C Advanced

A two day course for professional programmers

Agenda

Day 1: 0930-1530
Standard
Incomplete Types
Exercise and Lunch
Structs and Enums
Objects
Sequencing
Exercise

Day 2: 0930-1530
Representation
Lvalues
Arrays
Exercise and Lunch
Pointers
Conversions
Threading

Standard

chapter and verse

- **http://www.open-std.org/jtc1/sc22/wg14**

Joint Technical Committee 1

Sub Committee 22

Working Group 14

John Wiley: ISBN 0470845732

- **http://www.knosof.co.uk/cbook/cbook.html**
  - **commentary on every sentence in the c99 standard+**
- **Peter Norvig advises**
  - **get involved in a language standardization**
  - **get off the language standardization asap :-)**

Clauses

3. Terms, definitions, and symbols
4. Conformance
5. Environment
5.1.1.3 Diagnostics
5.1.2.3 Program execution
1. Language
6.2 Concepts
6.2.5 Types
6.2.6 Representations of types
6.3 Conversions
6.3.2.1 Lvalues, arrays, and function designators
6.5 Expressions
6.7 Declarations
6.8 Statements and blocks

These are the main clauses we will be looking at

spirit of C

- trust the programmer
  - let them do what needs to be done
  - the programmer is in charge not the compiler
- keep the language small and simple
  - provide only one way to do an operation
  - new inventions are not entertained
- make it fast, even if its not portable
  - target efficient code generation
  - int promotion rules
  - sequence points
- rich expressions
  - lots of operators
  - expressions combine into larger expressions

3.12 implementation
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports executions of functions in, a particular execution environment.

1. Environment
para 1 - An implementation translates C source files and executes C programs in two data-processing system environments, which will be called the translation environment and the execution environment...

Implementation means compiler/translator

implementation

3.4 behavior
external appearance or action

3.4.1 implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made.

3.4.3 undefined behavior
behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirement.

3.4.4 unspecified behavior
use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and impose no further requirements on which is chosen in any instance.

behavior

- **behaviour in Java/C# is completely defined**
- **a lot of behaviour in C is not - why not?**

> ***3.4 behavior***
> ***...***
>
> ***3.4.1 implementation-defined behavior***
> ***...***
>
> ***3.4.3 undefined behavior***
> ***...***
>
> ***3.4.4 unspecified behavior***
> ***...***

question

diagnostics

3.8 constraint
restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted.

3.10 diagnostic message
message belonging to an implementation-defined subset of the implementation's message output.

5.1.1.3 Diagnostics
para 1 - A conforming implementation shall produce at least one diagnostic message ... if a ... translation unit contains a violation of any syntax rule or constraint ...
Diagnostic messages need not be produced in other circumstances.

- **an implementation is required to produce a diagnostic message for a *syntax* violation***

6.5.16 Assignment operators

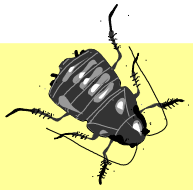
Syntax

assignment-operator:

= *= /= %= += -= <<= >>= &= ^= |=

syntax.c

```
int x = 0;
x := 42;
```



```
>gcc syntax.c
  → error: expected expression before '=' token
```

diagnostics

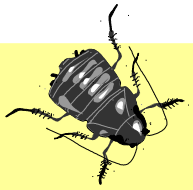
*if it is the first violation

- **an implementation is required to produce a diagnostic message for a *constraint* violation***

6.5.16 Assignment operators
Constraints
As assignment operator shall have a modifiable lvalue as its left operand.

constraints.c

```
const int x = 0;
x = 42;
```



```
>gcc constraints.c
  → error: assignment of read-only variable 'x'
```

diagnostics

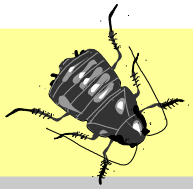
*if it is the first violation

- **an implementation is *not* required to produce a diagnostic message for any other violatio**

> ***5.1.2.3 Program execution***
> ***para 2 – At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.***

semantics.c

```
int x = 0;
x = x++;
```



```
>gcc semantics.c
  → ...no diagnostic...
```

```
>gcc –Wall semantics.c
  → warning: operation on 'x' may be undefined
```

conformance

4. Conformance
para 1 - ... "shall" is to be interpreted as a requirement on an implementation or on a program; conversely, "shall not" is to be interpreted as a prohibition.

para 2 - If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior", or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined"

Annex J.2 lists 190+ undefined behaviours.

conformance

4. Conformance
para 5 - A strictly conforming program shall use only those features of the language and library specified in this International Standard.
It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior and shall not exceed any minimum implementation limit.

para 6 - A conforming ... implementation shall accept any strictly conforming program.

para 7 - A conforming program is one that is acceptable to a conforming implementation.

The standard never writes about "correct" programs.
The only proper terms are conformance/conforming.

- # 3. Terms, definitions, and symbols
  - ## 3.1 access

Execution time action to read or modify the value of an object.
NOTE 1: where only one of these two actions is meant, "read" or "modify" is used.
NOTE 2: "Modify" includes the case where the new value being stored in the same as the previous value.
NOTE 3: Expressions that are not evaluated do not access objects.

access

What use is an expression that is not evaluated?

3. Terms, definitions, and symbols

3.15 object

A region of data storage in the execution environment, the contents of which can represent values

What C calls objects, other languages call variables.

An object's representation is held in a contiguous sequence of bytes.

An object is addressable.

objects

3. Terms, definitions, and symbols

3.15 object

NOTE: when referenced, an object may be interpreted as having a particular type.

A reference that does not interpret the contents of an object, for example as an argument to memcpy, does not need to interpret it as having a particular type.

Objects have storage duration but no linkage. Identifiers have linkage but no storage duration. Constants have only a value.

objects

3. Terms, definitions, and symbols

3.6 byte

Addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

Each byte is at least 8 bits wide.
A char, whether signed or unsigned, occupies exactly one byte.

An implementation cannot hide the internal bits of an object.

Why not?

bytes

- # 3. Terms, definitions, and symbols
  - ## 3.19 value

Precise meaning of the contents of an object when interpreted as having a specific type.

A literal also has a value. Its type is determined by both the lexical form of the token and its numeric value.

values

- 6. Language
  - 6.2 Concepts
    - 6.2.5 Types

para 1 - The meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it.

para 1 – Types are partitioned into
- ***object types (types that describe objects)***
- ***function types (types that describe functions), and***
- ***At various points within a translation unit an object may be incomplete (lacking sufficient information to determine the size of objects of that type) or complete (having sufficient information).***

types

C has a relatively weak type system. C++ added much greater support for "typing".

- 6. Language
  - 6.5 Expressions

> *para 1 - An expression is a sequence of operators and operands that ...*
> • *specifies computation of a value, or*
> • *that designates an object or a function, or*
> • *that generates side effects, or*
> • *that performs a combination thereof.*

expressions

- rich expressions
  - lots of operators
  - expressions combine into larger express

- 5. Environment
  - 5.1 Conceptual models
    - 5.1.2 Execution environment
      - 5.1.2.3 Program execution

para 2 –
• Accessing a volatile object,
• modifying an object,
• modifying a file,
• or calling a function that does any of those operations are all side effects which are changes in the state of the execution environment.

side effects

- 5. Environment
  - 5.1 Conceptual models
    - 5.1.2 Execution environment
      - 5.1.2.3 Program execution

sequence points

para 3 – The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced-before every value computation and side effect associated with B. (A summary of sequence points is given in annex C.)

Why do so few expressions cause a sequence point?

- 5. Environment
  - 5.1 Conceptual models
    - 5.1.2 Execution environment
      - 5.1.2.3 Program execution

para 3 – In the abstract machine, all expressions are evaluated as specified by the semantics.
An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

Why does the as-if rule exist?

as-If

contents

- Spirit of C
- Main clauses
- Behaviour
- Diagnostics
- Conformance
- Key terms
- Types
- Expressions
- Program execution

Incomplete Types

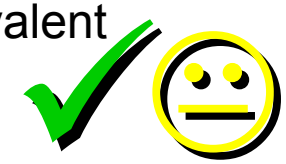
- **6.2.5 Types**
  - **para 1 - Types are partitioned into**
    - ***object types*** **(types that describe objects),**
    - ***function types*** **(types that describe functions),**
    - **At various points within a translation unit an object may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).**
- **6.2.5 Types**
  - **para 19 – The void type ... is an *incomplete type* that cannot be completed.**
  - **para 22 – An array of unknown size is an *incomplete type*.**
  - **para 22 – A structure or union type of unknown content is an *incomplete type*.**

6.7.6.3 Function declarators

- para 14 – An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters.

```
int f(void) { ... }
int f()     { ... }
```

equivalent



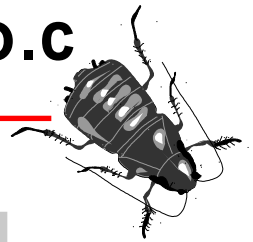
- para 14 – The empty list in a function declarator that is<u>not</u>part of a definition of that function specifies that no information about the number o

```
int f(void);
int f();
```

supplied.

not equivalent

- when do you get a diagnostic?
  - gcc clo.c
  - gcc –Wall clo.c
  - gcc –Wmissing-prototypes clo.c

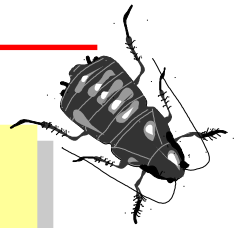


```
int f();

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

int f(int value)
{
    return value;
}
```

clo.c

exercise-1

- when do you get a diagnostic?
  - gcc clo.c→ NO
  - gcc –Wall clo.c→ NO! why not?
  - gcc –Wmissing-prototypes clo.c→ YES



```
int f();

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

int f(int value)
{
    return value;
}
```

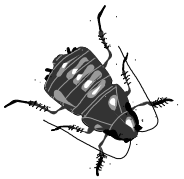
clo.c

answer-1

- why do you still get a diagnostic?
  - gcc –Wmissing-prototypes clo.c
    → warning: no previous prototype for 'f'

```
int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

clo.c



exercise-2

- because f() is not declared with external linkage and there is no previous function declaration
  - option : if f() is not used in another source file then define it with internal linkage

```
static int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

clo.c

answer-2a

- because f() is not declared with external linkage and there is no previous function declaration
  - option : if f() is used in aother source file then declare it in a header and define it with external linkage

```c
int f(int value);
```

clo.h

```c
#include "clo.h"

int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

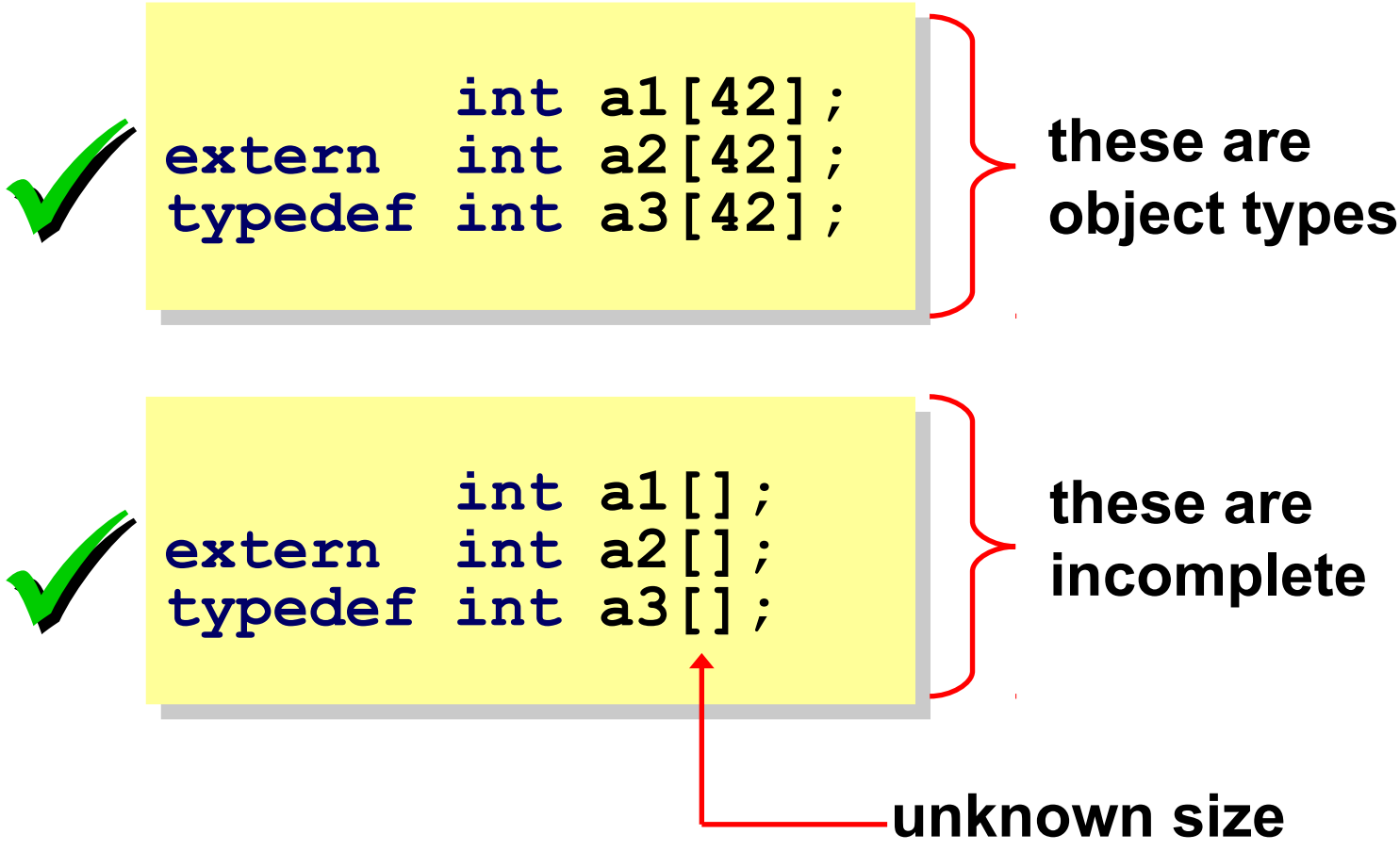
clo.c

answer-2b

6.2.5 Types

- para 22 – An array of unknown size is an incomplete type.

incomplete arrays



- are these fragments conforming or not?

