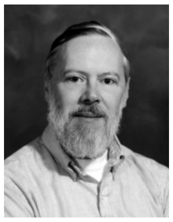
Chapter 27 The C Programming Language


Dennis M. Ritchie

Bjarne Stroustrup
www.stroustrup.com/Programming

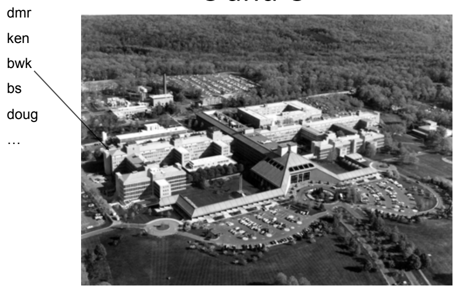
Abstract

- This lecture gives you the briefest introduction to C from a C++ point of view. If you need to use this language, read an introductory book (e.g. K&R). This lecture gives you a hint what to look for.
- C is C++'s closest relative, and compatible in many areas, so much of your C++ knowledge carries over.

Overview

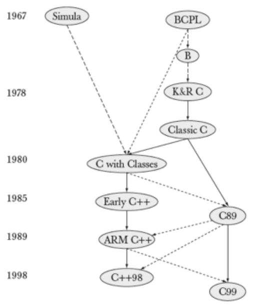
- C and C++
- Function prototypes
- **printf()/scanf()**
- Arrays and strings
- Memory management
- Macros
- **const**
- C/C++ interoperability

C and C++



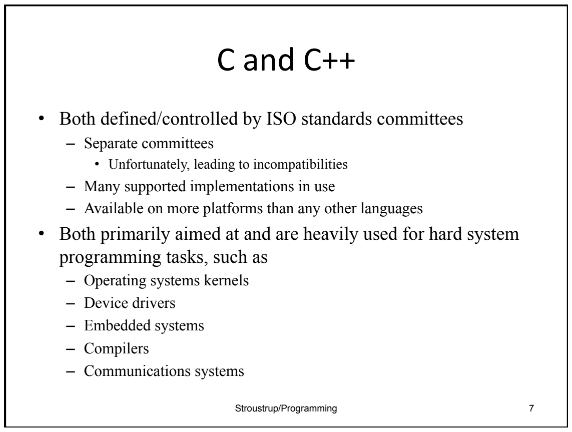
- Both were "born" in the Computer Science Research Department of Bell Labs in Murray Hill, NJ

Modern C and C++ are siblings



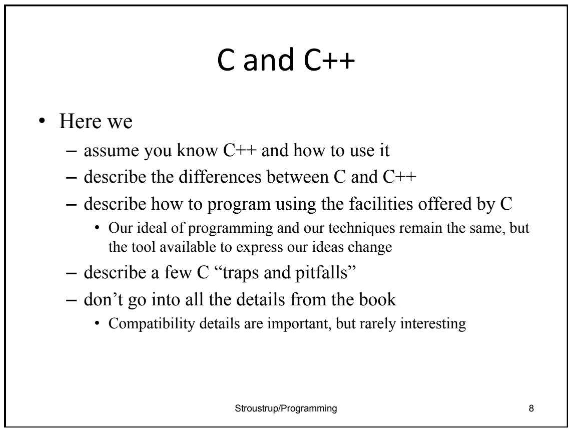
C and C++

- In this talk, I use "C" to mean "ISO C89"
  - That's by far the most commonly used definition of C
    - Classic C has mostly been replaced (though amazingly not completely)
    - C99 is not yet widely used
- Source compatibility
  - C is (almost) a subset of C++
    - Example of excepion: **int f(int new, int class, int bool);** /* *ok in C* */
  - (Almost) all constructs that are both C and C++ have the same meaning (semantics) in both languages
    - Example of exception: **sizeof('a')** /* *4 in C and 1 in C++* */
- Link compatibility
  - C and C++ program fragments can be linked together in a single program
    - And very often are
- C++ was designed to be "as close as possible to C, but no closer"
  - For ease of transition
  - For co-existence
  - Most incompatibilities are related to C++'s stricter type checking



