

Introduction to C++ and Object-Oriented Programming Part 1

August, 2016

Dan Saks

1

Saks & Associates

393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-937-324-3601 (voice)
dan@dansaks.com
www.dansaks.com

These notes are Copyright © 2016 by Dan Saks.

2

Legal Stuff

- If you have attended this course:
 - 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 course, you may possess these notes provided either:
 - You acquired the notes, directly or indirectly, from Saks & Associates or a Saks & Associates authorized distributor.
 - You acquired the notes, either directly or indirectly, from someone who attended the course, and the person from whom you acquired the notes no longer possesses any printed or electronic copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

4

About the Author and Presenter

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

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

5

About the Author and Presenter

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.

6

Part 1 Contents

1. The C Part of C++
2. C++ as a Better C
3. Design Principles in Practice
4. Classes and Objects
5. Language Features for Libraries
6. Function Name Overloading
7. Resource Management

7

Part 2 Contents

8. Packaging Refinements
9. Foundations for Operator Overloading
10. Operator Overloading
11. Compiler-Generated Functions
12. Derived Classes
13. Virtual Functions

8

Overview

- This course is an introduction to C++.
- It covers the essentials of the Standard C++ language and library.
- It covers enough features in enough detail for programmers:
 - to start doing productive work with C++, and
 - to avoid the most common pitfalls.

9

Overview

- C++ is an evolving language.
- Recent additions to the C++ Standard have added many new features.
- Many programmers find the new programming styles preferable to older ones.
- However, the older styles are still very much in use in existing code.
- This course covers both:
 - older styles so that you can work with existing code, and
 - more modern styles so that you can use them as appropriate.

10

Goals

- The goals of this course are to:
 - give you a solid understanding of the core features of Standard C++ and the C++ Standard Library,
 - help you see a clear connection between object-oriented design principles and C++ coding practices, and
 - give you a look “under the hood” at how compilers and linkers implement C++ language features to help you avoid many performance pitfalls.

11

Prerequisites

- This course assumes you have experience programming in some language such as Ada, Java, or Python.
- It assumes you are familiar with concepts such as:
 - declarations
 - expressions
 - functions
 - statements
- It doesn't assume knowledge of C.

12

[XX] Accompanying Source Code

- These notes come with accompanying source code for:
 - many of the programming examples, and
 - all of the programming exercises.
- The code folders have names of the form **CC-XX**, where:
 - **CC** is the chapter number.
 - **XX** is the example/exercise number.
- For example,
 - A slide in **Chapter 3** that has **[07]** in the slide title contains code found in folder **03-07**.
 - Folders **07-01.1** and **07-01.2** contain code for variants of the same example.

13

[XX] Accompanying Source Code

- A folder with a lowercase **s** in the name contains the solution to an exercise.
- For example,
 - Folder **03-08s** contains the solution to **Exercise 8** in **Chapter 3**.
 - Folders **07-07s.1** and **07-07s.2** contain alternate versions of the solution to **Exercise 7** in **Chapter 7**.
- You rarely solve any exercise from scratch.
- Rather, start with the code in the folder just prior to the solution.

14

[XX] Accompanying Source Code

- A code folder with a lowercase **p** at the end of the name contains a partial solution.
 - Use it to complete the corresponding exercise.
- For example, you should solve **Exercise 8** in **Chapter 3** by starting with the code in folder **03-08p**.
- A partial solution:
 - might be an earlier version of a program, or
 - might not be a complete program.

15

Classroom Dynamics

- Please ask questions about the material as soon as you suspect you might be lost.
 - The material builds on itself.
 - If you let yourself stay lost, you'll miss much of what follows.
- If you want me to go back over something, just ask.
 - We all zone out now and then.
- If you get lost in details, please ask questions to gain perspective.
 - For example, ask "Why is this useful to know?"
- Please save questions on tangential topics for the end of the course.
 - I'll try to answer them if time permits.
- Enjoy yourself! This is interesting stuff.

16

[1]

The C Part of C++

1

The Nature of C

- C is a ***general-purpose*** programming language.
 - You can use it for almost every kind of application.
- C provides many high-level language facilities:
 - primitive data types
 - character, integer, floating-point
 - aggregate types
 - array, structure
 - arithmetic, logical, and relational operators
 - add, subtract, and, or, less-than, equal-to
 - flow control
 - if, for, while
 - functions

2

The Nature of C

- C is a *“low-level” high-level* language.
- It provides machine-level data objects:
 - bits
 - unsigned as well as signed integers
 - integers of different size
 - floating-point numbers of different size and precision
 - pointers to data
 - pointers to functions
- It also provides machine-level operators:
 - bitwise-and and bitwise-or
 - left-shift and right-shift

3

The Nature of C++

- C++ is an imperfect superset of C.
 - Most, but not all, C code will compile as C++.
 - If it compiles, it will produce the same results when executed.
- C++ retains C’s ability to deal efficiently with bits and bytes.
- C++ extends C with features that support large-scale programming, including:
 - better compile-time program verification
 - object-oriented programming
 - generic programming

4

Language Dialects

- Both C and C++ have changed over time.
- They continue to change.
- An international committee governs the standard specification for C.
- Another international committee governs the standard specification for C++.
- An international standards organization, **ISO**, oversees both committees.

5

Standard Libraries

- Some programming languages have built-in support for common operations such as:
 - character string manipulation
 - file input and output
- C and C++ don't.
- Rather, C and C++ provide these operations, and many others, via functions available in a library.
- Every Standard C compiler comes with a **C Standard Library**.
- Every Standard C++ compiler comes with a **C++ Standard Library**:
 - It includes the C Standard Library, and much more.

6

C Timeline

- **1972:** Dennis Ritchie of AT&T Bell Labs began developing C.
- **1978:** Brian Kernighan and Dennis Ritchie published *The C Programming Language* (Kernighan and Ritchie [1978]).
 - this early dialect is often called “K&R” or “Classic” C
- **1990:** ISO approved “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 [2011a])

7

C++ Timeline

- **1980:** Bjarne Stroustrup of AT&T Bell Labs began developing C++.
- **1985:** Stroustrup published *The C++ Programming Language* (Stroustrup [1985]).
- **1998:** ISO approved “C++98”
 - the first international C++ standard (ISO [1998])
- **2003:** ISO approved “C++03”
 - a revised international C++ standard (ISO [2003])
 - basically C++98 with bug fixes
- **2005:** ISO approved “TR1”
 - “Library Technical Report 1” (ISO [2005])
 - not a new standard, but rather proposals for library extensions

8

C++ Timeline

- **2011**: ISO approved “C++11”
 - yet another international C++ standard (ISO [2011b])
 - most of TR1, plus new language features and more library components
 - this is “Modern” C++
- **2014**: ISO approved “C++14”.
 - the latest international C++ standard (ISO [2014])
 - mostly small improvements to C++11 features

9

Source Files

- The source code for a C or C++ program is a text file.
- The naming conventions for the file depend on:
 - the development tools and
 - the host operating system.
- For C source, the file name almost always has a **.c** extension.
- For C++ source, the file name often has a **.cpp** extension.
 - Other possible extensions are **.C**, **.cxx**, or **.c++**.

10

[01] Saying “Hello”

- Here’s a first C program, “*hello.c*”:

```
// "Hello, world" in Standard C and C++  
  
#include <stdio.h>  
  
int main() {  
    printf("Hello, world\n");  
    return 0;  
}
```

- This is also a C++ program.

11

Building and Running a Program

- Before you can run the program, you have to build it.
 - That is, convert it to an executable program.
- Again, how you do this depends on:
 - the development tools and
 - the host operating system.
- Many development tools include an integrated development environment (IDE) in which you can edit the source and build the program.
- You can also build programs from a command line environment...

12

Building and Running a Program

- For example, using Microsoft Visual C++ within a Windows command prompt, you simply type:

```
cl hello.c
```

- `cl` is the command to compile and link a program.
- In this case, it:
 - compiles *“hello.c”* into an object file named *“hello.obj”*, and
 - links *“hello.obj”* with library components to produce an executable file named *“hello.exe”*.

13

Building and Running a Program

- To execute *“hello.exe”* from the command line, you simply type:

```
hello
```

- The program responds by displaying:

```
hello, world
```

14

Building and Running a Program

- When the source file name ends in *.c*, the *cl* command compiles the code as C.
- You can force *cl* to compile as C++ by using the */TP* option:

```
rem compile as C++  
cl /TP hello.c
```

- Or, you can just change the file extension to *.cpp*:

```
rem compile as C++  
cl hello.cpp
```

15

Building and Running a Program

- Using the GNU compiler from a Linux command line, you type:

```
gcc hello.c
```

- In this case,
 - it compiles "*hello.c*" (as C) into an object file named "*hello.o*", and
 - links "*hello.o*" with library components to produce an executable file named "*a.out*".
- To execute the program, type:

```
a.out
```

16

Building and Running a Program

- To produce an executable named *“hello”* instead of *“a.out”*, type:

```
gcc hello.c -o hello
```

- To compile as C++, type:

```
g++ hello.c -o hello
```

- Or, change the file extension to `.cpp`.
- You can compile as C++ using either:

```
gcc hello.cpp -o hello  
g++ hello.cpp -o hello
```

17

[01] Saying “Hello”

- Let’s look at our first program again:

```
// "Hello, world" in Standard C and C++  
  
#include <stdio.h>  
  
int main() {  
    printf("Hello, world\n");  
    return 0;  
}
```

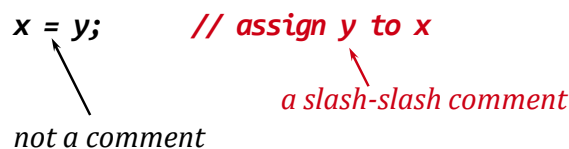
18

“Slash-Slash” Comments

- C++ source code can contain comments for human readers:

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

- A “slash-slash” or “double-slash” comment begins with `//`.
- It runs to the end of the line.
- The `//` need not be at the beginning of the line:

`x = y; // assign y to x`

not a comment *a slash-slash comment*

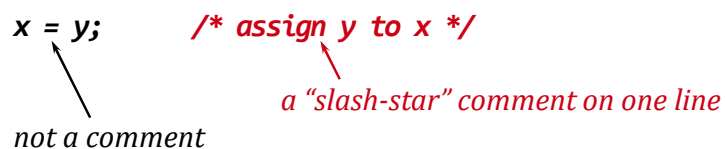
19

“Slash-Star” Comments

- C++ also allows comments that begin with `/*` and end with `*/`.
- “Slash-star” comments can span multiple lines, as in:

```
/*
 * This is a "slash-star" comment.
 * It spans 4 lines altogether.
 */
```

- However, they need not span multiple lines:

`x = y; /* assign y to x */`

not a comment *a “slash-star” comment on one line*

20

The main Function

- This defines a function named main:

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

- The part before the { is the *function heading*:

```
int main()
```

21

The main Function

```
int main()  
  ↑      ↑      ↑  
return type function name parameter list
```

- int is a **keyword**.
 - It specifies that the function returns an integer value to its caller.
- main is an **identifier**.
 - It names the function.
- The (and) after the function name are **operators**.
 - They are what make this a function declaration.

22

The main Function

```
int main()
```

- This heading says `main` is a function that:
 - accepts no arguments, and
 - returns an integer value to its caller.
- `main` is special — it's the program's *entry point*.
 - The target operating system (OS) calls `main`.
 - If `main` accepted arguments, it would be the OS that passed them in.
- Every Standard C++ program must have a function named `main`.

23

Functions

- The rest of this definition is the *function body*:

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

- The body:
 - is enclosed in curly braces, { and }, and
 - contains zero or more statements.
- Here, `main`'s body has two statements.

24

Functions

- C++ is largely free form.
- You can use horizontal and vertical spacing as you wish.
- For example, you can put each brace on its own line:

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

- You can put the entire function on one line:

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

25

Calling printf

- The first statement in main is:

printf("Hello, world\n");

function name *argument*

- printf is an identifier.
 - In this case, it names a Standard Library function.
- The () after the function name are **operators** that enclose a list of zero or more, comma-separated arguments.
 - In this case, the lone argument is "Hello, world\n".
- The ; at the end terminates the statement.

26

String Literals

- The construct "Hello, world\n" is a **string-literal**.
- A string-literal represents a specific sequence of characters.
- Within a string-literal, \ marks the beginning of an **escape sequence**.
- You use escape sequences to represent special characters:
 - \n represents a **newline** character.
 - It marks the end of a line of text.
 - On most devices, it displays as both a *carriage return* and a *line feed*.
 - \t represents a **horizontal tab** character.
 - \" represents a **double quote** character.
 - \\ represents a **backslash** character.

27

Function Returns

- The second (and last) statement in main is:

return 0;
↑ ↑
keyword integer-literal

- return is a keyword.
 - Executing this statement causes the function to return control to its caller.
 - For the main function, the caller is the OS.
- 0 is an **integer-literal**.
 - The function returns this value to its caller.
- The ; at the end terminates the statement.

28

Tokens

- The compiler reads the source program one character at a time.
- It groups those characters into tokens.
- A **token** is a sequence of characters that have meaning as a unit.
- We've seen that C++ has five kinds of tokens:
 - *keywords*, such as `int` and `return`
 - *identifiers*, such as `main` and `printf`
 - *literals*, such as `"hello, world\n"` and `0`
 - *operators*, such as `()`
 - *punctuation*, such as `{ }` and `;`

29

Tokens

- Inserting a space into a token changes its meaning.
- For example, this is not a keyword:

`re turn`
- It's two identifiers: `re` and `turn`.
- Removing the space(s) between tokens can alter their meanings.
- For example, this is not a keyword followed by an integer-literal:

`return0`

- It's a single identifier.

30

Function Declarations

- Again, this calls the Standard Library's `printf` function:

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

- A declaration for a function must appear before the first call to that function.
- A ***function declaration*** is a statement that provides the compiler with information needed to call the function.

31

Standard Headers

- For a Standard Library function, you don't write the declaration yourself.
- Rather, you obtain the declaration by including the appropriate header:

```
#include <stdio.h>      // stdio.h is a header...
```

- The C Standard Library provides numerous ***standard headers***.
- Each header contains declarations for a subset of the library components.

32

Standard Headers

- For example, `<stdio.h>` contains declarations for input and output functions such as:
 - `printf` — write formatted data
 - `scanf` — read formatted data
 - `putchar` — write a single character
 - `getchar` — read a single character
- With most compilers, each standard header is a separate file.
- You typically find the headers in a subfolder of the compiler installation folder, often named “*include*”.

33

Standard Output

- By default, `printf` and `putchar` write characters to ***standard output***.
 - It’s short for “the *standard output* stream”.
- Standard output usually defaults to the computer screen, or a window therein.
- Most operating systems offer a way to redirect the output, say to a file.

hello ***>hello.txt***

- From a Windows or Linux command line, this runs *hello* and sends the output to “*hello.txt*”.

34

Standard Input

- By default, `scanf` and `getchar` read characters from ***standard input***.
 - It's short for "the *standard input* stream".
- Standard input usually defaults to the keyboard.
- Most operating systems offer a way to redirect the input, say from a file.

`munge <data.txt`

- From a Windows or Linux command line, this runs the *munge* program, receiving input from "*data.txt*".

35

Objects, Addresses and Bytes

- C and C++ programs can manipulate data stored in objects.
- An ***object*** is simply a region of data storage that can hold a value.
- ***Data storage*** is often called ***memory***.
- Each object has a unique numeric memory ***address***.
- The smallest addressable memory unit is a ***byte***.
 - A byte is at least 8 bits.
 - On some processors, it might be more.

36

Objects and Words

- Every object occupies a sequence of 1 or more contiguous bytes.
- Most processors define a **word** as some multiple number of bytes.
- For example, on a typical 16-bit processor with 8-bit bytes:
 - a *word* is 2 bytes,
 - a *double word* is 4, and
 - a *quad word* is 8.
- Or, on a typical 32-bit processor with 8-bit bytes:
 - a *half word* is 2 bytes,
 - a *word* is 4, and
 - a *double word* is 8.

37

Alignment

- Some objects must have addresses that are a particular multiple of a byte address.
- That multiple is called an **alignment**.
- Each processor specifies its own alignment requirements.
- For example,
 - 4-byte words:
 - usually have to be **word aligned** (at a multiple of 4).
 - 8-byte double words:
 - might also have to be word aligned, or
 - might have to be **double-word aligned** (at a multiple of 8).

38

Object Declarations

- A program must declare an object before using that object.
- An object's declaration includes its type and name:

```
int i;      // object i has type int (integer)
double f;   // object f has type double (double-precision
           // floating-point)
```

- A declaration can declare more than one object:

```
char c1, c2; // objects c1 and c2 both have type char
           // (character)
```

39

Type Information is Static

- An object's declared type is a ***static data type***.
 - The object's type is set at compile time.
 - It can't change during program execution.
- In effect:
 - "Once an int, always an int."
 - "Once a double, always a double."
 - And so on...