```
struct wibble;
```

```
          struct wibble w1[42];
extern    struct wibble w2[42];
typedef   struct wibble w3[42];
```

```
          struct wibble w1[];
extern    struct wibble w2[];
typedef   struct wibble w3[];
```

exercise

- **neither are conforming!**

```
struct wibble;
```

```
          struct wibble w1[42];
extern    struct wibble w2[42];
typedef   struct wibble w3[42];
```

```
          struct wibble w1[];
extern    struct wibble w2[];
typedef   struct wibble w3[];
```

answer

Incomplete arrays can be incomplete in their size *but not* in their element type

- 6.7.6.2 Array declarators
  - para 1 – the element type shall not be an incomplete type

- are these fragments conforming or not?

```
struct wibble;

struct wibble x[4];

struct wibble x[];

struct wibble * ptr;
```

exercise

```
struct wibble;
```

```
struct wibble x[4];
```

```
struct wibble x[];
```

```
struct wibble * ptr;
```

answer

- are these code fragments conforming?
- if not, why not?

```
int table[] =
{
    1, 2, 3,
    4, 5, 6
};
```

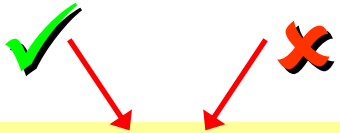
```
int table[][] =
{
    { 1, 2, 3 },
    { 4, 5, 6 }
};
```

exercise

answer

```
int table[] =
{
    1, 2, 3,
    4, 5, 6
};
```

this is conforming



```
int table[][] =
{
    { 1, 2, 3 },
    { 4, 5, 6 }
};
```

this is not conforming

- ## 6.7.6.3 Function declarators
  - **para 12 - If the function declarator is not part of a definition of that function, parameters may have incomplete types...**

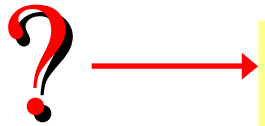
forward declaration

no #include

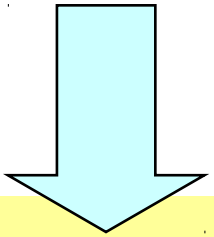
```
struct wibble;

struct wibble value_return(void);
struct wibble * ptr_return(void);

void value_parameter(struct wibble);
void   ptr_parameter(struct wibble *);
```

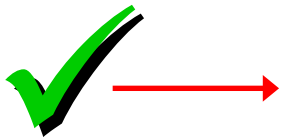
- manage dependencies aggressively
  - both physically and logically

dependencies



```
#include "wibble.h"

void many_includes(wibble * ptr);
void are_unnecessary(wibble value);
```

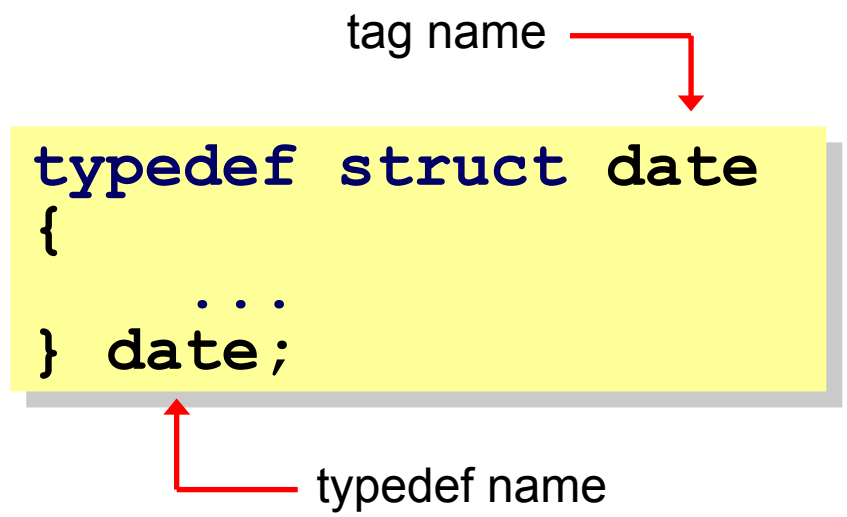




```
struct wibble;

void many_includes(struct wibble * ptr);
void are_unnecessary(struct wibble value);
```

Structs and Enums

- **these are the two recommended styles**
  - **any deviation from these two is unhelpful**



if you use a typedef make the typedef name the same as the tag name.

```
struct date
{
    ...
};
```

or don't typedef.

summary

- **making the tag_name and the typedef name different creates a name difference when the is no actual type difference**

```
struct date_tag
{
    ...
};
```

```
typedef struct date_tag date;
```

```
struct date_tag deadline;
...
date delay = deadline;
```

variation-1

- omit the tag name and typedef?
  - this forces a single syntax

```
typedef struct
{
    ...
} date;
```

```
struct date_tag deadline;
...
date delay = deadline;
```

```
date deadline;
...
date delay = deadline;
```

variation-2

- omit the tag name and don't typedef?
  - also changes the namespace
  - you can't use the type name as an object name

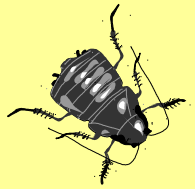
variation-2

```
struct date
{
    ...
};

struct date date;
```

```
typedef struct
{
    ...
} date;

date date;
```

- **omit the tag name?**
  - **also prevents a bizarre bug**
  - **struct X; means X does *not* have to already exist**
  - **struct X; *forces* X to name a possibly new type!**

```
struct date
{
    ...
};

void func(struct date_t *);
```



variation-2

```
typedef struct
{
    ...
} date;

void func(date_t *);
```

- **omit the tag name?**
  - **is not an option for cyclically dependent types**

```
struct link
{
     struct link * next;
     ...
};
```

variation-2

```
typedef struct
{
     struct link * next;
} link;
```

- **omit the tag name?**
  - **prevents forward declarations**
  - **forces unnecessary #includes**

date.h

```
typedef struct
{
    ...
} date;
```

use.h

```
#include "date.h"

void f(date d);
```

variation-2

- **writing a tag name?**
  - **allows forward declarations**
  - **allows omission of unnecessary #includes**

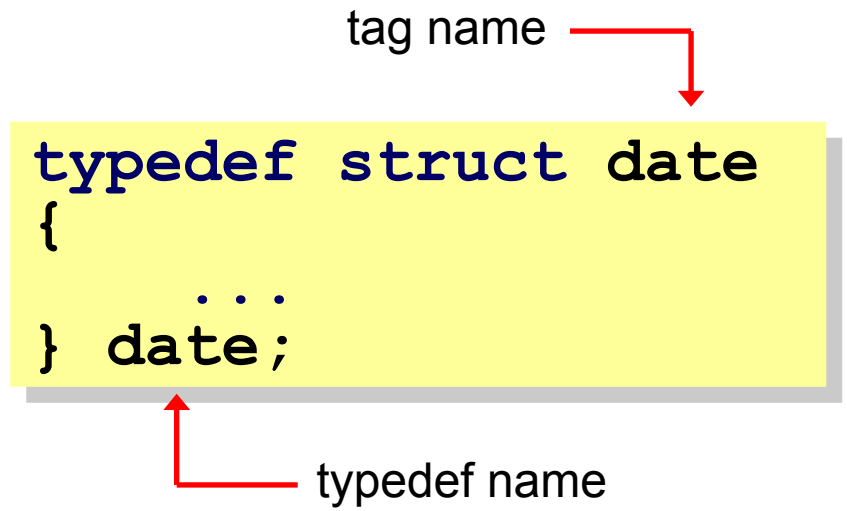
date.h

```
struct date
{
    ...
};
```

```
struct date;

void f(struct date d);
```

variation-2

- **these are the two recommended styles**
  - **any variation is unhelpful**



if you use a typedef make the typedef name the same as the tag name.

```
struct date
{
    ...
};
```

or don't typedef.

recap

- a typedef does ***not*** create a new type
  - perhaps it should have been named typedecl ?

typedef != definition

```
typedef int mile;
```

```
typedef int kilometer;
```

```
void weak(mile lhs, kilometer rhs)
{
    lhs = rhs;
    ...
}
```

- a struct *does* introduce a new type name
  - use a wrapper type instead...

```
struct mile
{
    int value;
};
```

```
struct kilometer
{
    int value;
};
```

```
void strong(struct mile lhs, struct kilometer rhs)
{
    lhs = rhs;
    ...
}
```

struct == definition

- **a struct wrapper does *not* introduces any extra runtime overhead**

mile1.c

```
struct mile
{
    int value;
};

void f(int n)
{
   struct mile m = { n };
   if (m.value % 2)
      ...
}
```

mile2.c

```
typedef int mile;

void f(int n)
{
   mile m = n;
   if (m % 2)
      ...
}
```

```
>gcc -O2 -S mile1.c
>gcc -O2 -S mile2.c
>diff mile1.s mile2.s
   → no difference
```

struct == definition

- **struct layout – the compiler...**
  - **can insert padding between members and after the last member**
  - **cannot insert padding before the first element or re-order the members**

```
#include <stddef.h> // offsetof

const size_t co = offsetof(struct wibble, c);
const size_t cs = offsetof(struct wibble, s);
const size_t ci = offsetof(struct wibble, i);
```

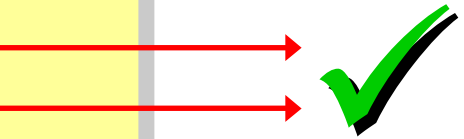
```
co == 0
cs >= co + sizeof(char)
ci >= cs + sizeof(short)
```

```
struct wibble
{
    char c;
    short s;
    int i;
};
```

- structs support initialization and assignmer
  - not necessarily implemented as a memcpy
  - some other implementation might be faster! e.g. member by member copying avoiding memcpy of many padding bytes
  - hence == and != are *not* supported

```
struct wibble
{
    ...
};
```

```
struct wibble w = { ... };
struct wibble copy = w;
              copy = w;
```



```
if (copy == w) ...
```

typedef == definition

- a struct can contain an array
  - struct assignment copies the whole array!
  - no extra overhead compared to a memcpy

```
struct name
{
    char letters[64];
};
```

```
void strong(struct name * dst, const struct name * src)
{
    *dst = *src;
}
```

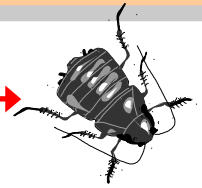
```
void strong(struct name * dst, const char * src)
{
    assert(strlen(src) < sizeof *dst);
    strcpy(dst->letters, src);
}
```

designators

- structs support designator identifiers
  - allows for a small useful degree of ignorance

```
<stdlib.h>
7.22.6.2 The div ... Functions
div_t div(int numer, int denom);
The div...functions return a structure of
type div_t...The structures shall contain
(in either order)
the members quot (the quotient) and rem
(the remainder)...
```

```
struct div_t d = { 9, 6 };
```



```
struct div_t d;
d.quot = 9;
d.rem = 6;
```

```
struct div_t d = { .quot = 9, .rem = 6 };
```

enums

```
#define MAX_LEN (1024)

char buffer[MAX_LEN];
```

Avoid the preprocessor when possible

worse

```
enum { MAX_LEN = 1024 };

char buffer[MAX_LEN];
```

All uppercase is reserved for the preprocessor

better?

```
enum { max_len = 1024 };

char buffer[max_len];
```

best?

```
struct x912_buffer
{
    char bytes[1024];
};
```

anonymous enums

- **useful for designators and switch labels**

```
enum { january, february, march,
       ...
       october, november, december
     };
```

```
const int days_in_month[] =
{
    [january] = 31,
    [february] = 28,
    ...
    [november] = 30,
    [december] = 31
};
```

```
switch (...)
{
     case january: ...
     case february: ...
     ...
     case november: ...
     case december: ...
}
```

- **enums cannot be forward declared**
  - **or have cyclic dependencies**
  - **so never any need for a tag name**

```
typedef enum suit
{
    clubs, diamonds, hearts, spades
} suit;

enum suit trumps;
suit lead;
```

```
typedef enum
{
    clubs, diamonds, hearts, spades
} suit;

enum suit trumps;
suit lead;
```

weak enum

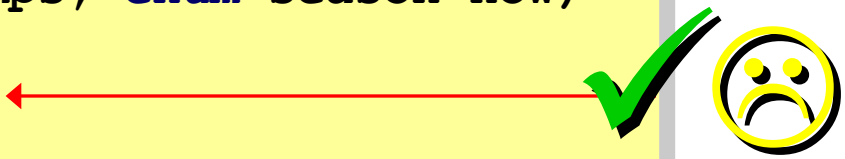
- **enums are very weakly typed**
  - **an enum's enumerators are of type integer, not of the enum type itself!**

weak enum

```
enum suit
{
    clubs, diamonds, hearts, spades
};
```

```
enum season
{
    spring, summer, autumn, winter
};
```

```
void weak(enum suit trumps, enum season now)
{
    trumps = now;
}
```



stronger enums

- an enum can also be wrapped in a struct

```
struct suit
{
    enum
    {
        clubs, diamonds, hearts, spades
    } value;
};
```

```
struct season
{
    enum
    {
        spring, summer, autumn, winter
    } value;
};
```

stronger enums

- struct wrapper provides strong typing
  - expressions that shouldn't compile don't
- enum enumerators still available
  - important for compile time constructs, eg switch
- wrapper type is a struct
  - forward declaration now possible

```
void strong(struct suit trumps, struct season now)
{
    trumps = now;

    switch (now.value)
    {
        case spring: ...
        case summer: ...
        case autumn: ...
        case winter: ...
    }
}
```

stronger enums

- **a struct wrapper does *not* introduce any extra runtime overhead**

suit1.c

```
struct suit
{
    enum { clubs, diamonds, hearts, spades } value;
};

void f(int n)
{
   struct suit trumps = { n % 2 ? clubs : diamonds };
   if (trumps.value == clubs)
      ...
}
```

stronger enums

- **a struct wrapper does *not* introduce any extra runtime overhead**

suit2.c

