LValues

lvalues

6.3.2.1 Lvalues, arrays, and function designators
para 1 - An lvalue is an expression with an object type or an incomplete type other than void;

If an lvalue does not designate an object when it is evaluated, the behaviour is undefined.

The name lvalue comes originally from the assignment expression E1 = E2, in which the left operand E1 is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called an "rvalue" is, in the Standard, described as the "value of the expression"

more lvalues

6.3.2.1 Lvalues, arrays, and function designators
para 1 - A modifiable-lvalue is an lvalue that

- *does not have array type*
- *does not have an incomplete type*
- *does not have a const-qualified type*
- *if it is a* struct *or* union *does not have any member...with a const-qualified type*

An lvalue might be unmodifiable because...

- it is an array that's decayed into a pointer
- it has unknown size
- it is const qualified

yet more lvalues

6.3.2.1 Lvalues, arrays, and function designators

para 2 - Except when it is the operand of

- ***the sizeof operator***
- ***the unary & operator***
- ***the ++ operator***
- ***the -- operator***
- ***the left operand of the . operator***
- ***the left operand of an assignment operator***

an lvalue that does not have an array type is converted to the value stored in the designated object (and is no longer an lvalue).

Lots of expressions start out as an lvalue and are implicitly converted into a value.

values

- these operators never yield lvalues

unary	! ~ + - ++ -- (T) &
arithmetic	* / % + -
shift	<< >>
relational	< > <= >= == !=
bitwise/boolean	& ^ |
boolean	&& || ?:
assignment	= *= /= %= += -= ...
comma	,

lvalues

- these expressions sometimes yield lvalues

primary	`identifier`
parentheses	`( )`

- these operators sometimes yield lvalues

subscript	`[ ]`
arrow	`->`
dot	`.`
dereference	`*`

identifiers

6.5.1 Primary Expressions

para 2 - An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue)

```
int function(int m)
{
    m = 42;
}
```

✓

- m designates an object and is an lvalue
- m is a modifiable lvalue
  - (not array, incomplete, const)
- m is *not* converted to a value
  - (left hand side of assignment)

parentheses

6.5.1 Primary Expressions
para 5 - A parenthesized expression ... is an lvalue ... if the unparenthesized expression is ... an lvalue

```
int function(int m)
{
      (m)++;
}
```

✓ ?

- m is an lvalue (previous slide)
- so (m) is an lvalue
- (m) is a modifiable lvalue
    - (not array, incomplete, const)
- (m) is *not* converted to a value
    - (operand of ++)

. operator

6.5.2.3 Structure and union members
para 3 - A postfix expression followed by the . operator and an identifier designates a member of a structure or union. The value is that of the named member, and is an lvalue if the first expression is an lvalue

```
void f(wibble w)
{
    w.member = 42;
}
```

✓

- w designates an object and is an lvalue
- w is a modifiable lvalue
  - (not array, incomplete, const)
- w is *not* converted to a value
  - (left operand of . operator)
- so w.member is also an lvalue
- w.member is *not* converted to a value
  - (left hand side of assignment)

exercise

- is this a conforming program?
- if not why not?
    - what clause? what paragraph? what sentence?

```
typedef struct
{
      int member;
}
wibble;

wibble f(void)
{
      wibble w;
      ...
      return w;
}

void use(void)
{
      f().member--;
}
```

answer

- no, it's not a conforming program
  - 6.3.2.1 paragraph 1, sentence 1

```
typedef struct
{
    int member;
}
wibble;

wibble f(void)
{
    wibble w;
    ...
    return w ;
}

void use(void)
{
    f().member--;
}
```

- w designates an object and is an lvalue
- w is converted to a value
- f( ) is a value so f ( ).member is also a value
- you can't do -- on a value

→ operator

6.5.2.3 Structure and union members

para 4 - A postfix expression followed by the → operator and an identifier designates a member of a structure or union. The value is that of the named member of the object to which the first expression points to, and is an lvalue.

```
void f(wibble w)
{
    wibble * ptr = &w;
    ptr->member = 42;
}
```

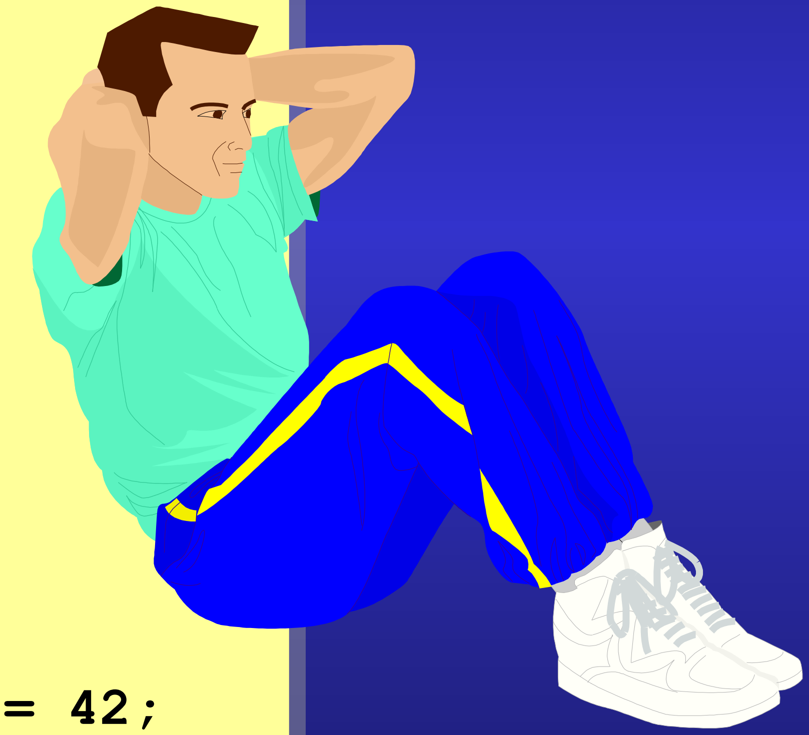
✓

- ptr→member is an lvalue
- ptr→member is a modifiable lvalue
  - (not array, incomplete, const)
- ptr→member is *not* converted to a value
  - (left hand side of assignment)