40

Data Types

- What exactly is a data type?
- A (static) data type is a bunch of compile-time properties for an object:
 - its size in bytes
 - its alignment (also in bytes)
 - the set of values it can hold
 - the set of operations that you can apply to it

41

Type `int`

- An object of type `int` stores a signed integer.
- On typical 32-bit processors, type `int` has:
 - size and alignment of 4 (bytes)
 - values from -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$), inclusive
 - integers only
 - unary operations including:
 - arithmetic-negate, address-of, increment, decrement
 - binary operations including:
 - assign, add, subtract, multiply, divide, less-than, equal-to

42

Type char

- An object of type `char` stores a character.
- On all processors, type `char` has:
 - size and alignment of 1
 - values that can represent the 96 basic source characters:
 - *space, horizontal tab, vertical tab, form feed, and new-line*
 - *a through z, A through Z, and 0 through 9*
 - `_ { } [] # () < > % : ; . ? * + - / ^ & | ~ ! = , \ " '`
 - unary operations including:
 - same as `int`
 - binary operations including:
 - same as `int`

43

Type bool

- An object of type `bool` stores a logical value: `false` or `true`.
- On typical 32-bit processors, type `bool` has:
 - size and alignment of 1
 - but it could be 2 or 4
 - values `false` (equal to 0) and `true` (equal to 1)
 - `false` and `true` are keywords
 - unary operations including:
 - logical-negate, address-of
 - binary operations including:
 - assign, and, or, equal-to, not-equal-to

44

Integer Types

- Integer types can be:
 - signed or unsigned
 - signed is the default
 - short, “plain”, long, or long long
 - “plain” is the default
- For example,

```
int spi;           // signed "plain" int
unsigned short int usi; // exactly what it says
long long int slli; // signed long long int
signed long int sli; // exactly what it says
```

45

Optional int

- If signed, unsigned, short, or long is present, then int is optional.
- For example,

```
short ssi;           // signed short int
unsigned short usi;  // unsigned short int
long sli;            // signed long int
signed spi;          // signed "plain" int
long long slli;      // signed long long int
unsigned long uli;    // unsigned long int
```

46

Arithmetic Types

- The arithmetic types are:
 - bool
 - char
 - the integer types
 - the floating-point types
- The three floating-point types are:
 - float
 - double
 - long double

47

sizeof

- sizeof is a keyword that is also an operator.
- The expression sizeof(x) yields the size of x in bytes.
 - It's an unsigned value because sizes are never negative.
- Here, x can be an object or a type, as in:

```
char c;  
unsigned long long ull;  
~~~  
sizeof(c)           // always 1  
sizeof(char)        // same value  
~~~  
sizeof(unsigned long long) // typically 8 or 16  
sizeof(ull)         // same value
```

48

Sizes of Integer Types

- Standard C++ allows considerable slack in the range of values and sizes for integer types.
- This is both good and bad:
 - It lets the compiler choose object sizes that are optimal for the target processor.
 - It can lead to portability problems.
- Here's what you can count on...

49

Sizes of Integer Types

- The Standard guarantees that:
 - `sizeof(char) <= sizeof(short)`
 - `sizeof(short) <= sizeof(int)`
 - `sizeof(int) <= sizeof(long)`
 - `sizeof(long) <= sizeof(long long)`
- Each unsigned type is the same size as its corresponding signed type.
- On almost all processors, either:
 - `sizeof(short) == sizeof(int) <= sizeof(long)`
 - `sizeof(short) <= sizeof(int) == sizeof(long)`
- All sizes (`char`, `short`, `int`, `long`, and `long long`) might be the same.

50

Sizes of Integer Types

- The Standard also guarantees that:
 - bits per char ≥ 8
 - bits per short ≥ 16
 - bits per long ≥ 32
 - bits per long long ≥ 64
- For example, this means the maximum value:
 - for a signed short is at least 32,767 ($2^{15}-1$), and
 - for an unsigned short is at least 65,535 ($2^{16}-1$).

51

Sizes of Integer Types

- For instance, a 16-bit processor might use:
 - 16 bits (2 bytes) for int and unsigned int, and
 - 32 bits (4 bytes) for long int and unsigned long int.
- A 32-bit processor might use:
 - 32 bits (4 bytes) for all four types, int, unsigned int, long int, and unsigned long int.
- A 64-bit processor might use:
 - 32 bits (4 bytes) for int and unsigned int, and
 - 64 bits (8 bytes) for long int and unsigned long int.

52

Arithmetic Operations

- For these examples, assume `i` and `j` are declared as:

```
int i, j;
```

- You can do arithmetic with `i` and `j`, such as:

```
i + j      // add
i - j      // subtract
i * j      // multiply
i / j      // divide, yielding int quotient
```

- Arithmetic with `int` operands yields an `int` result.
- What happens if the result ***overflows*** — is too big to fit?

53

Undefined Behavior

- C compilers:
 - can catch many erroneous language constructs, and
 - reject offending programs before they get to run.
- C++ compilers are even better at this.
- However, some errors just can't be caught until run time.
- Runtime error detection often imposes a burden on programs:
 - more object code
 - slower execution time.
- To avoid imposing these costs on all programs...
- The Standard allows that certain erroneous program constructs simply have ***undefined*** behavior.

54

Undefined Behavior

- For instance, a program exhibits undefined behavior when:
 - it executes an `int` operation that yields a result larger than an `int`, or
 - it divides an `int` by zero.
- The notion of undefined behavior is effectively a license for each compiler to cope with each error as it sees fit...

55

Undefined Behavior

- The compiler might generate code that intercepts the error at run time.
 - Compilers rarely do this.
- Typically, the compiler generates code that acts as if nothing's wrong:
 - Sometimes, this produces reasonable behavior.
 - More often, something bad happens.
- Sometimes, the compiler can detect the error at compile time and issue a diagnostic (error or warning) message.

56

Unsigned vs. Signed Types

- Unsigned integer arithmetic is modular:
 - When unsigned integer arithmetic overflows, the program discards the high-order bits of the result.
 - In effect, unsigned arithmetic that overflows simply “wraps around” to a small value.
- In contrast, signed integer overflow produces *undefined behavior*.
 - On many machines, the result is modular.
 - But don’t count on it...
 - Something bad could happen.

57

Type char, Revisited

- C++ actually has 3 char types:
 - “plain” char
 - signed char
 - unsigned char
- signed char and unsigned char are for storing small integer values.
 - They aren’t for text.
- Plain char has the same implementation as either signed char or unsigned char.
 - Whether plain char is signed or unsigned can vary from compiler to compiler.
- However, the compiler treats each char type as a distinct type.

58

Literals

- A literal is a token that represents a specific value.
- In addition to *string-literals*, C++ provides:
 - ***integer-literals***:
 - *decimal-literals*, such as **10**
 - *hexadecimal-literals*, such as **0x1C**
 - ***floating-literals***:
 - without an exponent, such as **1.0**
 - with an exponent, such as **6.022e+23**
 - ***character-literals***:
 - such as **'a'** or **'\n'**
 - *hexadecimal-character-literals*, such as **'\x10'**

59

Literals Have Types

- Every literal has a type.
 - 10 and 0x1C have type [signed] int.
 - 1.0 and 6.022e+23 have type double (not float).
 - 'a' and '\n' have type char.
- Some literals allow suffixes that specify the type:
 - 10**U** is an unsigned [int].
 - 10**L** is a [signed] long [int].
 - 10**UL** is an unsigned long [int].
 - 10.0**F** is a float.
 - 10.0**L** is a long double.

60

Binary Operators

- C++ provides a wide variety of binary operators:

- ***additive and multiplicative*** operators:

```
x + y    // add x and y
x - y    // subtract y from x
x * y    // multiply x times y
x / y    // divide x by y, yielding quotient
x % y    // divide x by y, yielding remainder
```

- ***shift*** operators:

```
x >> y    // shift x right by y bits
x << y    // shift x left by y bits
```

61

Binary Operators

- C++ provides a wide variety of binary operators:

- ***equality*** operators:

```
x == y    // true if x is equal to y
x != y    // true if x is not equal to y
```

- ***relational*** operators:

```
x < y      // true if x is less than y
x <= y     // true if x is less than or equal to y
x > y      // true if x is greater than y
x >= y     // true if x is greater than or equal to y
```

62

Binary Operators

- C++ also provides:

- **bit-wise** operators:

```
x & y      // and x with y
x | y      // or x with y
x ^ y      // exclusive-or x with y
```

- **logical** operators:

```
x && y     // true if both x and y are true
x || y     // true if either x or y is true
```

- C++ provides other binary operators explained later.

63

Assignment Operators

- C++ provides a **simple assignment** operator:

```
x = y      // copy value in y to x
```

- It also provides **compound assignment** operators, such as:

```
x += y     // short for: x = (x + y)
x *= y     // short for: x = (x * y)
```

- C++ provides these compound assignments:

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

64

Unary Operators

- C++ provides an assortment of **unary** operators, such as:

```

+x          // x
-x          // 0 - x
!x          // logical negate of x
~x          // ones-complement of x
sizeof(x)   // size of x (in bytes)
alignof(x)  // alignment of (x in bytes)

```

65

Increment and Decrement Operators

- The increment and decrement operators are the only operators that are both prefix and postfix operators.
- The **prefix** operators are simple:

```

++x          // short for: x += 1
--x          // short for: x -= 1

```

- The **postfix** operators mean pretty much the same thing:

```

x++          // short for: x += 1 (almost)
x--          // short for: x -= 1 (almost)

```

- Here's the difference...

66

Increment and Decrement Operators

- A prefix operators yields the result of the increment:

```
int x, y;
```

```
~~~~
```

```
y = ++x;           // short for: x += 1; y = x;
```

- A postfix operator yields the operand's value prior to the increment:

```
y = x++;           // short for: y = x; x += 1;
```

67

Grouping Expression

- You can use () to explicitly group operations and operands in a expression:

```
w = (2 * (x + y)) - z;
```

- If you remove the (), the expression looks like:

```
w = 2 * x + y - z;
```

- This groups as if you had written:

```
w = ((2 * x) + y) - z;
```

- That's because operators have precedence and associativity...

68

Operator Precedence and Associativity

- The * and / have higher precedence than + and -.
- Thus, these expressions are the same:

$$2 * x + y / z$$
$$(2 * x) + (y / z)$$

- + and - have the same precedence and associate left to right.
- Thus, these expressions are the same:

$$2 + x - y + z$$
$$((2 + x) - y) + z$$

69

Operator Precedence and Associativity

- Here's the precedence and associativity for many of the operators we've seen thus far:

Precedence	Operators	Associativity
highest	x++ x--	left
	++x --x +x -x !x ~x	right
	x*y x/y x%y	left
	x+y x-y	left
	x<<y x>>y	left
	x<y x<=y x>y x>=y	left
lower	x==y x!=y	left

- Continued...

70

Operator Precedence and Associativity

Precedence	Operators	Associativity
even lower	x&y	left
	x^y	left
	x y	left
	x&&y	left
	x y	left
lowest	x=y x+=y x-=y x*=y x/=y x%=y x<<=y x>>=y x&y x^y x y	right

✓ When in doubt, parenthesize.

71

Chaining Assignment

- Assignment is unusual — it's **right** associative.
- That is, these are equivalent:

a = b = c = d;
a = (b = (c = d));
- Each assignment expression returns the value of its left operand after the assignment.
- The assignment chains above are shorthand for:

```
c = d;  
b = c;  
a = b;
```

72

Arithmetic Conversions

- The operands in an expression may have different types.
- In some cases, the compiler quietly inserts an implicit conversion, as in:

```
int i;  
long li;  
double d;  
~~~  
li = li + i;    // convert i to long; long result  
d = d / 2      // convert 2 to double; result is double
```

- Sometimes, your compiler will issue a warning:

```
i = d;          // probable warning: loss of significance
```

73

Arithmetic Conversion Ranks

- When performing arithmetic conversions, C++ ranks the types:

Rank	Type
highest	long double
	double
	float
	[un]signed long long int
	[un]signed long int
	[un]signed int
	[un]signed short int
lowest	char

74

Arithmetic Conversions

- Again, an arithmetic expression can have operands of different types.
- In general, the expression converts the operand with the lower rank into the type of the higher rank.
- The signed and unsigned version of each integer have the same rank.
- The compiler breaks ties in favor of the unsigned type:

```
int i;
unsigned u;
~~~
u = u + i; // convert i to unsigned; result is unsigned
```

75

Narrowing Conversions

- C++ lets you perform conversions that lose a part of the value.
- For example,

```
float f;
int i;
~~~
i = f; // discards fractional part of f
```

- Even this might lose part of the value:

```
f = i; // might lose precision
```

- If `sizeof(int) == sizeof(float)`, then `f` has fewer significant digits than `i`.

76

[02] Narrowing Conversions

- A **narrowing conversion** is any implicit conversion that potentially loses some portion of the converted value.
- For example,

```
double d;
float f;
int i;
unsigned u;
~~~
f = d;      // narrows double to float
i = f;      // narrows float to int
f = i;      // narrows int to float
i = u;      // narrows unsigned int to signed int
u = i;      // narrows signed int to unsigned int
```

77

Narrowing Conversions

- Some compilers issue warnings for some narrowing conversions.
- You often can configure the compiler to be pickier about narrowing conversions.
- For example, with g++ you can use command-line options:

```
g++ -Wconversion -Wsign-conversion x.cpp
```

- With these options, g++ will warn about all conversions on the previous slide.

78

Casts

- Sometimes, you have no choice but to use narrowing conversions.
- In that case, the warnings can be an annoyance.
- You can tell the compiler that you're doing the conversion intentionally by using a cast:

```
double d;  
int i;  
unsigned u;  
~~~  
f = (float)d;      // casts double to float  
i = (int)f;        // casts float to int  
u = (unsigned)i;   // casts signed int to unsigned int
```

79

Initializers

- An object declaration can specify an initial value, as in:
- You can initialize more than one object in a single declaration:

```
int level = 42;  
  
unsigned horizontal = 1920, vertical = 1080;
```

80

Separate Compilation

- A C++ program can consist of more than one source file.
- You compile each source file separately.
- Each compilation produces a separate object file.
- You link the object files together, along with any other pre-compiled libraries, to build the executable file.

81

[03] Separate Compilation

- As a crude example, suppose you'd like to package printing "hello, world\n" as a separate function that anyone can use:

```
// hello.cpp

#include <stdio.h>

void hello() {                // void return type
    printf("Hello, world\n");
    return;                   // no return value
}
```

- The keyword `void` indicates that the function returns nothing.
- In this case, the return statement is optional.

82

[03] Separate Compilation

- Here's a separate file with a new main function that calls hello:

```
// greet.cpp

int main() {
    hello();           // compile error: hello undeclared
    return 0;
}
```

- Unfortunately, it doesn't compile.

83

[03] Function Declarations

- The source file requires a declaration for hello:

```
// greet.cpp

void hello();           // function declaration

int main() {
    hello();             // OK: hello declared
    return 0;
}
```

- Now it does compile.

84

Compiling and Linking

- Now, you can compile each source file separately, as in:

```
g++ -c greet.cpp
```

- With most compilers, the `-c` option means “compile only”.
- The above command generates an object file named “`greet.o`”.
- Without `-c`, the compiler will try to compile and link:

```
g++ greet.cpp
```

- The link will fail because “*greet.o*” calls but does not define the `hello()` function.

85

Compiling and Linking

- To build the entire, you compile each source file separately, as in:

```
g++ -c greet.cpp  
g++ -c hello.cpp
```

- You can then link the program using:

```
g++ greet.o hello.o -o greet
```

- Alternatively, you can do all of the above in one step:

```
g++ greet.cpp hello.cpp -o greet
```

86

Declarations and Definitions

- A name can designate a program entity such as a function or an object.
- A **declaration** is a statement that says to the compiler:
 - “Here’s a name and some attributes for an entity that’s somewhere in this program, possibly here.”
- A **definition** is a declaration that says:
 - “Here’s a name and the complete set of attributes for an entity that’s right here.”
- All definitions are declarations.
- Not all declarations are definitions.

87

Declarations and Definitions

- This is a **function declaration**:


```
void hello();           // non-defining declaration
```
- It lacks a function body, so it isn’t a definition.
- This function declaration is also a **function definition**:


```
void hello() {           // a definition
    printf("Hello, world\n");
}
```
- It has a brace-enclosed body.
- It tells the compiler everything it needs to know to actually create the function.

88

User-Defined Headers

- A large program may contain many separate source files.
- Each file might define many functions.
- What happens if you change the function heading in the definition, but forget to change the declaration(s) in other source files?
- The program might build anyway.
 - The resulting program will have *undefined behavior*.
 - This is a potential maintenance nightmare.
- User-defined headers avoid these problems...

89

[03] User-Defined Headers

- Consider the *greet* program:

```
// greet.cpp

void hello();           // a potential maintenance problem

int main() {
    hello();             // OK: hello declared
    return 0;
}
```

- Here's the preferred way...

90

[03] User-Defined Headers

- Rather than declare the `hello` function here, declare it in a header:

```
// hello.h - a user-defined header  
  
void hello();
```

91

[03] User-Defined Headers

- Now, rewrite "*greet.cpp*" to include that header:

```
// greet.cpp  
  
#include "hello.h" // better practice  
  
int main() {  
    hello();        // OK: hello declared  
    return 0;  
}
```

92

[03] User-Defined Headers

- When you compile *greet.cpp*, the compiler replaces the header with its contents:

```
// greet.cpp

void hello();           // after header inclusion

int main() {
    hello();           // OK: hello declared
    return 0;
}
```

93

[03] Separate Compilation

- It's good practice to include a header in its corresponding source file:

```
// hello.cpp

#include <stdio.h>
#include "hello.h"           // good practice

void hello() {
    printf("Hello, world\n");
}
```

- You can declare a function many times...
- ...as long as you define it only once.

94

User-Defined Headers

- Think of each separate source file as providing some services.
- In effect, the accompanying header declares the interface to those services.
- Each source file that uses the services simply includes the header.
- If you change the services, you simply change the header to match.
- Then recompile.
- Here's how it all works...