```
enum suit
{
    clubs, diamonds, hearts, spades
};

void f(int n)
{
    enum suit trumps = n % 2 ? clubs : diamonds;
    if (trumps == clubs)
       ...
}
```

```
>gcc -O2 -S suit1.c
>gcc -O2 -S suit2.c
>diff suit1.s suit2.s
   →  no difference
```

- **what might this look like if refactored to a wrapped enum?**

```
#define PKG_ID           0
#define PKG_VERSION      3
#define PKG_CKSUM        4
#define PKG_FILESIZE     8
#define PKG_FILE_CKSUM  12
...
```

exercise

an answer

```
enum
{
    package_offset_id = 0,
    package_offset_version = 3,
    package_offset_checksum = 4,
    package_offset_filesize = 8,
    package_offset_file_checksum = 12,
};
```

Objects

Users care about how easy objects are to ***use*** and how hard objects are to ***misuse***. This is called Affordance.

Users don't want to know or care about how hard objects are to ***make****. This is called Ignorance/Apathy.

Users care about only what ***they*** care about! This is called Selfishness.

users

(in)consistency

<stdio.h>

```
int fsetpos(FILE *, ...);
int fprintf(FILE *, ...);
int fscanf (FILE *, ...);
```

```
int    fputc (..., FILE *);
int    fputs (..., FILE *);
size_t fwrite(..., FILE *);
```

```
int main(int argc, char * argv[])...
```

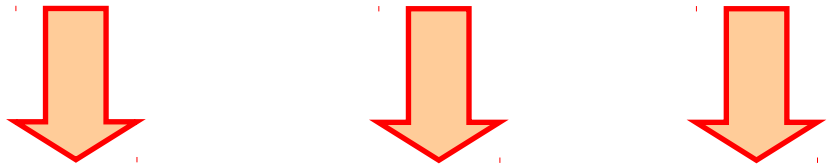
```
char * fgets(char * s, int n, FILE *);
```

- # 7.14.1.1 The signal function
  - if the request can be honoured, the signal function returns the value of func for the most recent successful call to signal for the specified signal sig.

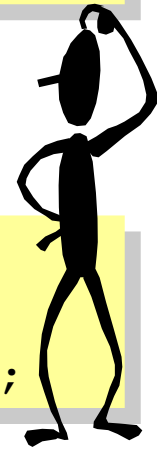
<signal.h>

```
void (* signal(int sig, void (*func)(int)))(int);
```

?



```
typedef void (*signal_handler)(int sig);

signal_handler signal(int sig, signal_handler func);
```



bad design

- ## 6.5.2.5 Compound literals
  - para 5 – if the compound literal occurs outside the body of a function, the object has static storage duration; ...

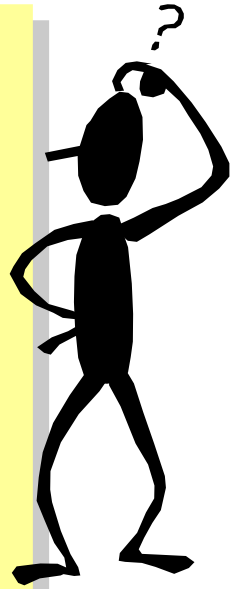
```
int * p = (int[]){1,2,3,4};
```

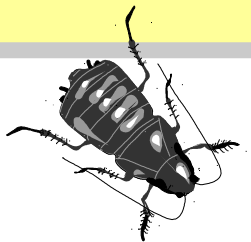
as-if

```
int $[] = { 1,2,3,4 };
int * p = $;
```

- ## 6.5.2.5 Compound literals
  - para 5 – if the compound literal occurs outside the body of a function, the object has static storage duration;
    otherwise it has automatic storage duration associated with the enclosing *block*

```
int eg(int i, int j)
{
    int * p;
    if (i == j)
        p = (int[]){ i, i };
    else
        p = (int[]){ j, j };
    return *p;
}
```





- ## 6.8.4 Selection statements
  - **para 3 – A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement i also a block whose scope is a strict subset of the scope of the selection statement.**

block scope

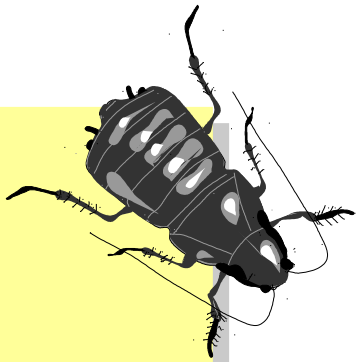
```
void f(void)
{
    if (...)
        ...
    else
        ...
}
```

blocks

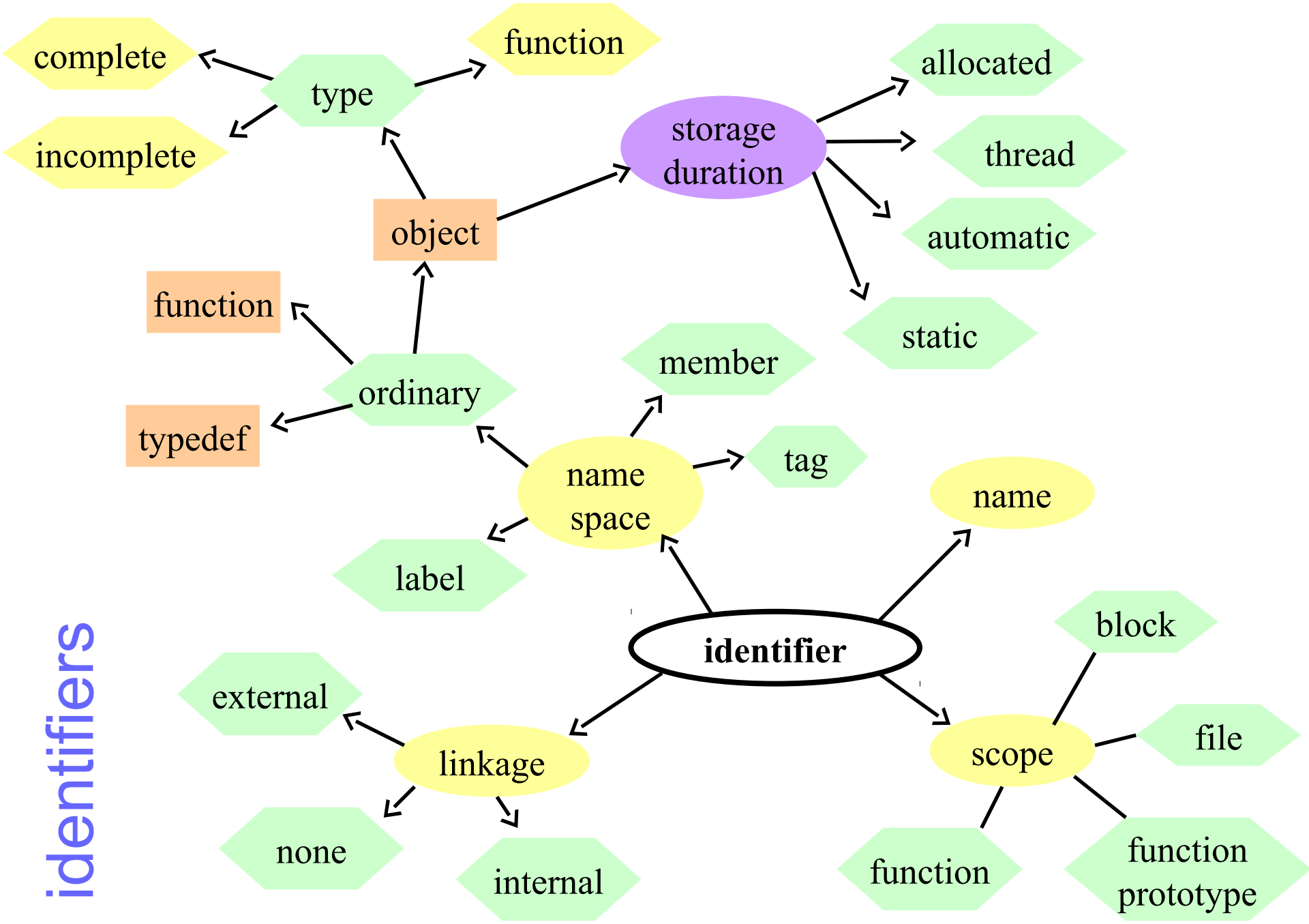
```
void f(void)
{
  {
    if (...)
    {   ...   }
    else
    {   ...   }
  }
}
```

c89 blocks != c99/c11 blocks

compound literals



```
int eg(int i, int j)
{
  int * p;
  {
    if (i == j)
    {
      int $[] = { i, i };
      p = $;
    }
    else
    {
      int $[] = { j, j };
      p = $;
    }
  }
  return *p;
}
```


identifiers
identifier
name space
member
tag
label
ordinary
typedef
function
object
type
complete
incomplete
function
storage duration
allocated
thread
automatic
static
name
linkage
external
none
internal
scope
block
file
function
function prototype

scope-linkage

```
/* file scope */

typedef int typedef__no_linkage;

struct stag__no_linkage { int member__no_linkage; };
union  utag__no_linkage { int member__no_linkage; };
enum   etag__no_linkage {     member__no_linkage, };

static int object__internal_linkage;
extern int object__external_linkage;
       int object__external_linkage;

static void function__internal_linkage(void) {}
extern void function__external_linkage1(void) {}
       void function__external_linkage2(void) {}
```

scope-linkage

```
void function_prototype_scope(

    /* typedef illegal here */

    struct stag__no_linkage { int member__no_linkage; },
    union  utag__no_linkage { int member__no_linkage; },
    enum   etag__no_linkage {     member__no_linkage, },

    int object__no_linkage
);
```

```
void function_scope(void)
{
    label__no_linkage: ;
}
```

scope-linkage

```
void block_scope(void)
{
  typedef int typedef__no_linkage;

  struct stag__no_linkage { int member__no_linkage; };
  union  utag__no_linkage { int member__no_linkage; };
  enum   etag__no_linkage {     member__no_linkage, };

         int object__no_linkage1;
  static int object__no_linkage2;
  extern int object__external_linkage;

  /* illegal - 6.7.1 paragraph 5 (pedantic) */
  static void function__internal_linkage(void);

  extern void function__external_linkage(void);
         void function__external_linkage(void);
}
```

Sequencing

sequence points

- **5.1.2.3 Program execution. Paragraph 3**
  - ***Sequenced before*** **is a … relation between evaluations executed by a single** ***thread*** **, which induces a partial order among those evaluations**
  - **The presence of a** ***sequence point*** **between the evaluations of expressions A and B implies that every value computation and side effect associated with A is** ***sequenced before*** **every value computation and side effect associated with B.**
  - **If A is not** ***sequenced before*** **or after B, then A and B are** ***unsequenced***

A sequence point is a point of stability; a point in the program's execution sequence where all previous side-effects will have taken place and where all subsequent side-effects will not have taken place.

sequence points

- **6.5 Expressions**
  - **para 2 – If a side effect on a scalar object is *unsequenced* relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.**

```
n = n++;
```

Undefined **if** n= (side effect on n) is unsequenced relative to n++ (different side effect on n)

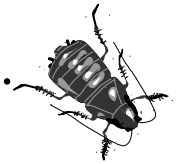
```
n + n++;
```

Undefined **if** n (value computation of n) is unsequenced relative to n++ (side effect on n)

- # 6.5 Expressions
  - **para 3 – Except as specified later, side effects and value computations of subexpressions are *unsequenced***

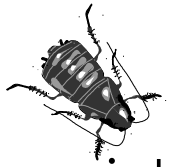
```
n = n++;
```

Undefined behaviour.
Assignment does not introduce a sequence point



```
n + n++;
```

Undefined behavior.
Addition does not introduce a sequence point.



The default is that subexpressions are *unsequenced*.

sequence points

- **sequence points occur…**
  - **at the end of a full expression**
    - **a full expression is an expression that is not a sub-expression of another expression or declarator (6.8 p4)**
  - **after the first operand of these operators**
    - **&& logical and (6.5.13 p4)**
    - **|| logical or (6.5.14 p4)**
    - **?: ternary (6.5.15 p4)**
    - **, comma (6.5.17 p2)**
  - **after evaluation of all arguments and function expression before a function call (6.5.2.2 p10)**
  - **at the end of a full declarator (6.7.6 p3)**

6.8 Statements and blocks

- para 4 – A full expression is an expression that is not part of another expression or declarator.

full expression

```
int function(int value)
{
    int inside = [          ];
    [                       ];
    if ([      ])...
    switch ([      ])...
    while ([      ])...
    do ([      ])...
    for ([      ];[      ];[      ])...
    return [      ];
}
```

- an initializer;
- the expression in an expression statement;
- the controlling expression in a selection statement…;
- the controlling expression of a while or do statement;
- each of the (optional) expressions in a for statement;
- the (optional) expression in a return statement;

- **when is a comma a sequence point?**

```
int x   = f( o++ , o++ );
```

```
int x   =    o++ , o++;
```

```
int x   =   (o++ , o++);
```

exercise

- **when it's an operator but not when it's a punctuator**

```
int x   = f( o++ , o++ );
```

punctuator

```
int x   =    o++ , o++;
```

```
int x   =   (o++ , o++);
```

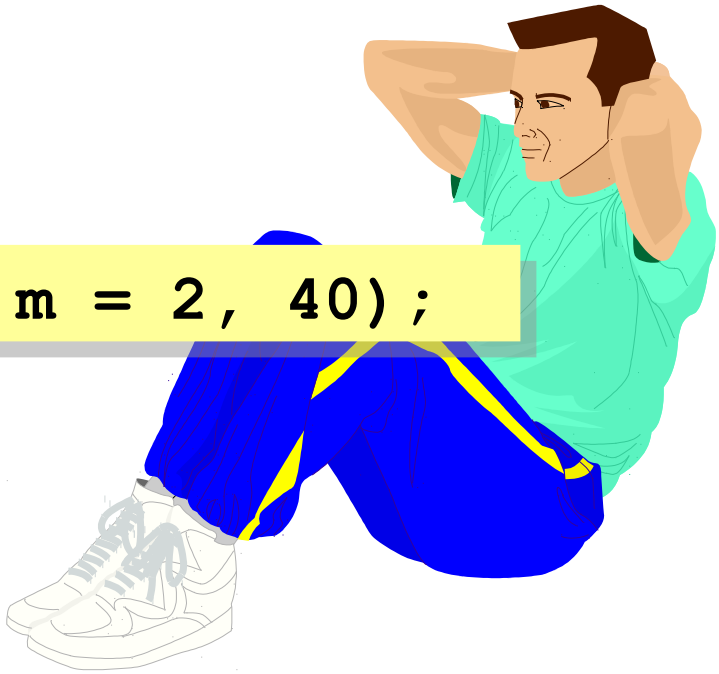
operator

answer

another exercise

- in this very tricky statement...
  - where are the sequence points?
  - what are the operators?
  - what is their relative precedence?
  - what is affected by what?
  - how many times is m modified between sequence points?
  - is it undefined?

