ACCU 2015

"New" Features in C

Dan Saks
Saks & Associates
www.dansaks.com

1

Abstract

The first international standard for the C programming language was C90. Since then, two newer standards have been published, C99 and C11. C99 introduced a significant number of new features. C11 introduced a few more, some of which have been available in compilers for some time. Curiously, many of these added features don't seem to have caught on. Many C programmers still program in C90.

This session explains many of these "new" features, including declarations in for-statements, typedef redefinitions, inline functions, complex arithmetic, extended integer types, variable-length arrays, flexible array members, compound literals, designated initializers, restricted pointers, type-qualified array parameters, anonymous structures and unions, alignment support, non-returning functions, and static assertions.

2

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. He currently writes the online "Programming Pointers" column for *embedded.com*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught thousands of programmers around the world. He has presented at conferences such as *Software Development* and *Embedded Systems*, and served on the advisory boards for those conferences.

3

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ Standards committees and as a member of the ANSI C Standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. Previously, he was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/ Information Science from Case Western Reserve University.

4

Legal Stuff

- These notes are Copyright © 2015 by Dan Saks.
- You are free to use them for self study.
- If you'd like permission to use these notes for other purposes, contact:

Saks & Associates
393 Leander Drive
Springfield, OH 45504-4906 USA
+1-937-324-3601
dan@dansaks.com

5

6

C Standards Timeline

- **1983**: C Standards committee formed.
- **1989**: ANSI approved "C89", the first US C Standard (ANSI [1989]).
- **1990**: ISO approved C89 as "C90", the first international C Standard (ISO [1990]).
- **1999**: ISO approved "C99", a revised international C Standard (ISO [1999]).
- **2011**: ISO approved "C11", the latest international C Standard (ISO [2011]).

7

Undocumented Identifiers

- The Standard C library uses identifiers to name library components.
- For example, <*stdio.h*> declares:
  - `EOF` (a macro)
  - `FILE` (a type)
  - `printf` (a function)
  - `stdout` (an object)
- Implementing a Standard C library requires additional, undocumented identifiers.
- For example…

8

Undocumented Identifiers

- The Standard hints that `FILE` is an alias for a structure type.
- Most implementations define `FILE` as something like:

```
typedef struct _iobuf FILE;
```

- The structure name might be something other than `_iobuf`.

9

Undocumented Identifiers

- The Standard requires that:
  - A compiled module may include any standard header more than once.
  - Including a header a second or third time has no effect.
- For example,

```
#include <stdio.h>
#include <stdio.h>      // OK; no effect
#include <stdio.h>      // still OK; still no effect
```

10

Undocumented Identifiers

- Most headers use an "include guard" such as:

```
#ifndef __STDIO_H
#define __STDIO_H

// body of <stdio.h>

#endif
```

- The Standard doesn't mandate this approach, but…
- It's hard to find a compiler that doesn't do this.

11

Undocumented Identifiers

- What happens if your code declares one of these undocumented names?
- It won't compile:

```
#include <stdio.h>
~~~

struct _iobuf {     // error: invalid redefinition
    int m, n;
    ~~~
};
```

- How do you avoid such name conflicts?

12

Reserved Identifiers

- The Standard specifies that certain identifiers are ***reserved*** for the compiler and library implementation.
- This means ***you*** shouldn't use them.
- The burden is on you to know which names to avoid…

13

Reserved Identifiers

- Identifiers with these forms are ***always reserved***:

```
__reserved  // two leading underscores
_Reserved   // a leading underscore and uppercase letter
```

- You should never use names with these forms.

14

Reserved Identifiers

- Identifiers with this form are ***reserved at global scope***:

```
_reserved   // a leading underscore and lowercase letter
```

- You can use names with this form, but only for structure members or local variables:

```
struct widget {
    double _cost;   // OK here
};

int f(int i) {
    int _size;
    ~~~
}
```

15

Preserving Existing C Programs

- C90 had no boolean type.
- C programmers used to define their own, such as:

```
typedef enum { false, true } bool;
```

or:

```
typedef int boolean;
```

- C99 added a boolean type.
- However, it couldn't name the type `bool` or `boolean` without "breaking" existing code…

16

Preserving Existing C Programs

- The standards committee named the new type ***_Bool***.
  - `_Bool` was a ***reserved identifier***.
  - Now, it's a ***keyword***.
- The C99 header *<stdbool.h>* defines prettier names as macros:
  - ***bool*** as an alias for `_Bool`
  - ***false*** as an alias for the constant 0
  - ***true*** as an alias for the constant 1
- You must include *<stdbool.h>* if you want to use these names.
- C99 and C11 used this tactic with other (new) features.

17

Version Detection

- C90 specifies that Standard C implementations must define:
  - `__STDC__`
    - a macro whose value is 1 indicating that the compiler fully implements Standard C.
- However, this you can't use this macro to tell if the compiler implements C90, or C99, or C11.
- C99 added another macro…

18

Version Detection

- The standard macro __STDC_VERSION__ indicates which standard the compiler supports.
- The following code determines whether the compiler supports C90, C99 or C11:

```
#if __STDC_VERSION__ == 201112L
    /* it's C11 */
#elif __STDC_VERSION__ == 199901L
    /* it's C99 */
#elif __STDC__ == 1
    /* it's C90 */
#else
    /* it's not standard conforming */
#endif
```

19

C99 vs. C90

- C99 added many features to C90.
- Many of these features haven't caught on.
- Many C programmers don't know these features exist.
- C compiler vendors have been slow to implement some features.
- Here's some of what you may be missing…

20

// Comments

- Comments in C90 begin with **/*** and end with ***/**.
- Comments can span multiple lines.
- C99 added single-line comments that begin with **//** and end at the next newline.

21

Relaxed Declaration Ordering

- C90 doesn't allow any declarations in a block after the first statement in that block.
- C99 lets you intermix declarations and statements within a block:

```
int foo() {
    int n;                  // declaration
    scanf("%d", &n);        // statement
    char *p = malloc(n);    // declaration:
                            //     OK in C99, but not C90
    ~~~
}
```

22

Declarations in For-Statements

- In C90, the initialization clause in a for-statement can only be an expression, such as:

```
for (i = 0; i < n; ++i) {
    ~~~
}
```

- The loop-control variable must be declared prior to the for-statement.

23

Declarations in For-Statements

- In C99, as in C++, the initialization clause can be a declaration.
- In this case, the loop-control variable is local to the loop.
- That is, its scope ends immediately after the statement that's the loop body, as in:

```
for (int i = 0; i < n; ++i) {
    ~~~
}
if (i == n)                     // error: i not declared
    ~~~
for (int i = n; i > 0; --i) {   // OK: different i
    ~~~~
}
```

24

Typedef Redefinition

- C doesn't allow multiple definitions for functions and objects.
- However, it does allow multiple declarations:

```
extern int total;       // an object declaration
extern int total;       // a benign redeclaration

int foo(int);           // a function declaration
int foo(int);           // a benign redeclaration
```

- It also allows macro redefinitions:

```
#define BUFSIZ 512      // a macro definition
#define BUFSIZ 512      // a benign redefinition
```

25

Typedef Redefinition

- A header that contains only such declarations doesn't need an "include guard":

```
// lib.h

#ifndef LIB_H_INCLUDED      // don't need ...
#define LIB_H_INCLUDED      // any of ...

#define LEVEL 42            // can be redefined

int foo(int);               // can be redeclared
char *bar(char const *);    // can be redeclared

#endif                      // this
```

26

Typedef Redefinition

- C90 doesn't allow redefinition of a typedef.
- C99 lets you redefine a typedef, as long as all definitions define the same type:

```
typedef int index_type;      // OK
typedef int index_type;      // OK in C99, but not C90
typedef int ix;              // OK
typedef ix index_type;       // OK in C99, but not C90
```

- Such type definitions no longer need include guards in C99.

27

Inline Functions

- Well-written programs are often self documenting.
- For example, this is OK:

```
if (n % 2 == 0)          // if n is even
```

- This is better:

```
int even(unsigned n) {
    return n % 2 == 0;
}

if (even(n))             // no comment needed
```

28

Inline Functions

- Unfortunately, the added overhead of calling and returning from the function call can degrade performance.
- C programmers traditionally avoid this overhead by using macros instead of functions:

```
#define even(n) ((n) % 2 == 0)

if (even(n))            // still no comment needed
```

- Macros work fine in some cases, but can have problems with side effects…

29

Inline Functions

- For example,

```
#define larger_of(x, y) ((x) > (y) ? (x) : (y))
~~~
larger = larger_of(*p++, *q++);
```

expands to:

```
larger = ((*p++) >= (*q++) ? (*p++) : (*q++));
```

- This expansion increments either p or q twice and probably returns an incorrect value.

30

Inline Functions

- C99 added support for inline functions, similar to C++:

```
inline int larger_of(int x, int y) {
    return x > y ? x : y;
}
```

- ***inline*** is a keyword in C99.

31

Inline Functions

- A call to an inline function typically generates the code for the function body at the point of each call.
- For example, the expression:

```
larger = larger_of(*p, m);
```

generates code equivalent to:

```
larger = *p > m ? *p : m;
```

- Unlike a macro, an inline function always behaves like a function in that…

32

Inline Functions

- A call to an inline function evaluates each argument exactly once.
- This avoids spurious side effects.
- For example, the call expression:

```
larger = larger_of(*p++, *q++);
```

translates into something like:

```
{
    int temp1 = *p++;
    int temp2 = *q++;
    larger = temp1 > temp2 ? temp1 : temp2;
} // temp1 and temp2 are no longer in scope
```

33

Inline Functions

- An inline function also behaves like a function in that…
- A program can take the address of an inline function.
- For example, this forces the compiler to generate a non-inline copy of `larger_of`:

```
int (*pf)(int, int) = &larger_of;
```

- Calls through `pf` won't expand the function body inline.
- However, calls using `larger_of` as the function name should continue to expand inline.

