ESC-205

C++ for Embedded C Programmers

Dan Saks
Saks & Associates
www.dansaks.com

1

Abstract

The C++ programming language is a superset of C. C++ offers additional support for object-oriented and generic programming while enhancing C's ability to stay close to the hardware. Thus, C++ should be a natural choice for programming embedded systems. Unfortunately, many potential users are wary of C++ because of its alleged complexity and hidden costs.

This session explains the key features that distinguish C++ from C. It sorts the real problems from the imagined ones and recommends low-risk strategies for adopting C++. Rather than tell you that C++ is right for you, this session will help you decide for yourself.

2

Legal Stuff

- These notes are Copyright © 2013 by Dan Saks.
- If you have attended this seminar, then:
  - You may make printed copies of these notes for your personal use, as well as backup electronic copies as needed to protect against loss.
  - You must preserve the copyright notices in each copy of the notes that you make.
  - You must treat all copies of the notes — electronic and printed — as a single book. That is,
    - You may lend a copy to another person, as long as only one person at a time (including you) uses any of your copies.
    - You may transfer ownership of your copies to another person, as long as you destroy all copies you do not transfer.

3

More Legal Stuff

- If you have not attended this seminar, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the seminar, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

4

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

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 is also a Microsoft MVP.

5

6

The “++” in C++

- C++ is a programming language based on the C language.
- Like C, C++ is a general-purpose language.
  - It’s not targeted toward any particular application domain.
- C++ retains C’s ability to deal efficiently with bits and bytes.
- C++ is particularly useful for embedded systems programming.

7

The “++” in C++

- C++ extends C with features that support large-scale programming.
- These features help you organize large programs into smaller, simpler units.
- Compared to C, C++ lets you draw boundaries between subunits:
  - more clearly
  - more reliably
  - no less efficiently (and sometimes even more efficiently)

8

The “++” in C++

- One way to simplify building large systems is to build them from libraries of components:
  - functions
  - objects
  - types
- You can produce better software in less time by:
  - using components that others have written and tested, and
  - returning the favor.
    - That is, when feasible, package parts of your application(s) as components to share.
- C++ offers rich features for building libraries of components.

9

The “++” in C++

- C++ provides better support for large-scale development:
  - object-oriented programming
    - classes
    - class derivation (inheritance)
    - virtual functions (polymorphism)
  - generic programming
    - templates
  - global name management
    - namespaces
- C++11 (the current Standard) provides better support for low-level programming.

10

Saying "Hello"

- Here's the classic "Hello, world" program in Standard C:

```
// "Hello, world" in Standard C

#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

- This is also a Standard C++ program.

11

Saying "Hello"

- Here's the same program in a distinctively C++ style:

```
// "Hello, world" in Standard C++

#include <iostream>

int main() {
    std::cout << "Hello, world\n";
    return 0;
}
```

- The ***bold italic*** text indicates the few places where the C++ program differs from the C program.

12

What's Different?

- The latter program uses the standard header *<iostream>* instead of *<stdio.h>*.
- *<iostream>* declares the Standard C++ Library's input and output components.
- C++ provides *<iostream>* in addition to, not instead of, *<stdio.h>*.

13

What's *Really* Different?

- This statement uses components declared in *<iostream>* to write the value of `"Hello, world\n"` to standard output:

```
std::cout << "Hello, world\n";
```

- The effect is essentially the same as calling:

```
printf("Hello, world\n");
```

- Most C programmers are already familiar with *<stdio.h>*.
- Using << as an output operator isn't obviously better than calling `printf`.
- Why bother mastering a different library?

14

Why Use a Different I/O Library?

- Again, C++ was designed to support large-scale programming.
- In a tiny program such as "Hello, world", it's hard to see an advantage for *<iostream>* over *<stdio.h>*.
- In a big program, it's much easier.

15

Why Use a Different I/O Library?

- Large programs deal with application-specific data formed from the primitive data types already in the language.
- For example, applications often handle data such as:
  - calendar dates
  - clock times
  - physical devices (ports, timers, etc.)
  - data collections (sequences, sets, etc.)
  - and so on

16

User-Defined Types

- In C, you might represent clock times as:

```
struct clock_time {
    unsigned char hrs, mins, secs;
};
~~~
struct clock_time t;
```

- That is, you'd invent a data type called `clock_time` and declare variables of that type representing clock times.
- A type such as `clock_time` is a ***user-defined type***.
  - The ***user*** is you, the programmer.

17

User-Defined Types

- How do you write a `clock_time` to a file?
- If `clock_time` were a built-in type, *<stdio.h>* would provide a format specifier for `clock_time`.
- That is, you can write an integer `i` to file `f` using the `%d` format:

```
fprintf(f, "The value is %d", i);       // can do
```

- You should be able to write a `clock_time t` using, say:

```
fprintf(f, "The time is %t", t);        // we wish
```

- Standard C doesn't have a `%t` format, or anything like it.