exercise

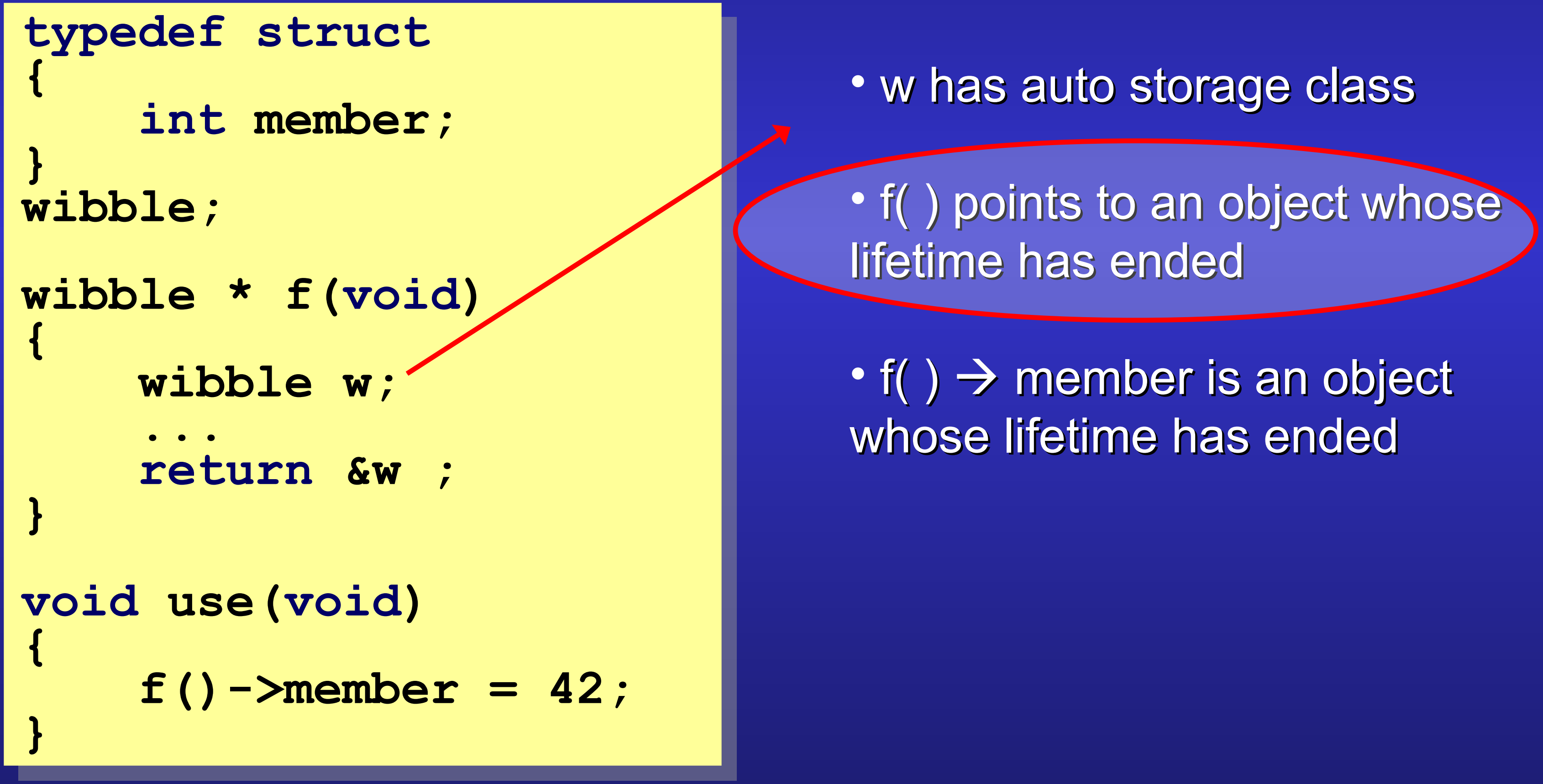
- is this a conforming program?
- if not why not?
  - what clause? what paragraph? what sentence?

```
typedef struct
{
    int member;
}
wibble;

wibble * f(void)
{
    wibble w;
    ...
    return &w;
}

void use(void)
{
    f()->member = 42;
}
```



answer

- no, it's not a conforming program
  - 6.3.2.1 paragraph 1, sentence 2



```
typedef struct
{
    int member;
}
wibble;

wibble * f(void)
{
    wibble w;
    ...
    return &w ;
}

void use(void)
{
    f()->member = 42;
}
```

- w has auto storage class
- f( ) points to an object whose lifetime has ended
- f( ) → member is an object whose lifetime has ended

* operator

6.5.3.2 Address and indirection operators
*para 4 - The unary * operator denotes indirection. If the operand ... points to an object, the result is an lvalue designating the object.*

```
void star(wibble w)
{
    wibble * ptr = &w;
    *ptr = w;
}
```

✓

- ptr points to an object
- *ptr is an lvalue
- *ptr is a modifiable lvalue
  - (not array, incomplete, const)
- *ptr is *not* converted to a value
  - (left hand side of assignment)

& operator

6.5.3.2 Address and indirection operators
para 3 - The unary & operator yields the address of its operand. ... If the operand is the result of a unary * operator, neither that operator, nor the & operator is evaluated and the result is as if both were omitted, ... and the result is not an lvalue.

```
void cancel_one_way(wibble w)
{
    *&w = w;      ✓
}
```

```
void cancel_other_way(wibble w)
{
    wibble * ptr;

      ptr = &w;   ✓

    &*ptr = &w;   ✗
}
```

The last operator controls the lvalueness

(cast) operator

6.5.4 Cast operators
para 2 - Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.
Footnote: A cast does not yield an lvalue.

```
void casting(void)
{
    int x;

        x = 42;

    (int)x = 42;

    (int[])42;

}
```

x = 42; ✓

(int)x = 42; ✗ cast is not lvalue

(int[])42; ✗ int[ ] is not scalar type

In C++ sometimes the result of a cast is an lvalue

compound literals

6.5.2.5 Compound literals
para 4 - A postfix expression that consists of a parenthesized type, followed by a brace enclosed list of initializers is a compound literal. It provides an unnamed object whose value is given by the initializer list. ...
para 5 - The result is an lvalue.

this is *not* a cast

```
void compound_literals(void)
{
    int x[] = (int[]){1,2,3};          ← ✓

    (int[]){1,2,3} [0] = 0;            ← ✓!

    (int){1} = 2;                      ← ✓!
}
```

= operators

6.5.16 Assignment operators
para 3 - An assignment stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue.

```
void assignment(void)
{
    int x;

    x = 42;          ←  ✓

     (x = 42) = 0;   ←  ✗
}
```

++ -- operators

6.5.3.1 Prefix increment and decrement operators
para 1 - The operand of the prefix increment or decrement operator <u>shall</u> ... be a <u>modifiable</u> <u>lvalue</u>.
para 2 - ... The result is the new <u>value</u> of the operand after incrementation. The expression ++E is equivalent to (E+=1)

```
void increment(void)
{
    int x = 0;

    x++ = 42;   ←————————— ✗

    ++x = 42;   ←————————— ✗
}
```

++x is accidentally an lvalue in C++

LValues

lvalues

- Click to add an outline

6.3.2.1 Lvalues, arrays, and function designators
para 1 - An lvalue is an expression with an object type or an incomplete type other than void;

If an lvalue does not designate an object when it is evaluated, the behaviour is undefined.