34

Inline Functions

- With non-inline functions that have external linkage, you typically declare the function in a header:

```
// my_lib.h

int larger_of(int x, int y);    // do this
```

- You typically define it in an associated source file:

```
// mylib.c

int larger_of(int x, int y) {   // do this as well
    return x > y ? x : y;
}
```

35

Inline Functions

- What happens if you define a non-inline external function in a header?

```
// my_lib.h

int larger_of(int x, int y) {   // don't do this
    return x > y ? x : y;
}
```

- If you call that function from more than one source file, then you'll get a link error when you try to link the resulting object files together.
- The linker will complain that the function is ***multiply-defined***.

36

Inline Functions

- In contrast, you should define, not just declare, an inline external function in a header:

```
// my_lib.h

inline int larger_of(int x, int y) {    // do this
    return x > y ? x : y;
}
```

- A compiler can't expand a function call inline unless it has seen the function definition (not just a declaration) prior to the call.

37

Inline Functions

- If necessary, a C99 compiler automatically generates a non-inline copy of an inline function…
- …but you must decide beforehand where that non-inline copy will go.
- You do this by declaring, but not defining, the inline function in some source file, as in:

```
// larger_of.c

#include "larger_of.h"

int larger_of(int, int);        // do this as well
```

38

Inline Functions

- The keyword `extern` is optional in the function declaration in the source file, as in:

```
int larger_of(int, int);                // OK
extern int larger_of(int, int);         // OK
```

- The declaration may include the keyword `inline`, but only if the keyword `extern` is there, too:

```
extern inline int larger_of(int, int);  // OK
```

- However, this is invalid:

```
inline int larger_of(int, int);         // not OK
```

39

New Integer Types

- A boolean type:
  - keyword `_Bool`
  - additional support in *<stdbool.h>* (as described earlier)
- `long long` integer types
  - signed and unsigned
  - additional library functions, such as:

```
long long int atoll(char const *nptr);
long long int llabs(long long int j);
```

  - additional library support, such as format specifiers:

```
printf("%lld\n", atoll(s));
```

40

Complex and Imaginary Types

- C99 added support for the concept of complex numbers.
- In mathematics, a complex number can have the Cartesian (rectangular) form:

  $x + yi$

- Here, x and y are real numbers and i is the imaginary unit such that:

  $i^2 = -1$

- C supports complex numbers in Cartesian form rather than in an alternative polar form, such as $re^{i\Theta}$.

41

Complex and Imaginary Types

- C supports three types with both a real part and an imaginary part:
  - `float _Complex`
  - `double _Complex`
  - `long double _Complex`
- C also supports three types with only an imaginary part:
  - `float _Imaginary`
  - `double _Imaginary`
  - `long double _Imaginary`
- The imaginary types are optional.
  - Compilers need not implement them.

42

Complex Arithmetic

- The complex types support:
  - the arithmetic operators +, -, *, and /
  - the equality operator == and !=
  - the simple assignment operator =
  - the compound assignment operators +=, -=, *= and /=
- For example,

```
double _Complex z1, z2, z3, z4;
~~~
z4 = z1 * z2 + z3;
```

43

<complex.h>

- The standard header *<complex.h>* makes the complex and imaginary types a little easier to use.
- The header defines a pair of macros:
  - `complex`, which expands to `_Complex`
  - `_Complex_I`, which expands to a constant expression of type `float _Complex` whose value is 0 + 1***i***
- If the compiler implementation supports imaginary types, then the header also defines macros:
  - `imaginary`, which expands to `_Imaginary`
  - `_Imaginary_I`, which expands to a constant expression of type `float _Imaginary` whose value is 1***i***.

44

<complex.h>

- The header also defines another macro:
  - `I`, which expands to a representation of ***i***.
- If the implementation supports imaginary numbers, then `I` expands to `_Imaginary_I`.
- Otherwise, it expands to `_Complex_I`.
- For example, you can declare a complex number, `z`, with double precision whose initial value is 8 + 6***i*** using:

```
#include <complex.h>
~~~
double complex z = 8 + 6 * I;
```

45

Complex Arithmetic

- C permits implicit conversion from real types (`float`, `double`, and `long double`) to complex types.
- The imaginary part of the resulting complex value is always zero.
- Implicit conversion from complex to real discards the imaginary part.
- For example,

```
double r = 42;
double _Complex z1 = r;      // z1 = 42 + 0 * I;
z1 += 3 * I;                 // z1 = 42 + 3 * I;
double d = z1;               // d = 42;
```