18

User-Defined Types

- *<stdio.h>* provides format specifiers only for built-in types.
- You can't extend *<stdio.h>* to provide format specifiers for user-defined types.
  - Not easily.
- Rather than use a single format specifier for `clock_time`, you must write something such as:

```
fprintf(
    f, "The time is %2u:%02u:%02u", t.hrs, t.mins, t.secs
);
```

- This isn't nearly as easy to write as:

```
fprintf(f, "The time is %t", t);
```

19

User-Defined Types

- In C, user-defined types don't look like built-in types.
  - They often introduce little details that complicate programs.
  - In large programs, the little details add up to lots of complexity.
- As in C, C++ lets you define new types.
- But more than that…
- C++ lets you define new types that look and act an awful lot like built-in types.
- For example, C++ lets you extend the facilities of *<iostream>* to work for user-defined types such as `clock_time`…

20

User-Defined Types

- In particular, you can define a function named `operator<<` such that you can display a `clock_time t` using:

```
std::cout << "The time is ";     // (1)
std::cout << t;                  // (2)
```

- Both lines use the same notation for operands of different types:
  1) displays a value of built-in type (array of `char`)
  2) displays a value of user-defined type (`clock_time`)
- You can even collapse (1) and (2) to just:

```
std::cout << "The time is " << t;
```

21

Operator Overloading

- This statement makes `clock_time` look like any other type:

```
std::cout << t;                  // (2)
```

- The compiler translates that statement into the function call:

```
operator<<(std::cout, t);
```

- Despite the function's odd-looking name, the call behaves just like any other call.
- The `operator<<` function is an overloaded operator.
- ***Operator overloading*** is the ability to define new meanings for operators.

22

Abstract Data Types

- Object-oriented design (OOD) and programming (OOP) emphasize building programs around data types.
  - Those data types should be abstractions.
- If done properly, an abstract type:
  - describes behavior (what an object of that type does)
  - hides implementation details (how the object does whatever it does)

23

Primitive Data Types

- C++ provides:
  - ***primitive types (arithmetic and pointer types)***
    - essentially the same as in C
  - ***enumeration types (user-defined scalar types)***
    - better type checking than in C
    - more powerful than in C
  - ***aggregate types (arrays, structures and unions)***
    - lacking some features of C99, but otherwise…
    - generally more powerful than in C

24

Classes

- C doesn't really have facilities for defining truly abstract types.
- C++ provides a general mechanism, ***classes***, for specifying new types that are truly abstract.
- Classes are the essential feature that distinguishes C++ from C.

25

Classes and Objects

- An ***object*** is a unit of data storage that has the properties associated with a class.
- To a great extent, saying:

  "An ***object*** is an instance of a class."

  is just another way to say:

  "A ***variable*** is an instance of a type."

26

Crafting New Data Types

- A central focus of object-oriented programming—and C++ programming—is crafting user-defined data types as classes.
- The basics of classes in C++ are not all that complicated.
- However, C++ is complicated, in large part because:
  - C++ goes to ***great*** lengths to let you fashion user-defined types that look and act very much as if they were built in.
- The language was designed assuming:
  - A user-defined type that looks and acts built-in should be ***easier to use correctly***, and ***harder to use incorrectly*** than it would be otherwise.

27

Contrasting C with C++

- The following example illustrates basic class concepts in C++.
- It does so by contrasting a fairly traditional procedure-oriented C program with an object-oriented C++ program.
- The example program is called *xr*.
  - It's a simple cross-reference generator.
  - Posed as exercise 6-3 in Kernighan and Ritchie [1988].
  - Solved by Tondo and Gimpel [1989].

28

What the Program Does

- *xr* reads text from standard input.
- It writes a cross-reference listing to standard output.
- A typical line of output looks like:

```
Jenny :   8   67 5309
```

word — *the numbers of the lines on which that word appears*

- Even if “Jenny” actually appears more than once on any line, each line appears only once in the output sequence of line numbers for “Jenny”.

29

xr’s Data Structure

- *xr* builds the cross-reference as a unbalanced binary tree.
- Each node in the tree contains:
  - the spelling of a word
  - a sequence of line numbers on which that word appears in the input
- The structure definition for the tree looks like:

```
struct tnode {
    char *word;
    linklist *lines;
    tnode *left, *right;
};
```

30

Watching a Tree Grow

- To visualize the data structure, suppose the input text contains these lyrics from “I am the Walrus” by the Beatles [1967]:

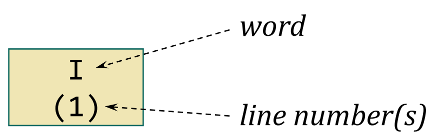
  I am the eggman.
  They are the eggmen.
  I am the Walrus.