The name lvalue comes originally from the assignment expression E1 = E2, in which the left operand E1 is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called an "rvalue" is, in the Standard, described as the "value of the expression"

more lvalues

- Click to add an outline

6.3.2.1 Lvalues, arrays, and function designators
para 1 - A modifiable-lvalue is an lvalue that
- *does not have array type*
- *does not have an incomplete type*
- *does not have a const-qualified type*
- *if it is a* struct *or* union *does not have any member...with a const-qualified type*

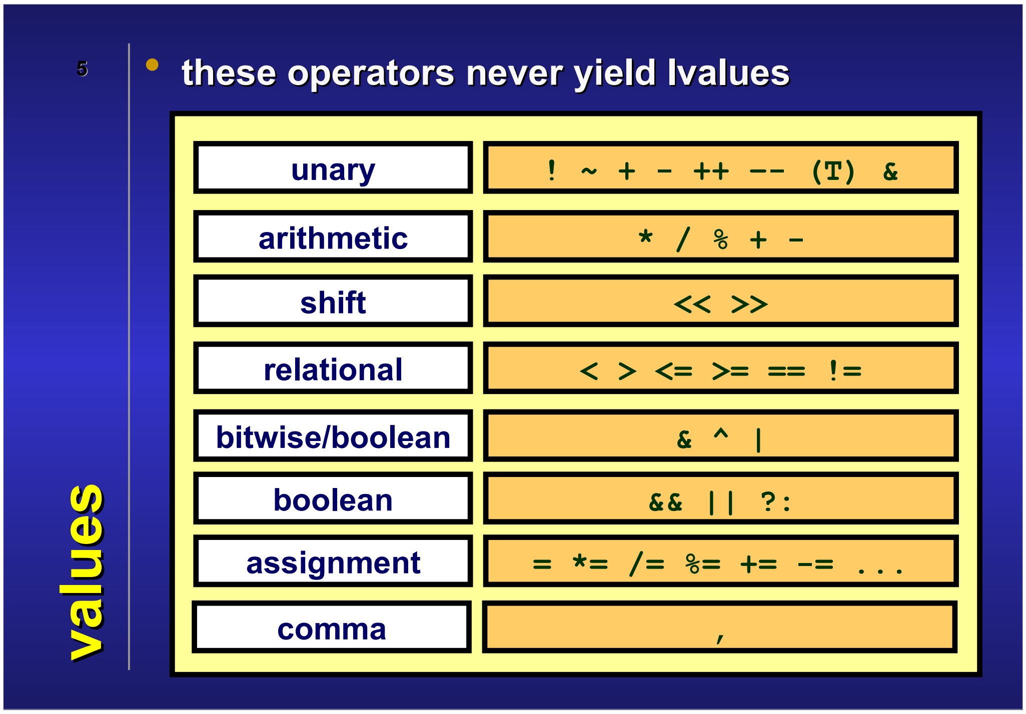
An lvalue might be unmodifiable because...
- it is an array that's decayed into a pointer
- it has unknown size
- it is const qualified

yet more lvalues

- Click to add an outline

6.3.2.1 Lvalues, arrays, and function designators
para 2 - Except when it is the operand of
• the sizeof operator
• the unary & operator
• the ++ operator
• the -- operator
• the left operand of the . operator
• the left operand of an assignment operator
an lvalue that does not have an array type is converted to the value stored in the designated object (and is no longer an lvalue).

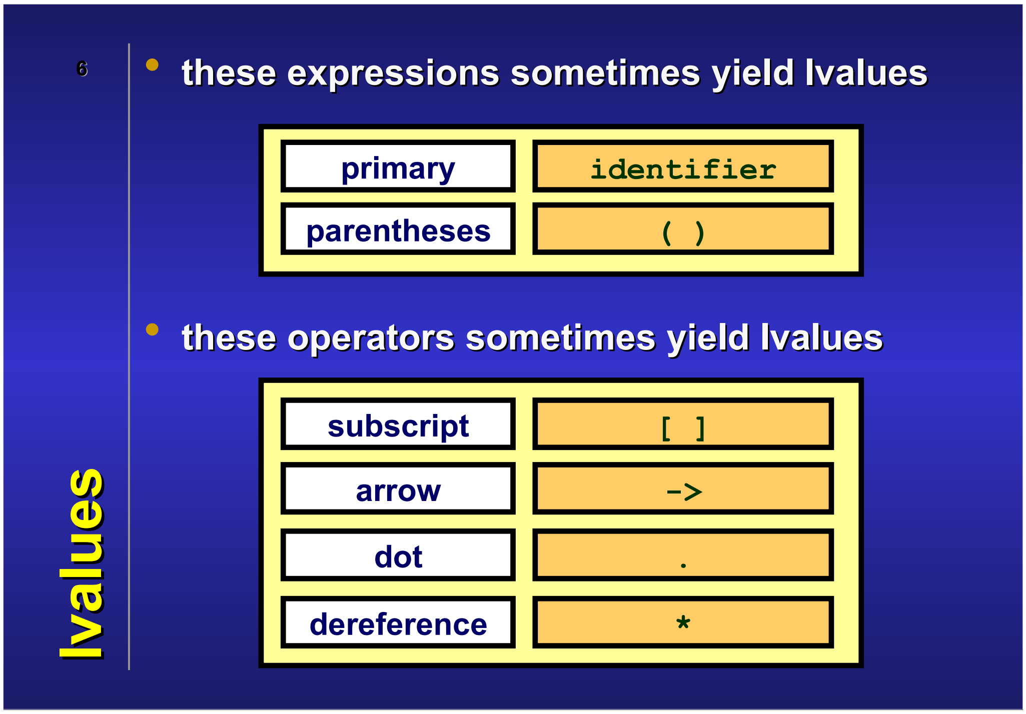
Lots of expressions start out as an lvalue and are implicitly converted into a value.