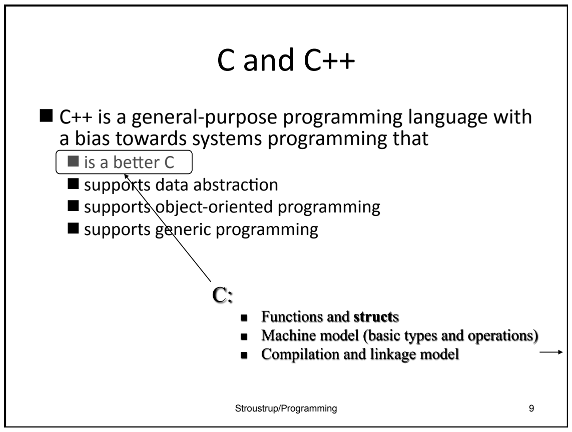
C and C++

- Both defined/controlled by ISO standards committees
  - Separate committees
    - Unfortunately, leading to incompatibilities
  - Many supported implementations in use
  - Available on more platforms than any other languages
- Both primarily aimed at and are heavily used for hard system programming tasks, such as
  - Operating systems kernels
  - Device drivers
  - Embedded systems
  - Compilers
  - Communications systems



C and C++

- Here we
  - assume you know C++ and how to use it
  - describe the differences between C and C++
  - describe how to program using the facilities offered by C
    - Our ideal of programming and our techniques remain the same, but the tool available to express our ideas change
  - describe a few C “traps and pitfalls”
  - don’t go into all the details from the book
    - Compatibility details are important, but rarely interesting

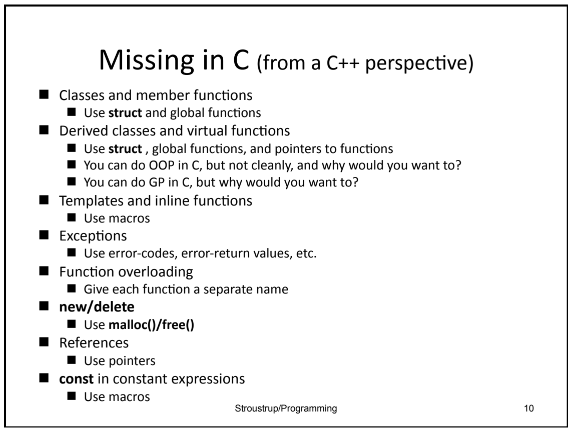


C and C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

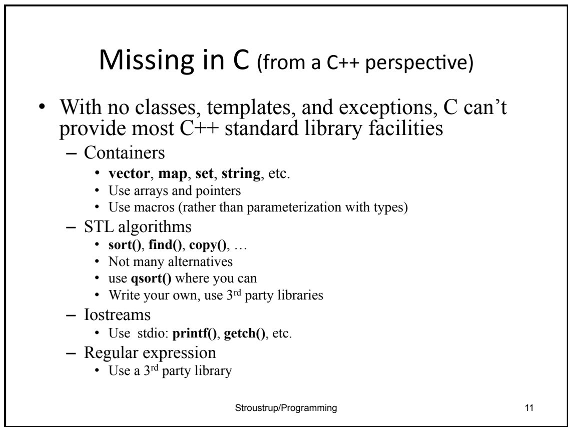
C:

- **Functions and structs**
- **Machine model (basic types and operations)**
- **Compilation and linkage model**



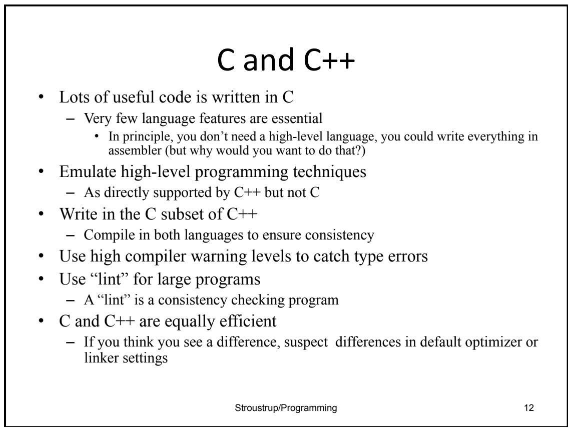
Missing in C (from a C++ perspective)