- In the ASCII collating sequence, uppercase letters are less than lowercase letters.
- However, the following illustration assumes that the ordering is case-insensitive…

31

Watching a Tree Grow

I *am the eggman.*
They are the eggmen.
I am the Walrus.



- Note that, in this and subsequent diagrams, every node contains:
  - exactly one word, and
  - at least one line number.

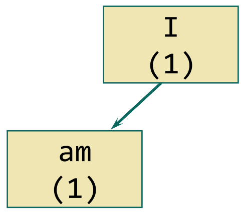
32

Watching a Tree Grow

I am *the eggman.*

They are the eggmen.

I am the Walrus.



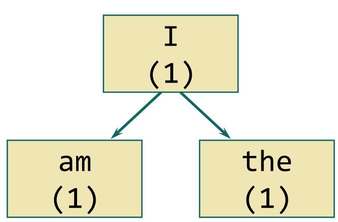
33

Watching a Tree Grow

I am the *eggman.*

They are the eggmen.

I am the Walrus.



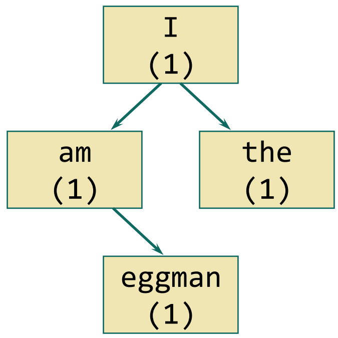
34

Watching a Tree Grow

I am the eggman.

They are the eggmen.

I am the Walrus.



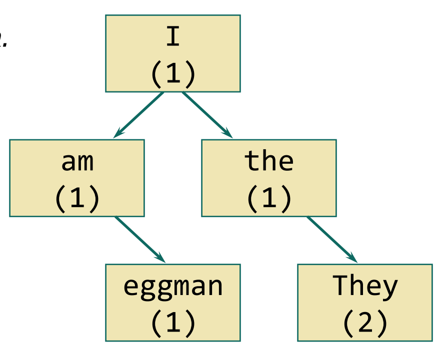
35

Watching a Tree Grow

I am the eggman.

They are the eggmen.

I am the Walrus.



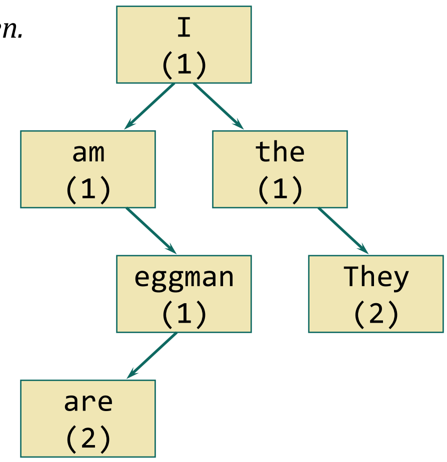
36

Watching a Tree Grow

I am the eggman.

They are *the eggmen.*

I am the Walrus.



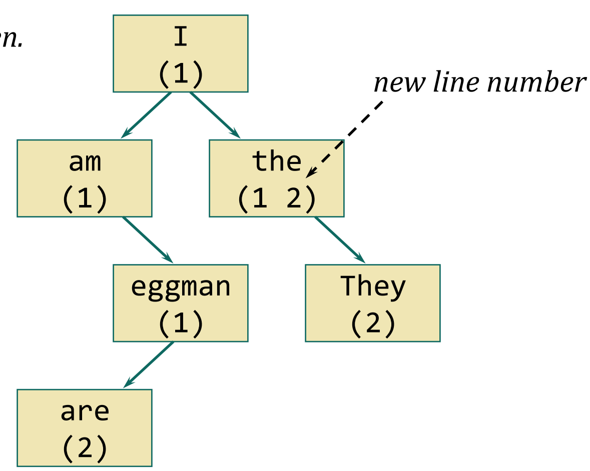
37

Watching a Tree Grow

I am the eggman.

They are the *eggmen.*

I am the Walrus.



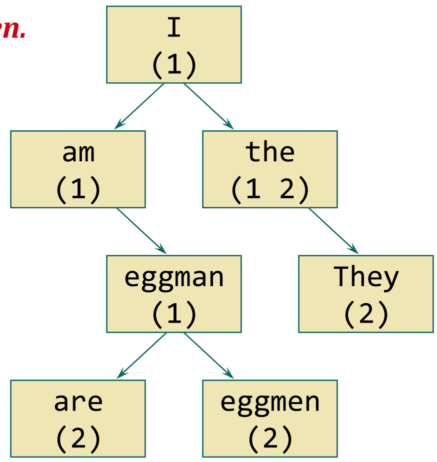
38

Watching a Tree Grow

I am the eggman.