values

- these operators never yield lvalues

unary	`! ~ + - ++ -- (T) &`
arithmetic	`* / % + -`
shift	`<< >>`
relational	`< > <= >= == !=`
bitwise/boolean	`& ^ |`
boolean	`&& || ?:`
assignment	`= *= /= %= += -= ...`
comma	`,`



lvalues

- these expressions sometimes yield lvalues

primary	`identifier`
parentheses	`( )`

- these operators sometimes yield lvalues

subscript	`[ ]`
arrow	`->`
dot	`.`
dereference	`*`

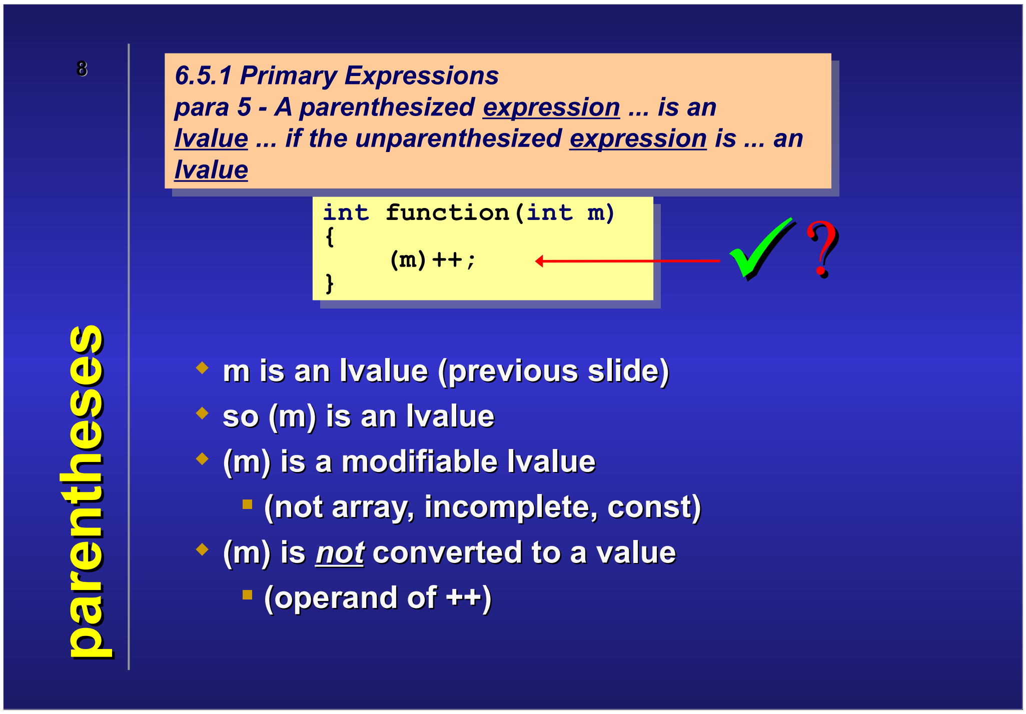
identifiers

6.5.1 Primary Expressions
para 2 - An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue)

```
int function(int m)
{
    m = 42;
}
```

✓

- m designates an object and is an lvalue
- m is a modifiable lvalue
  - (not array, incomplete, const)
- m is ***not*** converted to a value
  - (left hand side of assignment)



8

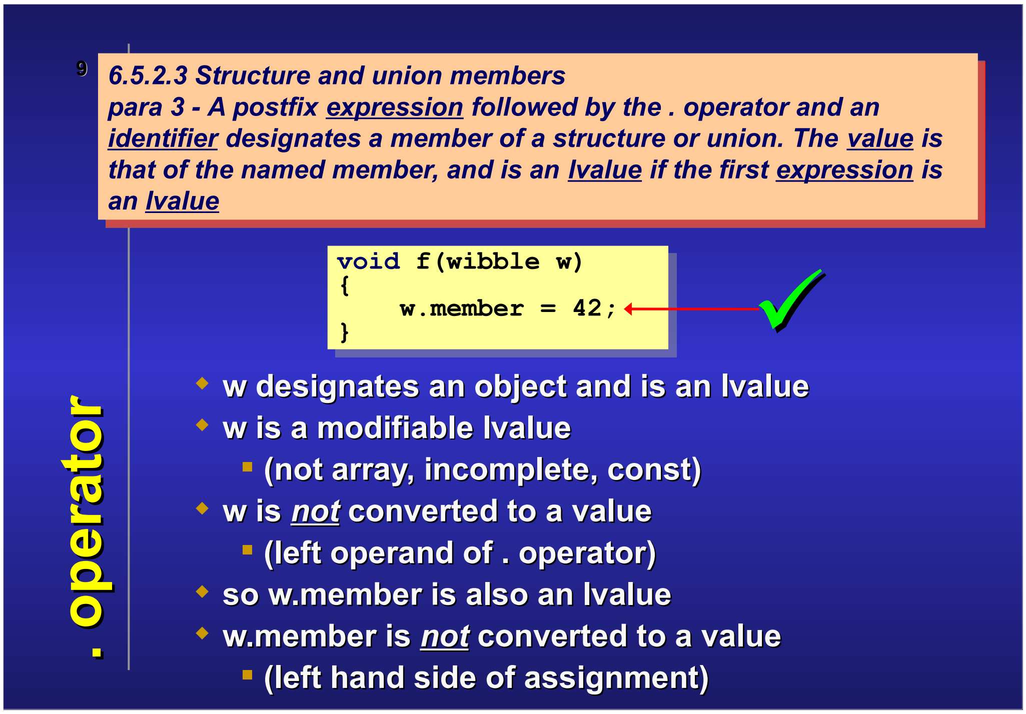
parentheses

6.5.1 Primary Expressions
para 5 - A parenthesized expression ... is an lvalue ... if the unparenthesized expression is ... an lvalue

```
int function(int m)
{
     (m)++;
}
```

← ✓?

- m is an lvalue (previous slide)
- so (m) is an lvalue
- (m) is a modifiable lvalue
  - (not array, incomplete, const)
- (m) is *not* converted to a value
  - (operand of ++)