- Classes and member functions
  - Use **struct** and global functions
- Derived classes and virtual functions
  - Use **struct** , global functions, and pointers to functions
  - You can do OOP in C, but not cleanly, and why would you want to?
  - You can do GP in C, but why would you want to?
- Templates and inline functions
  - Use macros
- Exceptions
  - Use error-codes, error-return values, etc.
- Function overloading
  - Give each function a separate name
- **new/delete**
  - Use **malloc()/free()**
- References
  - Use pointers
- **const** in constant expressions
  - Use macros



Missing in C (from a C++ perspective)

- With no classes, templates, and exceptions, C can’t provide most C++ standard library facilities
  - Containers
    - **vector**, **map**, **set**, **string**, etc.
    - Use arrays and pointers
    - Use macros (rather than parameterization with types)
  - STL algorithms
    - **sort()**, **find()**, **copy()**, …
    - Not many alternatives
    - use **qsort()** where you can
    - Write your own, use 3rd party libraries
  - Iostreams
    - Use stdio: **printf()**, **getch()**, etc.
  - Regular expression
    - Use a 3rd party library



C and C++

- Lots of useful code is written in C
  - Very few language features are essential
    - In principle, you don’t need a high-level language, you could write everything in assembler (but why would you want to do that?)
- Emulate high-level programming techniques
  - As directly supported by C++ but not C
- Write in the C subset of C++
  - Compile in both languages to ensure consistency
- Use high compiler warning levels to catch type errors
- Use “lint” for large programs
  - A “lint” is a consistency checking program
- C and C++ are equally efficient
  - If you think you see a difference, suspect differences in default optimizer or linker settings

Functions

- There can be only one function of a given name
- Function argument type checking is optional
- There are no references (and therefore no pass-by-reference)
- There are no member functions
- There are no inline functions (except in C99)
- There is an alternative function definition syntax

Function prototypes

(function argument checking is optional)

```
/* avoid these mistakes – use a compiler option that enforces C++ rules */

int g(int);      /* prototype – like C++ function declaration */
int h();         /* not a prototype – the argument types are unspecified */

int f(p,b) char* p; char b;      /* old style definition – not a prototype */
{ /* … */ }

int my_fct(int a, double d, char* p)     /* new style definition – a prototype */
{
    f();         /* ok by the compiler! But gives wrong/unexpected results */
    f(d,p);      /* ok by the compiler! But gives wrong/unexpected results */
    h(d);        /* ok by the compiler! But may give wrong/unexpected results */
    ff(d);       /* ok by the compiler! But may give wrong/unexpected results */

    g(p);        /* error: wrong type */
    g();         /* error: argument missing */
}
```

printf() – many people's favorite C function

Format string

```
/* no iostreams – use stdio */

#include<stdio.h>      /* defines int printf(const char* format, …); */

int main(void)
{
    printf("Hello, world\n");
    return 0;
}
void f(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %i char %c\n", d, s, i, ch);
    printf("goof %s\n", i);   /* uncaught error */
}
```

Arguments to be formatted

Format strings

Formatting characters

scanf() and friends

```
/* the most popular input functions from <stdio.h>: */
int i = getchar();    /* note int, not char;
                getchar() returns EOF when it reaches end of file */
p = gets();      /* read '\n' terminated line into char array pointed to by p */

void f(int* pi, char* pc, double* pd, char* ps)
{    /* read into variables whose addresses are passed as pointers:  */
     scanf("%i %c %g %s", pi, pc, pd, ps);
     /* %s skips initial whitespace and is terminated by whitespace  */
}
int i; char c; double d; char s[100]; f(&i, &c, &d, s);  /* call to assign to i, c, d, and s */
```

- Don't *ever* use **gets()** or **scanf("%s")**!
  - Consider them poisoned
  - They are the source of **many** security violations
  - An overflow is easily arranged and easily exploitable
  - Use **getchar()**

printf() and scanf() are not type safe

```
double d = 0;
int s = 0;
printf("d: %d , s: %s\n", d, s);  /* compiles and runs
                      the result might surprise you */
```

"s" for "string"

"d" for "decimal", not "double"