95

Translation Phases

- Once again, C++ programs consist of one or more source files and headers.
- Each source file and header is a sequence of text representing statements in the C++ language.
- You compile one source file at a time.
 - You don't compile headers by themselves.
- The compiler translates each source file via a series of phases:
 1. group the characters in the file into tokens
 2. replace each comment by a space
 3. include any headers
 4. analyze the syntax
 5. generate code

96

The Preprocessor

- Header inclusion is handled by a compiler subsystem called the ***preprocessor***.
- The preprocessor performs text transformations known as ***preprocessing***.
 - It performs other transformations covered later.
- You can think of the preprocessor as a program separate from the rest of the compiler.
 - It's not necessarily implemented that way.
- Most compilers have options that let you either:
 - run only the preprocessor, or
 - run the compiler without preprocessing.

97

Translation Units

- The preprocessor output is called a ***translation unit***.
- Some compilers actually generate the translation unit as a file with a ***.i*** or ***.ii*** extension.
- The remaining compile steps, such as syntax analysis and code generation, expect a translation unit as input.

98

Preprocessing

- Recall that header file “*hello.h*” contains:

```
// hello.h – a user-defined header  
  
void hello();
```

99

Preprocessing

- Source file “*greet.cpp*” contains:

```
// greet.cpp  
  
#include "hello.h" // better practice  
  
int main() {  
    hello();        // OK: hello declared  
    return 0;  
}
```

100

Preprocessing

- After stripping out spaces and comments, the translation unit looks like:

#include "hello.h"

```
int main() {  
    hello();  
    return 0;  
}
```

101

Preprocessing

- After header inclusion, the translation unit looks like:

void hello();

```
int main() {  
    hello();  
    return 0;  
}
```

- Later phases of the compiler see this preprocessor output as their input.

102

Including Headers

- A `#include` directive has either of two forms:

```
#include <header>    // with angle brackets
#include "header"    // with quotes
```

- The compiler replaces the `#include` directive with the contents of the named header.
- The only difference between the two forms is that they may look for the header in different ways.
- Oversimplifying somewhat:
 - `<header>` looks for `header` in folders the compiler provides.
 - `"header"` looks for `header` in folders you provide.

103

Macros

- Preprocessor macros provide symbolic string replacement:

```
#define INIT_VAL 42
~~~
int n = INIT_VAL;
```

- During translation, the preprocessor replaces all occurrences of the macro with its replacement value.
- It also removes the macro definition itself.
- The resulting translation unit simply contains:

```
int n = 42;
```

104

Function Arguments and Parameters

- When you call a function, you can pass it arguments:

```
v = volume(height, width, depth);
```

- The function must be declared to accept those arguments:

```
double volume(double h, double w, double d);
```

- It must be defined that way, too:

```
double volume(double h, double w, double d) {  
    return h * w * d;  
}
```

105

Function Arguments and Parameters

- Informally:
 - An **argument** is something you pass to a call.
 - A **parameter** is what the callee uses to refer to an argument.
- For example, this function heading declares a single parameter:

```
int f(int i);    // i is a parameter
```

- This call passes *n* as the argument to *f*'s parameter *i*:

```
m = f(n);        // n is an argument
```

- The distinction between arguments and parameters appears elsewhere in C++.

106

Unnamed Parameters

- This function declaration is not a definition:

```
double volume(double h, double w, double d);
```

- It lacks a function body.
- The parameter names don't matter at this point.
- You can omit them:

```
double volume(double, double, double);
```

- Of course, you must provide parameter names in the definition, which actually uses them.

107

Argument Conversions

- For a given parameter, the argument type need not match exactly.
- The compiler will try to convert it:

```
double volume(double h, double w, double d);
```

```
~~~
```

```
double v;
```

```
int height, width;
```

```
float depth;
```

```
~~~
```

```
v = volume(height, width, depth);      // OK
```

108

[04] More About printf

- printf can display values of any arithmetic type:

```
int k = 31;
double x = 19.2;
~~~
printf("k = %d; x = %f\n", k, x);
```

- When executed it displays:

```
k = 37; x = 19.200000
```

- Here's how it works...

109

More About printf

- printf's first argument is a string-literal.
- That literal can contain format specifiers such as %d and %f.
- Each format specifier has a corresponding additional argument:

```
printf("k = %d; x = %f\n", k, x);
```

- The %d specifier displays its corresponding argument as a decimal integer.
- The %f specifier displays its corresponding argument as a fixed-point number.

110

More About printf

- If the format specifiers don't match their corresponding arguments, the behavior is undefined.
- For example,

```
int k = 31;
double x = 19.2;
~~~
printf("k = %d; x = %f\n", x, k);    // mismatching order
printf("k = %d; x = %f\n", k);       // too few
```

- Both of these printf calls have undefined behavior.

111

[04] Fancier Formatting

- Using the %d specifier, printf displays exactly the number of decimal digits required for the value.
- Here are some sample alternatives:
 - **%4d** displays at least 4 characters, right-justified with leading spaces.
 - **%-6d** displays at least 6 characters, left-justified with trailing spaces.
 - **%x** displays the value in hexadecimal, with digits a through f in lowercase.
 - **%8X** displays the value in at least 8 hexadecimal digits, with digits A through F in uppercase, right-justified with leading spaces.

112

[04] Fancier Formatting

- Using `%f`, `printf` displays exactly the number of digits required for the value, with 6 digits to the right of the decimal point.
- Here are some sample alternatives:
 - `%6.2f` displays at least 6 characters, right-justified with leading spaces, and exactly 2 digits to the right of the decimal point.
 - `%-8.3f` displays at least 8 characters, left-justified with trailing spaces, and exactly 3 digits to the right of the decimal point.
 - `%e` displays the value in exponential form, with one digit to the left of the decimal point, 6 to the right, right-justified with leading spaces, and the `e` in the exponent in lowercase.
 - `%E` is the same as `%e`, with the `E` in uppercase.

113

Flow-Control Statements

- In addition to the return-statement, C++ has an assortment of flow-control statements, including:
 - `if`
 - `while`
 - `do`
 - `for`
 - `break`
 - `goto`

114

If-Statements

- The ***if-statement*** has either form:

```
if (expression)  
    statement
```

```
if (expression)  
    statement  
else  
    statement
```

- A statement may be a compound-statement — a sequence of statements enclosed in { }.

115

If-Statements

- The if-statement's expression must:
 - have type `bool`, or
 - have a type implicitly convertible to `bool`.
- An arithmetic value converts to `bool` as follows:
 - zero converts to `false`,
 - non-zero converts to `true`.
- For example, if `x` has an arithmetic type, then these are equivalent:

```
if (x)  
if (x != 0)
```

116

While-Statements

- The ***while-statement*** has the form:

```
while (expression)  
    statement
```

- It repeatedly executes the statement as long as the expression is true.
- It tests the expression each time before executing the statement.
- The statement may be a compound-statement.

117

Do-Statements

- The ***do-statement*** has the form:

```
do  
    statement  
while (expression);
```

- It repeatedly executes the statement as long as the expression is true.
- It tests the expression each time after executing the statement.
- The statement may be a compound-statement.

118

For-Statements

- The ***for-statement*** has the form:

```
for (expression1; expression2; expression3)  
    statement
```

- It is equivalent to:

```
expression1;  
while (expression2) {  
    statement  
    expression3;  
}
```

119

For-Statements

- Here's an actual for-statement:

```
for (i = 0; i < 10; ++i) {  
    product += (i * i);  
}
```

- It's equivalent to:

```
i = 0;  
while (i < 10) {  
    product += (i * i);  
    ++i;  
}
```

120

Break-Statements

- You can use a ***break-statement*** as an alternate way to terminate a loop.
- For example,

```
x = 0;
for (i = 0; i < 10; ++i) {
    t = i * i;
    if (x + t > 127) {
        break;
    }
    x += t;
}
```

121

Labels and Goto-Statements

- A ***label*** is simply an identifier and a : (colon) appearing before a possibly empty statement in a function body.
- A ***goto-statement*** transfers control to a labelled statement.
- Using a label and goto, you can write the previous example as:

```
x = 0;
for (i = 0; i < 10; ++i) {
    t = i * i;
    if (x + t > 127) {
        goto done;
    }
    x += t;
}
done:      // a label
```

122

Short-Circuit Evaluation

- The logical-and expression `x && y` is true if *both* `x` and `y` are true.
 - The program evaluates `x && y` from left to right.
 - If `x` is false, the entire expression is false.
 - In that case, the expression “short circuits” — it doesn’t evaluate `y`.
- The logical-or expression `x || y` is true if *either* `x` or `y` is true.
 - The program evaluates `x || y` from left to right.
 - If `x` is true, the entire expression is true.
 - Again, the expression “short circuits” — it doesn’t evaluate `y`.

123

Short-Circuit Evaluation

- Using short-circuit evaluation, these are equivalent:

```

if (a < b && n == 0) {
    statement1
} else {
    statement2
}

if (!(a < b)) goto else_part;
if (n == 0) {
    statement1
} else {
else_part:
    statement2
}

```

124

Short-Circuit Evaluation

- Using short-circuit evaluation, these are equivalent:

```

if (a < b || n == 0) {
    statement1
} else {
    statement2
}

if (a < b) goto if_part;
if (n == 0) {
if_part:
    statement1
} else {
    statement2
}

```

125

Arrays

- C++ lets you declare arrays using declarations of the form:

```
t x[n];    // x is "array with n elements of type t"
```

- x is the **array name**.
- t is the **element type**.
- n is the **array dimension**.
- sizeof(x) == n * sizeof(t).
- For example,

```
char buffer[256];    // buffer is an array of 256 chars
```

- The array dimension need not be a literal...

126

Constant Expressions

- An array-dimension may be a constant-expression:

```
#define DIM 32
~~~
unsigned scores[2 * DIM - 1];
```

- After preprocessing, we're left with:

```
unsigned scores[2 * 32 - 1];
```

- A **constant-expression** may have operators and multiple operands, as long as all the operands are themselves literals...
- ...so the compiler can compute it at compile-time.

127

Array Subscripting

- Array subscripts start at 0.

```
char buffer[256];
```

- This array has 256 elements:
 - The first is buffer[0].
 - The last is buffer[255].
- For example, this fills the buffer with spaces:

```
for (i = 0; i < 256; ++i) {
    buffer[i] = ' ';
}
```

128

Array Initializers

- You can declare an array with an initial value:

```
int number[7] = { 8, 6, 7, 5, 3, 0, 9 };
```

- This initializer is an *initializer-list*.
- If the array dimension is greater than the length of the list, the remaining array elements are initialized with 0.
- That is, these are equivalent:

```
int number[10] = { 8, 6, 7, 5, 3, 0, 9 };
int number[10] = { 8, 6, 7, 5, 3, 0, 9, 0, 0, 0 };
```

129

Array Initializers

- If the array dimension is less than the list length, it's a compile error:

```
int number[6] = { 8, 6, 7, 5, 3, 0, 9 };    // error
```

- If the array declaration has an initializer, you can omit the dimension.
- The array dimension becomes the length of the list.
- Thus, these are equivalent:

```
int number[7] = { 8, 6, 7, 5, 3, 0, 9 };
int number[ ] = { 8, 6, 7, 5, 3, 0, 9 };
```

130

Array Parameters

- You can declare functions with array parameters:

```
int sum(int x[]);      // an array parameter
~~~
int s;
int v[100];
~~~
s = sum(v);           // pass v's address
```

- C++ doesn't copy argument `v` to parameter `x`.
- Rather, it just passes `v`'s base address...

131

Array Parameters

- If you provide a dimension, the compiler ignores it:

```
int sum(int x[100]);   // an array parameter
~~~
int s;
int v[100];
~~~
s = sum(v);           // still just the address
```

- How then, does `sum` know how many elements are in `v`?
- The common convention is to...

132

Array Parameters

- Pass the array dimension as another argument:

```
int sum(int x[], int n);    // n is x's dimension
~~~
int s;
#define V_DIM 100
int v[V_DIM];
~~~
s = sum(v, V_DIM);        // conventional
```

133

[05] Exercise: Array Parameters

- Implement:

```
int sum(int x[], int n);
```

- Calling `sum(v, m)` should return the sum of the first `m` elements in array `v`.
- Use the provided program stub to test your function.

134

Array Dimension vs. Array Size

- Again, when you pass an array to a function, you often pass its dimension:

```
#define V_DIM 100
int v[V_DIM];
~~~
s = sum(v, V_DIM);           // conventional
```

- But, when you define an array with an initializer, you usually omit the dimension:

```
int number[] = { 8, 6, 7, 5, 3, 0, 9 };
```

- What do you use as the dimension?

135

Array Dimension vs. Array Size

- This doesn't work:

```
int number[] = { 8, 6, 7, 5, 3, 0, 9 };
~~~
s = sum(number, sizeof(number));    // wrong value
```

- sizeof returns the size in bytes, not the number of elements.
- This does work:

```
s = sum(number, sizeof(number)/sizeof(number[0]));
```

- It's just not very convenient and readable...

136

Function-Like Macros

- A preprocessor macro can have one or more parameters, as in:

```
#define DIM_OF(a) (sizeof(a)/sizeof(a[0]))
```

- The use of this macro looks like a function call:

```
s = sum(number, DIM_OF(number));
```

- Hence, it's called a *function-like macro*.
- After preprocessing, the translation unit contains:

```
s = sum(number, sizeof(number)/sizeof(number[0]));
```

137

String Literals

- As with all other literals, each string-literal has a type.
 - It's an "array of char".
- Every string-literal actually has an additional character at the end, the *null-character*, '\0'.
- That is, these are equivalent:

```
char name[] = "Nancy";  
char name[] = { 'N', 'a', 'n', 'c', 'y', '\0' };
```

- Any character array that has a null at the end is a *null-terminated character sequence* (NTCS).
- The C Standard calls them *strings*, but *string* has another meaning in C++.

138

[06] NTCS Functions

- The Standard Library has a header `<string.h>` that declares numerous functions that manipulates NTCSs.
- They use the null-character to mark the end of the string.
- For example, the standard `strlen` function computes the length of an NTCS more-or-less as this function does:

```
int mystrlen(char s[]) {
    int i = 0;
    while (s[i] != '\0') {
        ++i;
    }
    return i;
}
```

139

Pointers and Addresses

- A pointer object holds the address of another object.
- A pointer type includes the type of object to which it points:

```
int *pi;           // a "pointer to int"
unsigned long *pul; // a "pointer to unsigned long"
```

- For any object `x`, the expression `&x` returns the address of `x`.
- If `x` has type `t`, the `&x` has type "pointer to `t`".

```
int i;
unsigned long ul;
~~~
pi = &i;
pul = &ul;
```

140

Pointer Dereferencing

- You use the unary `*` operator to dereference a pointer.
- If `p` is a pointer, then `*p` is the object to which it points.

```
int *pi;           // a "pointer to int"
unsigned long *pul; // a "pointer to unsigned long"
~~~
int i = 13;
unsigned long ul = 42;
~~~
pi = &i;
pul = &ul;
~~~
*pi = 14;           // store 14 into i
*pul += 2;          // add 2 to ul
```

141

Pointer Conversions

- If t_1 and t_2 are distinct types, then there's no implicit conversion from "pointer to t_1 " into "pointer to t_2 ", and vice versa.

```
int i;
unsigned long ul;
~~~
int *pi;           // a "pointer to int"
unsigned long *pul; // a "pointer to unsigned long"
~~~
pi = &ul;           // compile error: type mismatch
pul = &i;           // compile error: type mismatch
```

142

Null Pointers

- There's a special built-in conversion from the integer value 0 to any pointer type:

```
int *pi = 0;  
double *pd = 0;
```

- The result of the conversion is a ***null pointer value***.
- In effect, a pointer with a null value doesn't point to anything.
- The standard header `<stddef.h>` defines a macro representing the null pointer value:

```
#define NULL 0
```

143

Passing by Value

- By default, C++ passes function arguments by value.
- The compiler copies the argument to the parameter.
- The parameter's copy is separate from the original.
- Changes to the parameter copy don't affect the original argument.

144

Passing by Value

- Now, consider a function named `swap`.
- A typical call might look like:

```
int i, j;  
~~~  
swap(i, j);
```

- We want the call to place the value that was in `i` into `j` and the value that was in `j` into `i`.

145

[07] Passing by Value

- You might write the function as:

```
void swap(int v1, int v2) {  
    int tmp = v1;  
    v1 = v2;  
    v2 = tmp;  
}
```

- Unfortunately, calling `swap(i, j)` has no outward effect:
 - It swaps the copies of `i` and `j` in `v1` and `v2`, respectively.
 - Then it throws away the copies.
- `i` and `j` remain unchanged.

146

[08] Pointer Parameters

- You can use pointer parameters instead:

```
void swap(int *v1, int *v2) {
    int tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
```

- Now, you have to pass the addresses of i and j as the arguments:

```
swap(&i, &j);           // swap values of i and j
```

- This works as expected.

147

Pointers and Arrays

- You can initialize a pointer with an expression whose type is array:

```
int x[10]; // x is "array with 10 elements of type int"
int *p;    // p is "pointer to int"
~~~
p = x;     // Why does this compile?
```

- It compiles because, in standards committee vernacular:
 - Arrays “decay” into pointers.**
- More precisely...

148

Arrays “Decay” to Pointers

- The assignment expression below performs an implicit ***array-to-pointer conversion***.
- That is, the compiler treats `x` as if it were a “pointer to `int`” whose value is `&x[0]`:

```
int x[10];
int *p;
~~~
p = x;      // x decays to &x[0]
```

- The “decay” is momentary — just long enough to assign the pointer value of `x` to `p`.
- All the while, the compiler’s symbol table says that `x` is an array.

149

Momentary Conversions

- Momentary conversions occur frequently during expression evaluation, as in:

```
double d;
int i;
~~~
d = d + i;
```

- We often say that the expression `d + i` “converts `i` to double” before adding `d` and `i`.
- However, the program doesn’t really change `i` into a double...

