Conversions

assignment

- 6.5.16 Assignment operators
  - para 3 – The type of an assignment expression is the type of the left operand...
- 6.5.16.1 Simple assignment
  - para 2 – The value of the right operand is converted to the type of the assignment expression

```
unsigned short x = 0;
x = UINT_MAX;
```

Assignment is the only binary operator that can cause the type of one of its operands to be implicitly converted to a "narrower" type.
So x = y; → x == y; is not true

integer promotions

- **6.3.1.1 Booleans, characters, and integers**
  - **para 2 – If an** int **can represent all the values of the original type, the value is converted to an** int**; otherwise, it is converted to an** unsigned int**. These are called the *integer promotions*.**

```
char c1, c2;

c1 + c2

(int)c1 + (int)c2
```

Why does the compiler prefer ints?

argument promotion

- 6.5.2.2 Function calls
  - para 6 – if the expression that denotes the called function does not include a prototype, the *integer promotions* are performed on each argument, and arguments that have type float are promoted to double. These are called the *default argument promotions*.

```
#include <stdio.h>

int main(void)
{
    return call(4, 2);
}

int call(double a, double b)
{
    return printf("%f, %f\n", a, b);
}
```

argument promotion

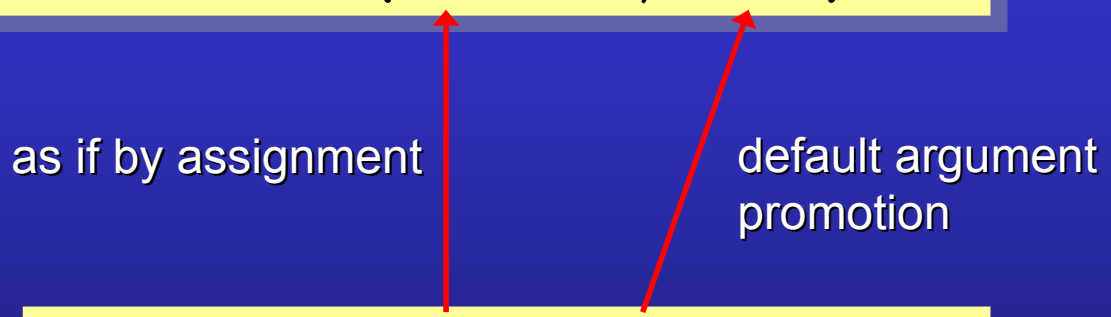
- 6.5.2.2 Function calls
  - para 7 – if the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters...

```
#include <stdio.h>

int call(double, double);

int main(void)
{
    return call(4, 2);
}

int call(double a, double b)
{
    return printf("%f, %f\n", a, b);
}
```

argument promotion

- 6.5.2.2 Function calls
  - para 7 – The ellipsis notation in a function prototype declarator causes argument type conversions to stop after the last declared parameter. The *default argument promotions* are performed on the trailing arguments.

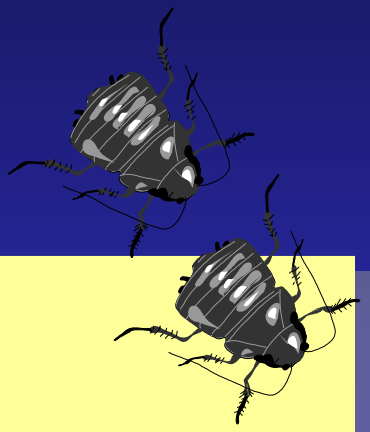
```
void variadic(char c, ...);
```




```
variadic('X', 'X');
```

exercise

- spot the bugs

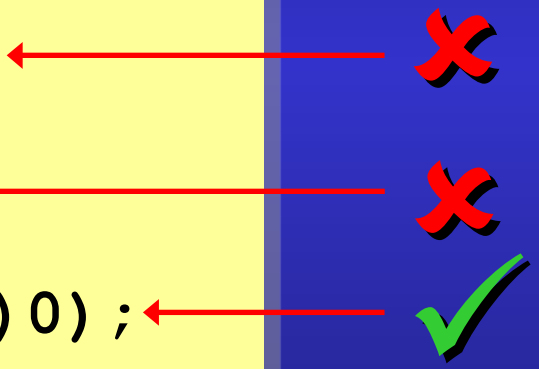


```c
#include <stdio.h>

int main(void)
{
    printf("%p", NULL);

    printf("%p", 0);

    printf("%p", (void*)0);

}
```

answer

- only (void*)0 is guaranteed to be a pointer
  - 0 is an int
  - NULL could be 0 too

```
#include <stdio.h>

int main(void)
{
    printf("%p", NULL);

    printf("%p", 0);

    printf("%p", (void*)0);

}
```