9

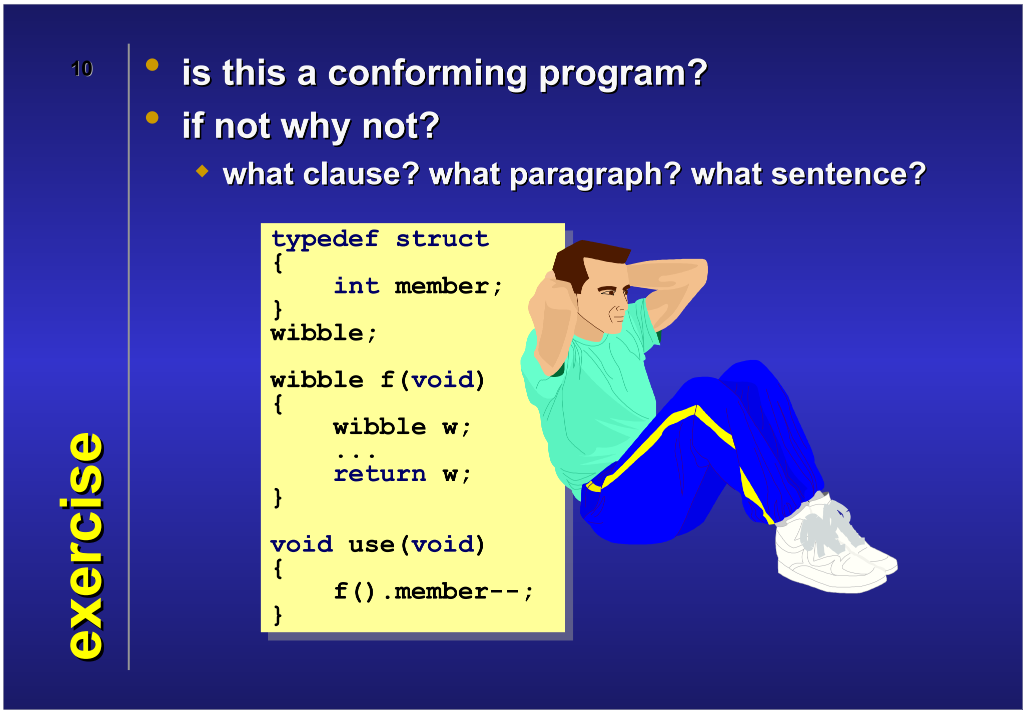
. operator

6.5.2.3 Structure and union members
para 3 - A postfix __expression__ followed by the . operator and an __identifier__ designates a member of a structure or union. The __value__ is that of the named member, and is an __lvalue__ if the first __expression__ is an __lvalue__

```
void f(wibble w)
{
    w.member = 42;
}
```

✓

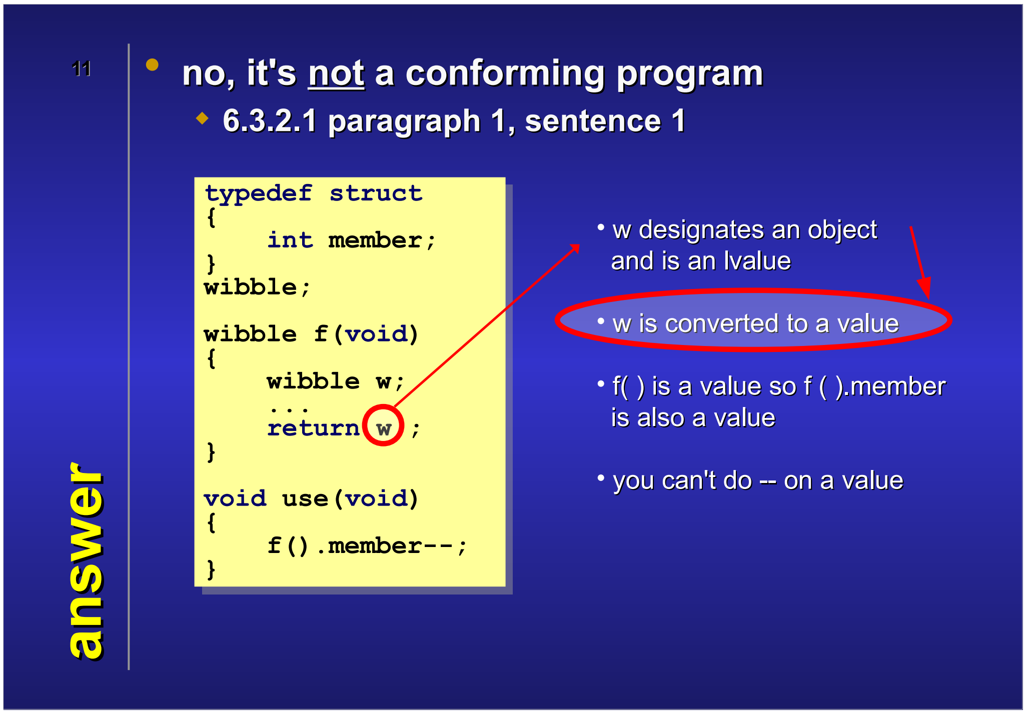
- w designates an object and is an lvalue
- w is a modifiable lvalue
  - (not array, incomplete, const)
- w is *__not__* converted to a value
  - (left operand of . operator)
- so w.member is also an lvalue
- w.member is *__not__* converted to a value
  - (left hand side of assignment)



exercise

- is this a conforming program?
- if not why not?
  - what clause? what paragraph? what sentence?

```
typedef struct
{
    int member;
}
wibble;

wibble f(void)
{
    wibble w;
    ...
    return w;
}

void use(void)
{
    f().member--;
}
```

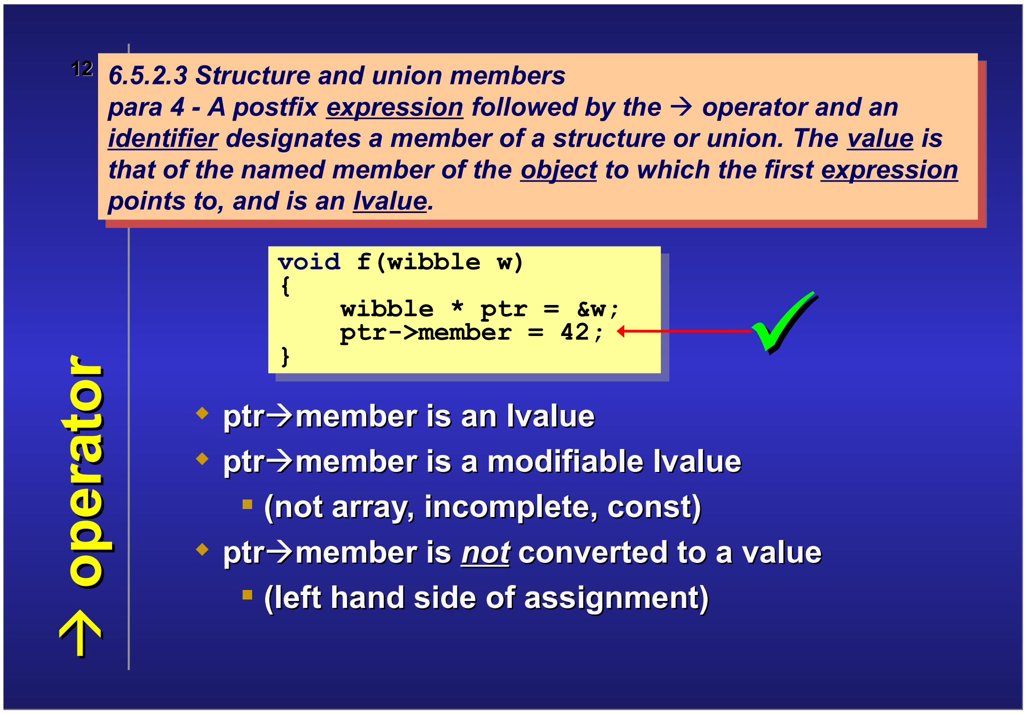


answer

- no, it's <u>not</u> a conforming program
  - 6.3.2.1 paragraph 1, sentence 1

```
typedef struct
{
    int member;
}
wibble;

wibble f(void)
{
    wibble w;
    ...
    return w ;
}

void use(void)
{
    f().member--;
}
```