They are the eggmen.

I am the Walrus.



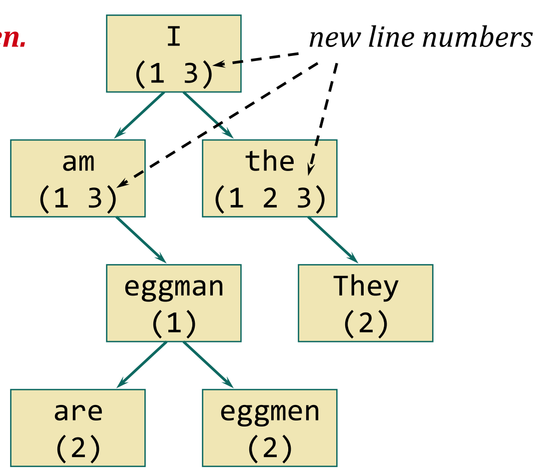
39

Watching a Tree Grow

I am the eggman.

They are the eggmen.

I am the *Walrus.*



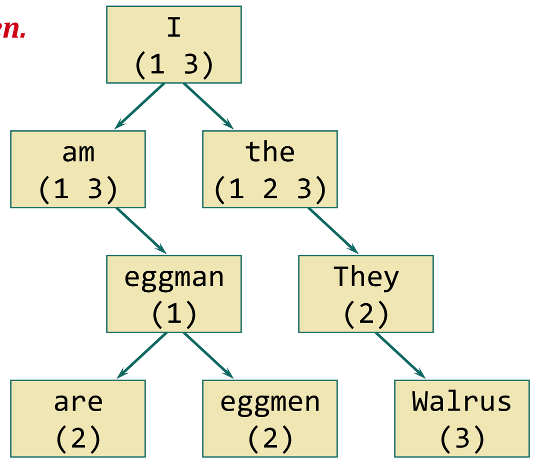
40

Watching a Tree Grow

I am the eggman.

They are the eggmen.

I am the Walrus.



41

Implementing *xr*

- *xr* uses this function to read the input:

```
int getword(char *word, int lim);
```

- Calling `getword(w, m)` reads (from standard input) the next word or single non-alphabetic character.
  - It copies at most the first `m` characters of that word or that single character into `w` along with a null character, and returns `w[0]`.

42

Implementing *xr*

- *xr* uses this function to add words and line numbers to the tree:

```
tnode *addtreex(tnode *p, char *w, int ln);
```

- Calling `addtreex(p, w, n)` adds word `w` and line number `n` to the tree whose root node is at address `p` (but only if they're not already in the tree).
- *xr* uses this function to display the results:

```
void treexprint(tnode *p);
```

- Calling `treexprint(p)` writes (to standard output) the contents of the tree whose root is at address `p`.

43

Implementing *xr*

- The `main` function is defined as:

```
int main() {
    int linenum = 1;
    tnode *root = NULL;
    char word[MAXWORD];
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtreex(root, word, linenum);
        else if (word[0] == '\n')
            ++linenum;
    treexprint(root);
    return 0;
}
```

44

Evidence of Excess Complexity

- `main`'s job is to:
  - keep track of the input line number
  - determine when a word and its line number should go into the cross reference
  - determine when to print the table
- `main` need not "know" how the cross-reference table is implemented.
- In fact, "knowing" only makes `main` more complex than it has to be.

45

Evidence of Excess Complexity

- Unfortunately, it's evident from reading `main` that the cross-reference table is a tree:
  - The cross-reference object is declared as:

```
tnode *root = NULL;
```

  - Each cross-referencing function has a parameter of type **`tnode *`**.
  - Each cross-referencing function has **`treex`** in its name.

46

Evidence of Excess Complexity

- Suppose you later changed the program to use a different data structure, say a hash table.
- This interface, particularly names using the word `tree`, would be inappropriate, if not downright confusing.
- Again, this evidence that the cross-reference table is implemented as a tree adds conceptual complexity to `main`.
- Fortunately, that complexity is avoidable…

47

Encapsulating with Classes

- *xr* should be organized so that the implementation of the cross-reference table is completely hidden from the main function.

✓ *Encapsulate design decisions inside classes.*

- You can define a `cross_reference_table` class that encapsulates the data representation and functionality of a cross-reference table.
- Then implement *xr* using that class…

48

Encapsulating with Classes

- The class definition should look something like…

```
class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;
    tnode *root;
};
```

49

Class Concepts

- A C++ class is a C structure, and then some.
- A class can contain data declarations, just like a C structure:

```
class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;
    tnode *root;
};
```

50

Class Concepts

- A class can also contain function declarations:

```
class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;
    tnode *root;
};
```

51

Class Concepts

- A class can also contain constant and type declarations.
- This class contains a type declaration:

```
class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;
    tnode *root;
};
```

52

Class Concepts

- The constants, data, functions and types declared in a class are its ***members***.
  - The ***data members*** specify the data representation for every object of that class.
  - The ***member functions*** specify fundamental operations that a program can apply to objects of that class.
  - The ***member constants*** and ***member types*** specify additional properties associated with the class.
- Why "data members" aren't "member data" is a mystery.

