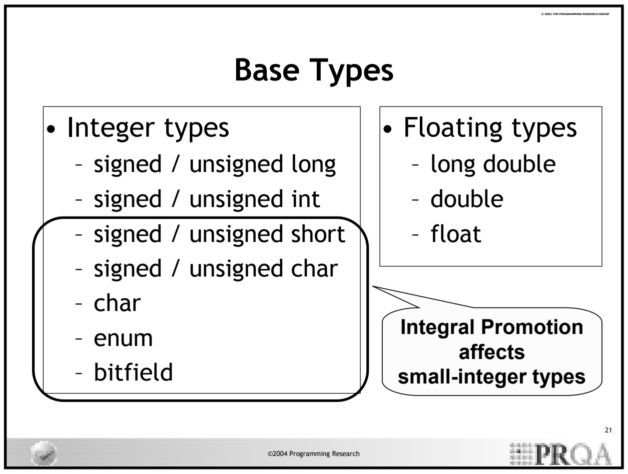
Integral Promotion

int + long	→ long
int + unsigned int	→ unsigned int
int + short	→ int
short + short	→ [unsigned] int
short + unsigned short	→ [unsigned] int
int + unsigned short	→ [unsigned] int

implementation defined

A result of type "signed int" is frequently generated from integer operands of unsigned type.

©2004 Programming Research 20

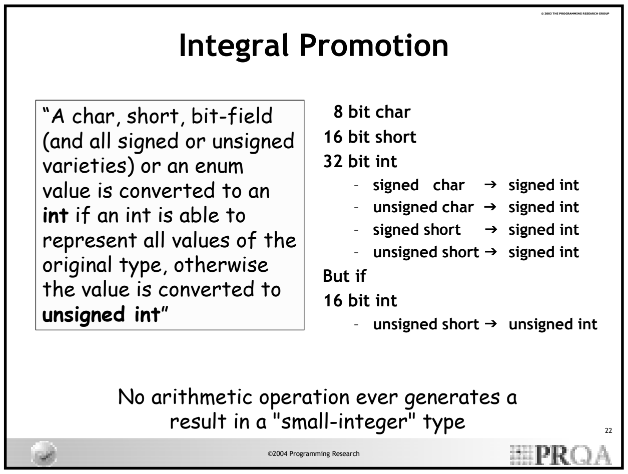


Base Types

- Integer types
  - signed / unsigned long
  - signed / unsigned int
  - signed / unsigned short
  - signed / unsigned char
  - char
  - enum
  - bitfield
- Floating types
  - long double
  - double
  - float

Integral Promotion affects small-integer types

©2004 Programming Research 21



Integral Promotion

"A char, short, bit-field (and all signed or unsigned varieties) or an enum value is converted to an **int** if an int is able to represent all values of the original type, otherwise the value is converted to **unsigned int**"

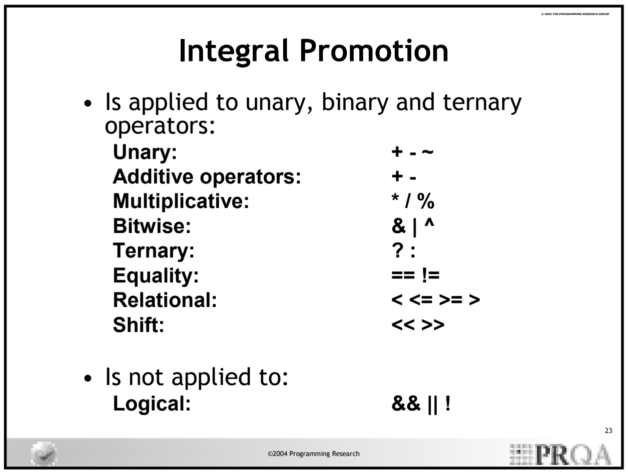
8 bit char
16 bit short
32 bit int

- signed char → signed int
- unsigned char → signed int
- signed short → signed int
- unsigned short → signed int