- <stdarg.h> provide type-unsafe access
  - restrictions on all the va_ macros

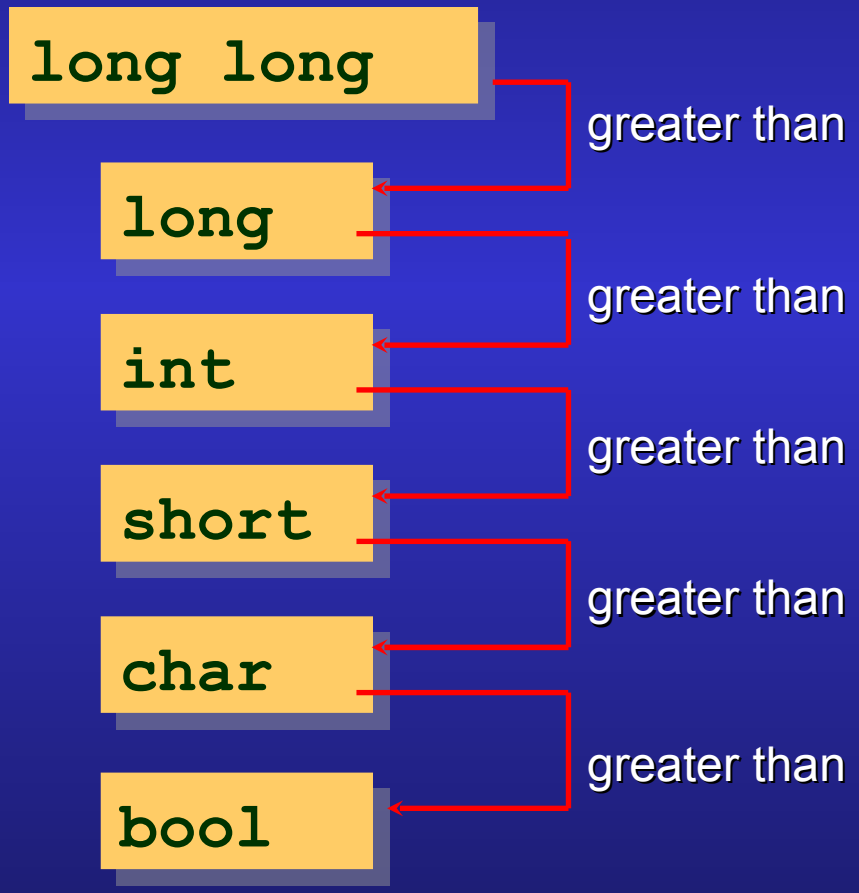
```
#include <stdarg.h>

int printf(const char * format, ...)
{
  va_list args;
  va_start(args, format);
  for (size_t at = 0; format[at]; at++)
  {
    switch (format[at])
    {
      case 'c':
        {
          int param = va_arg(format, int);
          char passed = (char)param;
          ...
        }
      ...
    }
  }
  va_end(args);
}
```

6.3.1.1 Booleans, characters, and integers

…

Every integer type has an *integer conversion rank*…

…



rank

6.3.1.8 Usual arithmetic conversions

part 1: the obvious conversion rule (safe)

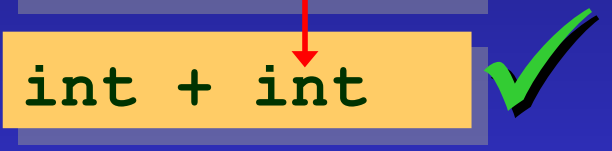
…

[both operands have integer type]

…

• the *integer promotions* are performed on both operands

```
int + char
int + int
```



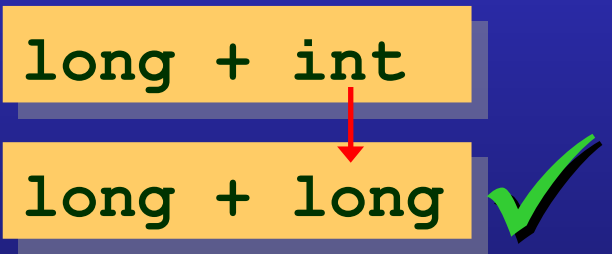
• If both operands have the same type, then no further conversion is needed.

```
long + long
```

• Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

…

```
long + int
long + long
```

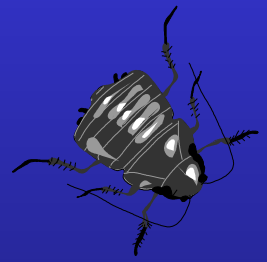


- **6.3.1.8 Usual arithmetic conversions**
  - **part 2: the signed → unsigned rule (lossy)**

…
[one signed and one unsigned operand]
…

• Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other [signed] operand, then the operand with the signed integer type is converted to the type of the operand with the unsigned integer type.

```
unsigned long + int
```



```
unsigned long + unsigned long
```

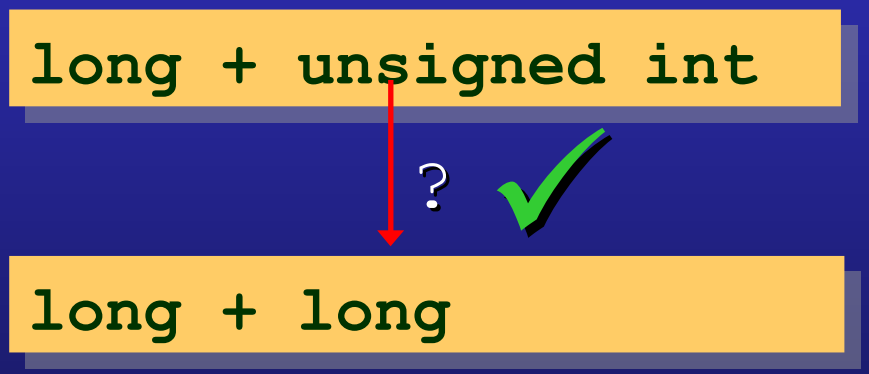
a negative signed integer value can be converted into a large positive unsigned integer value!!