53

Encapsulating with Classes

- A class can, and often does, contain ***access specifiers***:

```
class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;
    tnode *root;
};
```

54

Access Specifiers

- The public class members are:
  - the ***interface*** to the services that a class provides to its users.
  - accessible everywhere in the program that the class is visible.
- The private class members are:
  - the ***implementation details*** behind the class interface.
  - accessible only to other members of the same class.
    - (This last statement is oversimplified, but sufficient for now.)

55

Encapsulating with Classes

- Here's a more complete view of the header that define the class:

```
// table.h - a cross reference table class
~~~

class cross_reference_table {
public:
    cross_reference_table();
    void insert(char const *w, int ln);
    void put();
private:
    struct tnode;               // tnode is incomplete
    tnode *root;
};
~~~
```

56

Encapsulating with Classes

```
// table.h – a cross reference table class (continued)

struct cross_reference_table::tnode {
    char *word;
    linklist *lines;
    tnode *left, *right;
};                              // tnode is now complete

inline
cross_reference_table::cross_reference_table():
    root (NULL) {
}

~~~
```

57

Encapsulating with Classes

```
// table.h – a cross reference table class (continued)

inline void
cross_reference_table::insert(char const *w, int ln) {
    root = addtreex(root, w, ln);
}

inline
void cross_reference_table::put() {
    treexprint(root);
}
```

58

main Before

- Here's the `main` function as it was originally:

```
int main() {
    int linenum = 1;
    tnode *root = NULL;
    char word[MAXWORD];
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtreex(root, word, linenum);
        else if (word[0] == '\n')
            ++linenum;
    treexprint(root);
    return 0;
}
```

59

main After

- And here it is using the `cross_reference_table` class:

```
int main() {
    int linenum = 1;
    cross_reference_table table;
    char word[MAXWORD];
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            table.insert(word, linenum);
        else if (word[0] == '\n')
            ++linenum;
    table.put();
    return 0;
}
```

60

Encapsulation Support

- The `cross_reference_table` class:
  - completely hides the table implementation from `main`, and
  - prevents future maintainers from inadvertently violating the table abstraction.
- Using inline functions avoids adding any run-time cost.

61

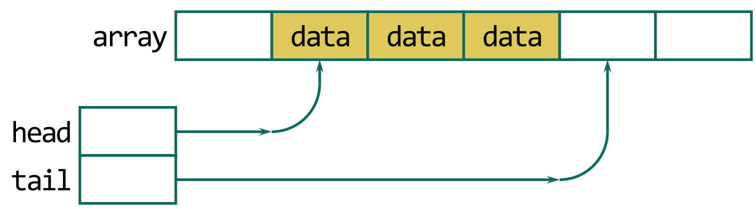
Encapsulation Support

- C programmers typically implement abstract types using some combination of:
  - incomplete types
  - separate compilation
  - internal linkage (via the keyword `static`)
- In C, you get to choose your poison:
  - poor compiler enforcement of the abstraction
  - loss of performance because you can't use inlining
  - excessively restrictive storage management policies
- For example…

62

A Ring Buffer

- Consider the implementation of a ***circular queue*** or ***ring buffer***.
- You might use a ring buffer to buffer character data coming from or going to a device such as a serial port.
- A ring buffer is a first-in-first-out data structure:
  - You insert data at the buffer's tail (the back end).
  - You remove data from the head (the front end).
- Visualize something like:



63

A Ring Buffer

- A typical implementation for a character ring buffer uses three variables:

```
char array[N];
sig_atomic_t head, tail;
```

- `N` is the dimension for `array` (presumably declared previously).
- `sig_atomic_t` is the standard integer type of an object that can be accessed atomically.
  - For thread safety.

64

A Ring Buffer

- In effect, the head and tail chase each other around the array.
- Initially, the head and tail have the same value, indicating an empty ring buffer.
- As the tail pulls away from the head, the buffer fills up.
- If the tail gets so far ahead that it wraps around and catches up to the head, the buffer will be full.
- As the head catches up to the tail, the buffer empties.
- When the head completely overtakes the tail, the buffer is empty once again.

65

A Ring Buffer "Class" in C

- You can implement the ring buffer "class" in C as:
  - a structure, with
  - associated functions.
- You can try to pretend that it's an abstract type, but you get no help from the compiler.
- The data members are all "public"...

66

A Ring Buffer "Class" in C