But if
16 bit int

- unsigned short → unsigned int

No arithmetic operation ever generates a result in a "small-integer" type



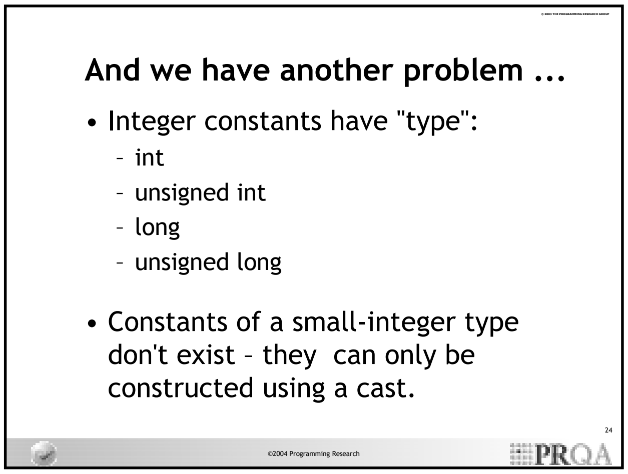
Integral Promotion

- Is applied to unary, binary and ternary operators:

Unary:	**+ - ~**
Additive operators:	**+ -**
Multiplicative:	*** / %**
Bitwise:	**& | ^**
Ternary:	**? :**
Equality:	**== !=**
Relational:	**< <= >= >**
Shift:	**<< >>**

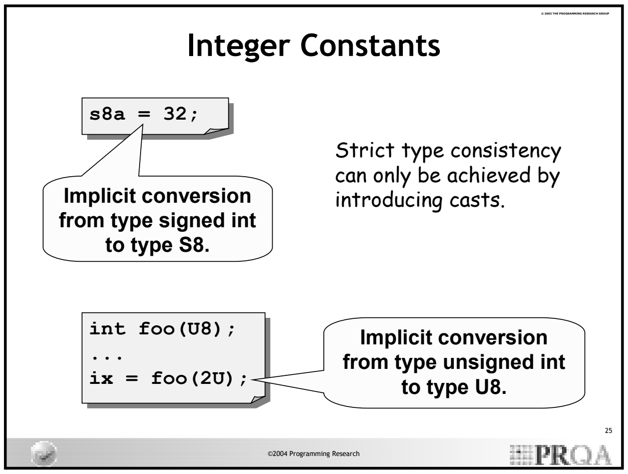
- Is not applied to:

Logical:	**&& || !**



And we have another problem ...

- Integer constants have "type":
  - int
  - unsigned int
  - long
  - unsigned long
- Constants of a small-integer type don't exist – they can only be constructed using a cast.



Integer Constants

```
s8a = 32;
```

Implicit conversion from type signed int to type S8.

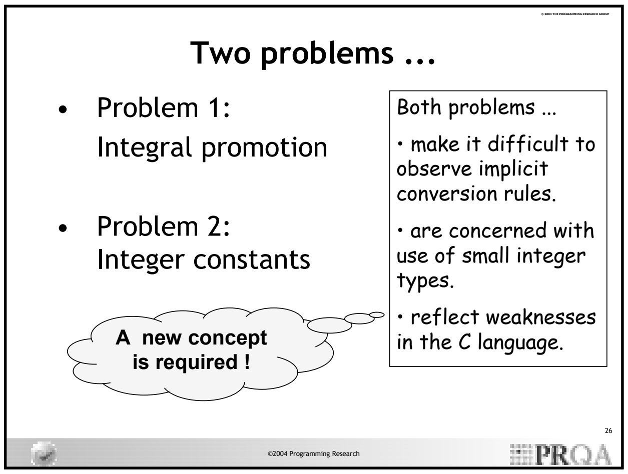
Strict type consistency can only be achieved by introducing casts.

```
int foo(U8);
...
ix = foo(2U);
```

Implicit conversion from type unsigned int to type U8.

25

©2004 Programming Research



Two problems ...

- Problem 1:
  Integral promotion
- Problem 2:
  Integer constants

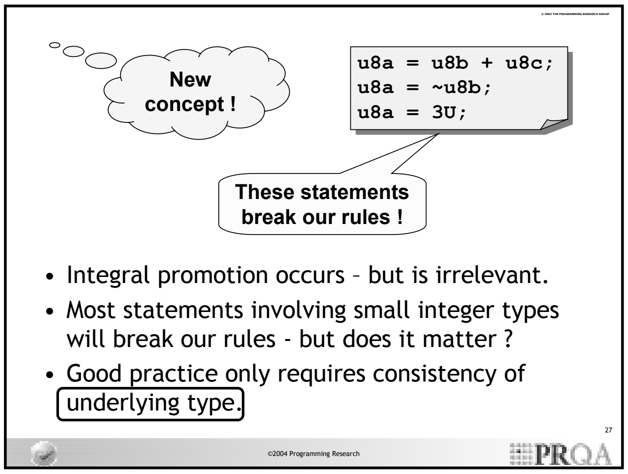
Both problems ...