150

Momentary Conversions

- Rather, it creates a temporary double object (possibly in a CPU register) initialized with the value of `i`.
- Then it adds `d` and the temporary:

```
double d;
int i;
~~~
d = d + i;  →  double temp = i;
               d = d + temp;
```

- The temporary vanishes soon thereafter.
- Object `i` remains an `int`.

151

Array-to-Pointer Conversions

- If `x` has type “array of `t`”, then the array-to-pointer conversion yields a “pointer to `t`”.
- That pointer is ***non-modifiable***.
- For example:

```
t x[N];           // x is an array
t *p;             // p is a pointer
~~~
p = x;            // OK: can assign to p
x = p;            // compile error: can't assign to x
```

152

Telling Arrays from Pointers

- Again, an array “decays” to a pointer when you use the array in an expression.
- How can you ever tell that it’s an array, and not just a pointer?
- Because ***the conversion doesn’t happen every time.***
- It doesn’t occur in a `sizeof` expression.
- For example:

```
int v[10];      // sizeof(v) == 10 * sizeof(int)
```

- `sizeof(v)` is not `sizeof(int *)`.

153

Pointer Arithmetic

- You can use pointers instead of subscripts to access array elements.

```
int x[10];
int *p = x;      // same as p = &x[0];
~~~
++p;             // p now points to &x[1]
```

- By definition, these all mean the same thing:

```
++p;
p += 1;
p = p + 1;
```

154

Pointer Arithmetic

- Effectively, `++p` means “increment `p` so that it points to the next array element”.
- If `p` were “pointer to `char`”:
 - `sizeof(char)` is 1.
 - `++p` would add 1 to `p`.
- In this example, `p` is actually “pointer to `int`”.
 - `sizeof(int)` is typically greater than 1, say 4.
 - `++p` adds 4 to the pointer representation so that it points to the next `int` in the array.

155

Pointer Addition

```
t x[10];
int k;
t *p = x;
```

- You can add any integer value to a pointer:

```
p = p + k;
```

- The compiler generates code roughly equivalent to:

```
int tmp = (int)p;           // cast the pointer to an int
tmp += (k * sizeof(t));     // scale k by sizeof(t)
p = (t *)tmp;               // cast the int back to a pointer
```

156

Pointer Subtraction

- Adding an integer to a pointer yields another pointer.

```
int k;
t *p, *q;
~~~
q = p + k;      // p + k yields a pointer
```

- Subtracting a pointer from a pointer yields an integer:

```
k = q - p;      // q - p yields an integer
```

157

Momentary Conversions, Again

- Subscripting expressions such as `x[i]` also treat arrays as pointers.
- C++ expects `x` in `x[i]` to have some pointer type:
 - If `x` is indeed a pointer, it remains as such.
 - If `x` is an array, the compiler converts `x` to a pointer (`x` “decays”) before compiling the rest of the expression.
- Again, the compiler doesn’t really change `x` into a pointer:
- It creates a temporary pointer object whose value is `&x[0]`.

158

Subscripting

- By definition, `x[i]` means the same thing `*(x + i)`.
- These are equivalent:

```
int i;
for (i = 0; i < n; ++i) {
    sum += x[i];
}
```

```
int i;
for (i = 0; i < n; ++i) {
    sum += *(x + i);
}
```

159

Using Pointers to Traverse Arrays

- Again, here's the loop using pointer arithmetic:

```
int i;
for (i = 0; i < n; ++i) {
    sum += *(x + i);
}
```

- Alternatively, you can use a pointer as the loop control:

```
int *p;
for (p = x; p < x + n; ++p) {
    sum += *p;
}
```

160

Pointer Dereferencing

- Be careful with operator precedence.
- `++*p` groups as `++(*p)`.
 - It increments `*p` and returns the incremented value.
- `*++p` groups as `*(++p)`.
 - It increments `p` and then returns `*p`.
- `*p++` groups as `*(p++)`.
 - It returns `*p` and then increments `p`.
 - The `*` operator applies to the value `p` had before incrementing.

161

Array Parameters

- An array declaration in a parameter list really declares a pointer.

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

- It means the same as:

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

- Declaring both of these is not an error:

```
int f(t x[N]);
int f(t *x);        // OK: a benign redeclaration
```

- It's just two declarations for the same function.

162

Array Dimensions in Parameters

- Since a parameter declared as an array really has a pointer type, the compiler ignores the array dimension.
- Thus, the array dimension is optional.
- For example, all of these are equivalent:

```
int f(t x[N]);      // OK: x is "pointer to t"
int f(t x[]);       // OK: x is "pointer to t"
int f(t *x);        // OK: x is "pointer to t"
```

- All declare x as a parameter of type “pointer to T”.

163

sizeof Revisited

- You can tell that an array parameter is really a pointer by its size:

```
int g(int v[10]) {
    return sizeof(v);      // returns sizeof(int *)
}
```

- A call to g accepts any argument whose type converts to “pointer to int”, even if refers to an array with a different dimension:

```
int w[20];
~~~
g(w);          // compiles OK
```

164

[09] Exercise: Traversing with Pointers

- Rewrite the `sum` function from an earlier exercise using pointer arithmetic instead of subscripts.

165

[10] Exercise: Traversing with Pointers

- Rewrite the `strlen` function from the earlier example using pointer arithmetic instead of subscripts.
- Try using a `while` loop.
- Try it again using a `for` loop.

166

Structures

- A **structure** is a type composed of a sequence of member objects of possibly different types.
- For example, the following defines a structure named `date` with three members named `day`, `month`, and `year`:

```
struct date {
    unsigned char day;
    char month[4];
    unsigned short int year;
};

date today;           // today is an object of type date
date *when;           // when is a "pointer to date"
```

167

Elaborated Type Specifiers

- In C, a structure name by itself is not a type.
- It's called a **tag**.
- In C, you must use the keyword `struct` before each structure tag:

```
struct date today;           // required in C; OK in C++
struct date *when;          // required in C; OK in C++
```

- In C++, you can use a structure name by itself as a type:

```
date today;                  // error in C; OK in C++
date *when;                  // error in C; OK in C++
```

168

Accessing Structure Members

- The . (dot) operator selects a member of a structure.

```
date flux;
```

```
~~~
```

```
flux.day = 5;
```

```
strcpy(flux.month, "Nov");
```

```
flux.year = 1955;
```

- strcpy is a Standard Library function declared in *<string.h>*.
- strcpy(dst, src) copies NTCS src to character array dst.

169

Initializing Structures

- You can use a brace-enclosed initializer with a structure.

```
struct date {  
    unsigned char day;  
    char month[4];  
    unsigned short int year;  
};
```

```
~~~
```

```
date flux = { 5, "Nov", 1955 };
```

170

Accessing Structure Members

- The `->` (arrow) operator selects a member of a structure that's referenced via a pointer.

```
date *when = &flux;
~~~
printf("%d %s %u", when->day, when->month, when->year);
```

- The `%s` specifier formats an NTCS.
- The `%u` specifier formats an unsigned `int`.

171

Accessing Structure Members

- For any pointer `p` to a structure with member `m`:
 - `p->m` is equivalent to `(*p).m`.
- What happens if you omit the parentheses?
 - `.` (dot) has higher precedence than `*`.
 - Thus, `*p.m` means the same as `*(p.m)`.
- Take a moment to think how you can declare `p` and `m` in some way so that `*p.m` is a valid expression.
- An answer appears on the next slide...

172

Accessing Structure Members

- Consider the following declarations:

```
struct thing {  
    char *m;  
};  
thing p;
```

- With these declarations, `*p.m` is a valid expression, equivalent to `*(p.m)`.
- It's the character to which `p.m` points.

173

Unions

- Suppose you have two or more objects of different types.
- Only one of those objects can exist at any given time.
- Rather than allocate storage for each object separately, you can place them in a union.
- A ***union*** is simply a structure whose members are all at offset zero.
- The size of a union is the size of its largest member, plus any padding to preserve member alignment.

174

Unions

- This union places an `int` and a “pointer to `int`” at the same location:

```
union arithmetic_pointer {  
    int *p;  
    int i;  
};  
  
arithmetic_pointer ap;
```

175

Enumeration Types

- An **enumeration** definition specifies a type and a corresponding set of named constants.

```
enum day {  
    Sunday, Monday, Tuesday, Wednesday,  
    Thursday, Friday, Saturday  
};
```

- This defines a type `day` with seven constants named for each day of the week.

176

Enumeration Constants

- By default:
 - The first enumeration constant in an enumeration definition has the value of 0.
 - Each subsequent constant has the value which is one more than the value of the previous constant.
- Thus, Sunday's value is 0, Monday's is 1, and so on.
- Saturday's value is 6.

177

Enumeration Constants

- You can specify an explicit value for any enumeration constant:

```
enum month {
    January = 1, February, March,
    ~~~
    October, November, December
};
```

- This defines January with the value 1 instead of 0.
- Any enumeration constant without an explicitly specified value has the value one more than the value of the previous constant.
- Thus, February's value is 2, March's is 3, and so on.

178

Scope

- When the compiler encounters the declaration of a name, it stores that name and its attributes into a symbol table.
- When the compiler encounters a reference to a name, it looks up the name in the symbol table to find those attributes.
- Each declared name is valid only within some region of the program text called its ***scope***.
- That region need not be contiguous.

179

Scope Regions

- In C, declarations can appear at:
 - ***block scope***: local to a function definition (including that function's parameter list) or a block nested therein
 - ***file scope***: outside of any function, structure, or union
- In C++, declarations can appear in the same scopes as in C, except:
 - C++ refers to *block scope* as ***local scope***.
 - C++ refers to *file scope* as ***global scope***.
- C++ has other scopes, to be discussed.

180

Scope vs. Linkage vs. Storage Duration

- The concept of scope is meaningful ***only within a single translation unit.***
 - For example, a name that's at global scope is visible at the outermost level of a single translation unit.
 - It might or might not be accessible outside the translation unit.
- A name declared in one translation unit can refer to a name defined in another translation unit when the name has an attribute called ***external linkage.***
- Many programmers have trouble separating scope from linkage and from another attribute called ***storage duration.***
- Here's why...

181

Storage-Class Specifiers

- C provides four storage-class specifiers:

auto extern register static
- Until recently, C++ did also.
- However, C++11 made changes:
 - The keyword `auto` became a ***type specifier***, with a completely different meaning.
 - Nobody was using `auto`, so C++11 put it to work elsewhere.
 - The keyword `register` has been *deprecated*.
 - That is, it's still in C++, but may be removed in the future.
 - Modern compilers permit it, but ignore it.

182

Storage-Class Specifiers

- The keyword `static` is particularly confusing because it has multiple personalities.
 - Sometimes it affects the way a program allocates storage.
 - Sometimes it affects how object files link together.
 - In C++, it can alter the behavior of class member functions.
- Storage-class specifiers affect the linkage and storage duration of names appearing in a declaration.
- They don't affect scope.

183

Linkage

- The ***linkage*** of a name is the extent to which that name might refer to a name declared elsewhere.
- C++ provide for three categories of linkage:
 - external linkage
 - internal linkage
 - no linkage

184

External Linkage

- A name with **external linkage** denotes an entity that a program can reference via the same name declared in multiple translation units.
- For example, these declarations appear in separate translation units, but refer to the same object:

```
// file1.cpp
extern int total = 0;    // a definition
~~~

// file2.cpp
extern int total;        // a non-defining declaration
~~~                          // for the same object
```

185

Internal Linkage

- A name with **internal linkage** denotes an entity that a program can reference via the same name declared only in the same translation unit.
- For example, these declarations for `total` appear in different scopes of the same translation unit, but refer to the same object:

```
// file1.cpp
static int total = 0;    // a definition
~~~

int f(int i) {
    extern int total;    // a non-defining declaration
    ~~~                          // for the same object
}
```

186

Internal Linkage

- Again, this declaration has internal linkage:

```
// file1.cpp
static int total = 0;    // a definition
~~~
```

- Thus, this declaration in another translation unit refers to a different object:

```
// file2.cpp
extern int total;        // a non-defining declaration
~~~                      // for a different object
```

187

No Linkage

- A name with ***no linkage*** denotes an entity that a program can't reference via names from other scopes.
- For example, each of these objects named `total` has no linkage:

```
int f(int i) {
    int total;           // a definition
    ~~~
}

int g(int i) {
    int total;           // a definition
    ~~~                  // for a different object
}
```

188

Storage Duration

- The ***storage duration*** of an object defines the lifetime of the storage containing the object.
- It can be:
 - static
 - automatic
 - dynamic

189

Storage Duration

- The storage for an object with ***static storage duration***:
 - is allocated at program startup;
 - persists for the duration of program execution.
- The storage for an object with ***automatic storage duration***:
 - is allocated incidental to a function call, and
 - is deallocated incidental to the corresponding function return.
- The storage for an object with ***dynamic storage duration***:
 - is allocated by a call to an allocation function (such as `malloc` or C++'s operator `new`) and
 - is deallocated by a corresponding call to a deallocation function (`free` or C++'s operator `delete`).

190

Storage-Class Specifiers and Functions

- Here's a sampling of how the storage-class specifier and scope of a function declaration determine the function's linkage.
 - It's not a complete set of rules.
- In the following, *r* and *t* represent types...

191

Storage-Class Specifiers and Functions

- `static r f(t);`
 - at file (global) scope:
 - `f` has internal linkage.
 - at block (local) scope:
 - The declaration is invalid.
- For example:

```
static int foo(int);           // foo has internal linkage
~
int main() {
    static int bar(int);      // error
    ~
}
```

192

Storage-Class Specifiers and Functions

- `extern r f(t);`
 - at file (global) or block (local) scope:
 - `f` has internal linkage if previously declared with internal linkage in a visible declaration;
 - otherwise it has external linkage.
- For example:

```
static int foo(int);           // foo has internal linkage
~~~
int main() {
    extern int foo(int);       // foo has internal linkage
    extern int bar(int);       // bar has external linkage
    ~~~
}
```

193

Storage-Class Specifiers and Functions

- `r f(t);` // no storage-class specifier
 - at file (global) or block (local) scope:
 - `f` has the same linkage as if it were declared `extern`.
- For example:

```
static int foo(int);           // foo has internal linkage
int bar(int);                  // bar has external linkage
~~~
int main() {
    int foo(int);              // foo has internal linkage
    int bar(int);              // bar has external linkage
    ~~~
}
```

194

Storage-Class Specifiers and Functions

- These declarations can appear in the same scope:

```
static int foo(int n);    // foo has internal linkage
~~~~~
extern int foo(int n);    // still true
```

- This is valid in both C and C++.
- However, just because you can do this doesn't mean you should.
- Writing function declarations that differ only in their storage-class specifiers is rarely necessary, if ever, and can be confusing.

195

Storage-Class Specifiers and Objects

- Here's a sampling of how the storage-class specifier and scope of an object declaration determine the object's linkage and storage duration.
- Again, it's not a complete set of rules.
- In the following, *t* represents a type...

196

Storage-Class Specifiers and Objects

- `static t v;`
 - at file (global) scope:
 - `v` has internal linkage and static storage.
 - at block (local) scope:
 - `v` has no linkage and static storage.

197

Storage-Class Specifiers and Objects

- `extern t v;`
 - at file (global) or block (local) scope:
 - `v` has internal linkage if previously declared with internal linkage in a visible declaration;
 - otherwise it has external linkage.
 - `v` has static storage.

198

Storage-Class Specifiers and Objects

- `t v; // no storage-class specifier`
 - at file (global) scope:
 - `v` has the same linkage as if it were declared `extern`.
 - `v` has static storage.
 - at block scope:
 - `v` has no linkage and automatic storage.

199

Storage-Class Specifiers and Objects

- For example,

	<code>//</code>	<code>linkage=</code>	<code>storage=</code>
<code>static int i;</code>	<code>// i:</code>	<code>internal</code>	<code>static</code>
<code>extern int i;</code>	<code>// i:</code>	<code>internal</code>	<code>static</code>
<code>extern int j;</code>	<code>// j:</code>	<code>external</code>	<code>static</code>
~~~~			
<code>int foo(int k) {</code>	<code>// k:</code>	<code>no</code>	<code>automatic</code>
<code>extern int i;</code>	<code>// i:</code>	<code>internal</code>	<code>static</code>
<code>static int j;</code>	<code>// j:</code>	<code>no</code>	<code>static</code>
<code>int m;</code>	<code>// m:</code>	<code>no</code>	<code>automatic</code>
<code>extern int n;</code>	<code>// n:</code>	<code>external</code>	<code>static</code>
~~~~			
<code>}</code>			

200

Default Initial Values

- Objects with static storage are default initialized to zero.
 - That zero will be converted to the object's type.
- Otherwise, uninitialized objects have *indeterminate* values.
 - That is, you can't be sure what's there.
- Accessing an object with indeterminate value produces undefined behavior.

```
float total;           // initially 0.0f

int main() {
    static char *p;     // initially null
    int i;              // indeterminate initial value
    ~~~
}
```

201

The Structure of Declarations

- *Insight: Every object and function declaration has two parts:*
 - A sequence of one or more **declaration specifiers**
 - A **declarator** (or a sequence thereof, each with an optional initializer, separated by commas)
- For example:

static unsigned long int *x[N];
↙ ↖
declaration specifiers *declarator*

202

Declaration Specifiers

- A **declaration specifier** can be:
 - a **type specifier**:
 - a keyword such as `int`, `unsigned`, `long`, or `double`
 - a **storage class specifier**:
 - a keyword such as `extern` or `static`
- A **declarator** is the name being declared, possibly surrounded by operators:
 - `*` means “pointer”
 - `[]` mean “array”
 - `()` mean “function”

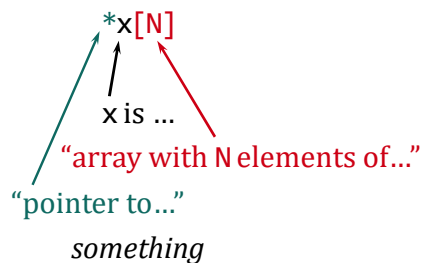
203

The Structure of Declarations

- For example, let’s dissect the declaration:

```
static unsigned long int *x[N];
```

- Start with the declarator:



204

The Structure of Declarations

- That *something* is specified by the declaration specifiers:

static unsigned long int *x[N];

“array with N elements of pointer to...”