```
(10, m = 1, 20) + (30, m = 2, 40);
```



another exercise

- what does this print?

```
#include <stdio.h>

int glob = 0;

int f(int value, int * ptr)
{
    if (value == *ptr)
    {
        printf("value == *ptr\n");
    }
    if (value != *ptr)
    {
        printf("value != *ptr\n");
    }
    return 1;
}

int main(void)
{
    return (glob = 1) + f(glob, &glob);
}
```

Representation

representation?

A general principle of coding

Use as little representation information as you can.

Use as little representation information as you can.

6.2.6 Representations of types
6.2.6.1 General
para 4 – Values stored in non-bit-field objects of any other object type consist of n x* CHAR_BIT *bits, where n is the size of that type, in bytes. The value may be copied into an object of type* unsigned char [n] *(e.g., by* memcpy*); the resulting set of bytes is called the object representation of the value.

representation

To gain access to the bits comprising the representation of an object you must use the address of the object as a pointer to an unsigned char. This type alone guarantees access to all bits in all bytes.

representation

6.2.6 Representations of types
6.2.6.2 Integer types
para 1 – For unsigned integer types other than **unsigned char**, ***the bits of the object representation shall be divided into two groups; value bits and padding bits...If there are N value bits, each bit shall represent a different power of 2 ... using a pure binary representation; this shall be known as the value representation. The value of any padding bits are unspecified.***

Suppose CHAR_BIT == 8 and sizeof(int) == 4
This does not guarantee that int is a 32 bit integer type! Use INT_MAX to determine if int is a 32 bit integer.

x === y; implies memcmp(&x, &y, sizeof(x)) == 0
true or false?

?

LValues

6.3.2.1 Lvalues, arrays, and function designators para 1 – sentence 1
***An lvalue is an expression (with an object type other than* void) that potentially designates an *object*;**
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"

lvalues

6.3.2.1 Lvalues, arrays, and function designators para 1 – sentence 3
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***

more lvalues

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.

- **these operators never yield lvalues**

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

values

- these expressions sometimes yield lvalues

primary	`identifier`
parentheses	`( )`

- these operators sometimes yield lvalues

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

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)

6.5.1 Primary Expressions
para 5 - A parenthesized *expression* is a primary expression. Its type and *value* are identical to those of the unparenthesized expression.

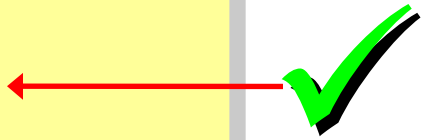
```
int function(int m)
{
     (m)++;
}
```

parentheses

- 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 ++)

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(struct 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)

. operator

exercise

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

```
struct wibble
{
    int member;
};


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

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

- **no, it's not a conforming program**
  - **6.3.2.1 paragraph 2 (see slide 93)**

```
struct wibble
{
    int member;
};

struct wibble f(void)
{
    struct 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

answer

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

→ operator

- 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)

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

```
struct wibble
{
    int member;
};

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

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

exercise

answer

- **no, it's not a conforming program**
  - **6.3.2.1 paragraph 1 (see slide 91)**

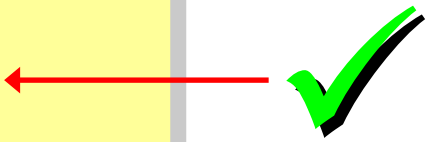
```
struct wibble
{
    int member;
};

struct wibble * f(void)
{
    struct 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(struct wibble w)
{
    struct 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(struct wibble w)
{
    *&w = w;
}
```

```
void cancel_other_way(struct 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 atomic, 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;
}
```

cast is not lvalue

int[ ] is not scalar type

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

compound literals

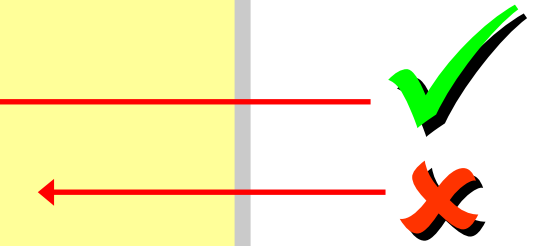
6.5.2.5 Compound literals
para 4 - A postfix expression that consists of a parenthesized type, followed by a brace enclosed list of *initializers* is a compound literal. It provides an unnamed object whose value is given by the *initializer list*. ...
para 5 - The result is an *lvalue*.

this is *not* a cast

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

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

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

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 *shall* have atomic, qualified, or unqualified real or pointer type, and shall be a *modifiable lvalue*.
para 2 - ... The result is the new value of the operand after incrementation. The expression ++E is equivalent to (E+=1)

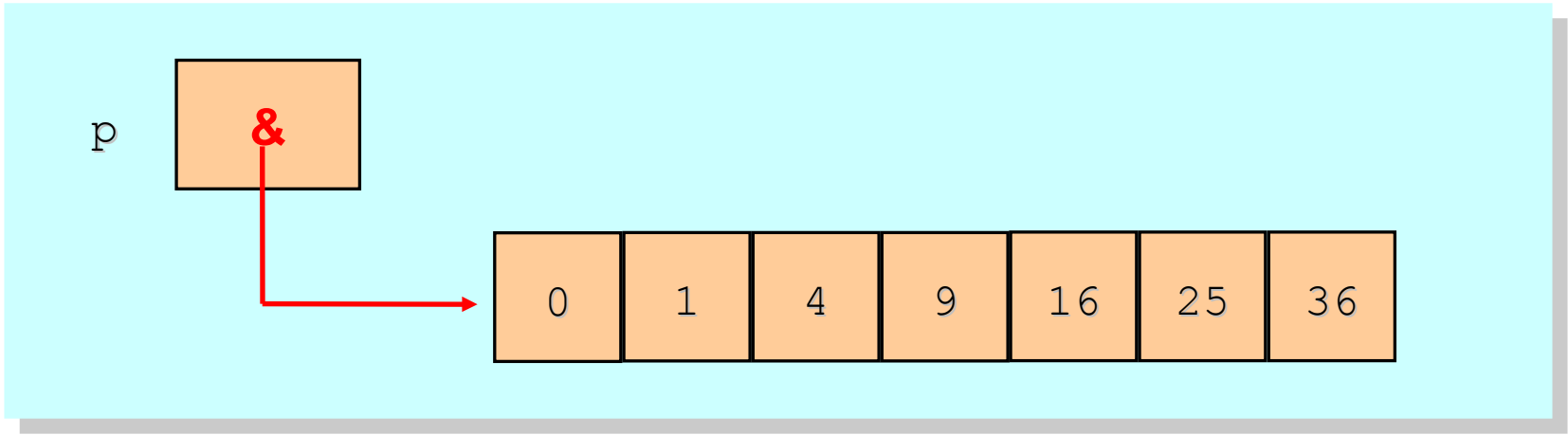
```
void increment(void)
{
    int x = 0;

    x++ = 42;

    ++x = 42;
}
```

++x is accidentally an lvalue in C++

Arrays

- **an aggregate initializer list can apparently be cast to an array type**
  - **known as a compound literal**

```
int * p =
     (int []){ 0,1,4,9,16,25,36 };
```



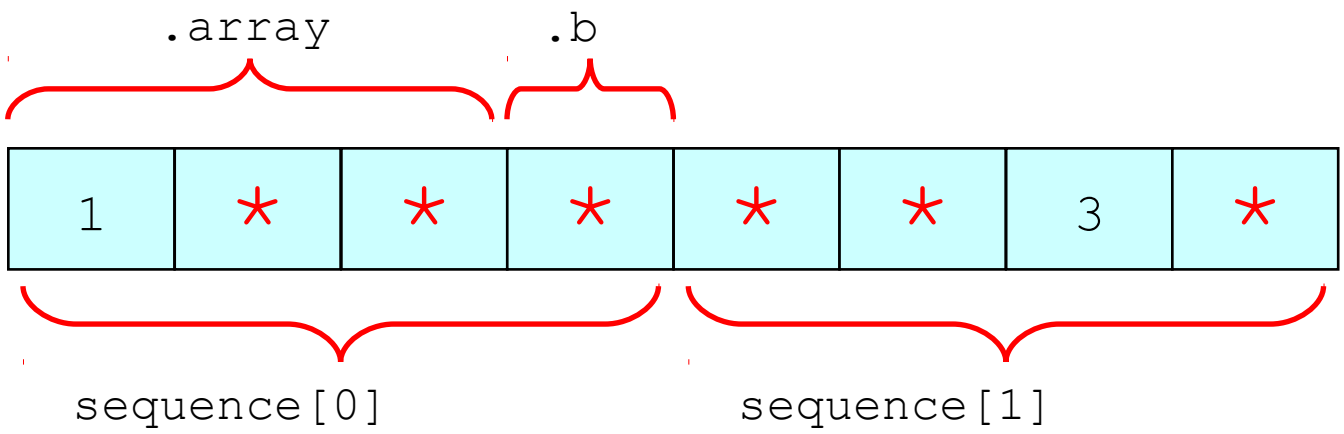
array literal

- **arrays and struct may contain each other**
  - **[int] and .identifier designators can be combined**

struct containing array

```
struct s
{
    int array[3];
    int b;
};
```

```
struct s sequence[] =
{
    [0].array = { 1 },
    [1].array[2] = 3
};
```



*default value

array rectangular

- only the top-level array decays into a pointe
  - the size of sub arrays remains part of the type

```
void print(int nrows, int matrix[2][3]);
void print(int nrows, int matrix[ ][3]);
void print(int nrows, int (*matrix)[3]);
```

equivalent

```
int main(void)
{
    int grid[2][3] = {{0,1,2},{3,4,5}};
    ...
    print(2, grid);
}
```

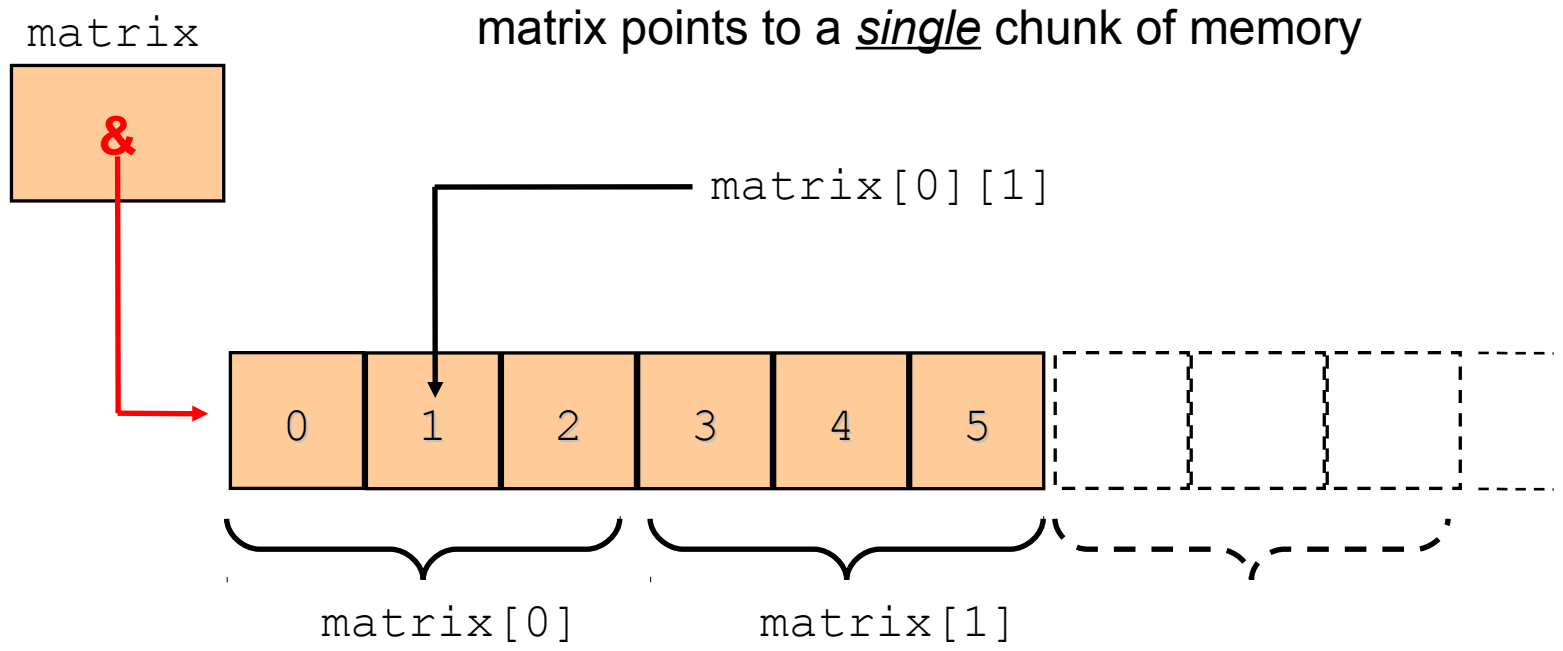
```
void illegal(int matrix[ ][ ]) ...
```

```
int (*matrix)[3]
```

```
int (*matrix)[3]

int (*matrix)[3]
```

matrix is a pointer
to zero, one, or more
array(s) of three ints

pointer to array



ragged arrays

- an array of pointers can mimic a 2d array
  - each pointer points to an array
  - aka Illiffe vector aka dope vector