6.3.1.8 Usual arithmetic conversions

part 3: the unsigned → signed rule (safe)

…
[one signed and one unsigned operand]
[rank(signed operand) > rank(unsigned operand)]
…

•Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

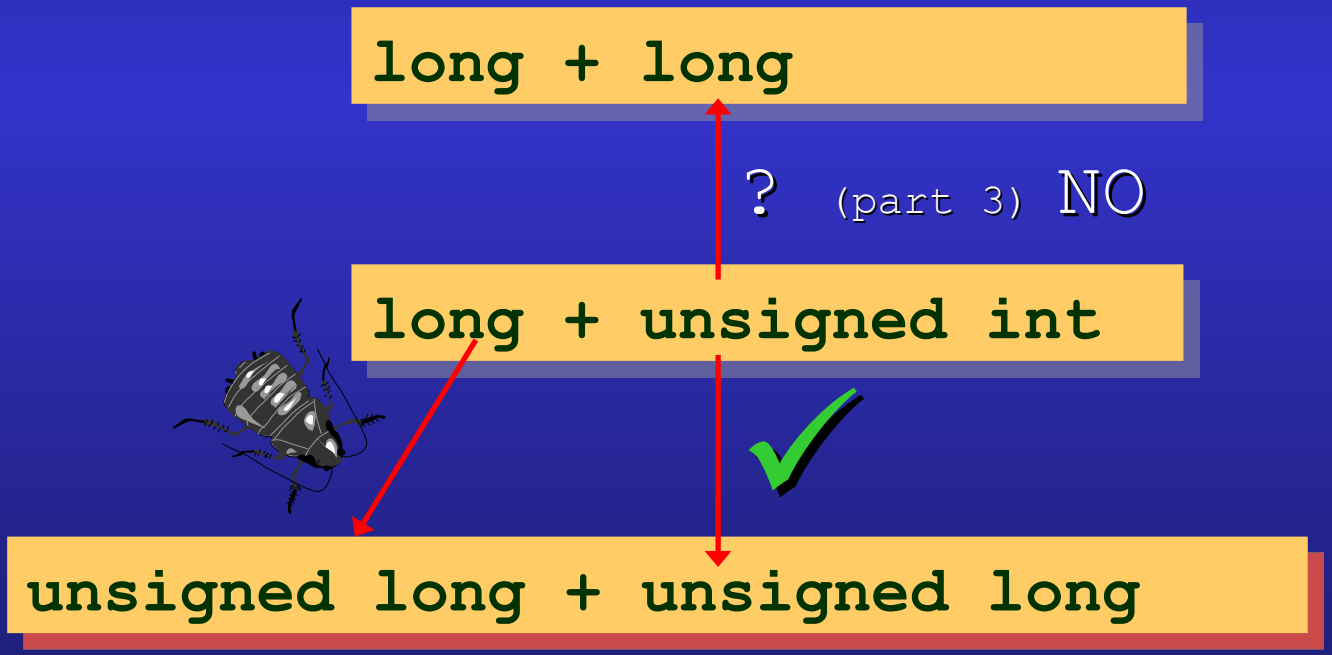


this depends on their value representations, as specified in <limits.h>

- ## 6.3.1.8 Usual arithmetic conversions
  - ### part 4 – the last resort rule (lossy)

…

•Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.



exercise

- is this program's behaviour
  - undefined?
  - unspecified?
  - implementation-defined?
  - conforming?
  - strictly conforming?

```c
#include <stdio.h>

int main(void)
{
    unsigned long a = 0;
    signed int b = -42;
    unsigned long long c = a + b;
    printf("%llu\n", c);
}
```

answer

- is this program's behaviour
  - undefined? NO
  - unspecified? NO
  - implementation-defined? YES
  - conforming? YES
  - strictly conforming? NO

```c
#include <stdio.h>

int main(void)
{
    unsigned long a = 0;
    signed int b = -42;
    unsigned long long c = a + b;
    printf("%llu\n", c);
}
```

18446744073709551574

signed ↔ unsigned

- 6.3.1.3 Signed and unsigned integers
  - para 1 – When a value with integer type is converted to another integer type, other than _Bool, if the value can be represented by the new type, it is unchanged.
  - para 2 – Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
  - para 3 – Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

conversions

unary	!
boolean	&& ||

→ int 0/1

unary	++ --
assignment	= *= /= %= += -= ...
comma	,

unary	~ + -
shift	<< >>

- these operators perform the *usual arithmetic conversions*

arithmetic	`* / % + -`
relational	`< > <= >= == !=`
bitwise	`& ^ |`
ternary	`?:`

exercise

- what does this program print?
  - assume sizeof(short) == 2
  - assume sizeof(int) == 4

```
void exercise(void)
{
    short s = 42;

    printf("%zd\n", sizeof(s));

    printf("%zd\n", sizeof(s && s);

    printf("%zd\n", sizeof(+s));

    printf("%zd\n", sizeof(s = s));
}
```