- make it difficult to observe implicit conversion rules.
- are concerned with use of small integer types.
- reflect weaknesses in the C language.

A new concept is required !

26

©2004 Programming Research



New concept !

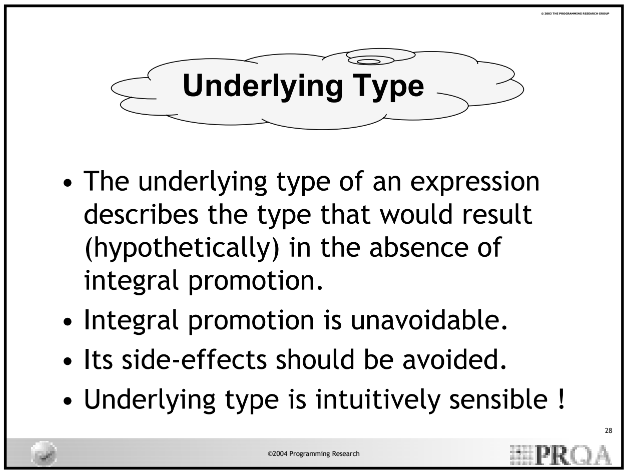
```
u8a = u8b + u8c;
u8a = ~u8b;
u8a = 3U;
```

These statements break our rules !

- Integral promotion occurs – but is irrelevant.
- Most statements involving small integer types will break our rules - but does it matter ?
- Good practice only requires consistency of $\boxed{\text{underlying type.}}$

27

©2004 Programming Research



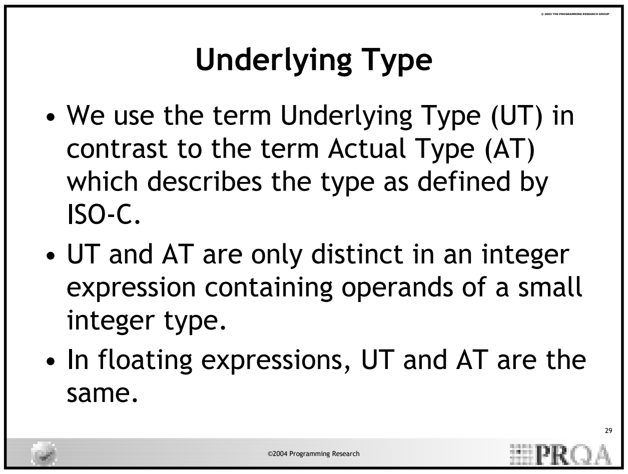
Underlying Type

- The underlying type of an expression describes the type that would result (hypothetically) in the absence of integral promotion.
- Integral promotion is unavoidable.
- Its side-effects should be avoided.
- Underlying type is intuitively sensible !

28

©2004 Programming Research

PRQA



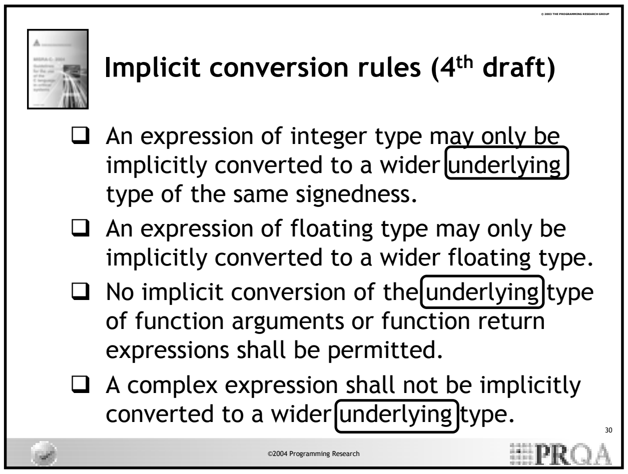
Underlying Type

- We use the term Underlying Type (UT) in contrast to the term Actual Type (AT) which describes the type as defined by ISO-C.
- UT and AT are only distinct in an integer expression containing operands of a small integer type.
- In floating expressions, UT and AT are the same.