- Though error-prone, **printf()** is convenient for built-in types
- **printf()** formats are not extensible to user-defined types
  - E.g. no **%M** for **My_type** values
- Beware: a **printf ()** with a user-supplied format string is a cracker tool

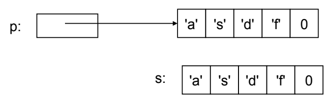
Arrays and pointers

- Defined almost exactly as in C++
- In C, you have to use them essentially all the time
  - because there is no **vector, map, string**, etc.
- Remember
  - An array doesn't know how long it is
  - There is no array assignment
    - use **memcpy()**
  - A C-style string is a zero-terminated array

C-style strings

- In C a string (called a C-string or a C-style string in C++ literature) is a zero-terminated array of characters

```
char* p = "asdf";
char s[ ] = "asdf";
```



C-style strings

- Comparing strings

```
#include <string.h>
if (s1 = = s2) {            /* do s1 and s2 point to the same array?
                           (typically not what you want) */
}
if (strcmp(s1,s2) = = 0) { /* do s1 and s2 hold the same characters? */
}
```

- Finding the lengths of a string

```
int lgt = strlen(s); /* note: goes through the string at run time
                   looking for the terminating 0 */
```

- Copying strings

```
strcpy(s1,s2);      /* copy characters from s2 into s1
                   be sure that s1 can hold that many characters */
```

C-style strings

- The string copy function **strcpy()** is the archetypical C function (found in the ISO C standard library)
- Unless you understand the implementation below, don't claim to understand C:

```
char* strcpy(char *p, const char *q)
{
    while (*p++ = *q++);
    return p;
}
```

- For an explanation see for example K&R or TC++PL

Standard function libraries

- **<stdio.h> printf()**, **scanf()**, etc.
- **<string.h>** **strcmp()**, etc.
- **<ctype.c>** **isspace()**, etc.
- **<stdlib.h>** **malloc()**, etc.
- **<math.h>** **sqrt()**, etc.

- Warning: By default, Microsoft tries to force you to use safer, but non-standard, alternatives to the unsafe C standard library functions

Free store: malloc()/free()

```
#include<stdlib.h>

void f(int n) {
    /* malloc() takes a number of bytes as its argument */
    int* p = (int*)malloc(sizeof(int)*n);      /* allocate an array of n ints */
    /* … */
    free(p);    /* free() returns memory allocated by malloc() to free store */
}
```

Free store: malloc()/free()

- Little compile-time checking

```
/* malloc() returns a void*. You can leave out the cast of malloc(), but don't */
double* p = malloc(sizeof(int)*n);/* probably a bug */
```

- Little run-time checking

```
int* q = malloc(sizeof(int)*m);  /* m ints */
for (int i=0; i<n; ++i) init(q[i]);
```

- No initialization/cleanup
  - **malloc()** doesn't call constructors
  - **free()** doesn't call destructors
  - Write and remember to use your own **init()** and **cleanup()**
- There is no way to ensure automatic cleanup
- Don't use **malloc()/free()** in C++ programs
  - **new/delete** are as fast and almost always better

Uncast **malloc()**

- The major C/C++ incompatibility in real-world code
  - Not-type safe
  - Historically a pre-standard C compatibility hack/feature
- Always controversial
  - Unnecessarily so IMO

```
void* alloc(size_t x); /* allocate x bytes
                 in C, but not in C++, void* converts to any T* */
void f (int n)
{
    int* p = alloc(n*sizeof(int));          /* ok in C; error in C++ */
    int* q = (int*)alloc(n*sizeof(int));    /* ok in C and C++ */
    /* … */
}
```

void*

- Why does void* convert to T* in C but not in C++?
  - C needs it to save you from casting the result of **malloc()**
  - C++ does not: use **new**
- Why is a **void*** to **T*** conversion not type safe?

```
void f()
  {
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q;  /* unsafe, legal C; error in C++ */
    *pp = -1;     /* overwrite memory starting at &i */
  }
```

Comments

- // comments were introduced by Bjarne Stroustrup into C++ from C's ancestor BCPL when he got really fed up with typing /* … */ comments
- // comments are accepted by most C dialects including the new ISO standard C (C99)

const

```
// in C, a const is never a compile time constant
const int max = 30;
const int x;  // const not initialized: ok in C (error in C++)

void f(int v)
{
    int a1[max];  // error: array bound not a constant (max is not a constant!)
    int a2[x];  // error: array bound not a constant (here you see why)
    switch (v) {
    case 1:
      // …
    case max:     // error: case label not a constant
      // …
    }
}
```