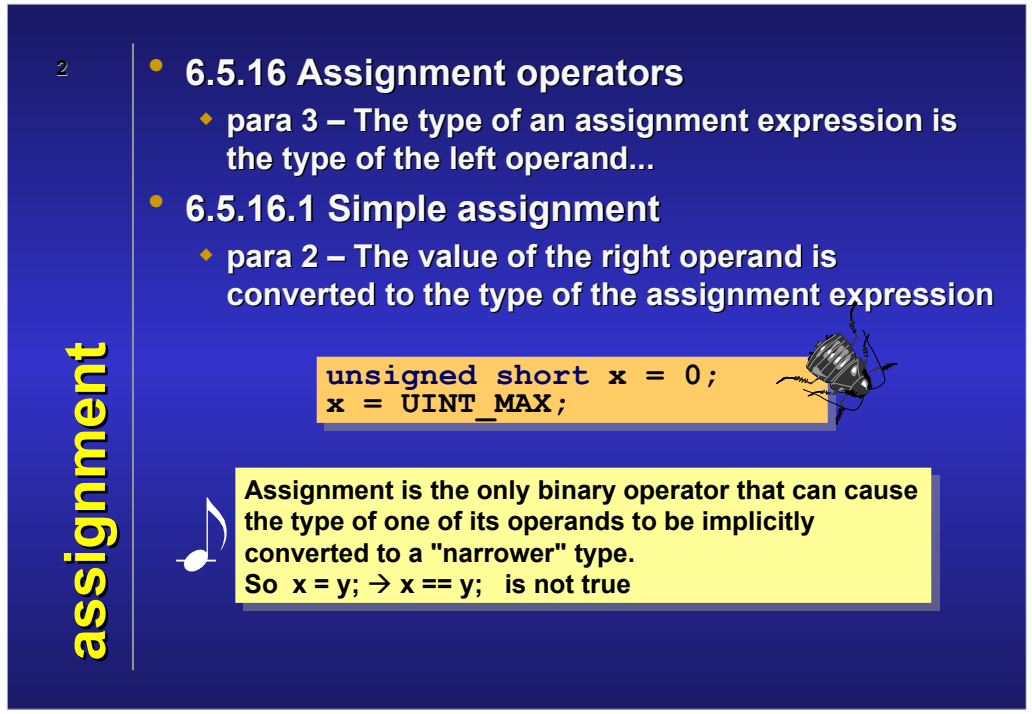
answer

```
void exercise(void)
{
    short s = 42;

    printf("%zd\n", sizeof(s));

    printf("%zd\n", sizeof(s && s));

    printf("%zd\n", sizeof(+s));

    printf("%zd\n", sizeof(s = s));
}
```

2 4 4 2

Advanced C

Conversions

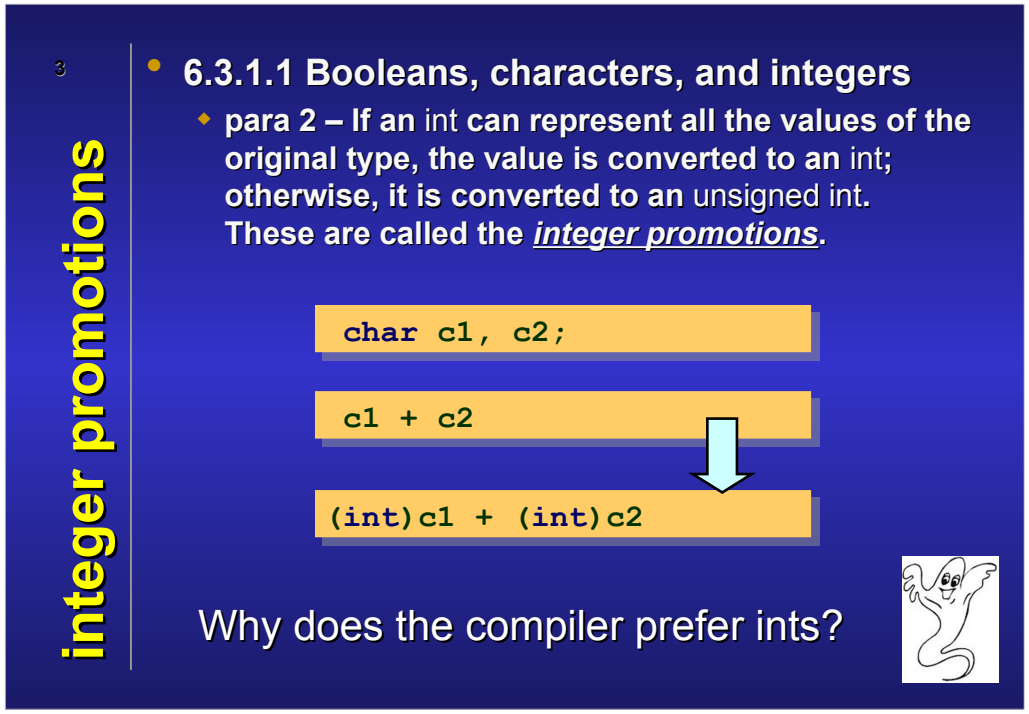


Note that these rules also apply to constants. For example a character constant such as 'x' is an *integer* character constant of type int. In C++ however, 'x' has type char.

Integer promotions do not take place in a simple assignment expression such as char=char.

Also, integer promotions are not applied to an object used as the operand to sizeof (however they are applied if the operand is not an object, viz cannot have its address taken).

```
sizeof('x') == sizeof(int);
/* but == sizeof(char) in C++ */
char x = 'x';
sizeof(x) == sizeof(char) == 1 ;
/* in both C and C++ */
```



Note that these rules also apply to constants. For example a character constant such as 'x' is an *integer* character constant of type int. In C++ however, 'x' has type char.

Integer promotions do not take place in a simple assignment expression such as char=char.

Also, integer promotions are not applied to an object used as the operand to sizeof (however they are applied if the operand is not an object, viz cannot have its address taken).

```
sizeof('x') == sizeof(int);
/* but == sizeof(char) in C++ */
char x = 'x';
sizeof(x) == sizeof(char) == 1 ;
/* in both C and C++ */
```

argument promotion

- 6.5.2.2 Function calls
  - para 6 – if the expression that denotes the called function does not include a prototype, the *integer promotions* are performed on each argument, and arguments that have type float are promoted to double. These are called the *default argument promotions*.

```
#include <stdio.h>

int main(void)
{
    return call(4, 2);
}

int call(double a, double b)
{
    return printf("%f, %f\n", a, b);
}
```