“unsigned long int”

- As explained later, static doesn't contribute to the type.
- It contributes in other ways.

205

The Structure of Declarations

- How do you know that *x[N] is:
 - an “array of ... pointer to ...”
- How do you know it's not:
 - a “pointer to an array of ...”?
- That knowledge comes from this insight...

206

Declarator Operators

- *Insight: The operators in a declarator group according to the same precedence as they do when they appear in an expression.*

Precedence	Operator	Meaning
Highest	()	grouping
	[]	"array"
	()	"function"
Lowest	*	"pointer"

- For example, [] has higher precedence than *.
- Thus the declarator *x[N] means that x is an "array of pointer" rather than a "pointer to array".

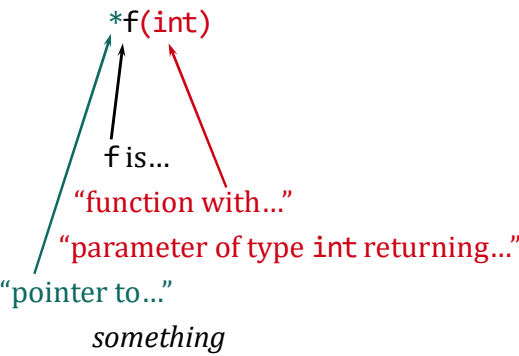
207

Parentheses in Declarators

- Parentheses serve two roles in declarators:
 - As the **function call operator**, () have the same precedence as [].
 - As **grouping**, () have the highest precedence of all.
- For example...

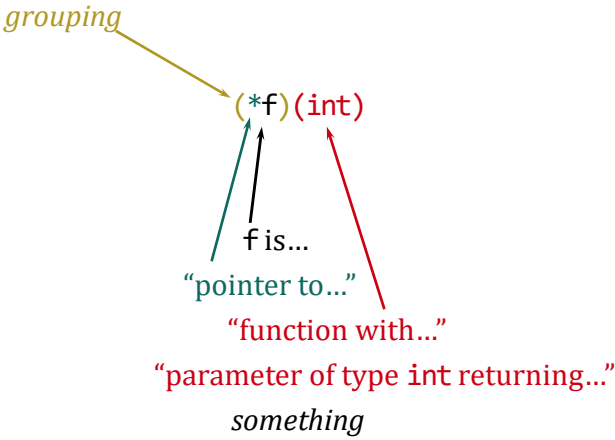
208

Parentheses in Declarators



209

Parentheses in Declarators



210

Declarator-Ids

- A declarator need not contain any operators at all:

declaration-specifier *declarator*

- A declarator may contain more than one identifier.
- One is the **declarator-id** — the one being declared.
- Others, if any, must have been declared previously.

declarator-id *other identifier*

211

Style and Syntax

- Some programmers use spacing to join the * with the declaration specifiers rather than with the declarator, as in:

```
int* p;      // unhelpful reality distortion
```

- Doing this is a disservice to yourself and others.
- ✓ *Use spacing to keep * operators with the rest of the declarator, as in:*

```
int *p;      // This is reality
```

- Recognizing the boundary between the declaration specifiers and the declarator is a key to understanding declarations.
- Deceptive spacing doesn't help.

212

Typedef

- Sometimes, it's convenient to give a complicated type a simple name.
- For example, pointers to functions have mildly complicated types:

```
int (*pf)(int *);
```

- Here, pf is a “pointer to function with a parameter of type ‘pointer to int’ returning an int”.
- Suppose you want to declare a few of these, all the same type...

213

Typedef

- You can define a **type alias** using a **typedef-declaration**.

```
typedef int (*func_ptr)(int *); // func_ptr is a type
```

- Now you can use func_ptr as the type name in other declarations.

```
func_ptr pf1, pf2; // pf1 and pf2 are function pointers
```

- Now, pf1 and pf2 are objects that are function pointers of exactly the same type.

214

Bibliography

- ISO [1990]. *ISO/IEC Standard 9899:1990, Programming languages—C.*
- ISO [1998]. *ISO/IEC Standard 14882:1998, Programming languages—C++.*
- ISO [1999]. *ISO/IEC Standard 9899:1999, Programming languages—C.*
- ISO [2003]. *ISO/IEC Standard 14882:2003, Programming languages—C++.*
- ISO [2005]. *ISO/IEC TR 19768, C++ Library Extensions.*
- ISO [2011a]. *ISO/IEC Standard 9899:2011, Programming languages—C.*
- ISO [2011b]. *ISO/IEC Standard 14882:2011, Programming languages—C++.*

215

Bibliography

- ISO [2014]. *ISO/IEC Standard 14882:2014, Programming languages—C++.*
- Kernighan and Ritchie [1978]. Brian Kernighan and Dennis Ritchie, *The C Programming Language*. Prentice-Hall.
- Stroustrup [1985]. Bjarne Stroustrup, *The C++ Programming Language*. Addison-Wesley.

216

[2] A Better C

1

xr: A Small Example of a Large Program

- A primary topic of this seminar is using C++ to build large software systems out of simpler abstractions.
- For this, we need to study a concrete programming example.
 - It must be big enough to demonstrate the problems of developing and maintaining really big programs.
 - It must be complicated enough so that you can see if the chosen techniques really do make it simpler.
- On the other hand, the example can't be so complicated that it drowns us in details.

2

xr: A Small Example of a Large Program

- *xr* is a simple cross-reference generator:
 - inspired by exercise 6-3 in Kernighan & Ritchie [1988], and
 - solved by Tondo & Gimpel [1989].
- *xr* writes to standard output an alphabetized cross-reference listing of words read from standard input.
- Here, a *word* is a contiguous sequence of letters, underscores and digits starting with a letter or underscore.
 - That is, a word is really an *identifier* as in C++.
- *xr* writes each word on a separate line followed by the set of line numbers on which that word appears in the input.

3

xr: A Small Example of a Large Program

- For example, you can use the source code for *xr* as input to the program, by running *xr* from a command line such as:

```
xr <xr.c
```

- The output should contain a line for the word `token` that looks something like:

```
token:      42      44      45      46      47
```

- `token` may actually appear more than once on some input lines, but each number appears only once in the output line.

4

xr: A Small Example of a Large Program

- For simplicity, the program assumes each word and all its line numbers will fit on a single output line.
- It doesn't break any long output lines into multiple lines.
- On most output displays, long lines just wrap around to the beginning of another line.

5

A High-level Algorithm for *xr*

- There are at least two ways to process the input to this program:
 - Read a **line at a time** into a buffer and scan each word from the buffer.
 - Read a **character at a time** and piece the characters into a word in a buffer.
- The program presented here uses the latter approach.
- Consequently, the input processing must recognize newline characters.
 - The program tracks line numbers by tracking newlines.
- Collectively, *words (identifiers)* and *newline characters* are called **tokens**.

6

A High-level Algorithm for *xr*

- At its highest level, the cross-referencing algorithm is pretty simple:
 - Set the current line number to 1.
 - While not at end of input, repeat the following steps:
 - Read the next token (a word or a newline character).
 - If that token is a word then...
 - ... record that it appeared on the current line number.
 - Otherwise, that token is a newline, so...
 - ... increment the current line number.
 - Write the alphabetized cross-reference listing.

7

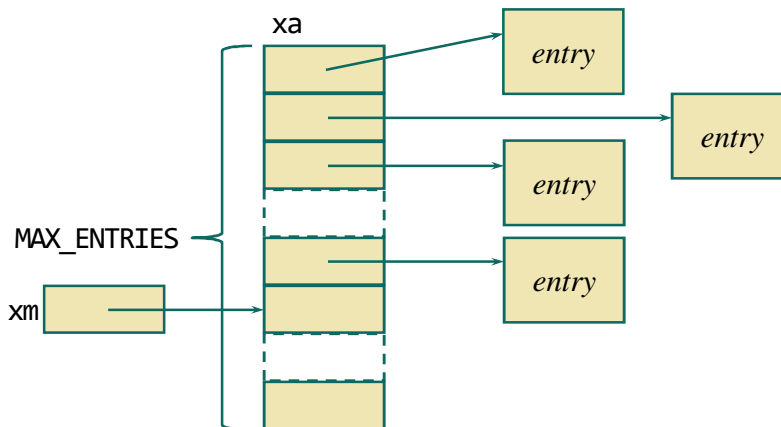
xr's Data Structure

- *xr* collects the words into an initially empty array, *xa*.
 - Each array element is a pointer to a cross-reference entry.
 - Each entry holds a word and its associated line numbers.
- *xa* has room for up to `MAX_ENTRIES` elements.
- *xm* is the index of the next available position in *xa*.
 - It starts at 0.
 - It must remain less than `MAX_ENTRIES`.

8

xr's Data Structure

- The data structure looks like:



9

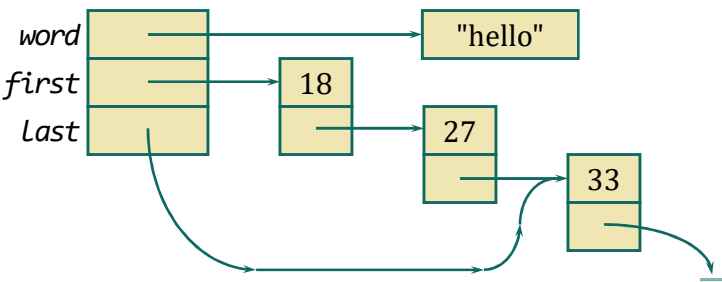
xr's Data Structure

- Each cross-reference entry holds:
 - **word**: a pointer to a null-terminated character array that holds the spelling of a word
 - **first**: a pointer to the first node in a singly-linked list of line numbers
 - **last**: a pointer to the last node in a singly-linked list of line numbers

10

xr's Data Structure

- For example, if the word “hello” appears on input lines 18, 27 and 33, then the corresponding cross-reference entry looks like:



11

An Implementation of *xr*

- The following implementation for *xr* uses simpler data structures than the original version in Tondo & Gimpel [1989].
- It's a procedure-oriented C program.
- It doesn't yet compile as C++.
 - It will, with just a few tweaks.
- It's not yet object-oriented.
 - It will be, but not right away.

12

Standard Library Stuff

- *xr*'s implementation uses a number of C Standard Library components.
- They come from these headers:
 - `<assert.h>` — diagnostics
 - `<ctype.h>` — character handling
 - `<stdio.h>` — input and output
 - `<stdlib.h>` — general utilities
 - `<string.h>` — “string” handling
- Here's a quick look at these components...

13

`<assert.h>`

- `<assert.h>` defines `assert` as a function-like macro.
- Calling `assert` expands to code that tests a condition:

`assert(condition);`
 - If *condition* is true (non-zero), nothing happens.
 - If it's false (zero), the program produces a diagnostic message and halts execution.
- *xr* uses `assert` to turn what would be undefined behavior into an overt failure.
- It's not the best approach, but better than nothing for now.

14

<stdio.h>

- `xr` uses `printf` to write formatted data to standard output.
- It uses `getchar` to read from standard input.
- Calling `getchar()` returns either:
 - the next input character, or
 - EOF, an end-of-file indicator (there's no more input).
- `<stdio.h>` defines EOF as a macro with a negative value, typically:

```
#define EOF (-1)
```

- On some processors, “plain” `char` objects can't hold negative values, but `int` can...

15

<stdio.h>

- Thus, `getchar` returns, not a `char`, but an `int`:

```
int getchar();
```

- Calls to `getchar` should capture the return value in an `int`.
- Here's a typical loop that gets characters until EOF:

```
int c;  
c = getchar();  
while (c != EOF) {  
    // we've got a character!  
    c = getchar();  
}
```

16

<stdio.h>

- Assignment returns a value — the left operand after the assignment takes place.

```
c = getchar();
```

- This assignment returns the value of the character copied to c.
- Thus, you can write the loop more concisely as:

```
int c;  
while ((c = getchar()) != EOF) {  
    // we've got a character!  
}
```

- This is a common idiom.

17

<stdio.h>

- *xr* also uses the putchar function to write to standard output.
- putchar often puts what getchar gets.
- Thus, putchar accepts an int, not a char:

```
int putchar(int c);
```

- Here's a loop that copies standard input to standard output until EOF:

```
int c;  
while ((c = getchar()) != EOF) {  
    putchar(c);  
}
```

18

<stdio.h>

- *xr* uses one other function from *<stdio.h>*, `ungetc`.
- Calling `ungetc` “ungets” a character to a input stream:

```
ungetc(c, stdin);
```

- `stdin` is the global object declared in *<stdio.h>* that represents standard input.
- The next call to `getchar` will return the “ungot” character.

19

<ctype.h>

- As *xr* reads input, it needs to detect characters that are letters and digits.
- *<ctype.h>* declares functions that do this, including:
 - `isdigit(c)` returns true if `c` is a decimal digit
 - `islower(c)` returns true if `c` is a lowercase letter
 - `isupper(c)` returns true if `c` is an uppercase letter
 - `isalpha(c)` returns true if `c` is any letter
 - `isalnum(c)` returns true if `c` is a letter or digit

20

<string.h>

- *xr* stores character strings as null-terminated character sequences (NTCSs).
- *<string.h>* provides functions to support this.
- `strcpy(dst, src)` copies NTCS `src` to the character array `dst`.
- `strcmp(s, t)` compares NTCSs `s` and `t` character-by-character.
 - It returns an `int` whose value is:
 - negative if `s < t`,
 - zero if `s == t`, or
 - positive if `s > t`.

21

<stdlib.h>

- *xr* uses dynamically allocated storage to store the words and line numbers it detects.
- It calls the standard `malloc` function to acquire that storage.
- When calling `malloc`, you pass the size (in bytes) of the object that will reside there:

```
t *p;                                // t is some type
~~~
p = malloc(sizeof(t));                // storage for one t
```

- To allocate storage for an array with `n` elements of type `t`, do the math:

```
p = malloc(n * sizeof(t));           // storage for n of them
```

22

[01] Declarations in *xr*

- Now, here's the initial implementation of *xr*.
- It's a single source file, "*xr.c*".
- As is common, it begins with `#include` directives and declarations:

```
// xr.c - a cross-reference generator
```

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
~~~
```

23

[01] Declarations in *xr*

```
// input processing components
```

```
#define MAX_TOKEN 64
```

```
int get_token(char *t, int n);
```

```
~~~~
```

- Calling `get_token(token, MAX_TOKEN)` gets the next token (a word or newline).
- If it finds a token:
 - It copies that token into array `token` of dimension `MAX_TOKEN`.
 - It returns non-zero.
- Otherwise, it returns zero.

24

[01] Declarations in *xr*

```
// cross-reference table components

struct list_node {
    unsigned number;
    struct list_node *next;
};

struct entry {
    char *word;
    struct list_node *first, *last;
};
~~~
```

25

[01] Declarations in *xr*

```
// more cross-reference table components

#define MAX_ENTRIES 1024

struct entry *xa[MAX_ENTRIES];
int xm = 0;
~~~
```

- The definitions for variables *xa* and *xm* allocate storage for the cross-reference table.

26

[01] Declarations in *xr*

```
// more cross-reference table components
```

```
void add_array(
    struct entry *a[], int *m, char *w, unsigned n
);
~~~
```

- Calling `add_array(xa, &xm, token, ln)` records in the cross-reference table represented by `xa` and `xm` that the word in `token` was found on line number `ln`.

27

[01] Declarations in *xr*

```
// more cross-reference table components
```

```
void put_array(struct entry *a[], int m);
void sort_array(struct entry *a[], int m);
~~~
```

- Calling `put_array(xa, xm)` writes to standard output the cross-reference table represented by `xa` and `xm`.
- Calling `sort_array(xa, xm)` sorts the cross-reference table represented by `xa` and `xm` into ascending order by words.

28

A High-level Algorithm for *xr*

- Here, again, is the cross-referencing algorithm:
 - Set the current line number to 1.
 - While not at end of input, repeat the following steps:
 - Read the next token (a word or a newline character).
 - If that token is a word then...
 - ... record that it appeared on the current line number.
 - Otherwise, that token is a newline, so...
 - ... increment the current line number.
 - Write the alphabetized cross-reference listing.
- Here it is, implemented in the `main` function...

29

[01] *xr*'s Main Function

```
int main() {
    char token[MAX_TOKEN];
    unsigned ln = 1;
    while (get_token(token, MAX_TOKEN)) {
        if (isalpha(token[0]) || token[0] == '_') {
            add_array(xa, &xm, token, ln);
        } else { // if (token[0] == '\n')
            ++ln;
        }
    }
    sort_array(xa, xm);
    put_array(xa, xm);
    return 0;
}
```

30

Getting Tokens

- Calling `get_token(token, MAX_TOKEN)` gets the next token (a word or newline) from standard input.
 - `MAX_TOKEN` must be at least 2.
- If it actually gets a token:
 - It copies that token into `token` as a null-terminated character sequence whose length doesn't exceed `MAX_TOKEN`.
 - It discards excess characters.
 - It return true (a non-zero value).
- Otherwise:
 - It returns false (zero).
- Here's the `get_token` function...