29

©2004 Programming Research

PRQA



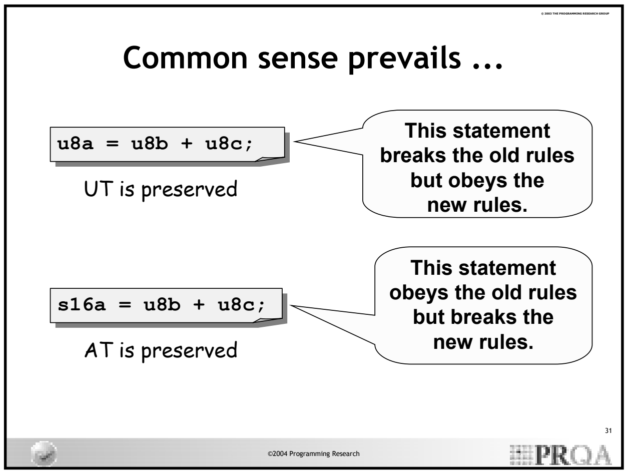
Implicit conversion rules (4th draft)

- ❑ An expression of integer type may only be implicitly converted to a wider $\boxed{\text{underlying}}$ type of the same signedness.
- ❑ An expression of floating type may only be implicitly converted to a wider floating type.
- ❑ No implicit conversion of the $\boxed{\text{underlying}}$ type of function arguments or function return expressions shall be permitted.
- ❑ A complex expression shall not be implicitly converted to a wider $\boxed{\text{underlying}}$ type.

30

©2004 Programming Research

PRQA



Common sense prevails ...

```
u8a = u8b + u8c;
```

UT is preserved

This statement breaks the old rules but obeys the new rules.

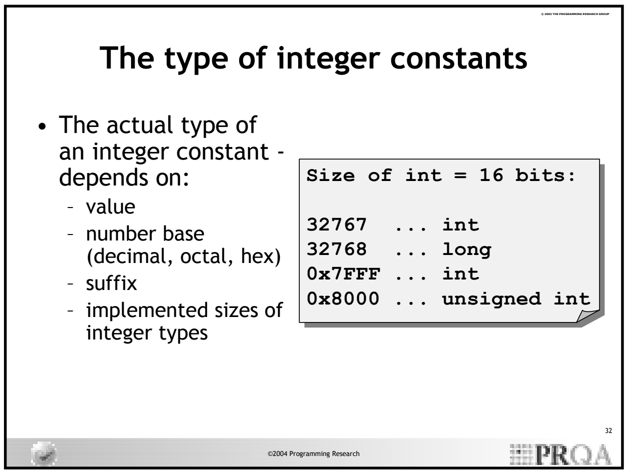
```
s16a = u8b + u8c;
```

AT is preserved

This statement obeys the old rules but breaks the new rules.

31

PRQA



The type of integer constants

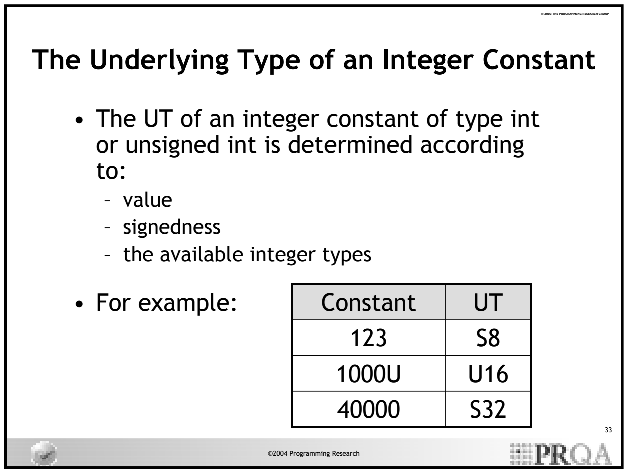
- The actual type of an integer constant - depends on:
  - value
  - number base (decimal, octal, hex)
  - suffix
  - implemented sizes of integer types

```
Size of int = 16 bits:

32767  ... int
32768  ... long
0x7FFF ... int
0x8000 ... unsigned int
```