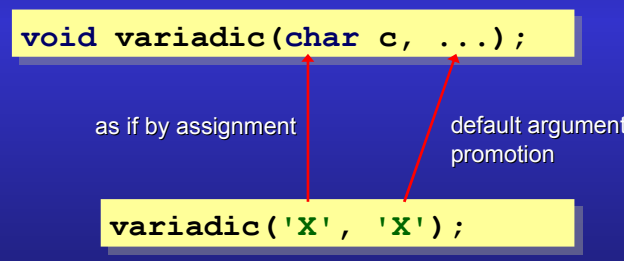
argument promotion

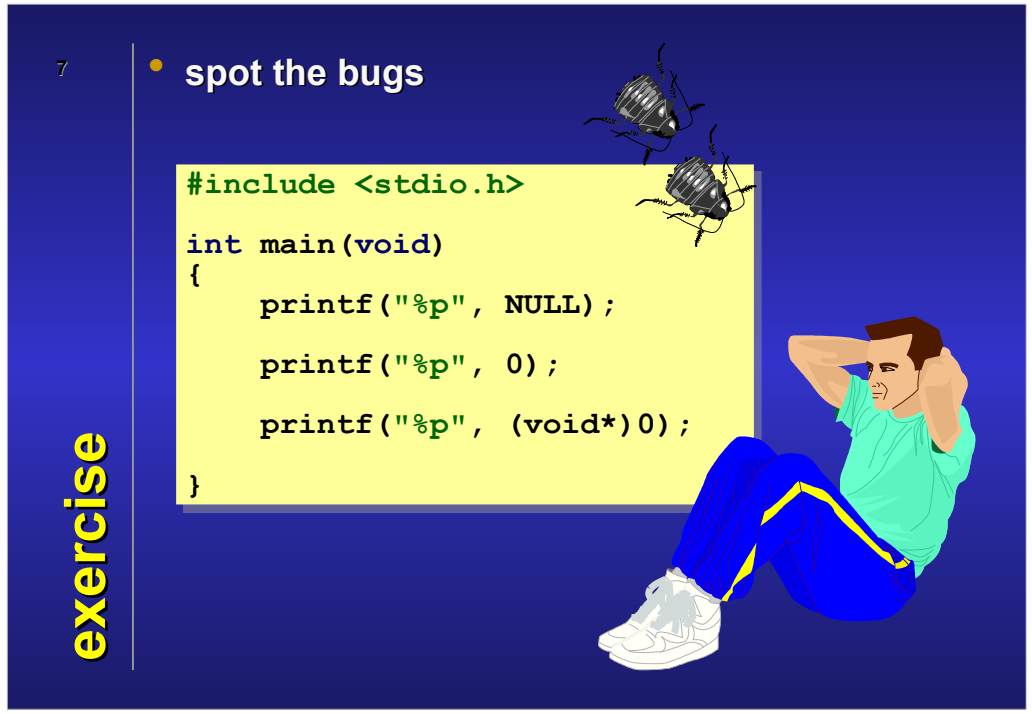
- 6.5.2.2 Function calls
  - para 7 – if the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters...

```
#include <stdio.h>

int call(double, double);

int main(void)
{
    return call(4, 2);
}

int call(double a, double b)
{
    return printf("%f, %f\n", a, b);
}
```

argument promotion

- 6.5.2.2 Function calls
  - para 7 – The ellipsis notation in a function prototype declarator causes argument type conversions to stop after the last declared parameter. The *default argument promotions* are performed on the trailing arguments.





The literal 0 (zero) is of type int. If you want to print the null pointer using the %p printf format specifier you must not write a plain zero since this will be of type int.

```
printf("%p", 0); // don't do this
```

Note that using the macro NULL is not safe either since NULL could be a macro for a plain zero:

```
printf("%p", NULL); // don't do this either
```

You must make sure the argument is a pointer:

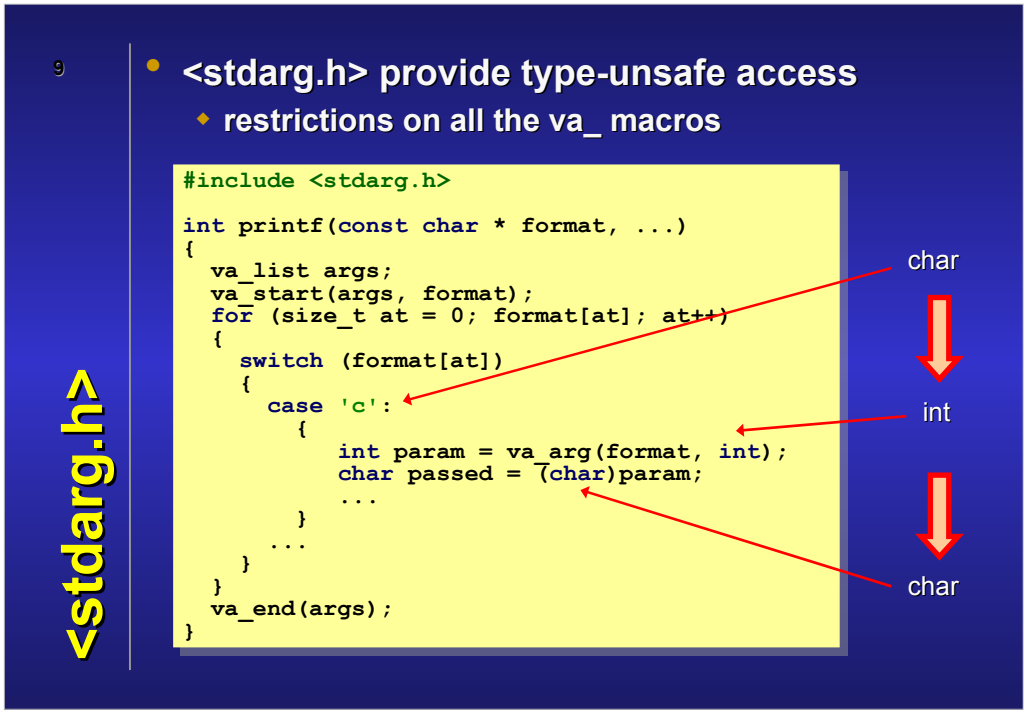
```
printf("%p", (void*)0); // do this
```

answer

- only (void*)0 is guaranteed to be a pointer
  - 0 is an int
  - NULL could be 0 too

```
#include <stdio.h>

int main(void)
{
    printf("%p", NULL);

    printf("%p", 0);

    printf("%p", (void*)0);

}
```