note this is `int*ragged[2]` and not `int(*ragged)[2]`

```
void print(int nrows, int ncols, int * ragged[2]);
void print(int nrows, int ncols, int * ragged[ ]);
void print(int nrows, int ncols, int * * ragged);
```

equivalent

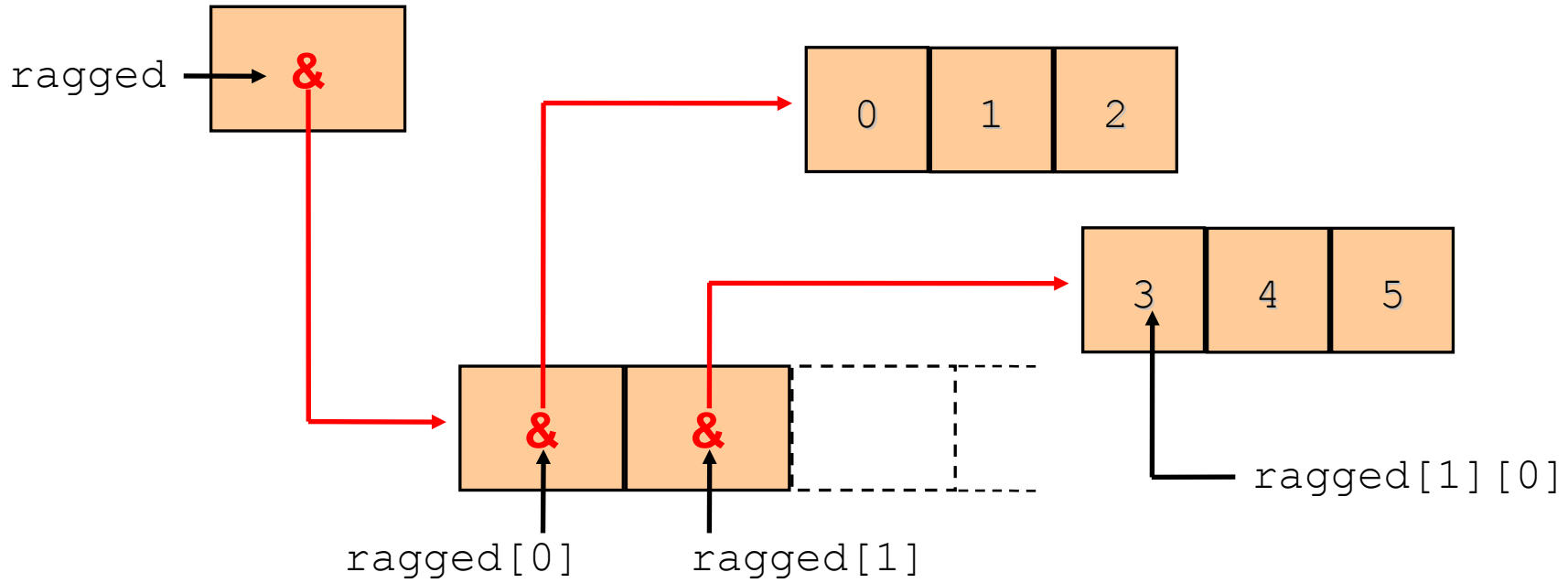
```
int main(void)
{
    int vec1[] = { 0, 1, 2 };
    int vec2[] = { 3, 4, 5 };
    int * grid[2] = { vec1, vec2 };

    print(2, 3, grid);
}
```

array of pointers

```
int * ragged[]
```

```
int * ragged[]

int * ragged[]

int * ragged[]
```

ragged is an array
of
pointers
to zero, one, or more
int(s)

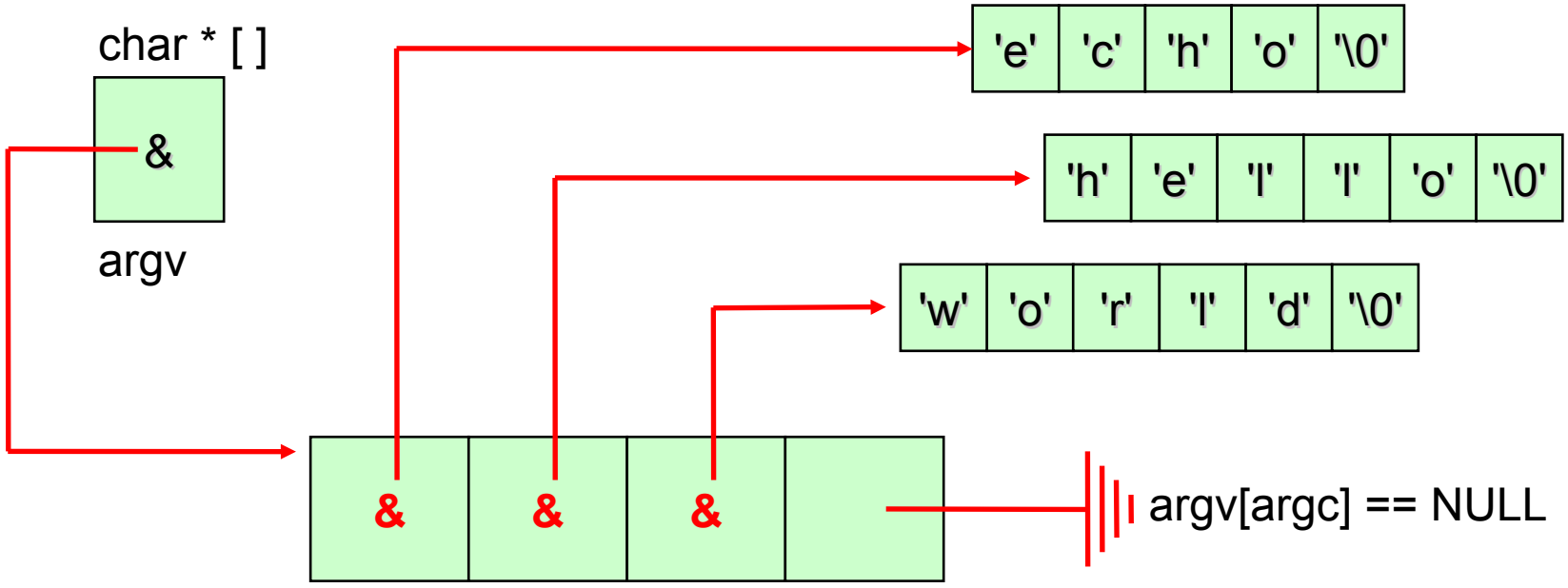


ragged array example

echo.c

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    for (int at = 0; at != argc; at++)
        printf("%s ", argv[at]);
    putchar('\n');
}
```

```
>echo hello world
```



variable length arrays

VLA's - four restrictions

```
void vla(int n)
{
     int ok[n];

     static int g[n];

     extern int f[n];

     struct tag
     {
          int x[n];
     };
}

extern int size;
int array[size];
```

An object with static storage duration cannot have a variable length array type

An object with linkage cannot have a variable length array type

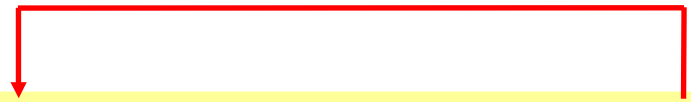
An identifier other than a simple identifier cannot have a variable length array type

A file scope identifier cannot have a variable length array type

variable length arrays

- **VLA's make multi-dimensional array parameters much more useful and reusable**

```
void print(int n, int m,  int matrix[ n ] [ m ] );
```

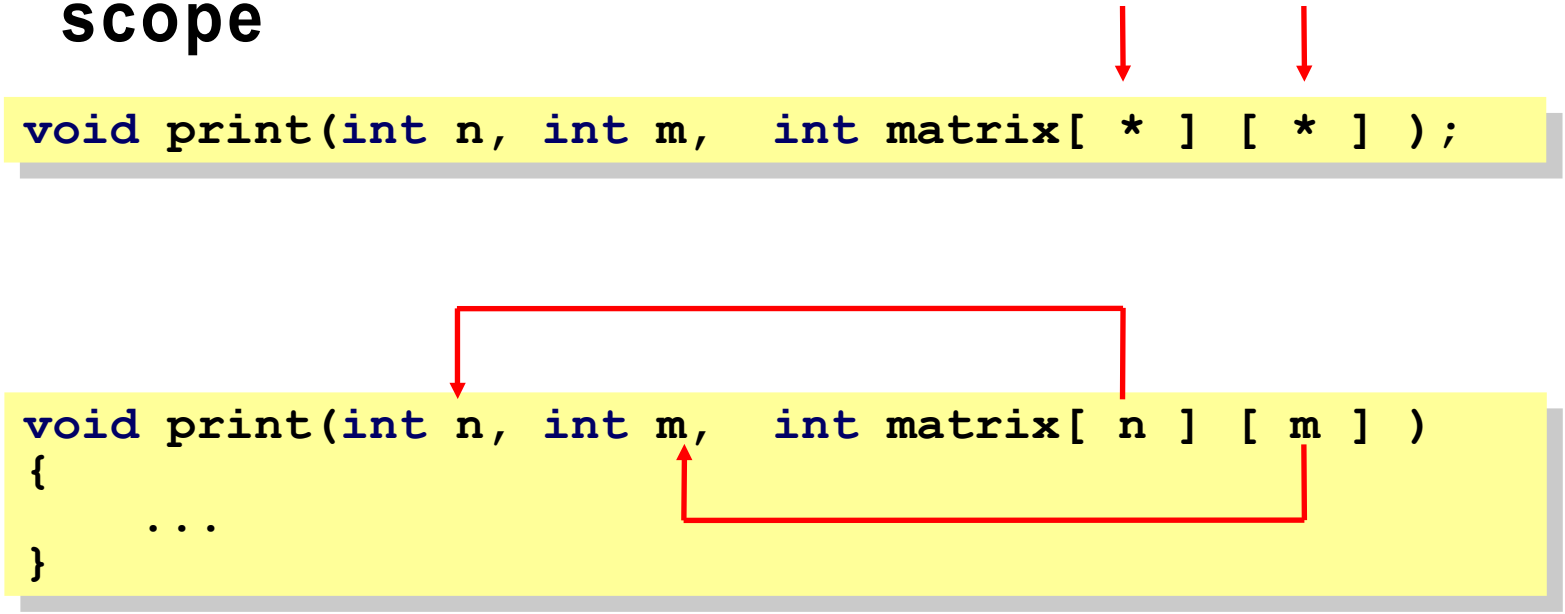


```
void print(int n, int m,  int matrix[ n ] [ m ] )
{
    ...
}
```

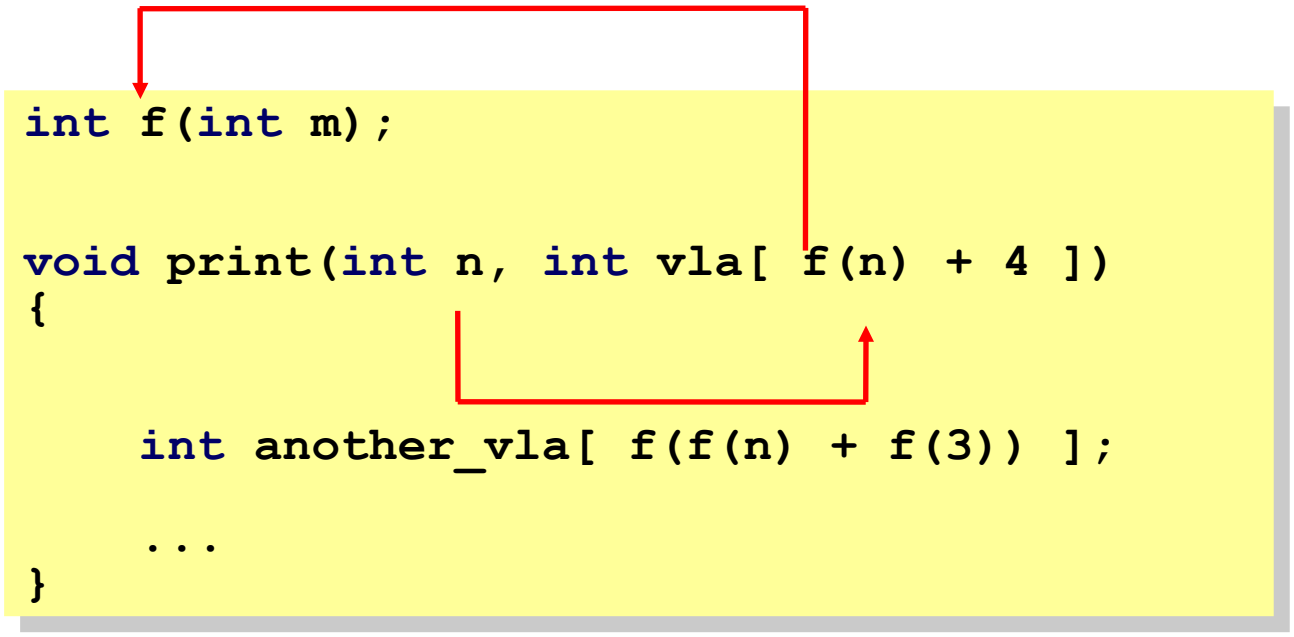
n and m must be declared before matrix

```
int main(void)
{
    int matrix[][3] = {{ 0, 1, 2 }, { 3, 4, 5 }};
    print(2, 3, matrix);
}
```

- # 6.7.6.2 Array declarators
  - para 4 – If the size is * instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations with function prototype scope



- # 6.7.6.2 Array declarators
  - para 5 – if the size is an expression that is not an integer constant expression ... each time it is evaluated it shall have a value greater than zero.