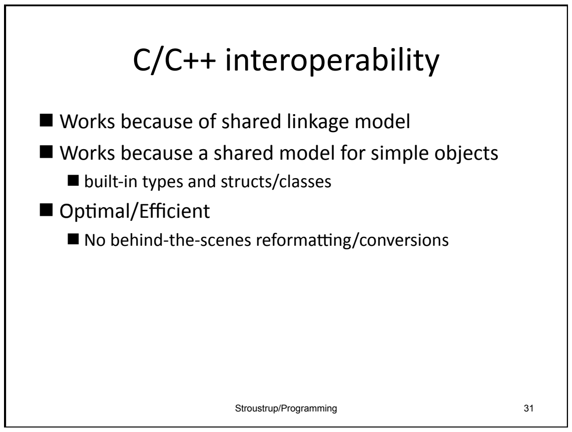
Instead of **const** use macros

```
#define  max 30

void f(int v)
{
    int a1[max];  // ok
    switch (v) {
    case 1:
      // …
    case max:     // ok
      // …
    }
}
```

Beware of macros

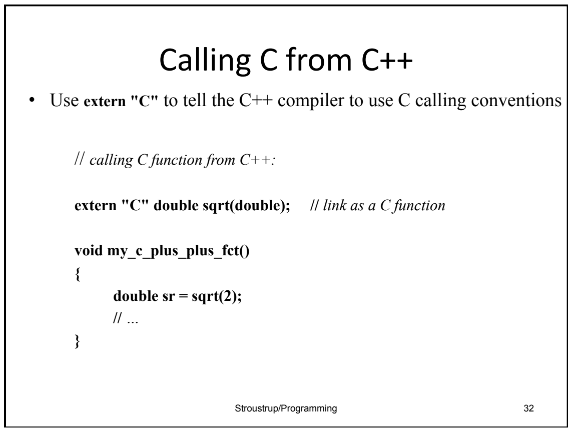
```
#include "my_header.h"
// …
int max(int a, int b) { return a>=b?a:b; }  // error: "obscure error message"
```

- As it happened **my_header.h** contained the macro **max** from the previous slide so what the compiler saw was
  int 30(int a, int b) { return a>=b?a:b; }
- No wonder it complained!
- There are tens of thousands of macros in popular header files.
- Always define macros with **ALL_CAPS** names, e.g.
  #define MY_MAX 30
  and never give anything but a macro an **ALL_CAPS** name
- Unfortunately, not everyone obeys the ALL_CAPS convention



C/C++ interoperability

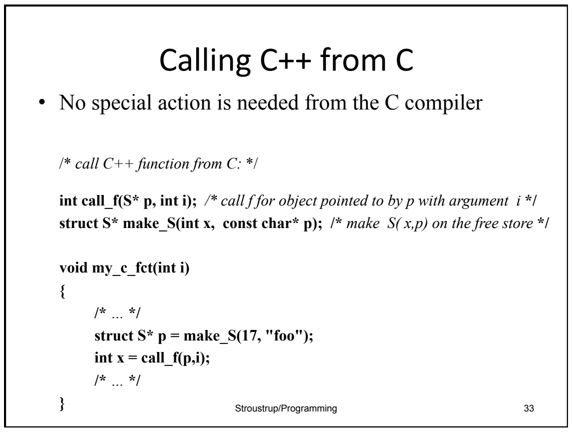
- Works because of shared linkage model
- Works because a shared model for simple objects
  - built-in types and structs/classes
- Optimal/Efficient
  - No behind-the-scenes reformatting/conversions