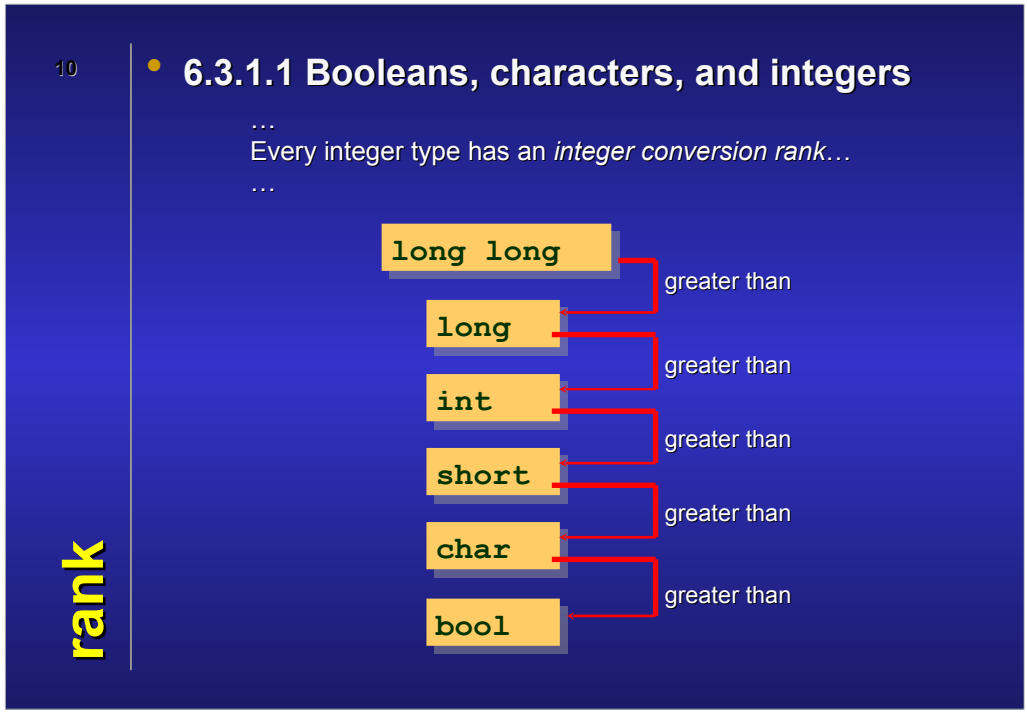




Expressions of a type smaller than int are automatically promoted to int when they bind to an ellipsis:

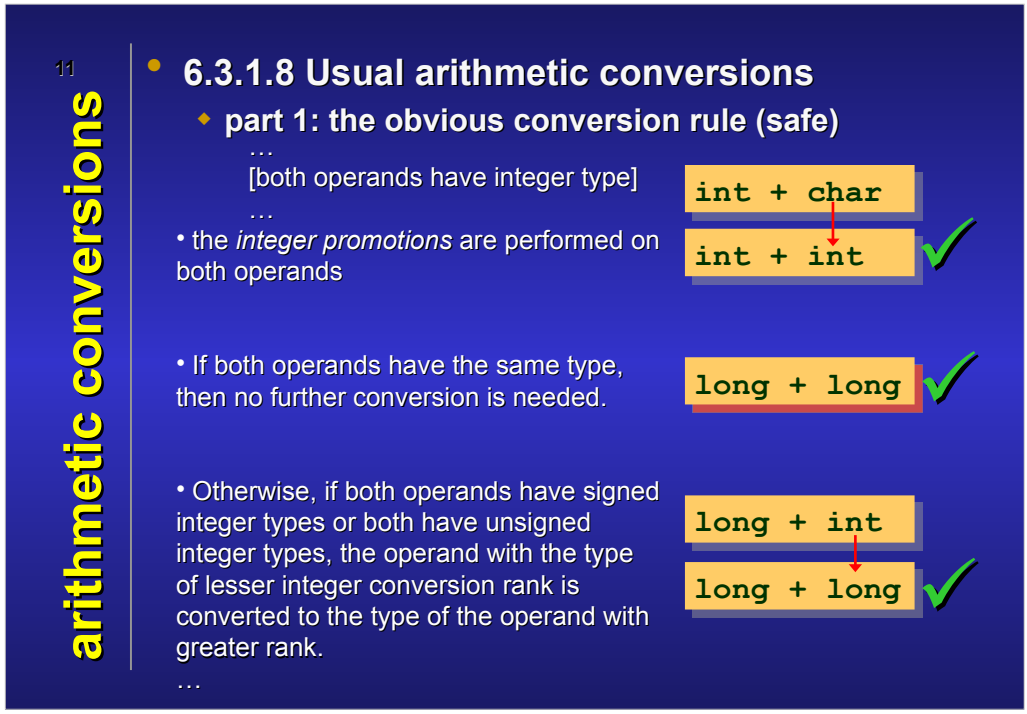
```
printf("%c", 'X'); // print a char
```

In this example 'X' will be promoted to an int – this means when the char is extracted using va_arg is must be extracted as an int and then cast to a char!