46

<complex.h>

- *<complex.h>* provides functions that explicitly return the real and imaginary parts of a complex number, as in:

```
#include <complex.h>
~~~
double complex z = 8 + 6 * I;
printf("%g + %gi\n", creal(z), cimag(z));
```

- The output is:

```
8 + 6i
```

- The header also provides trigonometric, hyperbolic, exponential, logarithmic, and other functions for complex operands.

47

Exact-Width Integer Types

- C has always allowed each compiler to choose the storage size and range of values of each integer type.
- For years, C programmers have been defining their own exact-width types, such as:

```
typedef char sint8;             // 8-bit signed int
typedef unsigned long uint32;   // 32-bit unsigned int
```

- You don't need to define these yourself (anymore).
- C99 provides a standard set of such typedefs in *<stdint.h>*…

48

Exact-Width Integer Types

- <*stdint.h*> defines exact-width integer types such as:
  - ***int8_t*** — a signed integer exactly 8 bits wide
  - ***uint8_t*** — an unsigned integer exactly 8 bits wide
  - ***int16_t*** — a signed integer exactly 16 bits wide
  - ***uint16_t*** — an unsigned integer exactly 16 bits wide
  - …and so on for 32 and 64
- However, these types are ***optional***.
- These types are widely, but not universally, available.
- For example, the smallest addressable unit on a processor might be a 32-bit word.
- In that case, <*stdint.h*> can't provide valid definitions for `int8_t` and `int16_t` and their unsigned companions.

49

Minimum-Width Integer Types

- Fewer programmers know about and use the minimum-width types in <*stdint.h*>, such as:
  - ***int_least8_t*** — the smallest signed integer type that's at least 8 bits wide
  - ***uint_least8_t*** — the smallest unsigned integer type that's at least 8 bits wide
  - ***int_least16_t*** — the smallest signed integer type that's at least 16 bits wide
  - ***uint_least16_t*** — the smallest unsigned integer type that's at least 16 bits wide
  - …and so on for 32 and 64.
- These types are ***required***.
- These types may satisfy your needs, and may be more portable.

50

Fastest Integer Types

- Also available are:
  - ***int_fast8_t*** — the fastest signed integer type that's at least 8 bits wide
  - ***uint_fast8_t*** — the fastest unsigned integer type that's at least 8 bits wide
  - ***int_fast16_t*** — the fastest signed integer type that's at least 16 bits wide
  - ***uint_fast16_t*** — the fastest unsigned integer type that's at least 16 bits wide
  - …and so on for 32 and 64.
- These types are also ***required***.
- In some cases, one of these types is what you really want.

51

Other Extended Integer Types

- *<stdint.h>* defines greatest-width integer types:
  - ***intmax_t*** is a signed integer type that can hold the value of any signed integer.
  - ***uintmax_t*** is an unsigned integer type that can hold the value of any unsigned integer.
- These types are ***required***.

52

Other Extended Integer Types

- *<stdint.h>* might define integer types capable of holding pointers to objects (not pointers to functions):
  - ***intptr_t*** is a signed integer that can hold the value of a pointer to an object.
  - ***uintptr_t*** is an unsigned integer that can hold the value of a pointer to an object.
- These types are ***optional***:
- *<stdint.h>* can't provide `intptr_t` on a platform where:

```
sizeof(void *) > sizeof(intmax_t)
```

53

Variable-Length Arrays (VLAs)

- This declares `x` to be an "array with `n` elements of `T`":

```
T x[n];
```

- In C90, the array dimension, `n`, must be a *constant expression*.
- That is, the compiler must be able to determine `n`'s value at compile time.
- C90 rejects this declaration unless `n` is a constant.

54

Variable-Length Arrays (VLAs)

- In C90, if you want to create an array with a dimension computed at run time, you typically do something like:

```
void f(size_t n) {
    int *x = malloc(n * sizeof(int));
    for (size_t i = 0; i < n; ++i) {
        // do something with each x[i]
    }
    free(x);
}
```

- You have to remember to call `free` to avoid a memory leak.

55

Variable-Length Arrays (VLAs)

- Using this technique with multi-dimension arrays is cumbersome:

```
void f(size_t m, size_t n) {
    int *x = malloc(m * n * sizeof(int));
    for (size_t i = 0; i < m; ++i)
        for (size_t j = 0; j < n; ++j) {
            // do something with each x[n * i + j]
        }
    free(x);
}
```

- This is easy to get wrong.

56

Variable-Length Arrays (VLAs)

- C99 introduced ***variable-length arrays (VLAs)*** to simplify using arrays with non-constant dimensions:

```
void f(size_t m, size_t n) {
    int x[m][n];                // a VLA
    for (size_t i = 0; i < m; ++i)
        for (size_t j = 0; j < n; ++j) {
            // do something with each x[i][j]
        }
}
```

- With a VLA, you don't need `malloc` and `free` to manage the array storage.

57

Variable-Length Arrays (VLAs)

- VLAs can be parameters:

```
void f(size_t m, size_t n, int x[m][n]) {
    for (size_t i = 0; i < m; ++i)
        for (size_t j = 0; j < n; ++j) {
            // do something with each x[i][j]
        }
}

int main() {
    int a[6][8];
    ~~~
    f(6, 8, a);
    ~~~
}
```

58

Variable-Length Arrays (VLAs)

- If `x` is a VLA , `sizeof(x)` is not a constant expression.
  - That is, it may not be computable at compile time.
- If `x` is anything other than a VLA, `sizeof(x)` is a constant expression.
- A VLA declaration can appear only in:
  - a function parameter list, or
  - a function body.
- A VLA can't be declared `extern` or `static`.
- Interestingly,
  - In C99, VLAs are a ***required*** feature.
  - In C11, VLAs are merely ***optional***.

59

Flexible Array Members

- Suppose you need to build packet-like structures with:
  - a fixed-format header, and
  - a trailing variable-length data sequence.
- Unfortunately, you can't use a VLA:

```
typedef struct packet packet;
struct packet {
    header h;
    data d[n];      // No: n must be constant expression
};
```

- VLAs can't be structure members.

60

Flexible Array Members

- The conventional approach in C90 is to define the data portion of the packet as an array whose dimension is 1:

```
typedef struct packet packet;
struct packet {
    header h;
    data d[1];
};
```

- To allocate storage for a packet with `n` data values, you might try:

```
packet *p = malloc(sizeof(header) + n * sizeof(data));
```

- However, this might not work…

61

Flexible Array Members

- There might be padding within the structure:

```
struct packet {
    header h;
    data d[1];
};
```

might be padding in there

- In that case, computing the packet size is more complicated:

```
packet *p
    = malloc(offsetof(packet, d) + n * sizeof(data));
```

- `offsetof(t, m)` (defined in *<stddef.h>*) returns the offset in bytes of member `m` from the beginning of structure type `t`.

62

Flexible Array Members

- In C99, the last member of a structure can be an array with unspecified dimension:

```
typedef struct packet packet;
struct packet {
    header h;
    data d[];       // OK in C99: flexible array member
};
```

- Such a member is called a ***flexible array member***.

63

Flexible Array Members

- The size of a structure with a flexible array member is:
  - the size of everything in the structure,
  - including any padding,
  - up to but not including the flexible array member.
- Using a structure with a flexible array member simplifies memory size computations:

```
packet *p = malloc(sizeof(packet) + n * sizeof(data));
```

64

Compound Literals

- A ***literal*** is a programming language construct that represents a fixed or invariant value.
- For example, "xyzzy" is a ***string-literal***.
- In C90, if you want a fixed value of a structure type, you have to create a named constant object:

```
typedef struct rational rational;
struct rational {
    long num, den;
};
~~~
rational const one_half = { 1, 2 };
rational const one_third = { 1, 3 };
```

65

Compound Literals

- In C99, you can use a ***compound literal*** to represent a fixed value of a structure type:

```
typedef struct rational rational;
struct rational {
    long num, den;
};
bool rat_eq(rational lo, rational ro) {
    return (lo.num == ro.num) && (lo.den == ro.den);
}
rational r;
~~~
if rat_eq(r, (rational){ 1, 2 }) {  // compound literal
    ~~~
}
```

66

Compound Literals

- A ***compound literal*** can have an array type:

```
enum { m = 7 }
enum { n = m * sizeof(int) };
int d[m];
~~~
if (memcmp(d, (int [m]) { 8, 6, 7, 5, 3, 0, 9 }, n) == 0)
    ~~~
```

67

Compound Literals

- A compound literal type can specify an array of unknown dimension.
- In that case, the length of the brace-enclosed list determines the array dimension:

```
if (memcmp(d, (int []) { 8, 6, 7, 5, 3, 0, 9 }, n) == 0)
    ~~~
```

68

Designated Initializers

- In C90, a brace-initializer for a union can initialize only the first member:

```
typedef union glop glop;
union glop {
    int i;
    double d;
};
glop g1 = { 10 };       // initializes g1.i with 10
glop g2 = { 12.3 };     // initializes g2.i with 12
```

- The first member, `i`, is an `int`.
- The compiler converts `12.3` from `double` to `int` by discarding `.3`.
- The truncation might trigger a warning.

69

Designated Initializers

- C99 lets you initialize any member of a union by using a ***designated initializer***:

```
typedef union glop glop;
union glop {
    int i;
    double d;
};
glop g1 = { .i = 10 };      // initializes g1.i with 10
glop g2 = { .d = 12.3 };    // initializes g2.d with 12.3
```

70

Designated Initializers

- You can use designated initializers with structures to avoid accidentally writing initial values in the wrong order:

```
typedef struct date date;
struct date {
    int d;
    month m;
    int y;
};
date flub = {
    Nov, 5, 1955                    // oops! wrong order
};
date flux = {
    .m = Nov, .d = 5, .y = 1955     // OK: no mistake
};
```

71

Designated Initializers

- You can also use designated initializers with arrays.
- This is especially convenient when you want to:
  - initialize only a few elements and
  - let the other elements default initialize to zero.
- For example, these two definitions are equivalent:

```
int x[10] = { 0, 0, 0, 8, 0, 0, 0, 2 };

int x[10] = { [3] = 8, [7] = 2 };
```

- The latter initializer is easier to get right.