```
int f(int m);


void print(int n, int vla[ f(n) + 4 ])
{

    int another_vla[ f(f(n) + f(3)) ];

    ...
}
```

- # 6.5.3.4 The sizeof operator
  - para 2 – If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant

```
void examples(int n, int vla[ f(n) ])
{
     ...sizeof vla

     int fla[42];

     ...sizeof fla

}
```

compile-time evaluation

run-time evaluation

- **6.7.6.3 Function declarators**
  - **para 7 - If the keyword static also appears within the [and] of the array type derivation, then for each call to the function, the value of the corresponding argument *shall* provide access to the first element of an array with at least as many elements as specified by the size expression**

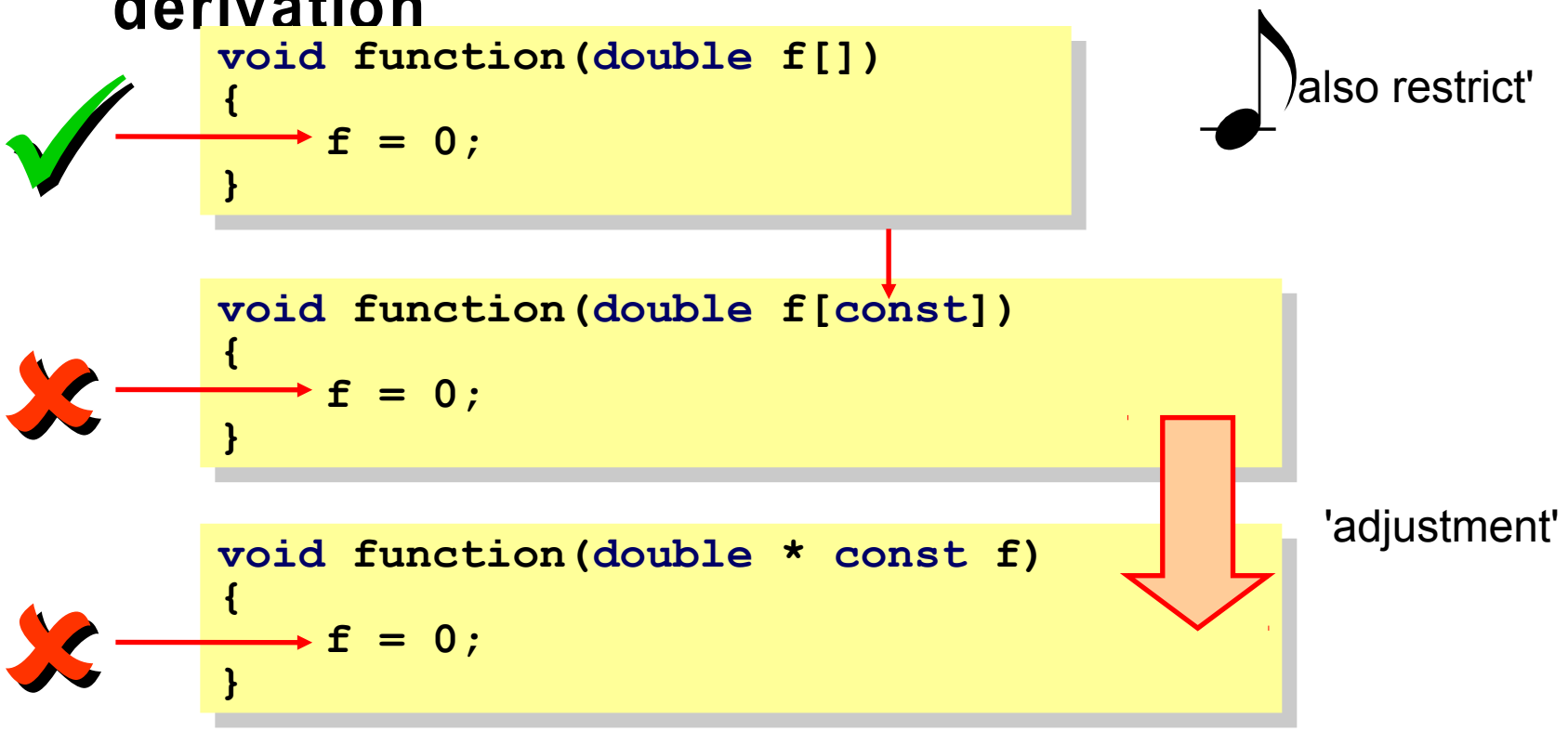
```
void function(double f[static 16])
{
    ...
}
```

f is non-null and points to at least 16 doubles

6.7.6.3 Function declarators

- para 7 – A declaration of a parameter as "array of type" shall be adjusted to "qualified pointer to type", where the type qualifiers (if any) are those specified within the [and] of the array type derivation

array declarators



- **complete arrays can be typedef'd**

```
void example(int m)
{
    typedef int fixed[42];

    typedef int variable[m];

    fixed f;
    variable v;
    ...
}
```

```
typedef int table[];

void example(void)
{
    table t1 = { 1,2 };

    printf("%zd\n", sizeof(t1));

    table t2 = { 1,2,3,4 };

    printf("%zd\n", sizeof(t2));

}
```

ef'd

2 * sizeof(int)

4 * sizeof(int)

Pointers

pointers =~= arrays

- **an array expression usually "decays" into a pointer to element zero**

```
void display(size_t size, wibble * first);
void display(size_t size, wibble first[]);
```

these two declarations are equivalent*

```
wibble table[42] = { ... };
```

```
display(42, table);
display(42, &table[0]);
```

these two statements are equivalent

- **6.3.2.1 Lvalues, arrays, and function designators**
  - **para 3 - Except when it is the operand of**
    - **the sizeof operator**
    - **or the unary & operator,**
    - **or is a string literal used to initialize an array,**
  - **an expression that has type "array of type" is converted to an expression with type "pointer to type" and points to the initial element of the array object and is not an lvalue**

array decay

There are three exceptions

exercise

- what does this program print?
  - assume sizeof(char*) == 8

```
struct wibble
{
    char x[42];
};

struct wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n", sizeof(f().x));

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

    printf("%zd\n", sizeof(0, x));
}
```

answers

```
struct wibble
{
    char x[42];
};

struct wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n", sizeof(f().x));
    printf("%zd\n", sizeof(x));
    printf("%zd\n", sizeof(0, x));
}
```

42

42

8

- **strings are arrays of char**
  - **automatically terminated with a null character, '\0'**
  - **a convenient string literal syntax**

```
char greeting[] = "Bonjour";
```

equivalent to

```
char greeting[] =
    { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

There is no decay in this case

- **what does this program print?**

```
void array_decay_or_not(void)
{
    char x[42];

    printf("%zd\n",
        (char*)( x+1) - (char*) x);

    printf("%zd\n",
        (char*)(&x+1) - (char*)&x);

    printf("%zd\n",
                (&x+1) -        &x);
}
```

exercise

answers

```
void array_decay_or_not(void)
{
    char x[42];

    printf("%zd\n",
        (char*)( x+1) - (char*) x);

    printf("%zd\n",
        (char*)(&x+1) - (char*)&x);

    printf("%zd\n",
                (&x+1) -       &x);
}
```

→ 1

→ 42

→ 1

- **C99/C11 library now uses restrict in its declarators where appropriate**
  - **says in code what would otherwise be said only in comments**

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

char * strcpy(char * s1,
              const char * s2);
```

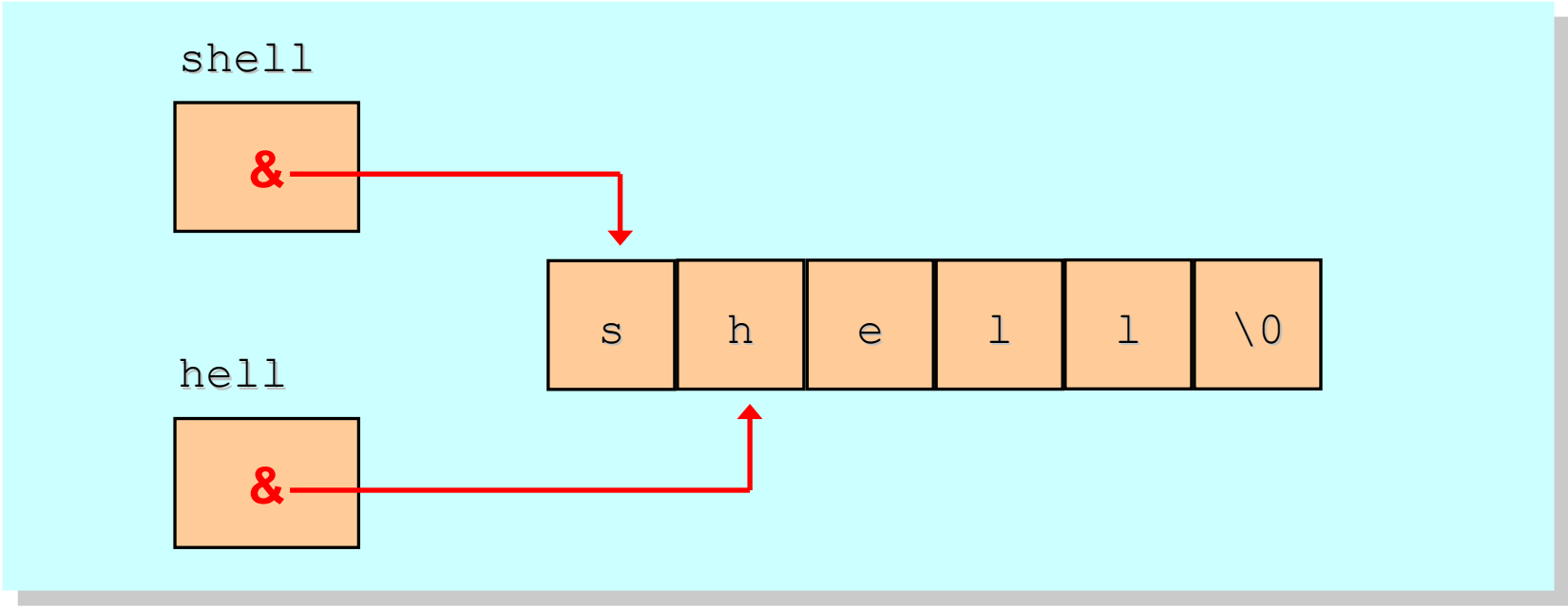
7.24 <string.h>

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

char * strcpy(char * restrict s1,
              const char * restrict s2);

If copying takes place between objects that
overlap the behaviour is undefined.
```

restrict

- **duplicate string literals can be assigned to the same (static) storage location**
  - **be careful with restrict**

```
const char * shell = "shell";
const char *  hell =  "hell";
```



- **6.5.2.1 Array subscripting**
  - **para 2 – The definition of the subscript operator [ ] is that E1[E2] is identical to (*((E1)+(E2)))**
- **6.5.6 Additive operators**
  - **para 8 – If both the pointer operand and the resul point to elements of the same array object, or on past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.**

Any pointer arithmetic that takes a pointer outside of the pointed to object...is undefined behavior. There is no requirement that the pointer be dereferenced.

subscripting

- **which of these are conforming?**

```
int array[10];

int * p1 = (array + 20) - 19;

int * p2 = array + 20 - 19;

int * p3 = array + (20 - 19);

int * p4 = array + 1;
```

exercise

answer

```
int array[10];
int * p1 = (array + 20) - 19;
int * p2 = array + 20 - 19;
int * p3 = array + (20 - 19);
int * p4 = array + 1;
```

- **typedef'ing a pointer...**

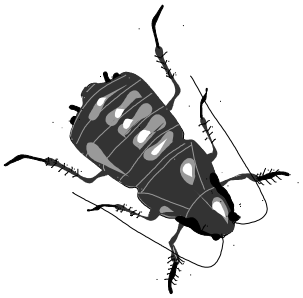
```
typedef struct date * date;

bool question(const date d);
```

...what is const?

is it the date pointer?

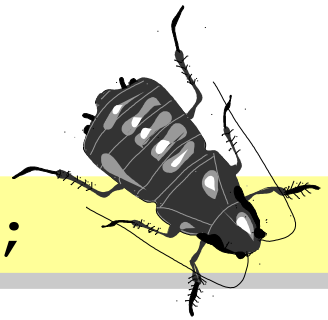
or

is it the date pointed to?



question

- **the pointer is const!**
- ***not*the date pointed to!**



```
typedef struct date * date;
```

```
bool question(const date d);
```

equivalent

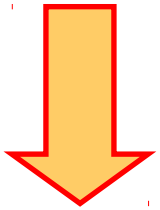
```
bool question(struct date * const d);
```

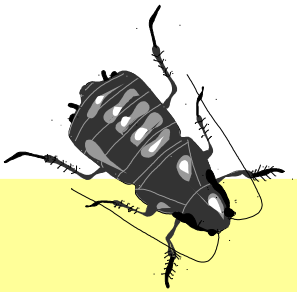
answer

pointer typedef?