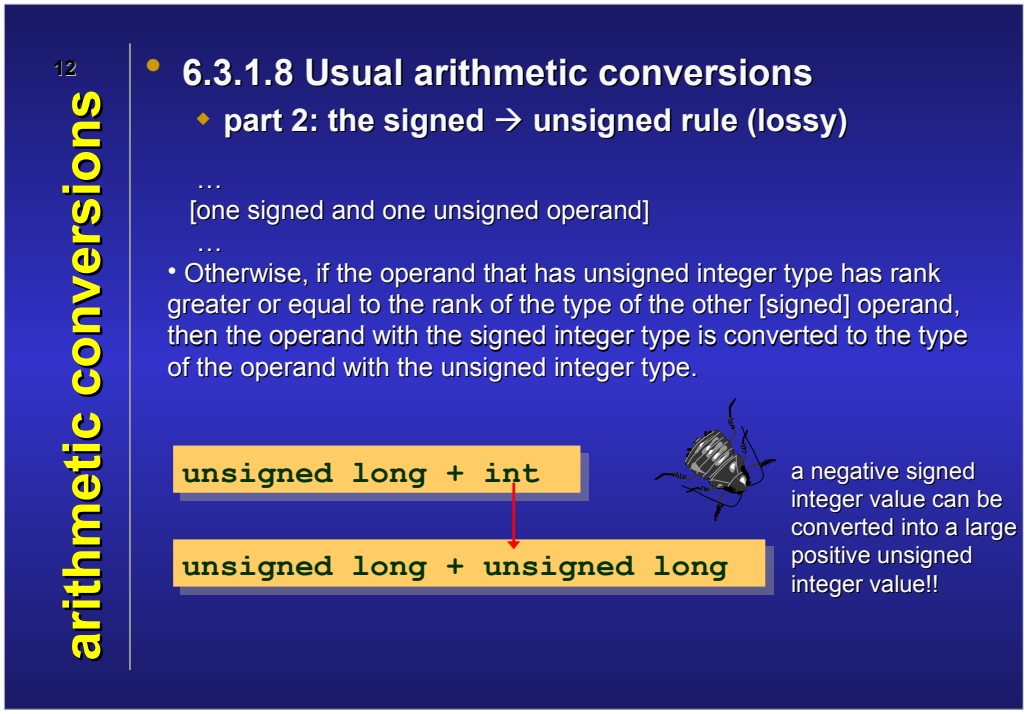


The rank of any unsigned integer type always equals the rank of the corresponding signed integer type. So for example if the rank of signed int is R then the rank of unsigned int is also R.

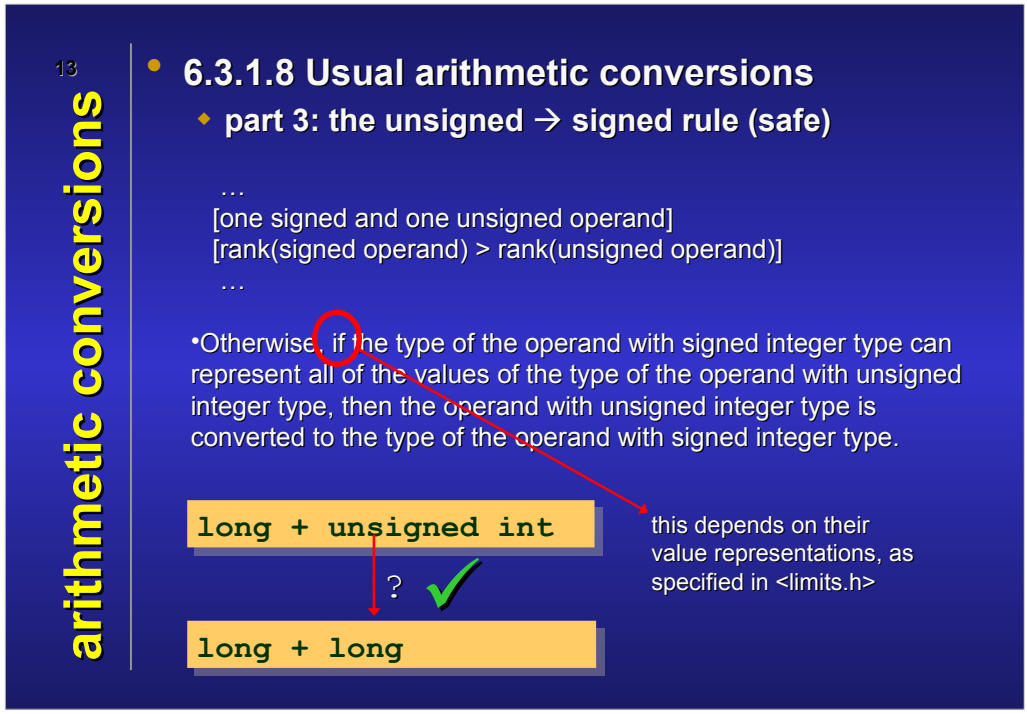
Ranking is transitive.



Note that if both operands have the same signedness, the standard guarantees that the value of the operand with lesser rank can be represented in the type of the operand with the greater rank. This preserves both the sign and the value. Note that from middle rule the ranks cannot be equal.



The reason for the bias towards converting to an unsigned integer type (rather than a signed integer type) is that the behaviour for a conversion to an unsigned integer type is always completely specified by the standard, whereas the behaviour for a conversion to a signed integer type is not.



Note that this rule is defined in terms of the type ("all the values of the type") and not in terms of the value of the individual operand.

Note that this conversion preserves the value of the converted operand but not its signedness.