72

Restricted Pointers

- In C90, *<stdlib.h>* declared the standard `strcpy` function as:

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

- In C99, the declaration looks like:

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

- As with `const` and `volatile`, the keyword `restrict` is a ***type qualifier***.
- However, `restrict` can apply only to:
  - pointers to object types, or
  - pointers to incomplete types.

73

Restricted Pointers

- ***Pointer aliasing*** occurs when a program uses two or more pointers to access the same storage.
- The potential for pointer aliasing inhibits optimizations such as:
  - caching memory into CPU registers, or
  - reordering memory accesses.
- Declaring a pointer with `restrict` enables such optimizations.
- For example…

74

Restricted Pointers

- In C99, `memcpy`'s pointer parameters are restrict-qualified:

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

- `memmove`'s pointer parameters are not:

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

- The compiler may assume that `memcpy` is copying between non-overlapping objects, and optimize code accordingly.
- It may not do so for `memmove`.

75

Type-Qualified Array Parameters

- Except when it has a non-constant dimension, an array declaration in a parameter list dimension actually declares a pointer.
- That is,

```
int f(T x[]);          // x is a "pointer to T"
```

means the same as:

```
int f(T *x);           // x is a "pointer to T"
```

- If the first array dimension is present and constant, the dimension is simply ignored.

76

Type-Qualified Array Parameters

- The transformation to pointer type preserves type qualifiers, if present.
- For example,

```
int f(T const x[]);     // x is a "pointer to const T"
int g(T volatile y[]);  // y is a "pointer to volatile T"
```

means the same as:

```
int f(T const *x);      // x is a "pointer to const T"
int g(T volatile *y);   // y is a "pointer to volatile T"
```

77

Type-Qualified Array Parameters

- In C90, there's no way to declare an array parameter that's equivalent to a pointer parameter with a top-level type qualifier.
- That is, in C90 you can declare:

```
int f(T *const x);      // x is a "const pointer to T"
int g(T *volatile y);   // y is a "volatile pointer to T"
```

- However, C90 offers no way to write this using array notation.
- In C99, you can declare the functions as:

```
int f(T x[const]);      // x is a "const pointer to T"
int g(T y[volatile]);   // y is a "volatile pointer to T"
```

78

Type-Qualified Array Parameters

- In practice, declaring parameters with top-level `const` or `volatile` qualifiers is not all that useful:

```
int f(T x[const]);      // not all that useful
int g(T y[volatile]);   // not all that useful, either
```

- However, declaring array parameters with top-level `restrict` qualifiers is useful:

```
int f(T x[restrict]);   // useful
```

- It's equivalent to:

```
int f(T *restrict x);   // useful
```

79

__func__

- C99 provides a predefined identifier, `__func__`.
- It's not a macro.
- Within each function body, it's an implicitly declared object.
- Its value is the function name as a null-terminated character sequence.
- It's as if the following declaration appeared immediately after the opening brace in each function definition:

```
static char const __func__[] = "function-name";
```

- `__func__` can appear only inside a function definition.
- For example, you can use `__func__` to implement simple function call tracing…

80

__func__

```
#define enter() printf("enter: %s\n", __func__)
#define leave() printf("leave: %s\n", __func__)

void foo() {
    enter();
    ~~~
    leave();
}

void bar() {
    enter();
    ~~~
    leave();
}
```

81

C11 vs. C99

- Compared to C99, C11 adds many fewer features.
- Your compiler might not implement all of them yet.
- Here's a sampling of some features that are already available somewhere…

82

Conditional Features

- C11 classifies certain features as ***conditional***.
- A compiler need not implement a conditional feature.
- Moreover, the compiler must define a standard object-like macro to indicate that it doesn't implement the feature.
- For example, these macros include:
  - `__STDC_NO_COMPLEX__`
    - If this macro is defined, the implementation doesn't support complex types or the *<complex.h>* header.
  - `__STDC_NO_VLA__`
    - If this macro is defined, the implementation doesn't support variable length arrays.