32

PRQA



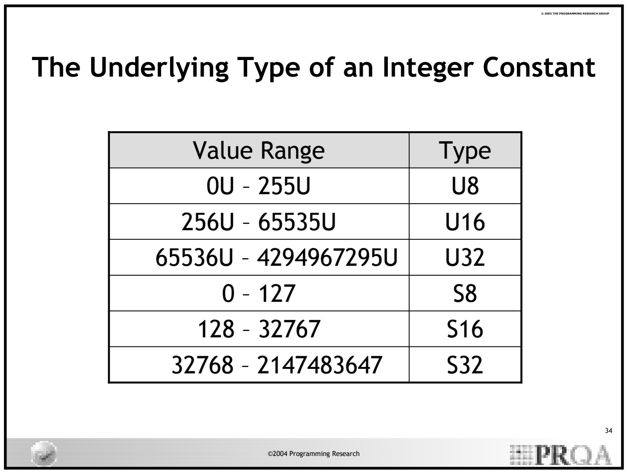
The Underlying Type of an Integer Constant

- The UT of an integer constant of type int or unsigned int is determined according to:
  - value
  - signedness
  - the available integer types
- For example:

Constant	UT
123	S8
1000U	U16
40000	S32

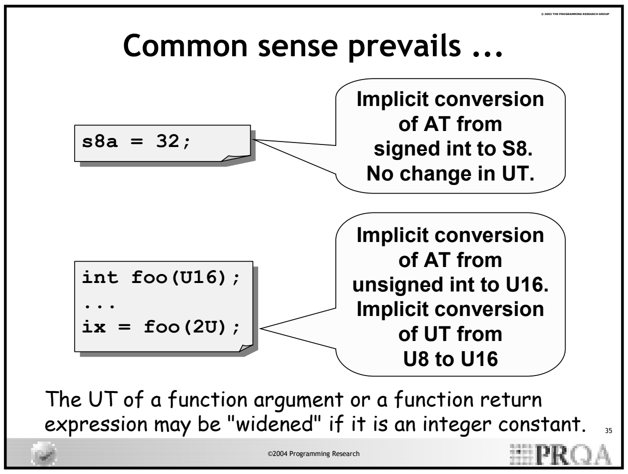
33

PRQA



The Underlying Type of an Integer Constant

Value Range	Type
0U – 255U	U8
256U – 65535U	U16
65536U – 4294967295U	U32
0 – 127	S8
128 – 32767	S16
32768 – 2147483647	S32



Common sense prevails ...

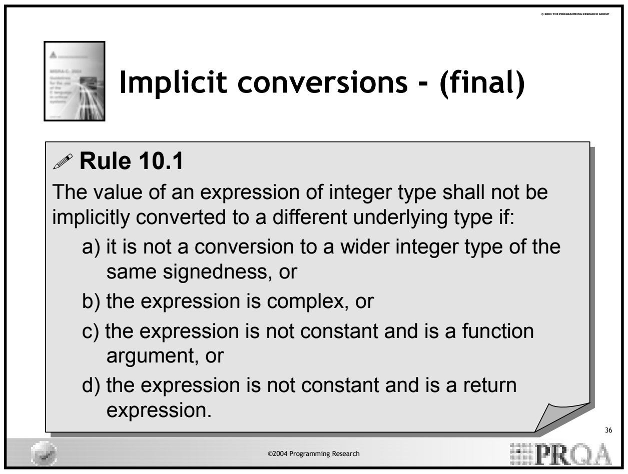
```
s8a = 32;
```

Implicit conversion of AT from signed int to S8. No change in UT.

```
int foo(U16);
...
ix = foo(2U);
```

Implicit conversion of AT from unsigned int to U16. Implicit conversion of UT from U8 to U16

The UT of a function argument or a function return expression may be "widened" if it is an integer constant.



Implicit conversions - (final)

Rule 10.1

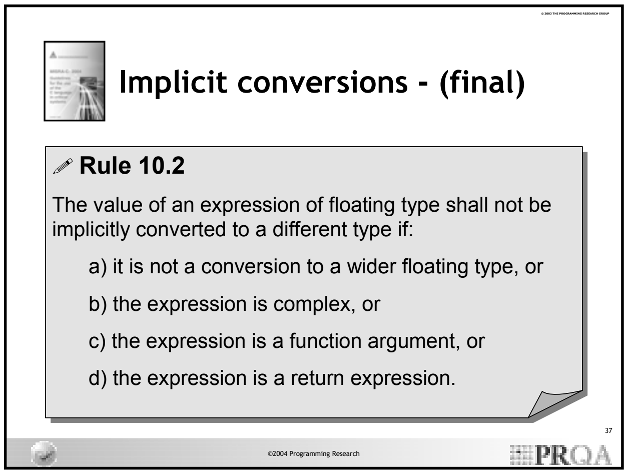
The value of an expression of integer type shall not be implicitly converted to a different underlying type if:

a) it is not a conversion to a wider integer type of the same signedness, or

b) the expression is complex, or

c) the expression is not constant and is a function argument, or

d) the expression is not constant and is a return expression.