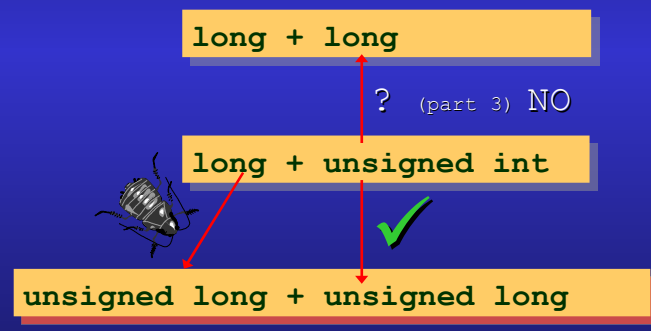
Example: If unsigned int is a 32 bit type and (signed) long is a 64 bit type then the conversion is safe and will occur (assuming no padding bits for either type).

arithmetic conversions

- 6.3.1.8 Usual arithmetic conversions
  - part 4 – the last resort rule (lossy)

…

•Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.



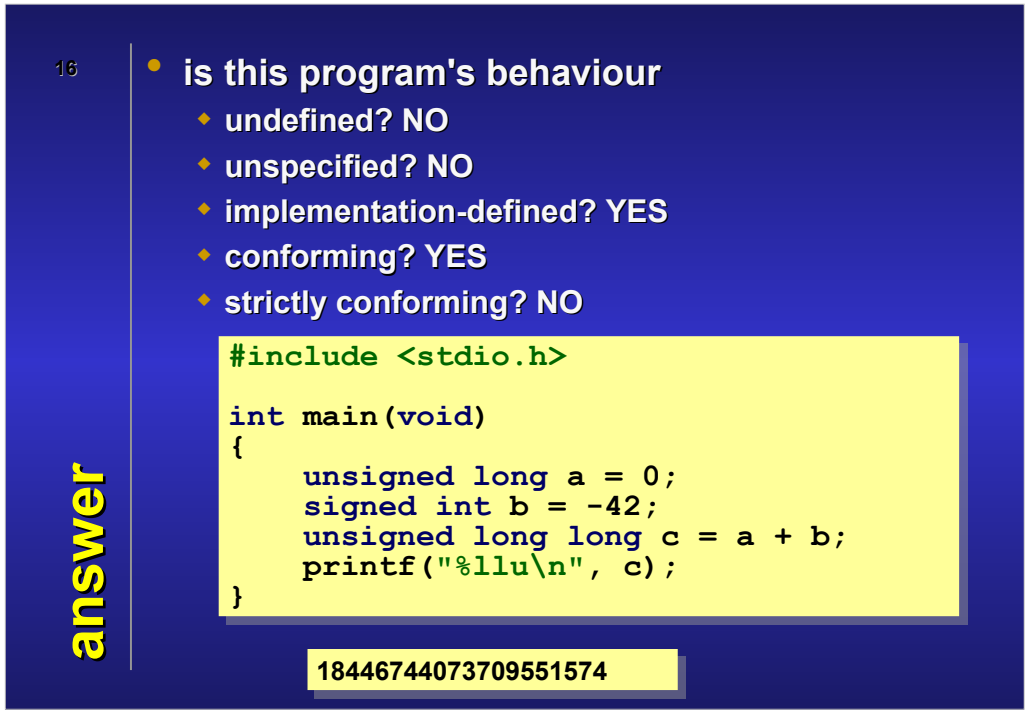
Note that in this case the type of the resulting expression is not the same as the type of either operand.

Example: If unsigned int is a 32 bit type and (signed) long is also a 32 bit type then both operands will be converted to unsigned long (again assuming no padding bits for either type).

exercise

- is this program's behaviour
  - undefined?
  - unspecified?
  - implementation-defined?
  - conforming?
  - strictly conforming?

```c
#include <stdio.h>

int main(void)
{
    unsigned long a = 0;
    signed int b = -42;
    unsigned long long c = a + b;
    printf("%llu\n", c);
}
```



The usual arithmetic conversions part 2 says that b is converted to an unsigned long.

unsigned long long c = a + (unsigned long)b;

Clearly -42 cannot be represented in an unsigned long.

The answer depends on the sizes of the integer types, but on a machine where sizeof(int)==4, sizeof(long)==8, CHAR_BIT==8 and there are no integer padding bits the answer is:

18446744073709551574

signed ↔ unsigned

- 6.3.1.3 Signed and unsigned integers
  - para 1 – When a value with integer type is converted to another integer type, other than _Bool, if the value can be represented by the new type, it is unchanged.
  - para 2 – Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
  - para 3 – Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

conversions

unary	`!`
boolean	`&& ||`

→ int 0/1

unary	`++ --`
assignment	`= *= /= %= += -= ...`
comma	`,`

unary	`~ + -`
shift	`<< >>`

conversions

- these operators perform the *usual arithmetic conversions*

arithmetic	`* / % + -`
relational	`< > <= >= == !=`
bitwise	`& ^ |`
ternary	`?:`

exercise

- what does this program print?
  - assume sizeof(short) == 2
  - assume sizeof(int) == 4

```
void exercise(void)
{
    short s = 42;

    printf("%zd\n", sizeof(s));

    printf("%zd\n", sizeof(s && s);

    printf("%zd\n", sizeof(+s));

    printf("%zd\n", sizeof(s = s));
}
```

- Click to add an outline

```
void exercise(void)
{
    short s = 42;

    printf("%zd\n", sizeof(s));

    printf("%zd\n", sizeof(s && s));

    printf("%zd\n", sizeof(+s));

    printf("%zd\n", sizeof(s = s));
}
```

answer

→ 2 4 4 2