83

Anonymous Structures and Unions

- Suppose your application employs two-dimensional shapes.
- Each shape contains some linear or angular distances that characterize the physical extent of the shape:
  - a ***circle*** has a ***radius***
  - a ***rectangle*** has a ***height*** and a ***width***
  - a ***triangle*** has ***side1***, ***side2*** and an ***angle***
  - etc.
- The shapes may also have common attributes, such as:
  - position (planar coordinates)
  - outline and fill colors

84

Anonymous Structures and Unions

- Here's a fairly traditional C implementation of the shape type:

```
typedef struct shape shape;
struct shape {
    coordinates position;
    color outline, fill;
    shape_kind kind;        // discriminator
    union {                 // discriminated union
        circle_part circle;
        rectangle_part rectangle;
        triangle_part triangle;
    } u;                    // union member name
};
```

85

Anonymous Structures and Unions

- A union paired with a value that indicates the active member of the union is called a ***discriminated union***.
- The discrete value is called a ***discriminator***.
- In C90, the union member (on the previous slide) must have a name.
- We don't need the name other than to make the compiler happy.
- So we usually give it a short name, such as ***u***.
- Still, it clutters up the code, as in…

86

Anonymous Structures and Unions

```
double shape_area(shape const *s) {
    switch (s->kind) {
    case sk_circle:
        return PI * s->u.circle.radius
            * s->u.circle.radius;
    case sk_rectangle:
        return s->u.rectangle.height
            * s->u.rectangle.width;
    case sk_triangle:
        return sin(s->u.triangle.angle)
            * s->u.triangle.side1
            * s->u.triangle.side2 / 2;
    }
    return -1;
}
```

87

Anonymous Structures and Unions

- C11 now permits anonymous structures and unions as structure or union members:

```
struct shape {
    ~~~
    shape_kind kind;
    union {                  // anonymous union
        circle_part circle;
        ~~~
    };                       // no union member name
};
```

- You can reference members of an anonymous structure or union as if they were members of the enclosing structure or union…

88

Anonymous Structures and Unions

```
double shape_area(shape const *s) {
    switch (s->kind) {
    case sk_circle:
        return PI * s->circle.radius * s->circle.radius;
    case sk_rectangle:
        return s->rectangle.height * s->rectangle.width;
    case sk_triangle:
        return sin(s->triangle.angle)
            * s->triangle.side1 * s->triangle.side2 / 2;
    }
    return -1;
}
```

- The wavy underline indicates where `u.` used to be.

89

Alignment Support

- Multibyte objects often have an alignment.
- The C Standard defines ***alignment*** as a:
  - "requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address".
- Each target processor specifies its own alignment requirements.
- 4-byte integers and pointers are often "word aligned" (at an address that's a multiple of 4).
- 8-byte floating point numbers might be:
  - word aligned, or
  - double-word aligned (at an address that's a multiple of 8).

90

Alignment Support

- A program that accesses a misaligned object produces undefined behavior.
- Possible outcomes include:
  - the processor issues a trap, or
  - the program executes properly, but more slowly than if the data were properly aligned.
- An object whose address requirement is a higher multiple than another is said to have a ***stricter alignment***.
- For example, double-word (=8) alignment is stricter than word (=4) alignment.
- Character objects always have a size of 1 (by definition).
- They have no alignment requirement.

91

Alignment Support

- For some tasks, it helps to have a type that's as strictly aligned as any on the current platform.
- Here's a common C99 way to define that type:

```
typedef union max_align_t max_align_t;
union max_align_t {
    long long int lli;
    long double ld;
    void *pv;
    void (*pfvv)(void);
 };
```

- C11 now defines `max_align_t` for you in *<stddef.h>*.

92

Alignment Support

- Again, character types have no alignment requirement.
- This could be aligned on any boundary:

```
char buffer[BUFSIZ];
```

- If you want to align the buffer to a particular boundary, you can declare it as a member of a union with an aligned member:

```
union {
    int i;                  // force word alignment
    char buffer[BUFSIZ];
} aligned;
```

93

Alignment Support

- To simplify alignment operations, C11 provides two new keywords, `_Alignas` and `_Alignof`.
- Standard header <*stdalign.h*> makes them look nicer:

```
#define alignas _Alignas
#define alignof _Alignof
```

94

Alignment Support

- `alignof` is an operator much like `sizeof`.
- `alignof(T)` yields an integer constant whose value is the alignment of type `T`:
  - `T` must be a complete object type.
  - The alignment of an array is the alignment of its element type.
- For example, here's how you can advance a "pointer to char" to the next address aligned for accessing an `int`:

```
char *p;
~~~
while ((uintptr_t)p % alignof(int) != 0)
    ++p;
```

95

Alignment Support

- Whereas `alignof` is an operator for use in expressions…
- `alignas` is a specifier for use in declarations.
- For example, this declares a character array aligned as an `int`:

```
alignas(int) char buffer[BUFSIZ];
```

- You can also specify the alignment as an integer constant.
- The above declaration is equivalent to:

```
alignas(alignof(int)) char buffer[BUFSIZ];
```

96

Alignment Support

- The standard header *<stdlib.h>* declares memory allocation functions:

```
void *calloc(size_t number, size_t size);
void *malloc(size_t size);
void *realloc(void *pointer, size_t size);
```

- If a call to any of these functions succeeds, it returns a pointer whose value is aligned so that:
  - it may be assigned to a pointer to any type of object, and
  - it may be used to access such an object or an array thereof in the space allocated.
- In practice, it usually means the returned pointer has a value aligned to `max_align_t`.

97

Alignment Support

- In C11, *<stdlib.h>* also declares:

```
void *aligned_alloc(size_t alignment, size_t size);
```

- This function lets you allocate storage at an alignment that's stricter than `max_align_t`, as in:

```
char *p = aligned_alloc(256, 4096);
```

- This allocates 4096 byes on a 256-byte boundary.

98

Alignment Support

- In general, a call to `aligned_alloc` has the form:

```
aligned_alloc(A, S)
```

- It allocates storage of size *S* aligned to boundary *A*.
- The behavior is undefined if:
  - *A* isn't a valid alignment supported by the implementation, or
  - the *S* isn't an integral multiple of *A*.

99

Non-Returning Functions

- Some functions never return to their caller.
- C11 provides the keyword `_Noreturn` to declare such functions.
- The standard header *<stdnoreturn.h>* makes it look a little nicer:

```
#define noreturn _Noreturn
```

- For example, the standard library now declares the `abort` and `exit` functions as:

```
_Noreturn void abort(void);
_Noreturn void exit(int status);
```

100

Non-Returning Functions

- Using `noreturn` has these advantages:
  - It suppresses compiler warnings on functions that don't return.
  - It enables some optimizations.
- The compiler should complain if a function declared with `noreturn` might return nonetheless.
- For example, this might return if `status` is nonnegative:

```
noreturn void bail(int status) {
    if (status < 0) {
        // do some cleanup
        exit(EXIT_FAILURE);
    }
}
```

101

Static Assertions

- The `assert` macro is defined in the standard header *<assert.h>*.
- Calling `assert(e)` expands to code that tests the value of expression `e` at run time:
  - If `e` is true (non-zero):
    - nothing happens.
  - If `e` is false (zero), the program:
    - writes a diagnostic message to `stderr`, and
    - aborts execution by calling the standard `abort` function.

102

Static Assertions

- For example, suppose you have an enumeration type defined as:

```
enum rating { worst, poor, okay, good, best };
```

- Suppose the program assumes that the range from `worst` to `best` doesn't exceed 7.
- Violating that constraint could lead to a subtle bug.
- Rather than let the program fail in some subtle way, you can force an overt failure:

```
assert(best - worst <= 7);
```

103

Static Assertions

- With most compilers, an assert failure message looks something like:

```
assert failed: condition, file file.c, line n
```

- `assert` writes to `stderr`.
- It may be useless in environments that lack support for the C file system.
- However, you can "roll your own" version of `assert`:
  - Copy the macro from *<assert.h>* to your own header.
  - Change the way it reports the failure.

104

Static Assertions

- Again, this assertion catches the constraint violation:

```
assert(best - worst <= 7);
```

- However:
  - It executes at run time.
  - It should be done at compile time.
  - An `assert` call is an executable expression.
  - It can appear only within a function.

105

Static Assertions

- Not every assertion can be checked ***statically*** (at compile time).
- An assertion that tests the value of a variable must be done ***dynamically*** (at run time).
- However, an assertion that tests the value of a constant expression can be done at compile time.
- For example, these can be tested statically:
  - the size or alignment of an object
  - the offset of a structure member
  - the value of an enumeration constant

106

Static Assertions

- C11 adds the keyword `_Static_assert` to support static assertions.
- A ***static_assert-declaration*** has the form:

```
_Static_assert(e, s);
```

- If constant expression *e* converted to `_Bool` is true, the declaration has no effect.
- Otherwise, the compiler generates a diagnostic message containing string literal *s*, and the program fails to compile.
- A static_assert-declaration is a declaration.
- It can appear anywhere that any other declaration can appear.

107

Static Assertions

- The standard header *<assert.h>* provides a macro that makes static assertions more pleasing to the eye:

```
#define static_assert _Static_assert
```

- For example, you can test the range of the enumerators using a static assertion such as:

```
static_assert(
    best - worst <= 7,
    "best shouldn't be more than 7 greater than worst"
);
```

108

No More gets

- Nearly all the differences between C11 and C99 are features that C11 added.
- However, C11 did remove one function from the standard library.
- In C11, *<stdio.h>* no longer declares:

```
char *gets(char *s);
```

- The function was just too unsafe:
  - It could easily cause an undetected buffer overrun.
- In place of gets, use:

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

109

Bibliographic References

- ANSI [1989]. *ANSI X3.159-1989, Programming Language C.*
- ISO [1990]. *ISO/IEC Standard 9899:1990, Programming languages—C.*
- ISO [1999]. *ISO/IEC Standard 9899:1999, Programming languages—C.*
- ISO [2011]. *ISO/IEC Standard 9899:2011, Programming languages—C.*

110

111

112