31

[01] Getting Tokens

```
int get_token(char *t, int n) {
    int c;
    if (n < 2) {
        return 0;
    }
    while ((c = getchar()) != EOF) {
        if (isalpha(c) || c == '_' || c == '\n') {
            break;
        }
    }
    if (c == EOF) {
        return 0;
    }
    // see the next slide for this part
}
```

32

[01] Getting Tokens

```
int get_token(char *t, int n) {
    // see the previous slide for this part
    *t++ = (char)c;
    --n;
    if (c != '\n') {
        // see the next slide for this part
    }
    *t = '\0';
    return 1;
}
```

33

[01] Getting Tokens

```
int get_token(char *t, int n) {
    // see the previous slide for this part...
    if (c != '\n') {
        while ((c = getchar()) != EOF) {
            if (!isalnum(c) && c != '_') {
                ungetc(c, stdin);
                break;
            } else if (n > 1) {
                *t++ = (char)c;
                --n;
            }
        }
    }
    // ...and for this part
}
```

34

Adding Words and Line Numbers

- Calling `add_array(xa, &xm, token, ln)` adds an entry for word `token` on line number `ln` to the cross-reference table represented by `xa` and `xm`, if no such entry already exists.
 - `xm` must be passed by address.
- `add_array` searches the first `xm` entries of array `xa` for an entry containing word `token`.
 - If no entry matches:
 - `add_array` creates a new entry in `xa[xm]` for word `token` on line number `ln`, and
 - increments `xm`.
 - If an entry matches, but `ln` isn't already in that entry's list:
 - `add_array` adds `ln` to that list.

35

[01] Adding Words and Line Numbers

```
void add_array(
    struct entry *a[], int *m, char *w, unsigned n
) {
    int i;
    for (i = 0; i < *m; ++i) {
        if (strcmp(a[i]->word, w) == 0) {
            break;
        }
    }
    // see the next slide for this part
}
```

36

[01] Adding Words and Line Numbers

```
void add_array(
    struct entry *a[], int *m, char *w, unsigned n
) {
    // see the previous slide for this part
    if (i >= *m) {
        // didn't find the word
        // add the new word and the line number
        // see the next slide for this part
    } else if (a[i]->last->number != n) {
        // found the word but not the line number
        // add a new line number for that word
        // see the next, next slide for this part
    }
}
```

37

[01] Adding Words and Line Numbers

- Here's how `add_array` adds a new word `w` and line number `n`:

```
if (i >= *m) {
    assert(i < MAX_ENTRIES);
    ++*m;
    a[i] = malloc(sizeof(struct entry));
    a[i]->word
        = malloc((strlen(w) + 1) * sizeof(char));
    strcpy(a[i]->word, w);
    a[i]->first = malloc(sizeof(struct list_node));
    a[i]->last = a[i]->first;
    a[i]->last->number = n;
    a[i]->last->next = NULL;
} else if (a[i]->last->number != n) {
```

38

[01] Adding Words and Line Numbers

- Here's how `add_array` adds a new line number `n` for a word:

```
    } else if (a[i]->last->number != n) {  
        a[i]->last->next  
            = malloc(sizeof(struct list_node));  
        a[i]->last = a[i]->last->next;  
        a[i]->last->number = n;  
        a[i]->last->next = NULL;  
    }
```

39

Displaying the Table

- `put_array(xa, xm)` writes to standard output the cross-reference table that's in the first `xm` elements of array `xa`.
- The outer loop visits each word in the table.
- The inner loop visits each line number for a given word...

40

[01] Displaying the Table

```
void put_array(struct entry *a[], int m) {
    int i;
    struct list_node *p;
    for (i = 0; i < m; ++i) {
        printf("%15s:", a[i]->word);
        for (p = a[i]->first; p != NULL; p = p->next) {
            printf(" %7d", p->number);
        }
        putchar('\n');
    }
}
```

41

Sorting the Entries

- `sort_array(xa, xm)` sorts the first `xm` entries in array `xa` into ascending order by the word in each entry.
- It uses an exchange sort...

42

[01] Sorting the Entries

```
void sort_array(struct entry *a[], int m) {
    int i, j;
    if (m == 0) {
        return;
    }
    for (i = 0; i < m - 1; ++i) {
        for (j = i + 1; j < m; ++j) {
            if (strcmp(a[i]->word, a[j]->word) > 0) {
                struct entry *temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

43

Warts and All

- *xr* compiles as C.
- It doesn't yet compile as C++.
- Again, C++ is a superset of C, but a slightly imperfect one.
- Nearly all of the primitive types, built-in operators, and statements in C are also in C++, warts and all.
- However, C++ differs from C in some noteworthy ways...

44

Macros as Symbolic Constants

- C programmers typically define symbolic constants as object-like macros:

```
#define MAX 16
~~~
int table[MAX];
```

- Unfortunately, some compilers don't preserve macro names among the symbols they convey to their debuggers.

45

Macros as Symbolic Constants

- Even worse, macros don't observe the scope rules:

```
void foo() {
    #define MAX 16 // MAX is non-local
    int table[MAX];
    ~~~
}

void bar() {
    #define MAX 32 // error: macro redefinition
    ~~~
}
```

46

Macros as Symbolic Constants

- Macros might substitute in places you don't want them to:

```
#define max 16
```

```
~~~~~
```

```
void sort(int a[], size_t max);
```

substitutes 16 here

- Fortunately, inadvertent macro substitution often produces a diagnostic, albeit a puzzling one.
- Some programmers try to avoid name collisions between macros and other identifiers by adhering to a style convention, such as spelling macro names entirely in UPPERCASE.

47

Enumerations as Symbolic Constants

- You can define enumeration constants with explicit values, as in:

```
enum color { red = 1, green = 2, blue = 4 };
```

- Most parts of this definition are optional.
- As a special case, an enumeration definition can define just a single constant:

```
enum { max = 16 };
```

- This defines `max` as a constant with value 16 of some unnamed enumeration type.

48

Enumerations as Symbolic Constants

- An enumeration constant is a compile-time constant, so you can use it as such:

```
int table[max];    // OK: array dimension

switch (v) {
case max:           // OK: case label
    ~~~
}

struct foo {
    int bf: max;    // OK: bitfield width
    ~~~
};
```

49

Enumerations as Symbolic Constants

- Enumeration constants obey the scope rules.
- Unfortunately, enumeration constants are integers, so you can't use them for floating constants, as in:

```
enum { pi = 3.14159 }; // truncates pi to 3
                        // maybe with a warning
```

- You can't use them for pointer constants, either:

```
int table[max];
enum { limit = table + max }; // error
```

50

Const Objects as Symbolic Constants

- Using enumeration constants of unnamed type is too subtle for some people's tastes.
- C++ provides ***const objects*** as an overt way to define symbolic constants.
- You can use const objects as integer constants:

```
int const max = 16;           // integer
```

- You can also use them as floating and pointer constants:

```
double const pi = 3.1415926;    // floating
int *const limit = table + max;  // pointer
```

51

Const Objects as Symbolic Constants

- Const object names observe the scope rules.
- C and C++ treat const objects differently:
 - In C, you can't use a const object in a compile-time constant expression.
 - In C++, you can.
- For example:

```
int const max = 16;

struct foo {
    int bf: max;    // OK in C++, but not in C
    ~~~
};
```

52

Symbolic Constants

- These are equivalent:

```
unsigned long n;  
long unsigned n;
```

- So are these:

```
const int N = 10;    // (1)  
int const N = 10;    // (2)
```

- The choice is purely a matter of style.
- Many C and C++ programmers use (1).
- For reasons explained in the next C++ course, these notes use style (2).

53

[02] Exercise: Better Symbolic Constants

- *xr* uses two symbolic constants: `MAX_TOKEN` and `MAX_ENTRIES`.
- Both are defined as object-like macros.
- Part 1:
 - Try defining the symbols as enumeration constants.
 - Note that the only uses of `MAX_TOKEN` are local to `main`.
 - Try moving `MAX_TOKEN`'s definition into `main`.
- Part 2:
 - Try defining the symbols as `const` objects.

54

Const in Parameter Lists

- C and C++ also use `const` in parameter declarations, as in:

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

- Here, parameter `s2` has type “pointer to const char”.
- It means that `s2` points to characters that `strcpy` won’t change.
- This is an important use for `const`.
- However, you can’t really understand `const` in C++ unless you understand classes.
- Thus, we’ll focus on classes first and defer further discussion of `const`.

55

Representing Sizes

- Early (non-standard) implementations of the C Library stored object sizes in variables of type `int` or `unsigned int`, as in:

```
int strlen(char *s);
```

or maybe:

```
void *malloc(unsigned int n);
```

- C programmers followed this practice.
- In time, this proved to be a source of portability problems...

56

If It's Not One Thing...

- Consider a machine with 16-bit integers and 32-bit pointers:
 - An `unsigned int` can represent an object size up to 64K.
 - A `signed int` can represent only half of that.
 - 32-bit pointers can address much more than 64K bytes, so the largest object size can be much greater than 64K.
- Again, you could declare `malloc` as:

```
void *malloc(unsigned int n);
```

- But then you limit the size of the largest allocation to only 64K.
- To allow larger allocations, you could declare `malloc` as:

```
void *malloc(unsigned Long int n);
```

57

Then It's Another

- Now, consider a machine with 16-bit integers and 16-bit pointers.
 - An `unsigned long int` will occupy two 16-bit integers.
- In this case, this declaration still works:

```
void *malloc(unsigned Long int n);
```

- However, it wastes both speed and space:
 - The size of the largest allocation can't exceed 64K.
 - Thus, the high-order 16 bits of parameter `n` will always be zero.
 - The compiler will generate extra machine instructions to create and move those 16-bits of zeroes.

58

size_t

- Standard C solved this problem by introducing a typedef name called `size_t`.
- Numerous Standard C headers define `size_t`, including:
 - `<stddef.h>`
 - `<stdlib.h>`
 - `<string.h>`

59

size_t

- By definition, `size_t` is the result of the built-in `sizeof` operator.
- Each platform may define `size_t` differently, but...
- It's always an unsigned integer type.
 - Sizes are never negative.
- `size_t` might be unsigned long, or even unsigned long long, but...
 - It's supposed to be no larger than necessary to represent the size of the largest object on the target machine.

60

size_t

- `size_t` appears as parameter and return types in numerous functions, such as:

```
void *malloc(size_t);  
void *memcpy(void *, void const *, size_t);  
size_t strlen(char const *);
```

- ✓ *Use `size_t` as the type of an object, parameter or return type representing an object size in bytes.*

61

[03] Exercise: Using `size_t`

- It's common practice to use `size_t` as the type for array indices as well as for object sizes.
- Why is this practical?
- Rewrite `xr` using `size_t` as appropriate.

62

Mutually Referential Types

- Some applications require *mutually referential structures* — structures that refer to each other.
- For example, suppose a chicken has an egg in it, and vice versa:

```
struct chicken {  
    egg e;           // error: egg undeclared  
};  
  
struct egg {  
    chicken c;  
};
```

63

Mutually Referential Types

- This declaration for chicken is an error:

```
struct chicken {  
    egg e;           // error: egg undeclared  
};
```

because chicken declares a member of type egg before egg has been declared.

- The compiler must know how big an egg is so it can determine how big a chicken is.
- If you place egg's declaration before chicken's declaration, then all you do is move the error from chicken to egg.

64

Mutually Referential Types

- Although you can't declare an entire `egg` inside a `chicken`, you can declare a pointer to an `egg` inside a `chicken`, as in:

```
struct egg;

struct chicken {
    egg *e;        // OK
};

struct egg {
    chicken c;
};
```

65

Incomplete Types

- Here, the declaration:

```
struct egg;    // incomplete type
```

declares `egg` as an *incomplete type*.

- Some programmers call it a *forward-declared type*.

66

Incomplete Types

- When `struct egg` is incomplete, you can't declare objects of type `egg`:

```
egg e;           // no: don't know sizeof(struct egg)
```

- The compiler doesn't know the size or contents of an `egg`.
- However, you can declare objects of type "pointer to `struct egg`".

```
egg *pe;         // OK
```

- The storage allocation for a pointer is fixed for a given platform.

67

Incomplete Types

- You complete an incomplete type with a declaration such as:

```
struct egg {  
    chicken c;  
};           // egg is now complete
```

- Only then can you declare objects of type `egg`.
- Fun fact: `void` is an incomplete type that you can't complete.

68

Self-Referential Structures

- Structures containing pointers to structures of the same type are called *self-referential structures*.
- For example, this structure is self-referential:

```
struct list_node {
    ~~~
    list_node *next;
};
```

- `list_node` contains, among other things, a pointer to another `list_node`.

69

Pointers Conversions and Casts

- Accidental pointer conversions were a common source of errors in early C programs.
- Many C compilers now issue warnings for conversions between incompatible pointer types.
 - But the C Standard doesn't insist.
- C++ treats such conversions as errors.
- For example:

```
gadget *pg;
widget *pw;    // gadget and widget are distinct types
~~~
pg = pw;       // likely warning in C; error in C++
pw = pg;       // likely warning in C; error in C++
```

70

Pointers Conversions and Casts

- You can silence the complaints by using casts:

```
pg = (gadget *)pw; // compiles quietly; still suspect
pw = (widget *)pg; // same here
```

- Nonetheless, such conversions easily lead to undefined behavior:

```
int i;
double *p = (double *)&i;
*p = 3;           // undefined behavior
```

71

Conversions To and From void *

- Some functions, such as Standard C's `malloc` and `free`, were designed to manipulate pointers to objects of any type.
- C and C++ provide `void *` as the *generic data pointer* type for declaring such pointer parameters, as in:

```
void *malloc(size_t n);
void free(void *p);
```

- An object of type `void *` can point to any data object.

72

Conversions To and From void *

- A C program can convert any pointer type to or from void * without a cast:

```
gadget *pg;
widget *pw;
void *pv;
~~~
pv = pg;           // permitted in C
pw = pv;           // permitted in C
```

- Unfortunately, these two assignments convert a “pointer to gadget” into a “pointer to widget” without using a cast.
- Accessing *pw yields undefined behavior.

73

Conversions To and From void *

- C++ attempts to cut down on such mishaps by:
 - allowing conversions to void *, but
 - disallowing conversions from void *.
- Thus, C++ requires a cast where C doesn't:

```
gadget *pg;
widget *pw;
void *pv;
~~~
pv = pg;           // permitted in C and C++
pw = pv;           // permitted in C; error in C++
pw = (widget *)pv; // permitted in C and C++
```

- Here's the rationale...

74

Conversions To and From `void *`

- **Converting from `T *` to `void *` is safe** (for any non-void `T`).
 - The program can't do much with a `void *` object.
 - That is, if `pv` has type `void *`, these expressions are all errors:

`*pv` `pv[i]` `++pv` `--pv`

- **Converting from `void *` to `T *` is not safe.**
 - In effect, the conversion turns an impotent, typeless object into a potent, typed one.
 - It might accidentally convert something that isn't really a `T` into a `T`.

75

Perspectives on Casts

- In C++ as well as in C, a cast-expression of the form `(T)e` converts expression `e` to type `T`.
- The presence of a cast-expression indicates potential danger...
- The program is doing a conversion that's typically unsafe.
- ✓ ***Use casts sparingly.***

76

Perspectives on Casts

- Some casts have portable behavior.
- Others don't.
- For example, the cast-expression below has common behavior across all Standard C++ implementations:

```
double d;  
int i;  
~~~  
i = (int)d;           // portable
```

- It converts the value of `d` to `int` by discarding `d`'s fractional part.

77

Perspectives on Casts

- On the other hand, converting an integer value to a pointer type is not necessarily portable:

```
device *p;  
~~~  
p = (device *)0xFFFFF70; // not strictly portable
```

- This is typically something that programs do when communicating directly with computer hardware.
- How the conversion transforms the integer value into a pointer value is **implementation-defined**.
- That is, it depends on the target platform.

78

New-Style Casts

- C++ provides a new notation that distinguishes portable casts from potentially non-portable ones:

```

                                // behavior:
static_cast<T>(e)              //      standardized
reinterpret_cast<T>(e)        //      implementation-defined

```

- These “new-style” casts don’t provide any additional functionality beyond the “old-style” casts.
- They just offer a chance to write code that reveals its intent more clearly.

79

Dynamic Cast

- The keyword `static_cast` was chosen to contrast it with:

```
dynamic_cast<T>(e)  // requires run-time validation
```

- C++ provides a mechanism called inheritance.
- Inheritance lets you build hierarchies of related types.
- The `dynamic_cast` operator provides run-time validation for conversions between types in an inheritance hierarchy.
- In general, the compiler can’t determine at compile time if a given `dynamic_cast` expression will succeed.
- You won’t know until the cast-expression executes at run time.

80

Static Cast

- For those conversions that can be validated at compile time, C++ provides the `static_cast` operator:

```
static_cast<T>(e)    // compile-time validation
```

- At compile time, the compiler determines if it can convert expression `e` to type `T`.
- When does it perform the conversion?
- If `e` is a constant-expression, the compiler might do the conversion at compile-time...