- so how about this...?
  - one typedef for plain ptr
  - another typedef for ptr to const

```
typedef       struct date *  date;
typedef const struct date * cdate;
```





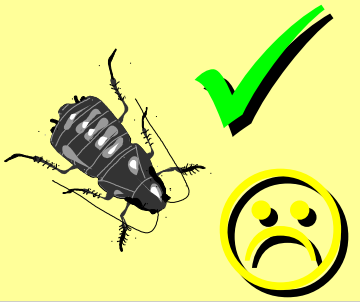
```
void hide_ptr(date delay);

void hide_const_too(cdate delay);
```

- **NO! it's a *bad bad* idea**
  - **hiding the pointer inside a typedef is not a good idea**

```
struct date
{
     int day;
     int month;
     int year;
};
```

```
typedef struct date * date;
```

```
bool date_equal(const date lhs, const date rhs)
{
   return lhs->day == rhs->day &&
          lhs->month == rhs->month &&
          lhs->year = rhs->year;
}
```

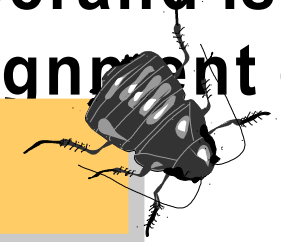


answer

Conversions

- **6.5.16 Assignment operators**
  - **para 3 – The type of an assignment expression is the type the left operand would have after lvalue conversion**
- **6.5.16.1 Simple assignment**
  - **para 2 – The value of the right operand is converted to the type of the assignment express**

```
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

assignment

- **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*.**

integer promotions

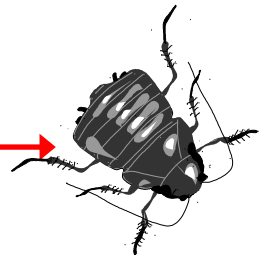
```
char c1, c2;

c1 + c2

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

Why does the compiler prefer ints?

argument promotion

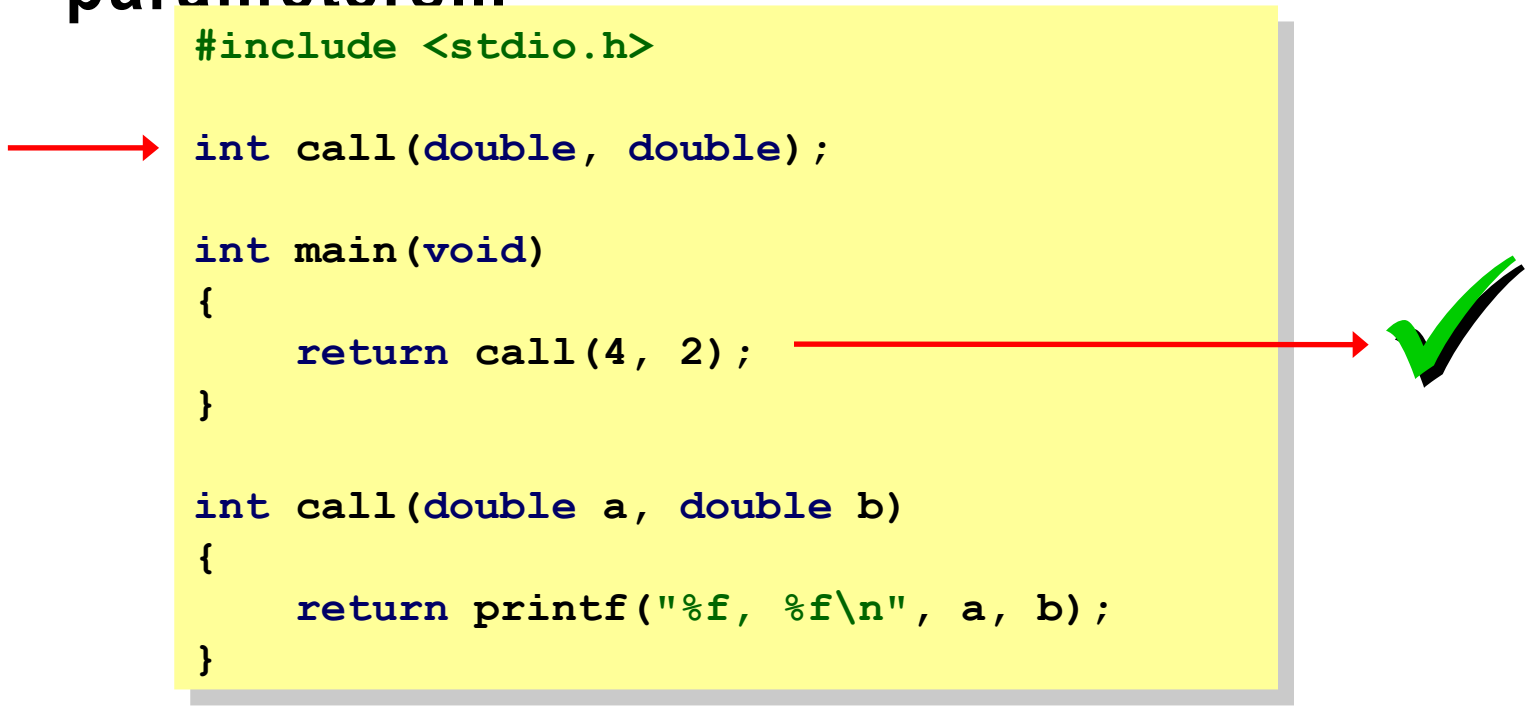
- **6.5.2.2 Function calls**
  - para 6 – if the expression that denotes the called function does not include a prototype, *integer promotions* are performed on each argument, and arguments that have type float are promoted to double. These are called the *default argument promotions*

```
#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...

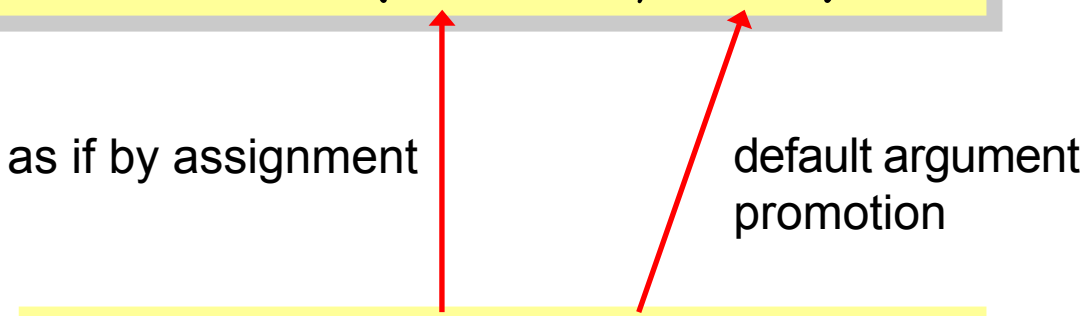


```c
#include <stdio.h>

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

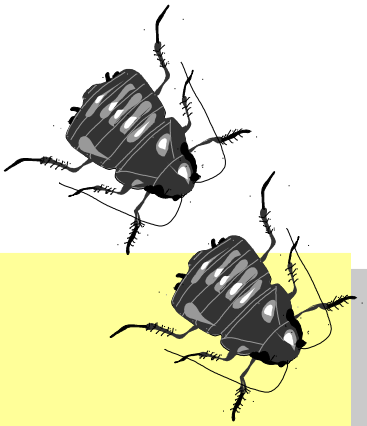
- **6.5.2.2 Function calls**
  - **para 7 – The ellipsis notation in a function prototype declarator causes argument type conversions to stop after the last declared parameter. The *default argument promotions* are performed on the trailing arguments.**

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

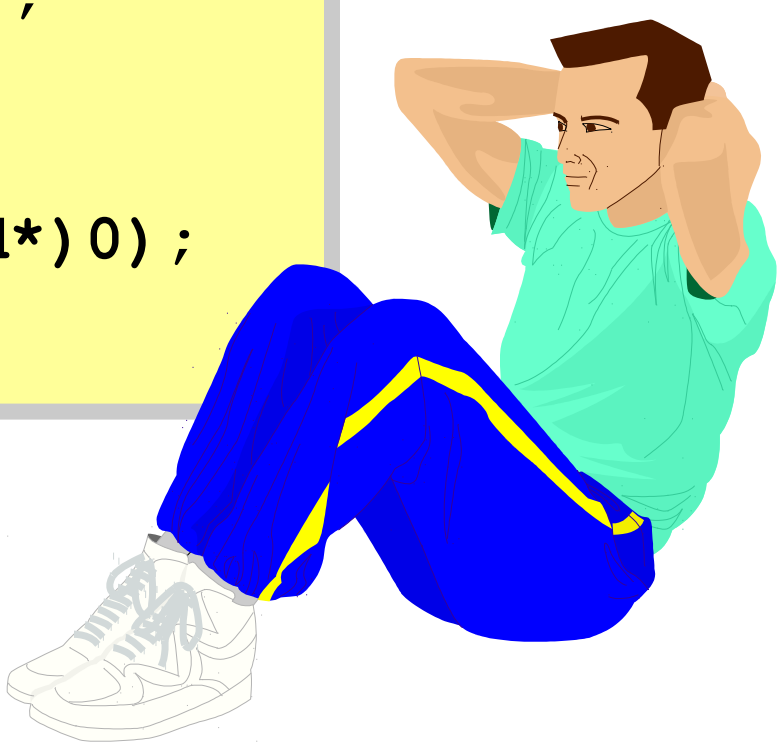



```
variadic('X', 'X');
```

spot the bugs



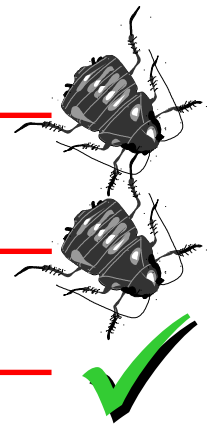
```
#include <stdio.h>

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

    printf("%p", 0);

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

}
```



exercise

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

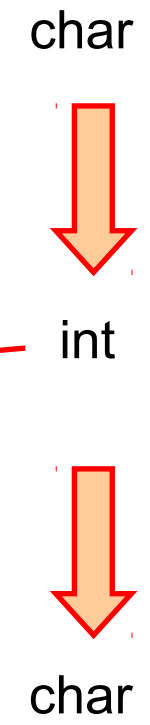
```
#include <stdio.h>

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

    printf("%p", 0);

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

}
```



answer

- **<stdarg.h> provide type-unsafe access**
  - **restrictions on all the va_ macros (7.16.1.1)**

```
#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);
}
```

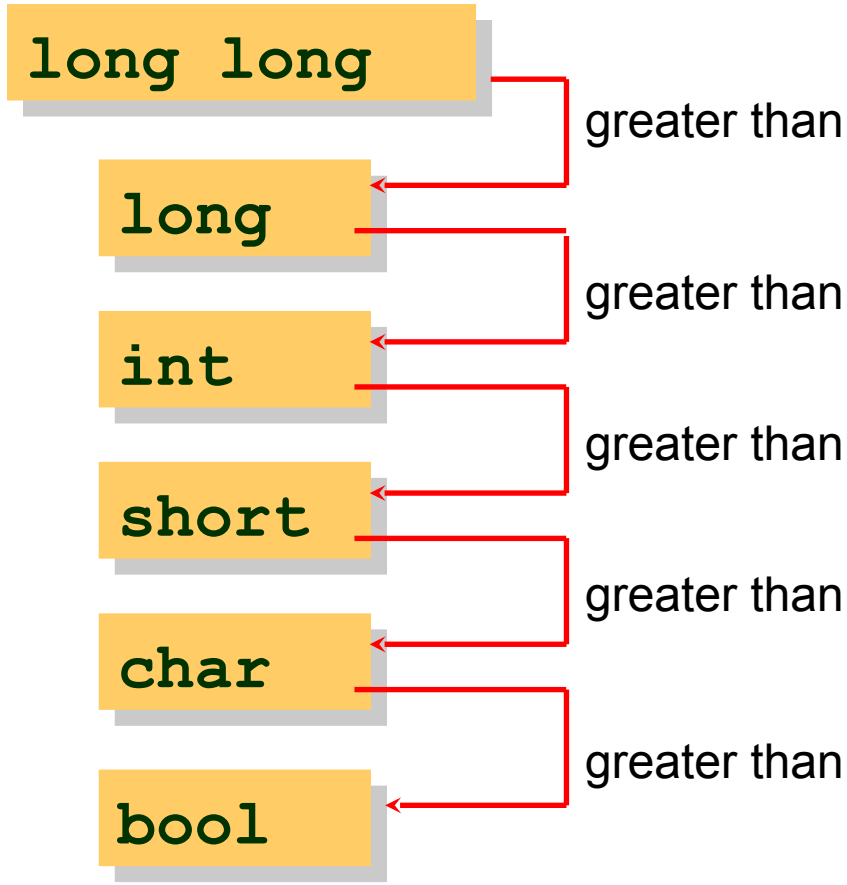


6.3.1.1 Booleans, characters, and integers

…

Every integer type has an *integer conversion rank*…

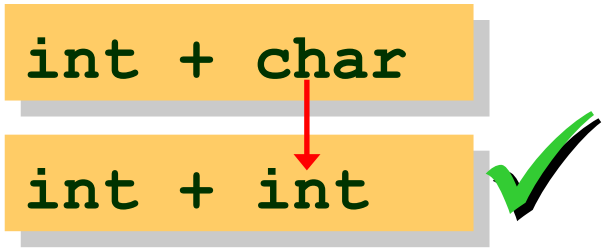
…



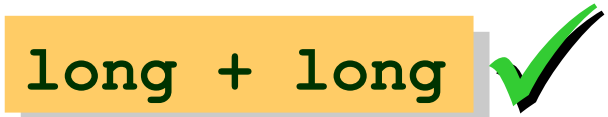
6.3.1.8 Usual arithmetic conversions

part 1: the obvious conversion rule (safe)

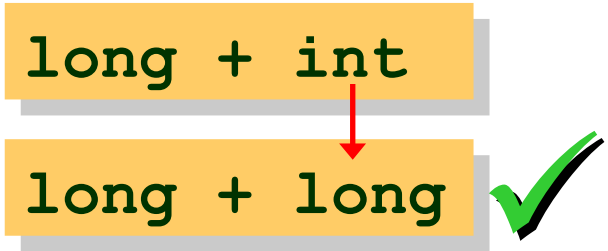
- …
- [both operands have integer type]
- …
- the *integer promotions* are performed on both operands