Calling C from C++

- Use **extern "C"** to tell the C++ compiler to use C calling conventions

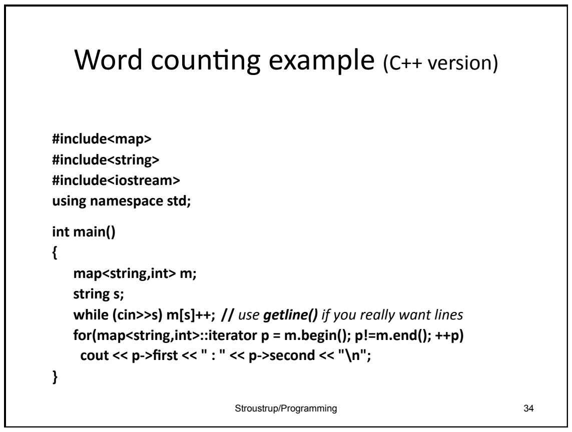
```
// calling C function from C++:

extern "C" double sqrt(double);     // link as a C function

void my_c_plus_plus_fct()
{
        double sr = sqrt(2);
        // ...
}
```



Calling C++ from C

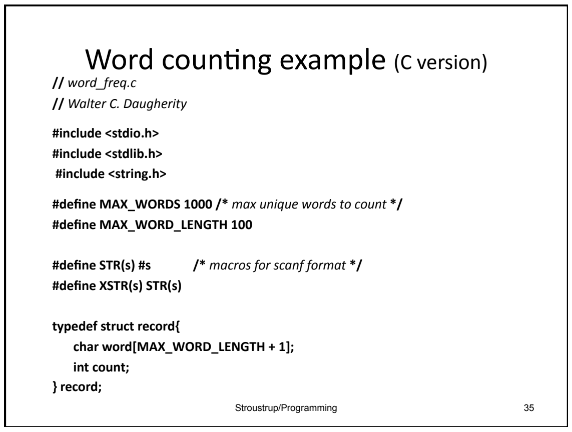
- No special action is needed from the C compiler

```
/* call C++ function from C: */

int call_f(S* p, int i);  /* call f for object pointed to by p with argument  i */
struct S* make_S(int x,  const char* p);  /* make  S( x,p) on the free store */

void my_c_fct(int i)
{
        /* ... */
        struct S* p = make_S(17, "foo");
        int x = call_f(p,i);
        /* ... */
}
```



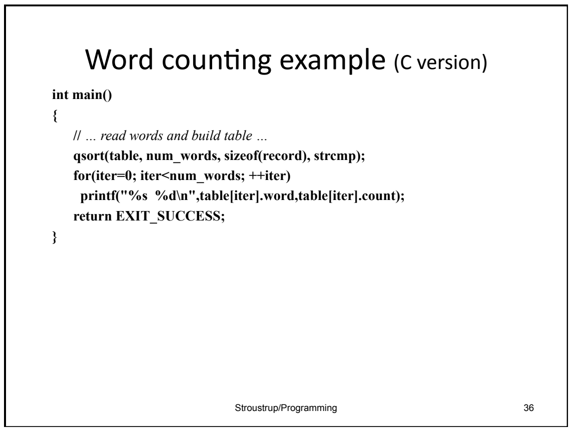
Word counting example (C++ version)

```
#include<map>
#include<string>
#include<iostream>
using namespace std;

int main()
{
    map<string,int> m;
    string s;
    while (cin>>s) m[s]++; // use getline() if you really want lines
    for(map<string,int>::iterator p = m.begin(); p!=m.end(); ++p)
      cout << p->first << " : " << p->second << "\n";
}
```



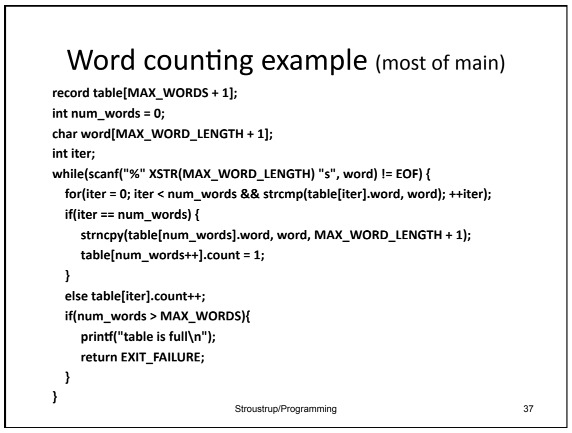
Word counting example (C version)

```
// word_freq.c
// Walter C. Daugherity

#include <stdio.h>
#include <stdlib.h>
 #include <string.h>

#define MAX_WORDS 1000 /* max unique words to count */
#define MAX_WORD_LENGTH 100

#define STR(s) #s          /* macros for scanf format */
#define XSTR(s) STR(s)

typedef struct record{
    char word[MAX_WORD_LENGTH + 1];
    int count;
} record;
```