81

Static Cast

- For example, consider:

```
double d;
int i, j;
~~~
i = static_cast<int>(3.14159); // maybe at compile time
scanf("%f", &d);
j = static_cast<int>(d);      // at run time
```

- The first static cast should be able to convert `3.14159` into `3` at compile time.
- However, the second cast can't convert `d` to `int` until it sees `d`'s value at run time.

82

Reinterpret Cast

- In a sense, a reinterpret cast is also a static cast.
- `reinterpret_cast` provides only compile-time validation for conversions:

```
reinterpret_cast<T>(e) // compile-time validation only
```

- However, `static_cast` and `reinterpret_cast` support a nearly disjoint set of conversions.
- That is, with few exceptions:
 - If you can convert expression `e` to type `T` using a `static_cast`, then you can't use a `reinterpret_cast`.
 - If you can convert expression `e` to type `T` using a `reinterpret_cast`, then you can't use a `static_cast`.

83

New-Style Casts

- For example, these have the same portable behavior:

```
i = static_cast<int>(d);  
i = (int)d;
```

- However, this is an error because conversion from `double` to `int` is portable:

```
i = reinterpret_cast<int>(d);
```

- `reinterpret_cast` is for implementation-defined conversions.

84

New-Style Casts

- Similarly, these have the same implementation-defined behavior:

```
p = reinterpret_cast<T *>(0xFFFFFFFF70);  
p = (T *)0xFFFFFFFF70;
```

- However, this is an error:

```
p = static_cast<T *>(0xFFFFFFFF70);
```

- `static_cast` is for portable conversions.

85

New-Style Casts

- Again, you should avoid using casts.
- ✓ ***If you must use a cast in C++, use a new-style cast.***
- These new-style casts make casts easier to spot in source code...
 - for humans and
 - for search tools such as `grep`.
- This is good, because casts are hazardous.
- A hazard that's easier to spot is usually less of a hazard.

86

Dynamic Allocation in C

- The Standard C `malloc` function allocates dynamic storage:

```
void *malloc(size_t n);
```

- Parameter `n` specifies the number of bytes to allocate.
- In C, you allocate an object of type `T` using:

```
T *p;
~~~
p = malloc(sizeof(T));
```

- This assignment involves a conversion from `void *` to `T *`.
- It's potentially hazardous...

87

Dynamic Allocation in C

- In C, `malloc` allows accidents such as:

```
widget *pw;
~~~
pw = malloc(sizeof(gadget));
```

- If a `gadget` is smaller than a `widget`, then accessing `*pw` yields undefined behavior.
- Again, C++ attempts to avoid such mishaps.
- The assignment won't compile without a cast...

88

New-Expressions

- You can use an old-style cast to get it right

```
pw = (widget *)malloc(sizeof(widget));
```

- A new-style cast is even better:

```
p = static_cast<widget *>(malloc(sizeof(widget)));
```

- A cast is generally an indication of possible danger.
- However, in this instance, the operations are safe.
- However, a safe operation shouldn't require a cast because...

89

Memory Allocation in C

- Using a cast invites accidents, as in:

```
widget *pw;
```

```
~~~
```

```
pw = static_cast<widget *>(malloc(sizeof(gadget)));
```

- Again, if a `gadget` is smaller than a `widget`, the behavior is undefined.
- C++ provides a nice alternative to `malloc`.

90

New-Expressions

- C++ provides an operator, called `new`, that does the pointer conversion automatically.
- A ***new-expression*** has the form:

```
new T
```

- It allocates storage for a T object and returns the address of that storage as a T *.
- This lets you allocate storage without using a cast:

```
T *p = new T;           // Look Ma.  No cast.
```

91

Array-New-Expressions

- In C, you allocate an “array of n elements of T” using a call of the form:

```
malloc(n * sizeof(T))
```

- In C++, you use an ***array-new-expression***:

```
new T [n];
```

- This array-new-expression returns a T *.
- Thus, you can allocate an array without using a cast:

```
T *p = new T [n];      // no cast here either
```

92

Array-New-Expressions

- In C++, as in C, a T * object can point to either:
 - a single T object or
 - the first element in an array of T.
- The expression inside the brackets of an array-new-expression can be any non-negative integer-valued expression.
- It need not be constant.
- For example, this is perfectly acceptable:

```
char *t = new char [strlen(s) + 1];
```

93

[04] Exercise: Compiling in Either C or C++

- Try compiling *xr* in C++.
 - It should fail.
- Add old-style cast expressions as needed so that *xr* compiles as C++.
- Try compiling it as C.
 - It should still succeed.

94

[05] Exercise: Converting to C++

- In *xr*, rewrite each `malloc` call as an appropriate new-expression.
- Hence, the program will compile only as C++, not as C.
- Now that the program is strictly C++, you can also:
 - redefine the symbols `MAX_ENTRIES` and `MAX_TOKEN` as `const` objects instead of as enumeration constants,
 - remove the keyword `struct` everywhere except as the leading keyword in a structure definition, and
 - replace each remaining old-style cast with an appropriate new-style cast.
- Remove include-directives for headers no longer in use.

95

Delete-Expressions

- The Standard C `free` function deallocates (reclaims) storage previously allocated by `malloc`:

```
void free(void *p);
```

- For example:

```
T *p;
~~~
p = malloc(sizeof(T)); // allocate a T object
~~~
free(p);                // deallocate it
```

96

Delete-Expressions

- In C++, you use a ***delete-expression*** to deallocate storage acquired by a new-expression.
- For example:

```
T *p;  
~~~  
p = new T;           // allocate a T object  
~~~  
delete p;            // deallocate it
```

97

Delete-Expressions

- In C++, you use an ***array-delete-expression*** to deallocate storage acquired by a previous array-new-expression.
- For example:

```
T *p;  
~~~  
p = new T [n];       // allocate an array of T  
~~~  
delete [] p;         // deallocate it
```

98

Delete-Expressions

- You don't — in fact you can't — specify the array dimension in an array-delete-expression.

```
delete [n] p;      // error: dimension not allowed
```

- The program somehow remembers the dimension specified in the prior array-new-expression.
- As with C's free function, deleting a null pointer is harmless.
 - You need not test a pointer for non-null before deleting it.

99

Delete-Expressions

- Mixing array operations with non-array operations for the same object produces undefined behavior.
- That is, don't do this:

```
T *p = new T;      // non-array new
~~~
delete [] p;       // no: array delete
```

- Don't do this, either:

```
T *p = new T [n];  // array new
~~~
delete p;          // no: non-array delete
```

100

Delete-Expressions

- Here we have a situation where C++ invites errors that you can't make in C:
 - In C++, it's the programmer's responsibility to use a delete-expression whose form (array or non-array) matches the new-expression used to allocate the storage.
 - In C, you simply call `free(p)` whether `p` points to a single object or an array.
- Isn't C++ supposed to be safer than C?
- Why doesn't C++ avoid these errors by providing only one form for delete-expressions?
 - This design for new- and delete-expressions makes it easier to implement and use efficient memory managers.

101

The Free Store vs. the Heap

- C++ programs should prefer using `new` and `delete` over `malloc` and `free`.
- However, a C++ program that uses `new` and `delete` can include legacy code that uses `malloc` and `free`, but should do so with caution.
- `malloc` and `free` manage memory from a region called the **heap**.
- `new` and `delete` manage memory from a logically separate region called the **free store**.

102

The Free Store vs. the Heap

- In many C++ environments, the default implementations of the free store and the heap are the same.
- In such implementations:
 - new-expressions call `malloc`, and
 - delete-expression call `free`.
- Nonetheless, you should program as if the free store and heap are separate:
 - If you allocate memory using `new`, you should `delete` it rather than `free` it.
 - If you allocate memory using `malloc`, you should `free` it rather than `delete` it.
- That is, put it back from where you got it.

103

Allocation Failures

- Whereas `malloc` returns `NULL` (a null pointer) if it can't allocate the requested memory...
- A new-expression ***throws an exception*** if the allocation request fails.
- For instance, the condition in the following if-statement will never be true:

```
p = new T;  
if (p == NULL) {  
    // do something about an allocation failure  
}
```

104

Allocation Failures

- C++ has a mechanism called *exception handling* by which you can “catch” the exception and handle it.
- By default, a program aborts if it throws an exception and doesn’t catch it.
- Most C++ compilers that support exception handling offer compilation options (switches or pragmas) that let you disable exception handling.
- With most compilers, when exception handling is disabled, a new-expression that fails returns NULL.

105

The Standard Boolean Type

- C90 didn’t support `bool` as a distinct built-in type.
- For many years, C programmers used `int` variables as booleans, interpreting zero as *false* and any non-zero value as *true*.
- For example,

```
int found = 0;      // found is false
~~~
if (!found) {
    ++found;        // found is now true
    ~~~
}
```

106

The Standard Boolean Type

- Conscientious C programmers used to define symbols to represent the boolean type and constants:

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

- This would let them write code such as:

```
bool found = false;  
~~~  
if (!found) {  
    found = true;  
    ~~~  
}
```

107

The Standard Boolean Type

- Unfortunately, different C programmers defined these symbols differently, sometimes in ways that conflicted with each other.
- For example, some programmers use macros instead of an enumeration, as in:

```
#define bool int  
#define false 0  
#define true 1
```

- This has led to name conflicts and other compilation problems.

108

The Standard Boolean Type

- Standard C++ provides a built-in type called `bool` with constants `false` (= 0) and `true` (= 1).
- `bool` is an integer type.
 - A program can convert `bool` values to and from other integer types without using casts.
- Standard C now provides a header named `<stdbool.h>` that defines `bool`, `false`, and `true`.
- In C++, `bool`, `false` and `true` are keywords.
 - You can't use them for any other purpose.
- In C, they're not keywords.

109

[06] Exercise: Using `bool`

- Currently, `get_token` returns an `int`:

```
int get_token(char *t, int n);
```
- That `int` is either zero (meaning *true*) or non-zero (meaning *false*).
- Thus, `get_token`'s return value acts like a `bool`.
- You should declare it as such and make other changes as needed to get the program to compile and execute correctly.

110

Bibliography

- Kernighan and Ritchie [1978]. Brian Kernighan and Dennis Ritchie, *The C Programming Language*. Prentice-Hall.
- Kernighan and Ritchie [1988]. Brian Kernighan and Dennis Ritchie, *The C Programming Language, 2nd ed.* Prentice-Hall.
- Tondo & Gimpel [1989]. Clovis Tondo and Scott Gimpel. *The C Answer Book, 2nd ed.* Prentice-Hall.

[3] Design Principles in Practice

1

So What's Not to Like?

- Let's look more critically at *xr*, the cross-reference generator, from a design and maintenance perspective.
- Here's the current version of the `main` function in C++...

2

xr's Main Function

```
int main() {
    size_t const MAX_TOKEN = 64;
    char token[MAX_TOKEN];
    unsigned ln = 1;
    while (get_token(token, MAX_TOKEN)) {
        if (isalpha(token[0]) || token[0] == '_') {
            add_array(xa, &xm, token, ln);
        } else { // if (token[0] == '\n')
            ++ln;
        }
    }
    sort_array(xa, xm);
    put_array(xa, xm);
    return 0;
}
```

3

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, and
 - determine when to print the cross-reference
- main need not “know” how the cross-reference table is implemented.
- But it does...

4

Evidence of Excess Complexity

- It's evident from reading `main` that the cross-reference table is an unordered array that must be sorted before printing:
 - The cross-reference table is declared as:


```
entry *xa[MAX_ENTRIES];
```
 - Each cross-referencing function has `array` in its name.
 - Each cross-referencing function has a parameter declared with type `entry *`.
 - `main` must call `sort_array` before it calls `put_array`.
- Let's look (again) at `main` in this light.
- Here it is with the evidence of arrays highlighted...

5

xr's Main Function

```
int main() {
    size_t const MAX_TOKEN = 64;
    char token[MAX_TOKEN];
    unsigned ln = 1;
    while (get_token(token, MAX_TOKEN)) {
        if (isalpha(token[0]) || token[0] == '_') {
            add_array(xa, &xm, token, ln);
        } else { // if (token[0] == '\n')
            ++ln;
        }
    }
    sort_array(xa, xm);
    put_array(xa, xm);
    return 0;
}
```

6

Evidence of Excess Complexity

- This evidence clutters `main` with unnecessary details about other parts of the program.
- The clutter make the program less readable.
- It also makes the program harder to maintain.
- Suppose you discover a better implementation for the cross-reference table...

7

Evidence of Excess Complexity

- When the data structure is no longer an array:
 - You must replace the declarations for `MAX_ENTRIES`, `xa` and `xm` with something else.
 - You must rewrite the bodies of `add_array` and `put_array`.
- Such changes are unavoidable.
- It then becomes clear that the function interfaces are inappropriate:
 - The functions shouldn't have `array` in their names.
 - The functions shouldn't have parameters of type `entry *`.
 - Maybe the sort function should be bundled another way.
- `main` is *too tightly coupled* to the cross-reference table implementation.

8

Evidence of Excess Complexity

- **Coupling** is the extent to which one software component depends on another.
- Two components can't communicate unless they are coupled in some way.
- For example:
 - `main` depends on the existence of a function that adds a word and a line number to the cross-reference table.
 - If that function goes away, then you must rewrite `main` to do the job some other way.
- Some minimal coupling is unavoidable.
- However, excess coupling makes for unnecessarily brittle software.

9

Naming Names

- You can create a greater illusion of simplicity by applying abstraction techniques.
- That is, you can reorganize the program into a small collection of simpler, cooperating components.
- A primary goal should be to:
 - ✓ *Separate what a component does from how it's implemented.*
- A good place to start is in your choice of names...

10

Naming Names

- Some entities in a program correspond directly to concepts in the original problem statement.
 - They are part of the ***problem domain***.
- Other entities in the program arise from implementation choices the programmer made.
 - They are part of the ***solution domain***.

11

Naming Names

- For example,
 - One ***problem*** in *xr* is: *xr* needs a cross-reference table.
 - The current ***solution*** is: Use an unsorted array.
- Thus,
 - “Cross-reference table” is in the ***problem*** domain.
 - “Unsorted array” is in the ***solution*** domain.

12

Naming Names

- ✓ *For entities in the problem domain, choose names that convey functionality, not implementation.*
- ✓ *For entities in the solution domain, choose names that clarify the implementation.*
- For example,
 - `main` shouldn't add a word and line number to an array.
 - It should add them to a cross-reference table.
- More specifically...

13

Naming Names

- `main` shouldn't call a function named `add_array`.
- It should call a function named `add_cross_reference` or `cross_reference_add`.
- A long prefix or suffix such as `cross_reference` might prove cumbersome.
- Let's try a shorter one, such as `xrt_`, as in `xrt_add`.

14

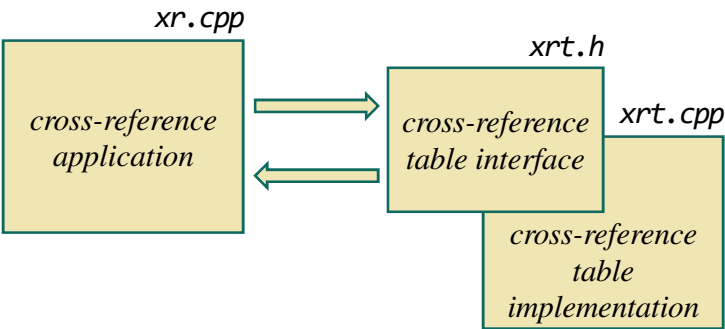
Data Abstraction By Separate Compilation

- Changing the names of the cross-referencing functions is just the first step in turning the table into an abstraction.
- The next step is to render the table’s data declarations invisible to `main`.
- This you can do by moving those data declarations to a separate module.
- As in C, a C++ program can be composed of separately compiled source files.
- Programmers often call these source files *modules*.

15

Data Abstraction By Separate Compilation

- To repack the cross-reference table as a module:
- Define the data and functions that implement the cross-reference table in a separate source file “`xrt.cpp`”.
- Define the interface to “`xrt.cpp`” in an accompanying header file “`xrt.h`”.



16

Data Abstraction By Separate Compilation

- When building the *xr* program:
 - Include “*xrt.h*” in “*xr.cpp*”.
 - This is so that `main` knows how to call the cross-referencing functions.
 - Include “*xrt.h*” in “*xrt.cpp*”.
 - This insures that the interface agrees with the implementation.
 - Link the object files resulting from “*xr.cpp*” and “*xrt.cpp*”.

17

[01] *xr* using Separate Compilation

- The header “*xrt.h*” is very simple:

```
// xrt.h - cross-reference table interface
// using abstraction by separate compilation

void xrt_add(char *w, unsigned n);
void xrt_put();
void xrt_sort();
```

18

xr using Separate Compilation

- This interface improves on the original program in several ways:
 - “*xrt.h*” declares function names that convey functionality instead of implementation:
 - `xrt_add` instead of `add_array`
 - `xrt_put` instead of `put_array`
 - `xrt_sort` instead of `sort_array`
 - No function in “*xrt.h*” has a parameter of type `entry *`.
 - The program completely hides the cross-reference data structures as types and static data inside “*xrt.cpp*”.
- Using separate compilation, the new “*xr.cpp*” contains a lot less code than before...

19

[01] *xr* using Separate Compilation

```
// xr.cpp - a cross-reference generator

#include <ctype.h>
#include <stdio.h>

#include "xrt.h"

// input processing components

bool get_token(char *t, size_t n);
~~~
```

20

[01] *xr* using Separate Compilation

```
int main() {
    size_t const MAX_TOKEN = 64;
    char token[MAX_TOKEN];
    unsigned ln = 1;
    while (get_token(token, MAX_TOKEN)) {
        if (isalpha(token[0]) || token[0] == '_') {
            xrt_add(token, ln);
        } else { // if (token[0] == '\n')
            ++ln;
        }
    }
    xrt_sort();
    xrt_put();
    return 0;
}
~
```

21

[01] *xr* using Separate Compilation

```
bool get_token(char *t, size_t n) {
    // same as before
}
```

- This version of main no longer offers any clues to the structure of the cross-reference table.
- That is, variables *xa* and *xm*, and functions with *_array* in their names are no longer visible.

22

[01] *xr* using Separate Compilation

- The source file “*xrt.cpp*” now contains all the data structures that implement the cross-reference table:

```
// xrt.cpp - cross-reference table implementation

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

#include "xrt.h"
~~~
```

23

[01] *xr* using Separate Compilation

```
struct list_node {
    unsigned number;
    list_node *next;
};

struct entry {
    char *word;
    list_node *first, *last;
};
~~~
```

24

[01] *xr* using Separate Compilation

```
size_t const MAX_ENTRIES = 1024;
```

```
static entry *xa[MAX_ENTRIES];
```

```
static size_t xm = 0;
```

```
~~~~
```

- *xa* and *xm* are the variables that represent the cross-reference data structure.
- “*xrt.cpp*” declares them using the keyword `static`.
- Thus, *xa* and *xm* have *internal linkage*.
- Thus, no code outside “*xrt.cpp*” can write a declaration that refers to either variable directly.