Implicit conversions - (final)

Rule 10.2

The value of an expression of floating type shall not be implicitly converted to a different type if:

a) it is not a conversion to a wider floating type, or

b) the expression is complex, or

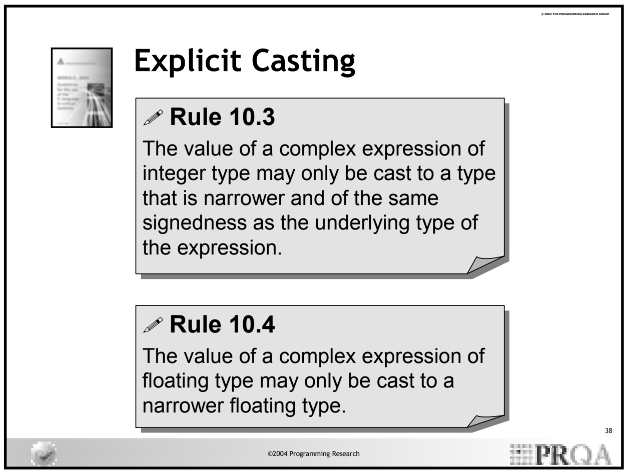
c) the expression is a function argument, or

d) the expression is a return expression.

37

©2004 Programming Research

PRQA



Explicit Casting

Rule 10.3

The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

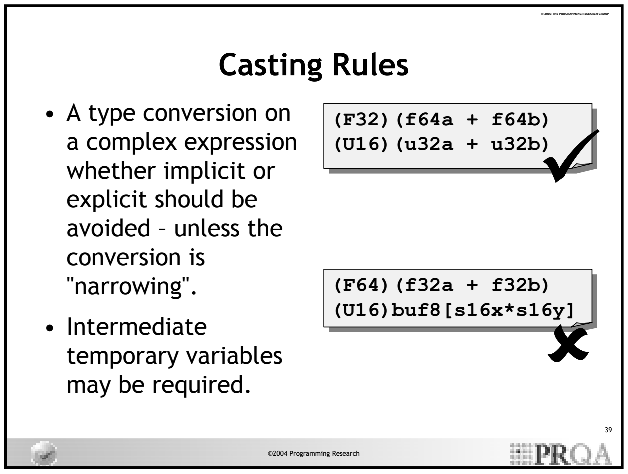
Rule 10.4

The value of a complex expression of floating type may only be cast to a narrower floating type.

38

©2004 Programming Research

PRQA



Casting Rules

- A type conversion on a complex expression whether implicit or explicit should be avoided – unless the conversion is "narrowing".
- Intermediate temporary variables may be required.

```
(F32)(f64a + f64b)
(U16)(u32a + u32b)
```

✓

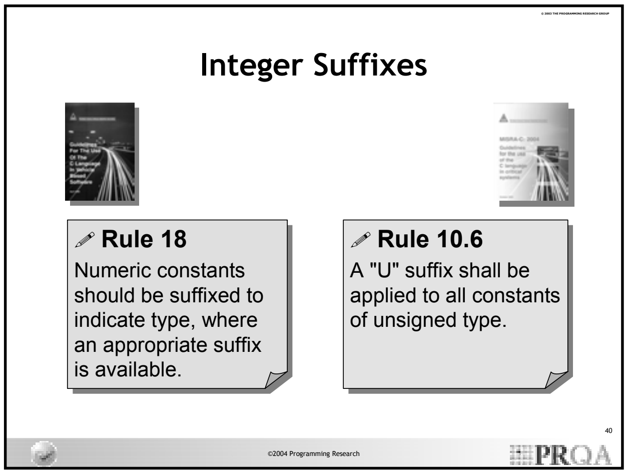
```
(F64)(f32a + f32b)
(U16)buf8[s16x*s16y]
```