Word counting example (C version)

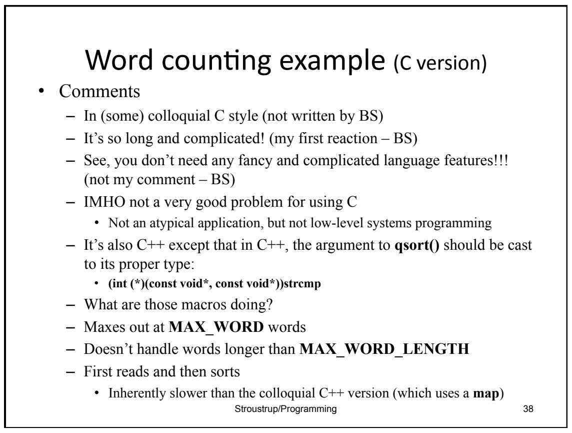
```
int main()
{
    // ... read words and build table ...
    qsort(table, num_words, sizeof(record), strcmp);
    for(iter=0; iter<num_words; ++iter)
      printf("%s  %d\n",table[iter].word,table[iter].count);
    return EXIT_SUCCESS;
}
```



Word counting example (most of main)

```
record table[MAX_WORDS + 1];
int num_words = 0;
char word[MAX_WORD_LENGTH + 1];
int iter;
while(scanf("%" XSTR(MAX_WORD_LENGTH) "s", word) != EOF) {
   for(iter = 0; iter < num_words && strcmp(table[iter].word, word); ++iter);
   if(iter == num_words) {
       strncpy(table[num_words].word, word, MAX_WORD_LENGTH + 1);
       table[num_words++].count = 1;
   }
   else table[iter].count++;
   if(num_words > MAX_WORDS){
       printf("table is full\n");
       return EXIT_FAILURE;
   }
}
```

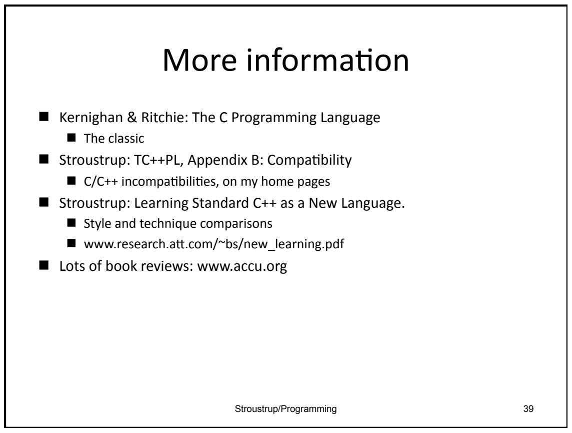
Stroustrup/Programming 37



Word counting example (C version)

- Comments
  - In (some) colloquial C style (not written by BS)
  - It's so long and complicated! (my first reaction – BS)
  - See, you don't need any fancy and complicated language features!!! (not my comment – BS)
  - IMHO not a very good problem for using C
    - Not an atypical application, but not low-level systems programming
  - It's also C++ except that in C++, the argument to **qsort()** should be cast to its proper type:
    - **(int (*)(const void*, const void*))strcmp**
  - What are those macros doing?
  - Maxes out at **MAX_WORD** words
  - Doesn't handle words longer than **MAX_WORD_LENGTH**
  - First reads and then sorts
    - Inherently slower than the colloquial C++ version (which uses a **map**)

Stroustrup/Programming 38



More information

- Kernighan & Ritchie: The C Programming Language
  - The classic
- Stroustrup: TC++PL, Appendix B: Compatibility
  - C/C++ incompatibilities, on my home pages
- Stroustrup: Learning Standard C++ as a New Language.
  - Style and technique comparisons
  - www.research.att.com/~bs/new_learning.pdf
- Lots of book reviews: www.accu.org

Stroustrup/Programming 39