- w designates an object and is an lvalue
- w is converted to a value
- f( ) is a value so f ( ).member is also a value
- you can't do -- on a value

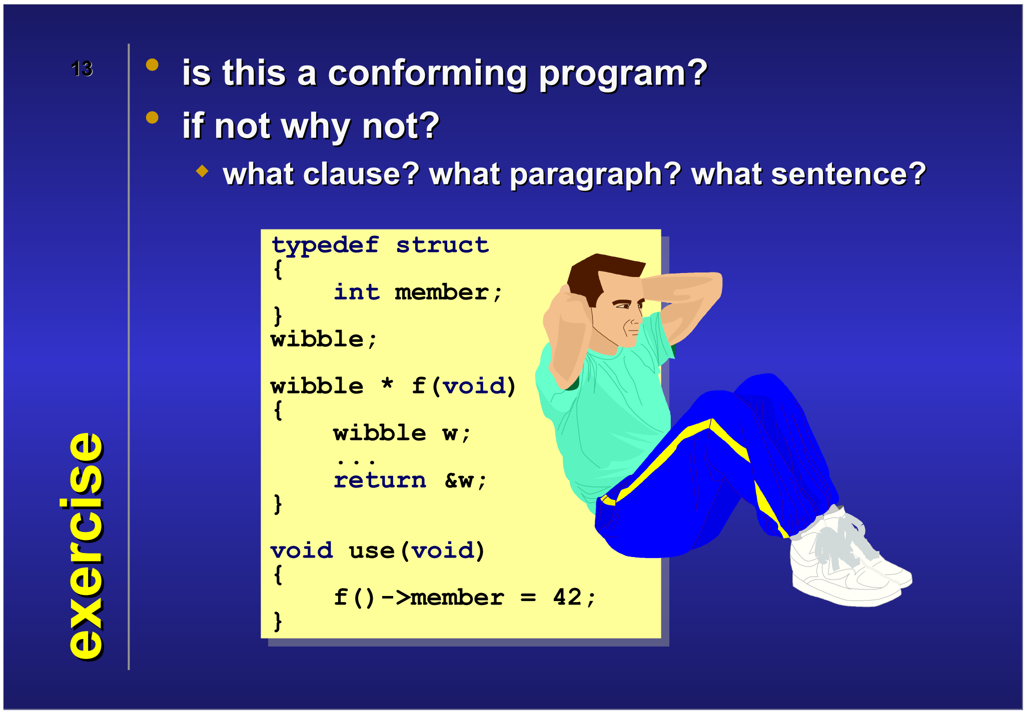


→ operator

6.5.2.3 Structure and union members
para 4 - A postfix expression followed by the → operator and an identifier designates a member of a structure or union. The value is that of the named member of the object to which the first expression points to, and is an lvalue.

```
void f(wibble w)
{
    wibble * ptr = &w;
    ptr->member = 42;
}
```

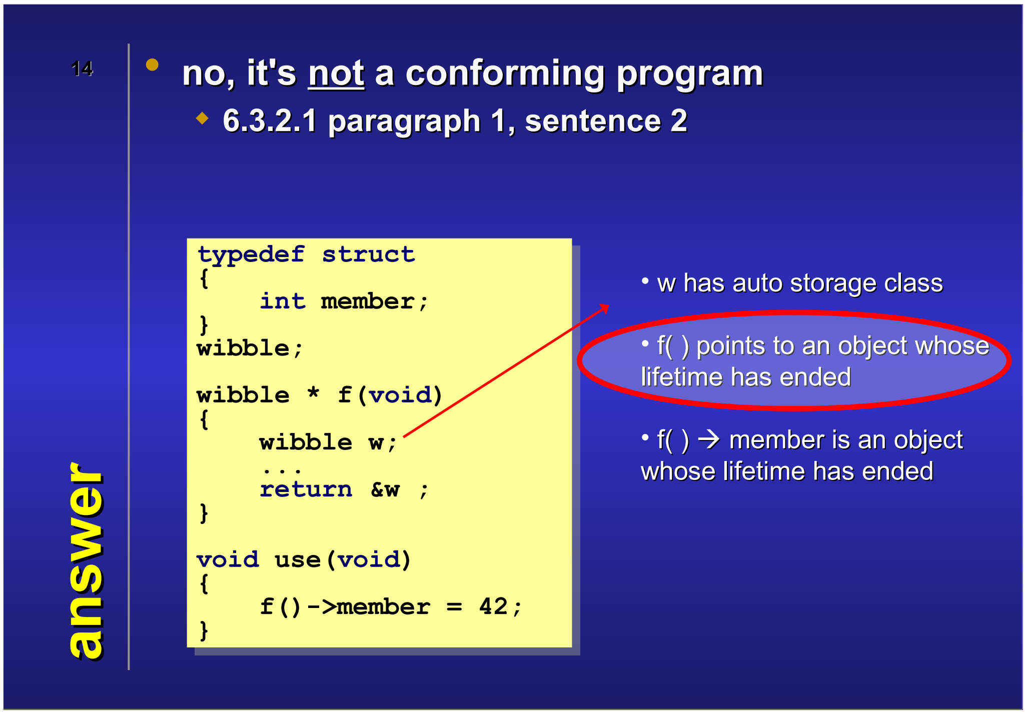
- ptr→member is an lvalue
- ptr→member is a modifiable lvalue
  - (not array, incomplete, const)
- ptr→member is *not* converted to a value
  - (left hand side of assignment)



exercise

- is this a conforming program?
- if not why not?
  - what clause? what paragraph? what sentence?

```
typedef struct
{
    int member;
}
wibble;

wibble * f(void)
{
    wibble w;
    ...
    return &w;
}

void use(void)
{
    f()->member = 42;
}
```