✗

39

©2004 Programming Research

PRQA



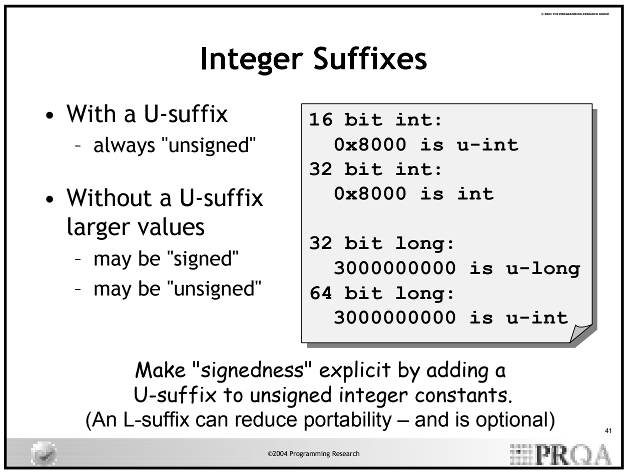
Integer Suffixes

Rule 18
Numeric constants should be suffixed to indicate type, where an appropriate suffix is available.

Rule 10.6
A "U" suffix shall be applied to all constants of unsigned type.

40

©2004 Programming Research



Integer Suffixes

- With a U-suffix
  - always "unsigned"
- Without a U-suffix larger values
  - may be "signed"
  - may be "unsigned"

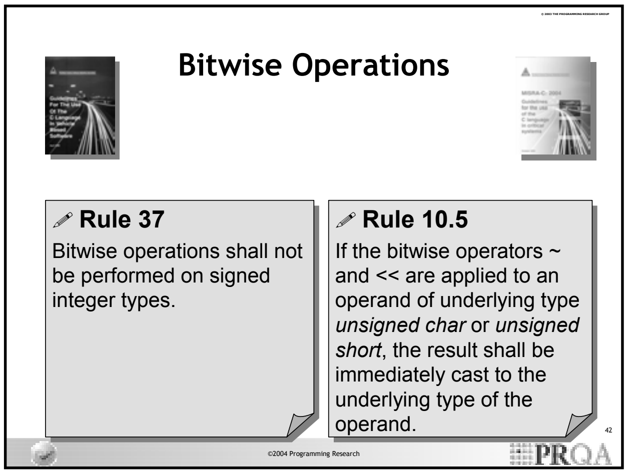
```
16 bit int:
  0x8000 is u-int
32 bit int:
  0x8000 is int

32 bit long:
  3000000000 is u-long
64 bit long:
  3000000000 is u-int
```

Make "signedness" explicit by adding a U-suffix to unsigned integer constants.
(An L-suffix can reduce portability – and is optional)

41

©2004 Programming Research



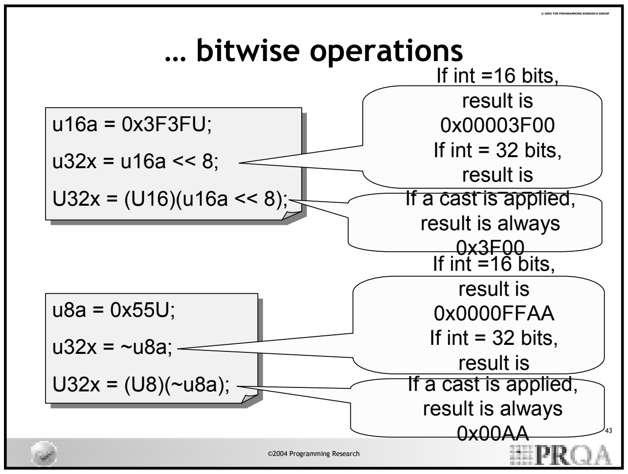
Bitwise Operations

Rule 37
Bitwise operations shall not be performed on signed integer types.

Rule 10.5
If the bitwise operators ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand.

42

©2004 Programming Research



… bitwise operations

```
u16a = 0x3F3FU;
u32x = u16a << 8;
U32x = (U16)(u16a << 8);
```