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



- 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.
- …

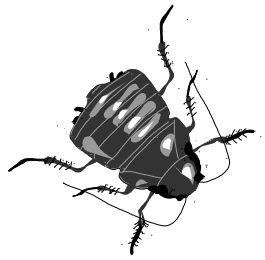


6.3.1.8 Usual arithmetic conversions

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
```

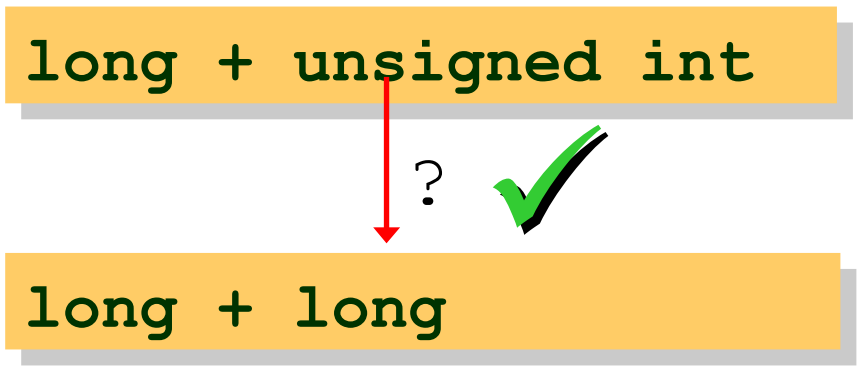
a negative signed integer value can be converted into a large positive unsigned integer value!!

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.



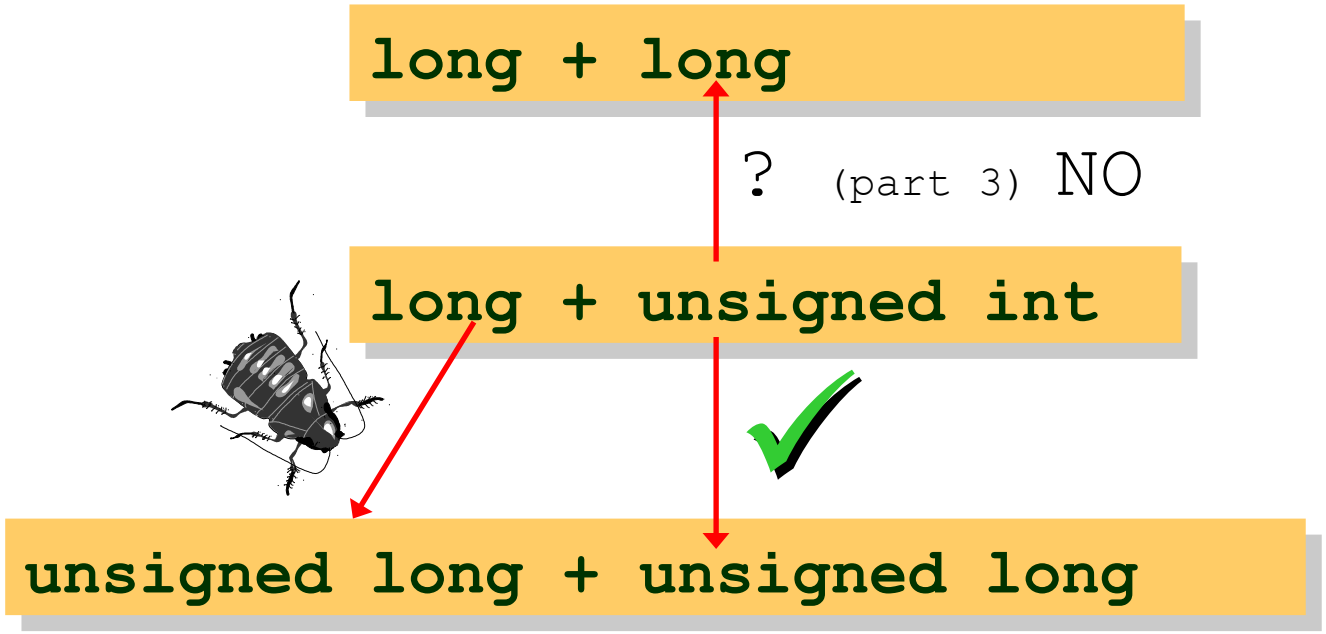
this depends on their value representations, as specified in <limits.h>

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);
}
```

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

answer

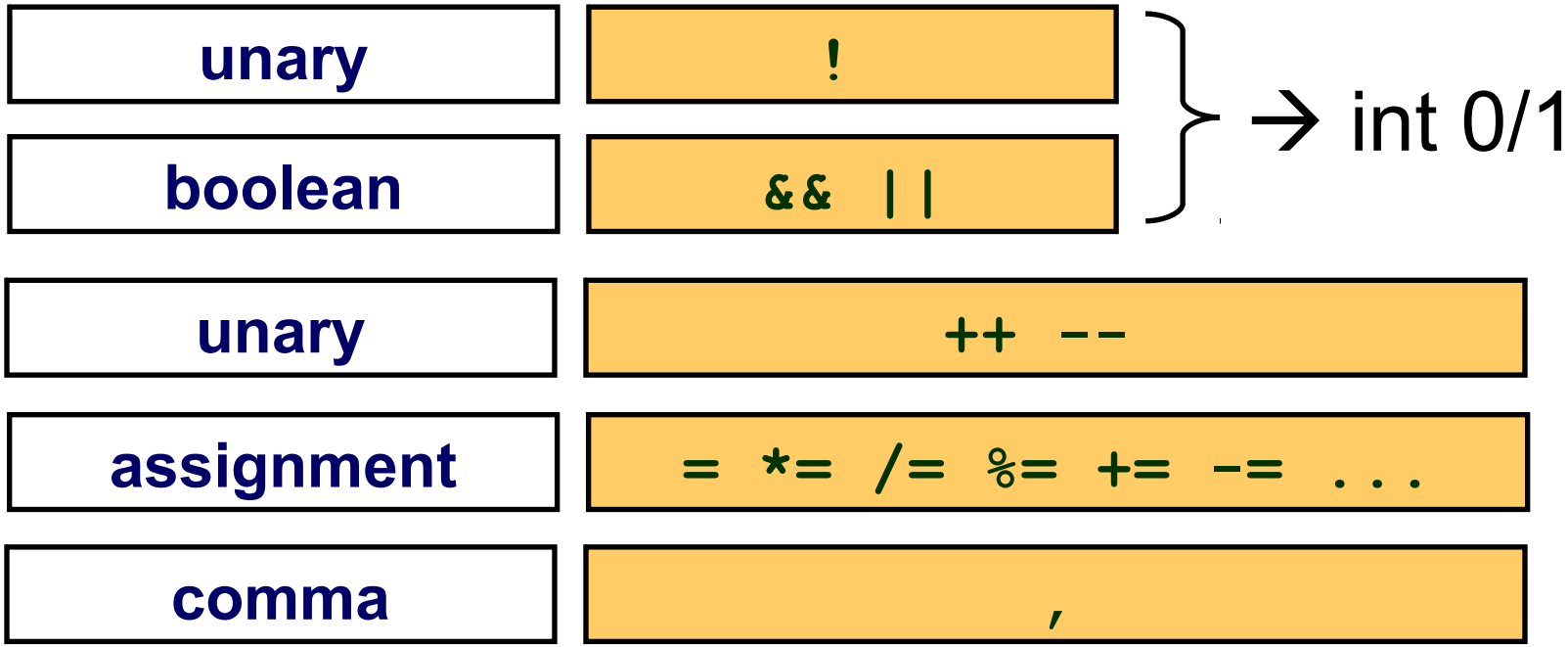
```c
#include <stdio.h>

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.**

- **these operators perform *no* conversions**

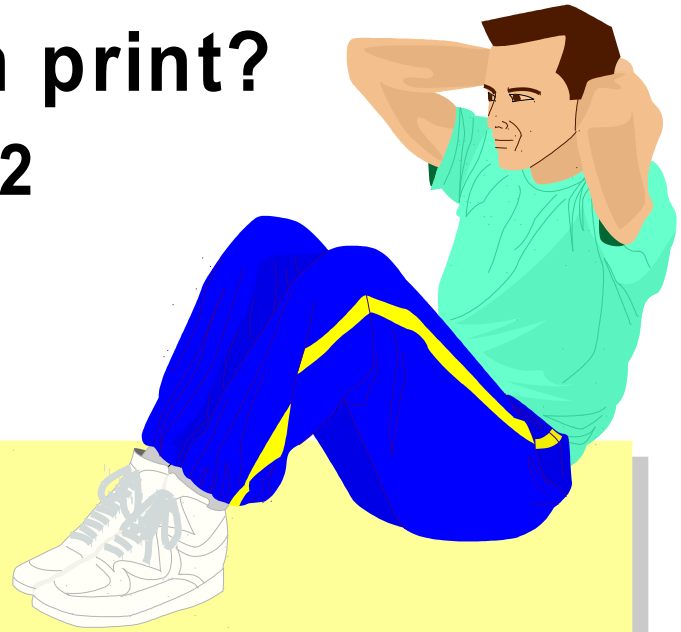


- **these operators perform *integer promotion***

- **these operators perform the *usual arithmetic conversions***

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

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



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

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

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

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

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

exercise

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

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

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

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

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

answer

4 2 4 2

Unit Testing

what is a unit?

- A Unit Test is
  - Isolated – it has no external dependencies
    - it passes or fails *solely* due to the test code and the code under test
  - Repeatable
    - No external dependencies means if it passed last time and the code hasn't changed it will pass this time
  - Automatable
    - No external dependencies means it can run all by itself
  - Fast
    - No external dependencies means nothing external is slowing the test execution down

How to unit test one function...

example.c

```
#include <stdio.h>
...
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

<stdio.h>

```
...

FILE * stdout = ...;

int fputc(int c, FILE * f)
{
   ...
}
```

testability

- **Move the function definition into its own file**
  - **Include this file from where it used to live**

example.c

```
#include <stdio.h>
...
#include "not_easily.func"
```

not_easily.func

```
void not_easily(void)
{
    fputc('4', stdout);
    fputc('2', stdout);
}
```

step 1

- **Include the *source* in the test**

not_easily.tests.c

```
#include <assert.h>

...
#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 2

Create a mock environment

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 3

Test from within the mock environment

not_easily.tests.c

```
#include <assert.h>

typedef int mock_file;

static mock_file * stdout = 0;

static int fputc(int c, mock_file * f)
{
    static int count = 0;
    static int expected[2] = { '4', '2' };
    assert(expected[count] == c);
    count = (count + 1) % 2;
    ...
}

#include "not_easily.func"

void not_easily_tests(void)
{
    not_easily();
}
```

step 4

- Write a simple utility to read given lines (eg 311 355) from a named file and save these lines to a new file as part of the test build
  - Simple verification is probably good enough
    - First line contains the named function
    - Last line contains only a }
- Include this made file from the source instead o including the .func file
  - This way leaves the original source file completely intact

too brutal?

for example

get_lines.rb

```
lines = STDIN.readlines
start = ARGV[0].to_i - 1
finish = ARGV[1].to_i - 1
STDOUT.write lines[start..finish]
```

How to unit test dependent functions?

example.c

```
#include "local.h"
#include <string.h>
#include <stdio.h>
...
int b(int value)
{ ...
}

int c(int value)
{ ...
}

int a(int value)
{
    return b(value) + c(value);
}
```

Do includes indirectly

```
#define LOCAL(x)   #x
#define SYSTEM(x) <x>
```

example.c

```
#include LOCAL(local.h)
#include SYSTEM(string.h)
#include SYSTEM(stdio.h)

...

int a(int value)
{
    return b(value) + c(value);
}
```

step 1

Divert includes when unit-testing

```
#ifdef UNIT_TESING

#  define LOCAL(x)   "mock/" ## #x
#  define SYSTEM(x) LOCAL(x)

#else

#  define LOCAL(x)   #x
#  define SYSTEM(x) <x>

#endif
```

step 2

- **Include the *source* in the test**

example.tests.c

```
#include <assert.h>

#define UNIT_TESTING
#include "example.c"

void example_test(void)
{
  assert(3 == a(42));
  ...
}
```

step 3

Create a mock environment

step 4

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
    ...
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
    ...
}
```

specific/mock/local.h

```
...
```

Test from within the mock environment

specific/mock/stdio.h

```
typedef int mock_file;

static mock_file * stdout = 0;

int fputc(int c, mock_file * f)
{
    assert(...);
}
```

specific/mock/string.h

```
int strcmp(const char * lhs, const char * rhs)
{
    assert(...);
}
```

step 5

- Write a simple utility copying input to new outpu file, except includes are commented
  - #include "abc.h➾ /* #include "abc.h" */
  - #include <def.h➾ /* #include <def.h> */
- Include this generated file in the test file
  - This leaves the original source file completely intact
  - This way all mock functions, mock data, and mock types can live inside the one test file

too brutal?

for example

excluder.rb

```
include  = Regexp.new('(\s*)#(\s*)include(.*)')

STDIN.readlines.each do |line|
  if m = include.match(line)
    line = "#if 0\n" + line + "#endif\n"
  end
  STDOUT.write line
end
```

```
cat wibble.c | ruby excluder.rb > wibble.isolated.c
```

wibble.tests.c

```
...
#include "wibble.ioslated.c"

void wibble_tests(void)
{
    ...
}
```

- Code that is impossible to unit-test...
- Is not impossible to test
- It might be hard to unit test
- It might be awkward to unit test
- It might be messy to unit test
- But it*will* be possible

impossible?

ask yourself...

- How hard are you willing to try?
- Only when you try will you find out...
- What makes code easy to unit test
- What makes code hard to unit test
- What tangled dependencies most code has

summary

- Writing unit tests takes effort
  - At first this effort is new and frightening
- That effort is rewarded many times over
  - It's never as bad as you think anyway
- Unit tests help you to write better code
  - You get better over time
- Unit tests create confidence to *refactor*
  - The codebase gets better over time
- Unit tests *improve* productivity
  - Why do cars have brakes?

books

- The C Programming Language
  - Kernighan and Ritchie
- The C Standard
  - BSI
- C: A Reference Manual
  - Harbison and Steele
- C FAQ's
  - Steve Summit
- Expert C Programming
  - Peter van der Linden
- Safer C
  - Les Hatton
- The Standard C Librar
  - P.J.Plauger

chapter and verse!

legacy code

Michael Feathers
Working Effectively With Legacy Code

> ***Legacy code is code that has no tests.***
>
> ***A unit test that takes 1/10th of a second to run is a slow unit test.***
>
> ***Characterisation tests...***

- Advice on test output

Threading

- # 5.1.2.4 Multi-threaded execution
  - para 25 - The execution of a program contains a *data race* if it contains two conflicting actions in different *threads*, at least one of which is not atomic, and neither *happens before* the other. Any such data race results in undefined behavior
  - para 18 – An evaluation A *happens before* an evaluation B if
    - A is *sequenced before* B or
    - A *inter-thread happens* before B

data races

- **5.1.2.4 Multi-threaded execution**
  - para 16 – An evaluation A *inter-thread happens before* an evaluation B if
    - A *synchronizes with* B,
    - A is *dependency ordered before* B, or
    - for some evaluation of X,
      - A *synchronizes* with X and X is *sequenced before* B,
      - A is *sequenced before* X and X *inter-thread happens before* B, or
      - A *inter-thread happens before* X and X *inter-thread happens before* B

7.17 <stdatomic.h>

sdsdsd

7.26 <threads.h>

- *sdsdsd*