14

answer

- no, it's not a conforming program
  - 6.3.2.1 paragraph 1, sentence 2

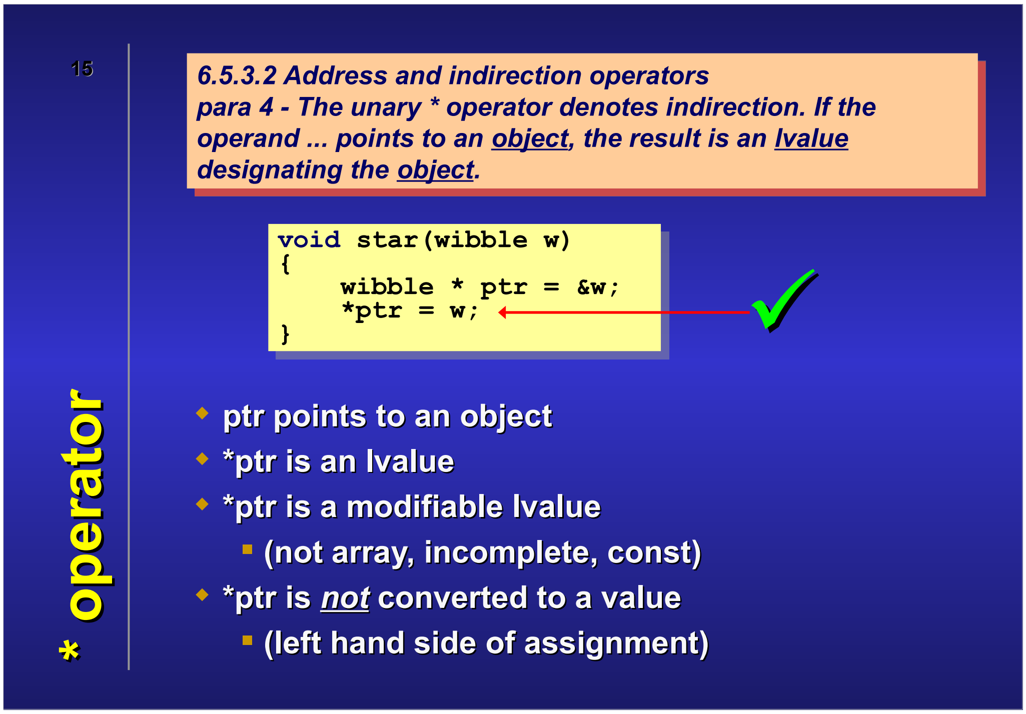
```
typedef struct
{
    int member;
}
wibble;

wibble * f(void)
{
    wibble w;
    ...
    return &w ;
}

void use(void)
{
    f()->member = 42;
}
```

- w has auto storage class
- f( ) points to an object whose lifetime has ended
- f( ) → member is an object whose lifetime has ended

6.5.3.2 Address and indirection operators

The unary * operator denotes indirection. ... If an invalid value has been assigned to the pointer, the behaviour of the unary * operator is undefined. 83)

83) Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of the object pointed to, and the address of an object after the end of its lifetime.



* operator

6.5.3.2 Address and indirection operators
*para 4 - The unary * operator denotes indirection. If the operand ... points to an object, the result is an lvalue designating the object.*

```
void star(wibble w)
{
    wibble * ptr = &w;
    *ptr = w;                ✓
}
```

- ptr points to an object
- *ptr is an lvalue
- *ptr is a modifiable lvalue
  - (not array, incomplete, const)
- *ptr is ***not*** converted to a value
  - (left hand side of assignment)

& operator

6.5.3.2 Address and indirection operators
*para 3 - The unary & operator yields the address of its operand. ... If the operand is the result of a unary * operator, neither that operator, nor the & operator is evaluated and the result is as if both were omitted, ... and the result is <u>not</u> an <u>lvalue</u>.*

```
void cancel_one_way(wibble w)
{
    *&w = w;            ✓
}
```

```
void cancel_other_way(wibble w)
{
    wibble * ptr;

      ptr = &w;         ✓

    &*ptr = &w;         ✗
}
```

The last operator controls the lvalueness

(cast) operator

- ***6.5.4 Cast operators***
  para 2 - Unless the type name specifies a void type, the type name <u>shall</u> specify qualified or unqualified <u>scalar</u> type and the operand <u>shall</u> have scalar type.
  Footnote: A cast does not yield an <u>lvalue</u>.

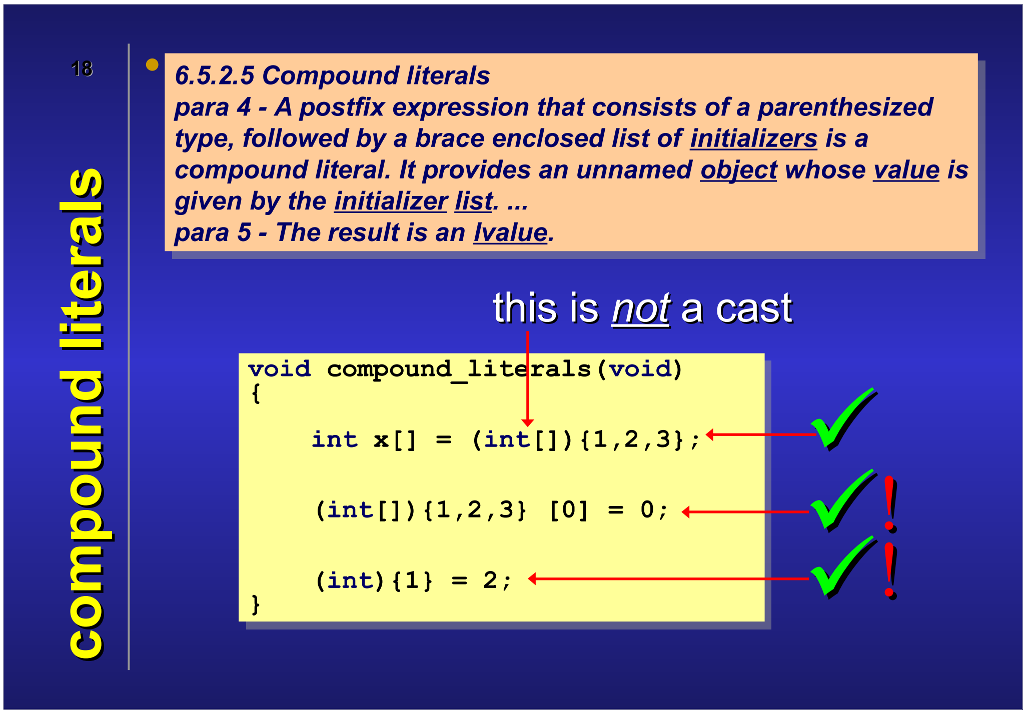
```
void casting(void)
{
    int x;
          x = 42;
     (int)x = 42;
     (int[])42;
}
```

x = 42; ✓

(int)x = 42; ✗ cast is not lvalue

(int[])42; ✗ int[ ] is not scalar type

In C++ sometimes the result of a cast <u>is</u> an lvalue



18

compound literals

- *6.5.2.5 Compound literals*
  para 4 - A postfix expression that consists of a parenthesized type, followed by a brace enclosed list of initializers is a compound literal. It provides an unnamed object whose value is given by the initializer list. ...
  para 5 - The result is an lvalue.