If int =16 bits, result is 0x00003F00
If int = 32 bits, result is

If a cast is applied, result is always 0x3F00

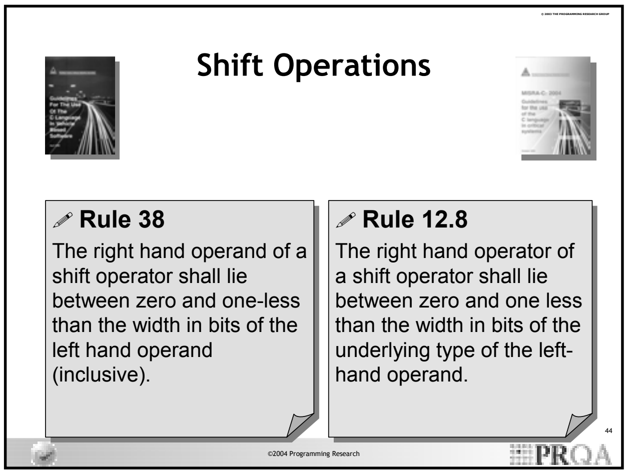
```
u8a = 0x55U;
u32x = ~u8a;
U32x = (U8)(~u8a);
```

If int =16 bits, result is 0x0000FFAA
If int = 32 bits, result is

If a cast is applied, result is always 0x00AA

43

©2004 Programming Research



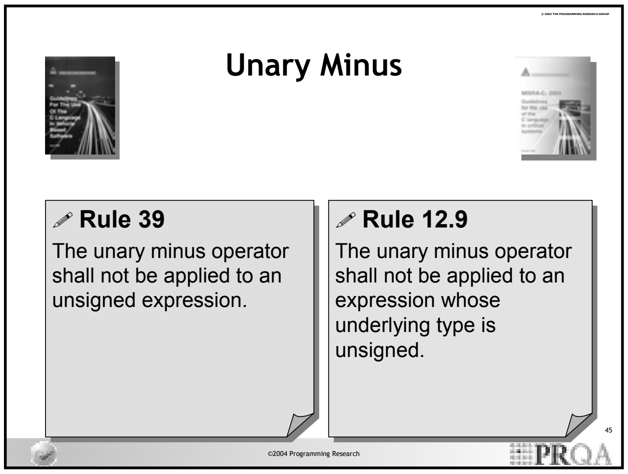
Shift Operations

Rule 38
The right hand operand of a shift operator shall lie between zero and one-less than the width in bits of the left hand operand (inclusive).

Rule 12.8
The right hand operator of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

44

©2004 Programming Research



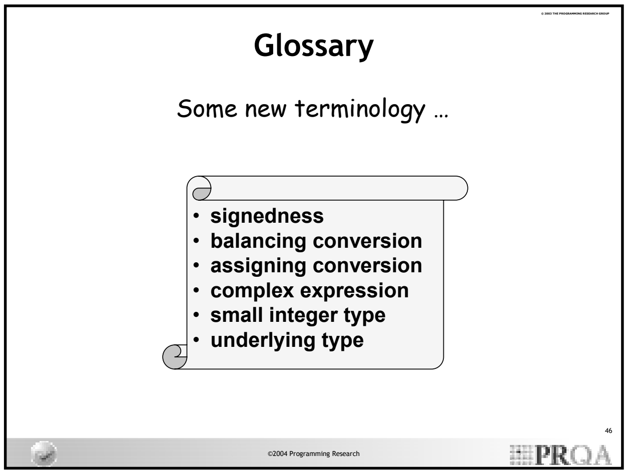
Unary Minus

Rule 39
The unary minus operator shall not be applied to an unsigned expression.

Rule 12.9
The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

45

©2004 Programming Research

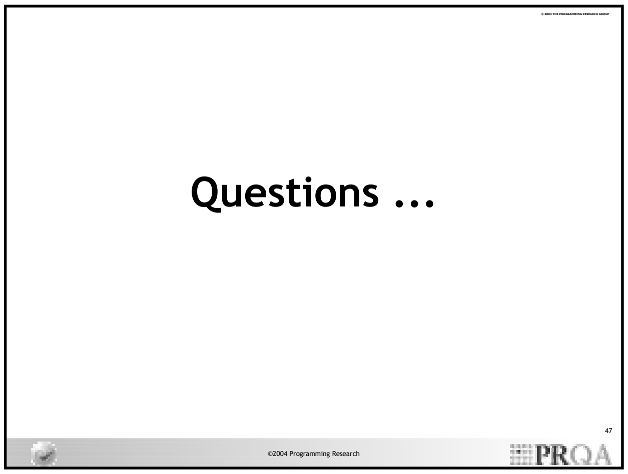


Glossary

Some new terminology …

- signedness
- balancing conversion
- assigning conversion
- complex expression
- small integer type
- underlying type

46



Questions ...

47