```
// ring_buffer.h – a ring buffer in C

enum { rb_size = 32 };

typedef struct ring_buffer ring_buffer;
struct ring_buffer {
    char array[rb_size];        // "public"
    sig_atomic_t head, tail;    // "public"
};

inline void rb_init(ring_buffer *rb) {
    rb->head = rb->tail = 0;
}

~~~
```

67

```
// ring_buffer.h – a ring buffer in C (continued)

inline bool rb_empty(ring_buffer const *b) {
    return b->head == b->tail;
}

inline char rb_front(ring_buffer const *b) {
    return b->buffer[b->head];
}

inline void rb_pop_front(ring_buffer *b) {
    if (++b->head >= rb_size)
        b->head = 0;
}

void rb_push_back(ring_buffer *b, char c);
```

68

A Ring Buffer "Class" in C

- Unfortunately, C can't warn you about simple misuses, such as:

```
int main() {
    char c;
    ring_buffer b;
    b.head = 0;            // improper initialization
    ~~~
    rb_push_back(b, c);    // ?
    ~~~
}
```

- The improper initialization causes the later call on `rb_push_back` to exhibit undefined behavior.

69

Using Incomplete Types

- If you replace the complete `ring_buffer` type in the header with an incomplete type, the type becomes more abstract.
  - The interface hides the data members.
- Unfortunately, you pay a run-time price.
  - You lose the ability to implement "public" functions as inline functions…

70

Using Incomplete Types

```
// ring_buffer.h – a ring buffer in C

typedef struct ring_buffer ring_buffer; // incomplete

inline void rb_init(ring_buffer *rb) {
    rb->head = rb->tail = 0;            // won't compile
}

~~~
```

- You have to:
  - Move the definition for `rb_init` from the header to the source file, and…
  - Remove the keyword `inline` from its declaration.

71

A Ring Buffer Class in C++

- Implementing the `ring_buffer` as a C++ class avoids these problems:
  - The ***private access specifier*** prohibits unauthorized access to the class's data representation…
    - even for class members declared in a header.
  - ***Constructors*** provide "guaranteed" automatic initialization for class objects.
- The class definition for a simplified implementation looks like…

72

A Ring Buffer Class in C++

```
// ring_buffer.h – a ring buffer in C++
~~~

class ring_buffer {
public:
    ring_buffer();
    bool empty() const;
    char &front();
    void pop_front();
    void push_back(char c);
private:
    enum { size = 32 };
    char array[size];
    sig_atomic_t head, tail;
};
```

73

A Ring Buffer Class in C++

```
// ring_buffer.h – a ring buffer in C++ (continued)

inline ring_buffer::ring_buffer():
    head (0), tail (0) {
}

inline bool ring_buffer::empty() const {
    return head == tail;
}

inline char &ring_buffer::front() {
    return array[head];
}
~~~
```

74

A More Flexible Ring Buffer

- The previous implementation provides a ring buffer of 32 characters.
- What if you want a ring buffer with:
  - 64 characters?
  - 96 unsigned characters?
  - 48 wide characters?
- You can get different buffer sizes by using a run-time parameter.
- But then you pay a run-time price.
- In C++, you can gain this flexibility without a run-time penalty.
- Simply transform the ring buffer class into a ***class template***…

75

A Ring Buffer Class Template

```
// ring_buffer.h - a ring buffer class template

template <sig_atomic_t N, typename element_type>
class ring_buffer {
public:
    ring_buffer();
    bool empty() const;
    element_type &front();
    void pop_front();
    void push_back(c);
private:
    enum { size = N };
    element_type array[size];
    sig_atomic_t head, tail;
};
```

76

A Ring Buffer Class Template

- Using the template is remarkably simple:

```
int main() {
    char c;
    ring_buffer<64, char> b;    // a buffer of 64 chars
    ~~~
    b.push_back(c);
    ~~~
}
```

77

Device Addressing

- Device drivers communicate with hardware devices through ***device registers***.
- ***Memory-mapped device addressing*** is very common:
  - It maps device registers into the conventional data space.
  - It's often called ***memory-mapped i/o*** for short.
- The following example assumes that:
  - the target is a 32-bit processor, and
  - the device registers are mapped to addresses beginning at hex value 0x3FF0000.

78

Device Registers

- A UART is a "***U***niversal ***A***synchronous ***R***eceiver/***T***ransmitter".
- The example assumes the system supports two UARTs.
- The UART 0 group consists of six device registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

- The UART 1 group consists of six more registers starting at offset 0xE000.

79

Modeling Individual Registers

- Each device register in this example occupies a four-byte word.
- Declaring each device register as an `unsigned int` or as a `uint32_t` works well, but…
- Using a meaningful typedef alias is better:

```
typedef uint32_t device_register;           // not quite
```

- Device registers are volatile, so you should declare them as such:

```
typedef uint32_t volatile device_register;  // quite
```

- As in C, using `volatile` inhibits overly-aggressive compiler optimizations that might cause the device driver to malfunction.

80

Placing Memory-Mapped Objects

- Normally, you don't choose the memory locations where program objects reside.
  - The compiler does, often with substantial help from the linker.
- For an object representing memory-mapped device registers:
  - The compiler doesn't get to choose where the object resides.
  - The hardware has already chosen.
- Thus, to access a memory-mapped object:
  - The code needs some way to reference the location as if it were an object of the appropriate type…

81

Pointer-Placement

- In C++, as in C, you can use ***pointer-placement***.
- That is, you cast the integer value of the device register address into a pointer value:

```
device_register *const UTXBUF0
    = (device_register *)0x03FFD00C;
```

- The device register has a fixed location.
- The pointer to that location should be `const`.
  - Its value never changes.

82

Placing Memory-Mapped Objects

- Once you've got the pointer initialized, you can manipulate the device register via the pointer, as in:

```
*UTXBUF0 = c;   // OK: send the value of c out the port
```

- This writes the value of character `c` to the UART 0's transmit buffer, sending the character value out the port.

83

Reference-Placement

- In C++, you can use ***reference-placement*** as an alternative to pointer-placement:

```
device_register &UTXBUF0
    = *(device_register *)0x03FFD00C;
```

- Using reference-placement, you can treat `UTXBUF0` as the register itself, not a pointer to the register, as in:

```
UTXBUF0 = c;    // OK: send the value of c out the port
```

84

UART Operations

- Many UART operations involve more than one UART register.
- For example:
  - The TBE bit (Transmit Buffer Empty) is the bit masked by 0x40 in the USTAT register.
  - The TBE bit indicates whether the UTXBUF register is ready for use.
  - You shouldn't store a character into UTXBUF until the TBE bit is set to 1.
  - Storing a character into UTXBUF initiates output to the port and clears the TBE bit.
  - The TBE bit goes back to 1 when the output operation completes.

85

A UART Structure in C

- In C, you would represent the UART as a structure:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
};

#define RDR 0x20    // mask for RDR bit in USTAT
#define TBE 0x40    // mask for TBE bit in USTAT
~~~
```

86

A UART Structure in C

- Here's a C function that sends characters from a null-terminated character sequence to any UART:

```
void put(UART *u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u->USTAT & TBE) == 0)
            ;
        u->UTXBUF = *s;
    }
}
```

87

A UART Class in C++

- A C++ class can package the UART as a better abstraction:

```
class UART {
public:
    ~~~     // see the next few slides
private:
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
    enum { RDR = 0x20, TBE = 0x40 };
    ~~~
};
```

88

A UART Class in C++

- These public members are for controlling transmission speed:

```
class UART {
public:
    ~~~
    enum baud_rate {
        BR_9600 = 162 << 4, BR_19200 = 80 << 4, ~~~
    };
    void set_speed(baud_rate br) { UBRDIV = br; }
    ~~~~
};
```

- set_speed is defined, not just declared, within its class definition.
- As such, it's implicitly an inline function.

89

A UART Class in C++

- These public members are for enabling and disabling the UART:

```
class UART {
public:
    ~~~
    void disable() { UCON = 0; }
    void enable() { UCON = RXM | TXM; }
    ~~~
private:
    ~~~
    enum mode { RXM = 1, TXM = 8 };
    ~~~
};
```

90

A UART Class in C++

- The class has two constructors:

```
class UART {
public:
    ~~~
    UART() { disable(); }
    UART(baud_rate br) {
        disable();
        set_speed(br);
        enable();
    }
    ~~~
};
```

91

A UART Class in C++

- And, it has three i/o functions:

```
class UART {
public:
    ~~~
    int get() {
        return (USTAT & RDR) != 0 ? (int)URXBUF : -1;
    }
    bool ready_for_put() { return (USTAT & TBE) != 0; }
    void put(int c) { UTXBUF = (device_register)c; }
    ~~~
};
```

92

A UART Class in C++

- Here (again) is the C function that sends characters from a null-terminated character sequence to any UART:

```
void put(UART *u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u->USTAT & TBE) == 0)
            ;
        u->UTXBUF = *s;
    }
}
```

93

A UART Class in C++

- And here it is using the C++ class:

```
void put(UART &u, char const *s) {
    for (; *s != '\0'; ++s) {
        while (!u.ready_for_put())
            ;
        u.put(*s);
    }
}
```

94

Modeling Devices More Accurately

- Objects of type `device_register` are read/write by default.
- But not all UART registers are read/write:

```
class UART {
    ~~~
private:
    device_register ULCON;
    device_register UCON;
    device_register USTAT;      // read-only
    device_register UTXBUF;     // write-only
    device_register URXBUF;     // read-only
    device_register UBRDIV;
};
```

95

Modeling Devices More Accurately

- Writing to a read-only register typically produces unpredictable run-time misbehavior that can be hard to diagnose.
- Enforcing read-only semantics at compile time is better.
- Declaring a member as read-only is easy — just declare it `const`:

```
class UART {
    ~~~
private:
    ~~~
    device_register const USTAT;     // read-only
    device_register UTXBUF;          // write-only
    device_register const URXBUF;    // read-only
    device_register UBRDIV;
};
```

96

Modeling Devices More Accurately

- Reading from a write-only register also produces unpredictable misbehavior that can be hard to diagnose.
- Again, you're better off catching this at compile time, too.
- Unfortunately, C++ doesn't have a write-only qualifier.
- Neither does C.
- However, you can enforce write-only semantics by using a class template…

97

A Write-Only Class Template

- `write_only<T>` is a simple class template for write-only types.
- For any type `T`, a `write_only<T>` object is just like a `T` object, except that it doesn't allow any operations that read the object's value.
- For example,

```
write_only<int> m = 0;
write_only<int> n;
n = 42;
m = n;  // compile error: attempts to read the value of n
```

98

A Write-Only Class Template

- The class template definition is:

```
template <typename T>
class write_only {
public:
    write_only(write_only const &) = delete;
    write_only &operator=(write_only const &) = delete;
    write_only() { }
    write_only(T const &v): m (v) { }
    void operator=(T const &v) { m = v; }
private:
    T m;
};
```

99

Modeling Devices More Accurately

- Using `const` and the `write_only<T>` template, the `UART` class data members look like:

```
class UART {
    ~~~
private:
    device_register ULCON;
    device_register UCON;
    device_register const USTAT;
    write_only<device_register> UTXBUF;
    device_register const URXBUF;
    device_register UBRDIV;
};
```

100

A Read-Only Class Template

- In truth, const class member don't always have the right semantics for read-only registers.
- Const class members require initialization.
- This can be a problem if the UART class has user-defined constructors.

101

A Read-Only Class Template

- You can use a `read_only<T>` class template instead of `const`, as in:

```
class UART {
    ~~~
private:
    device_register ULCON;
    device_register UCON;
    read_only<device_register> USTAT;
    write_only<device_register> UTXBUF;
    read_only<device_register> URXBUF;
    device_register UBRDIV;
};
```

102

A Read-Only Class Template

- The class template definition looks like:

```
template <typename T>
class read_only {
public:
    read_only(read_only const &) = delete;
    read_only &operator=(read_only const &) = delete;
    read_only() { }
    operator T const &() const { return m; }
    T const *operator&() const { return &m; }
private:
    T m;
};
```

103

Common C++ Misinformation

- Claim: C++ generates bigger and slower code than C does.
- ***Fact:***
  - When programming at lower levels of abstraction, C and C++ generate much the same code.
  - At higher levels of abstraction, C++ usually generates better code.

104

Common C++ Misinformation

- Claim: C++ features such as function overloading, friends, inheritance, namespaces, and virtual functions have an added run-time cost.
- ***Fact:***
  - Function overloading, friends, and namespaces have no run-time cost.
    - Moreover, overloading supports compile-time algorithm selection, which leads to faster code.
  - Inheritance without virtual functions also has no cost.
  - Virtual functions have a slight cost, but:
    - It's no different from using function call dispatch tables in C.
    - You don't pay for it unless you ask for it explicitly.

105

Common C++ Misinformation

- Claim: C supports encapsulation as well as C++ does.
- ***Fact:***
  - Absolutely not (as explained earlier).

106

Common C++ Misinformation

- Claim: C++ templates cause "code bloat".
- ***Fact:***
  - Templates make it easier to trade:
    - space for speed
    - space and/or speed for development time
  - Do templates make it too easy to generate numerous instances of nearly identical code?
    - Possibly.
  - However, function templates also support compile-time algorithm selection, which can lead to much faster code.

107

Common C++ Misinformation

- Claim: C++ hides too much of what's going on in programs.
- Claim: C++ encourages programmers to write unnecessarily complex software, while C does not.
- ***Fact:***
  - These are human factors issues supported only by poorly documented anecdotes.
  - However, you can program to reduce surprises:
    - Declare constructors using the keyword `explicit`.
    - Avoid declaring conversion operators.

108

References and Other Readings

- Kernighan and Ritchie [1988]. Brian Kernighan and Dennis Ritchie. *The C Programming Language, 2nd. ed*. Prentice Hall.
- Beatles [1967]. “I am the Walrus” by John Lennon and Paul McCartney. *Magical Mystery Tour*, Capital Record. LP.
- Tondo and Gimpel [1989]. Clovis Tondo and Scott Gimpel, *The C Answer Book, 2nd. ed*. Prentice Hall.

109

110

111

112