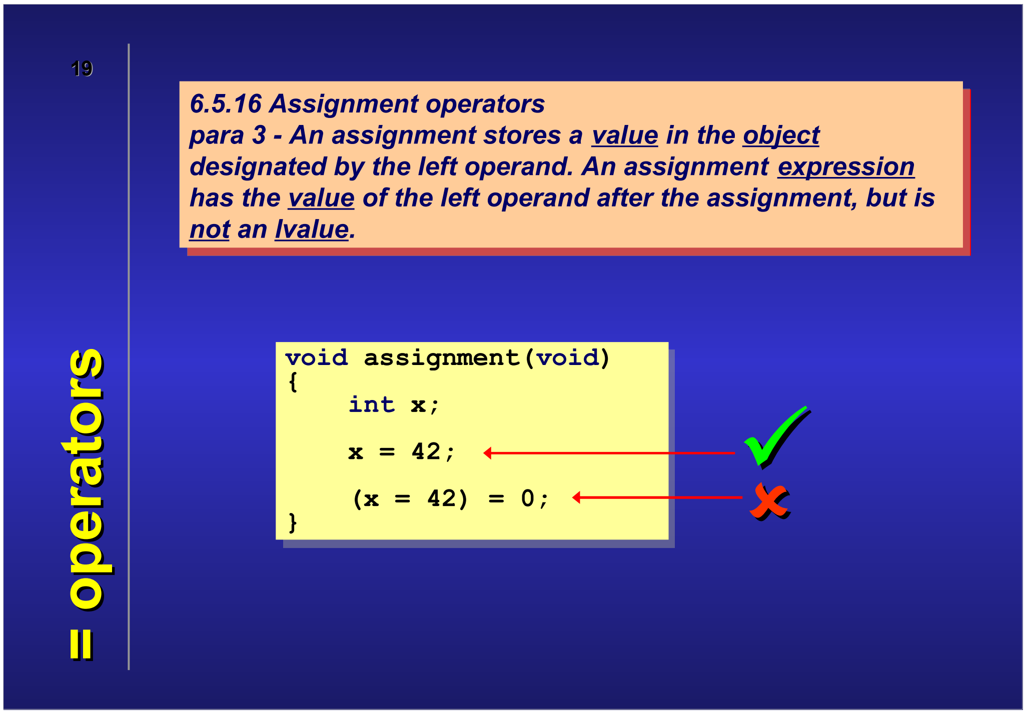
this is *not* a cast

```
void compound_literals(void)
{
    int x[] = (int[]){1,2,3};        ✓

    (int[]){1,2,3} [0] = 0;          ✓!

    (int){1} = 2;                    ✓!
}
```

(int)1 = 2; // not allowed

(int)(1) = 2; // not allowed

(int){1} = 2; // allowed!

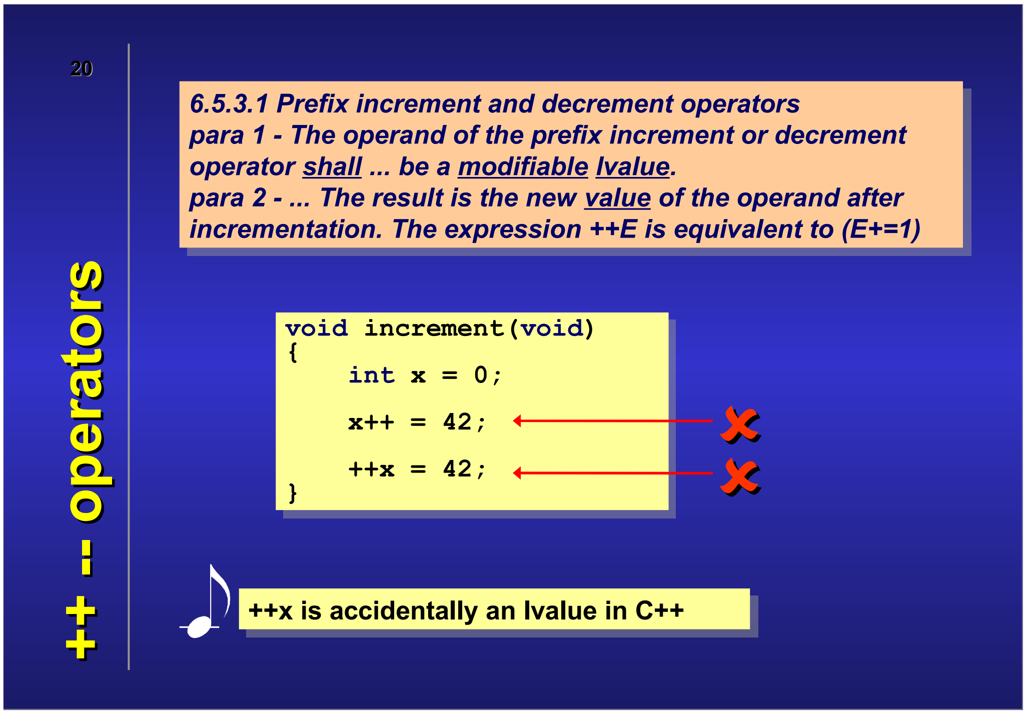


= operators

6.5.16 Assignment operators

para 3 - An assignment stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue.

```
void assignment(void)
{
     int x;

     x = 42;

      (x = 42) = 0;
}
```



++ -- operators

6.5.3.1 Prefix increment and decrement operators
para 1 - The operand of the prefix increment or decrement operator <u>shall</u> ... be a <u>modifiable</u> <u>lvalue</u>.
para 2 - ... The result is the new <u>value</u> of the operand after incrementation. The expression ++E is equivalent to (E+=1)

```
void increment(void)
{
    int x = 0;

    x++ = 42;

    ++x = 42;
}
```

++x is accidentally an lvalue in C++