25

xr using Separate Compilation

- Code outside “*xrt.cpp*” can manipulate *xa* and *xm* only by calling the interface functions (those declared in “*xrt.h*”), which have external linkage by default.
- The keyword `static` can also appear in function declarations at global scope.
- As with data, functions declared `static` have internal linkage.
- You need not declare `MAX_ENTRIES` with the keyword `static` because `const` objects declared at global scope have internal linkage by default.
- This differs from C, where `const` objects at global scope have external linkage by default.

26

[01] *xr* using Separate Compilation

- `xrt_add` (formerly `add_array`) no longer has parameters of type `entry *` and `size_t *`.
- It manipulates `xa` and `xm` as global variables:

```
void xrt_add(char *w, unsigned n) {
    size_t i;
    for (i = 0; i < xm; ++i) {
        if (strcmp(xa[i]->word, w) == 0) {
            break;
        }
    }
    // see the next slide for this part
}
```

~~~~~

27

## [01] *xr* using Separate Compilation

- The function body is still very much as it was before:

```
void xrt_add(char *w, unsigned n) {
    // see the previous slide for this part
    if (i >= xm) {
        // didn't find the word
        // add the new word and the line number
        // see the next slide for this part
    } else if (xa[i]->last->number != n) {
        // found the word but not the line number
        // add a new line number for that word
        // see the next, next slide for this part
    }
}
```

28

## [01] *xr* using Separate Compilation

- Here's how `xrt_add` adds a new word `w` and line number `n`:

```
if (i >= xm) {
    assert(i < MAX_ENTRIES);
    ++xm;
    xa[i] = new entry;
    xa[i]->word = new char [strlen(w) + 1];
    strcpy(xa[i]->word, w);
    xa[i]->first = new list_node;
    xa[i]->last = xa[i]->first;
    xa[i]->last->number = n;
    xa[i]->last->next = NULL;
} else if (xa[i]->last->number != n) {
```

29

## [01] *xr* using Separate Compilation

- Here's how `xrt_add` adds a new line number `n` for a word:

```
} else if (xa[i]->last->number != n) {
    xa[i]->last->next = new list_node;
    xa[i]->last = xa[i]->last->next;
    xa[i]->last->number = n;
    xa[i]->last->next = NULL;
}
```

30

## [01] *xr* using Separate Compilation

- `xrt_put` (formerly `put_array`) and `xrt_sort` (formerly `sort_array`) also manipulate `xa` and `xm` as global variables...

```
void xrt_put() {
    size_t i;
    list_node *p;
    for (i = 0; i < xm; ++i) {
        printf("%15s:", xa[i]->word);
        for (p = xa[i]->first; p != NULL; p = p->next) {
            printf(" %7d", p->number);
        }
        putchar('\n');
    }
}
~
```

31

## [01] *xr* using Separate Compilation

```
void xrt_sort() {
    size_t i, j;
    if (xm == 0) {
        return;
    }
    for (i = 0; i < xm - 1; ++i) {
        for (j = i + 1; j < xm; ++j) {
            if (strcmp(xa[i]->word, xa[j]->word) > 0) {
                entry *temp = xa[i];
                xa[i] = xa[j];
                xa[j] = temp;
            }
        }
    }
}
```

32

## Data Abstraction Using Structs

- This separate compilation technique hides implementation details almost completely.
- Unfortunately...
  - The program can have only one instance of the hidden data structure.
  - Modules with static data are not reentrant.
  - With some algorithms, the technique adds an extra layer of function calls, increasing object code and execution time.
- What we want is an abstraction technique that hides the implementation details of the data structure, yet avoids these problems.

33

## Data Abstraction Using Structs

- It turns out that we can get almost everything we want by using structure types as abstract types.
- That is, you can gather the variables that represent a cross-reference into a structure:

```
struct xrt {  
    entry *xa[MAX_ENTRIES];  
    size_t xm;  
};
```

- You can then speak of `xrt` as a type, because it is.

34

## Data Abstraction Using Structs

- When you want another `xrt` object, you simply declare one or allocate one:

```
xrt x;           // declare an xrt
~~~
xrt *p = new xrt; // allocate an xrt
```

- Now, no matter how you change the representation (the data members) of an `xrt`, the source code to create an `xrt` never changes.
- This is good, but it comes at a small price...
- Since there can be more than one `xrt` in a program, you must pass (the address of) an `xrt` object to each `xrt_` function.

35

## [02] Exercise: Abstraction Using Structs

- Rewrite `xr` to allow more than one cross-reference table in an application.
- Represent the cross-reference table type as a structure named `xrt`.
- Pass a pointer to an `xrt` as an argument to each `xrt_` function.
- For example, the new function heading for `xrt_put` should be:

```
void xrt_put(xrt *t)
```

- More...

36

## [02] Exercise: Abstraction Using Structs

- Inside every function with a parameter `xrt *t`, refer to the data in the `xrt` using `t->` to locate the data member(s).
- For example, this statement in `xrt_put`:

```
printf("%15s:", xa[i]->word);
```

becomes this:

```
printf("%15s:", t->xa[i]->word);
```

- More...

37

## [02] Exercise: Abstraction Using Structs

- Test your new `xrt` by modifying `xr` to build two tables:
  - `lower` containing words beginning with a lowercase letter or underscore, and
  - `upper` containing words beginning with an uppercase letter.
- ***Print the lower table first.***
  - When `xr` used only one table, the words beginning with uppercase letters appear before any words beginning with lowercase letters.
    - Such is the nature of ASCII.
  - Printing `lower` first changes what you see.
- More...

38

## [02] Exercise: Abstraction Using Structs

- Move type and constant definitions from headers to source files, and vice versa, as needed.
- Do the best you can to hide the cross-reference table implementation from the main function.
- *Try defining **lower and upper as global objects and then as local objects.***
  - The program should work either way.
- Note:
  - There's no perfect solution to this problem.
  - It's designed in part to expose the limitations of this abstraction technique.

39

## Why Structs Aren't Abstract Enough

- Again, since there can be more than one `xrt` in a program, you must pass (the address of) an `xrt` object to each `xrt_function`.
- For example, `xrt_add` becomes:

```
void xrt_add(xrt *t, char *w, unsigned n) {
    ~~~
}
```

- Consequently, this declaration must appear in "`xrt.h`":

```
// xrt.h - cross-reference table interface
~~~

void xrt_add(xrt *t, char *w, unsigned n);
```

40



## Why Structs Aren't Abstract Enough

- `xrt_add`'s declaration mentions `xrt`, so `xrt`'s definition must appear in the same header:

```
// xrt.h - cross-reference table interface
~~~

struct xrt {
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
~~~

void xrt_add(xrt *t, char *w, unsigned n);
```

41

## Why Structs Aren't Abstract Enough

- `xrt`'s definition mentions `MAX_ENTRIES`, so `MAX_ENTRIES`'s definition must be in the header:

```
// xrt.h - cross-reference table interface
~~~

size_t const MAX_ENTRIES = 1024;

struct xrt {
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
~~~

void xrt_add(xrt *t, char *w, unsigned n);
```

42

## Why Structs Aren't Abstract Enough

- xrt's definition also mentions entry, so entry's definition must be in the header:

```
// xrt.h - cross-reference table interface
~~~
```

```
struct entry {
    char *word;
    list_node *first, *last;
};
```

```
size_t const MAX_ENTRIES = 1024;
~~~
```

- And so on...

43

## Why Structs Aren't Abstract Enough

- In truth, you could avoid dragging all those definitions into the header by declaring xrt as an incomplete type:

```
// xrt.h - cross-reference table interface
~~~
```

```
struct xrt;
~~~
```

```
void xrt_add(xrt *t, char *w, unsigned n);
```

- Unfortunately, this can make the program less efficient.
- We'll explore this option later.

44

## Why Structs Aren't Abstract Enough

- Placing all these definitions in the header “*xrt.h*” compromises the abstraction of the separately compiled module “*xrt.cpp*”.
- Those structures are part of the cross-reference table implementation.
- They are not for public use.
- Yet they reside in a header, which makes them appear to be available as part of the interface.

45

## Why Structs Aren't Abstract Enough

- For example, when the entire *xrt* structure appears in the header, you can initialize *lower* as follows:

```
int main() {
    ~~~
    xrt lower;
    lower.xm = 0;
    ~~~
}
```

- This may be well-intentioned, but it's misguided.
- It explicitly mentions *xm* when we want to hide *xm* from *main*.

46

## Why Structs Aren't Abstract Enough

- In short, encapsulation using structures forces you to place implementation details, not just interface specifications, in header files.
- When this happens, the compiler:
  - can't distinguish interface from implementation and
  - can't diagnose violations of the abstraction.
- Proper encapsulation requires a language feature for restricting access to the implementation details of a widely available type, even if that type appears in a header.
- C++ has that feature, and it's called a ***class***.

47

48

## [4] Classes and Objects

1

### From Structures to Classes

- The most recent version of the “*xrt.h*” header defines *xrt* as a structure.
- The header contains a few other declarations:

```
// xrt.h - cross-reference table interface
```

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

```
struct list_node {
 unsigned number;
 list_node *next;
};
~~~~~
```

2

## From Structures to Classes

```
// xrt.h - cross-reference table interface (continued)

struct entry {
    char *word;
    list_node *first, *last;
};

size_t const MAX_ENTRIES = 1024;
~~~
```

3

## From Structures to Classes

```
// xrt.h - cross-reference table interface (continued)

struct xrt {
 entry *xa[MAX_ENTRIES];
 size_t xm;
};

void xrt_add(xrt *t, char *w, unsigned n);
void xrt_init(xrt *t);
void xrt_put(xrt *t);
void xrt_sort(xrt *t);
~~~
```

4

## From Structures to Classes

- The header contains not only the cross-reference table interface, but also many implementation details.
- The `xrt` structure itself is a weak abstraction.
- It does little to separate the implementation from the interface.
- You can improve the abstraction by transforming `xrt` into a C++ class.

5

## From Structures to Classes

- A C++ class is a C structure, with added capabilities.
- The syntax for C++ classes is an extension of the syntax for C structures.
- You can transform the `xrt` structure into a class step-by-step, as follows...

6

## From Structures to Classes

- Start by changing the `xrt` structure definition to:

```
class xrt {                // was struct
private:                // this is new
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

- The keyword `private` is an *access specifier*.
- It specifies that only class `xrt` itself can access `xa` and `xm`.

7

## From Structures to Classes

- Although class `xrt` has private members, the class itself isn't private.
- You can use the class name to declare variables just as you would use a structure name or any other type name:

```
xrt t;           // t is an xrt object
xrt *p;          // p is a "pointer to xrt" object
~~~
p = new xrt; // OK: allocate an xrt
```

8



## From Structures to Classes

- Now we've entered the world of object-oriented programming.
- So, instead of saying:

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

we now say:

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

- Still, C++ programmers often say "type" instead of "class" when it's obvious from the context that the type is a class type.

9

## From Structures to Classes

- Accessing a data member of a class is just like accessing a member of a structure, as in:

```
xrt t;
xrt *p;
~~~  
t.xm = 0;           // t.xm is the xm member of t  
p->xa[0] = NULL;    // p->xa is the xa member of *p
```

- In this case, xrt members xa and xm are private.
- The compiler rejects expressions that refer to xa or xm appearing outside the class.

10

## From Structures to Classes

- In the *xr* program, all references to *xa* and *xm* appear inside the cross-reference table functions:

```
void xrt_add(xrt *t, char *w, unsigned n);
void xrt_init(xrt *t);
void xrt_put(xrt *t);
void xrt_sort(xrt *t);
```

- However, these functions are declared outside class *xrt*.
- As such, they can't access *xa* and *xm*.
- But these functions can't do what they're supposed to do without access to *xa* and *xm*.

11

## From Structures to Classes

- If you declare the functions as members of class *xrt*, then they can access *xa* and *xm*:

```
class xrt {
    void xrt_add(xrt *t, char *w, unsigned n);
    void xrt_init(xrt *t);
    void xrt_put(xrt *t);
    void xrt_sort(xrt *t);
private:
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

12

## From Structures to Classes

- As you'll see shortly, each xrt member function has an implicitly declared pointer parameter of type xrt \*.
- You can, and should, remove the explicitly declared one:

```
class xrt {
    void xrt_add(xrt *t, char *w, unsigned n);
    void xrt_init(xrt *t);
    void xrt_put(xrt *t);
    void xrt_sort(xrt *t);
private:
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

13

## From Structures to Classes

- C programmers often use short prefixes on function names:
  - to show the logical grouping of functions, and
  - to reduce the chance of name conflicts.
- They rarely use such prefixes when naming structure members, because:
  - structures show grouping, and
  - each structure is effectively a distinct scope.
- Class members are just like structure members in this regard...

14

## From Structures to Classes

- When you move the `xrt_` function declarations into the `xrt` class definition, you can drop the `xrt_` prefix from each member function name:

```
class xrt {
    void xrt_add(char *w, unsigned n);
    void xrt_init();
    void xrt_put();
    void xrt_sort();
private:
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

15

## [01] From Structures to Classes

- Finally, you should specify the member functions as `public` so that code outside the class can call them:

```
class xrt {
    public:
        void add(char *w, unsigned n);
        void init();
        void put();
        void sort();
private:
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

16

## From Structures to Classes

- The `xrt` data members — `xa` and `xm` — define the data representation for the class.
  - They should be private so they're inaccessible to code outside the class.
- The `xrt` functions — `add`, `init`, `put`, and `sort` — define the basic operations on `xrt` objects.
  - They should be members so they can access the private data.
  - They should be public so they're callable from code outside the class.

17

## Class Concepts

- Again, a C++ class is a C structure, and then some.
- In addition to data declarations, a class can contain:
  - function declarations
  - the access specifiers `public` and `private`
- The functions and data declared in a class are its **members**:
  - The **data members** in a class specify the data representation for every object of that class.
  - The **member functions** in a class specify fundamental operations that a program can apply to objects of that class.
- For some reason, an object in a class is a “data member”, but a function in a class is not a “function member”.
  - Go figure.

18

## Access Specifiers

- The access specifiers in a class distinguish the public members from the private members.
- Each access specifier can appear more than once and in any order.
- Each access specifier applies to all the members that follow it, until the next access specifier or the end of the class.

19

## 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 bullet is oversimplified, but sufficient for now.

20

## Access Specifiers

- Member functions can be public or private.
- Data members can be public or private.
- In class `xrt`:
  - All the member functions are public.
  - All the data members are private.
- This is very common in well-written classes.

21

## `sizeof(Class Object)`

- The member function declarations and access specifiers in class definitions add nothing to the size of a class object.

```
class xrt {
  public:                                // adds no storage
    void add(char *w, unsigned n);      // adds no storage
    void init();                        // adds no storage
    void put();                         // adds no storage
    void sort();                       // adds no storage
  private:                             // adds no storage
    entry *xa[MAX_ENTRIES];
    size_t xm;
};
```

22

## sizeof(*Class Object*)

- Thus, sizeof applied to the xrt class yields the same result as sizeof applied to our earlier xrt structure:

```
struct xrt {  
    entry *xa[MAX_ENTRIES];  
    size_t xm;  
};
```

23

## Calling Class Member Functions

- Again, the member functions for a class provide the basic operations that you can apply to objects of that type.
- Each member function call applies to a specific object.
- The C++ community doesn't seem to have a commonly accepted term for that object.
- Some OOP communities use message passing terminology to talk about member function calls.
- Those communities describe applying a member function call to an object as "sending a message to a **receiver** object".
- Other OOP communities simply say that "each member function call applies to a **target** object".
- These notes use the term **target**.

24



## Calling Class Member Functions

- A C++ member function call looks a bit different from an ordinary (non-member) function call.
- For example, the call below applies `xrt`'s `init` member function to target object `t`:

```
xrt t;
~~~
t.init(); // call xrt's init applied to t
```

- However, it's not really a new syntax.
- It's simply a combination of the syntax for selecting class members and the syntax for calling functions...

25

## Calling Class Member Functions

- Here's how the compiler evaluates the call expression:

```
t.init() // evaluate the . before the ()
```

- The `.` (dot) operator and the function call `()` operator have the same operator precedence, but they're left-associative.
  - Both operator group implicitly from left to right.
- The compiler evaluates `t.init` first.
- This expression selects the `init` member from `xrt` object `t`.
- That member is a function.
- What can you do with a function?
- Call it...

26

## Calling Class Member Functions

- xrt's `init` member is a function with an empty parameter list:

```
void init();
```

- The only way you can call this function is with an empty argument list:

```
t.init()
```

- This is where we started.
- Again, the call means "call xrt's `init` member function applied to target `t`".

27

## Calling Class Member Functions

- A call can specify the target object through a pointer.
- The call below applies xrt's `add` member function to target `*p` passing `w` and `n` as additional arguments:

```
xrt *p;
```

```
~~~
```

```
p->add(w, n);    // call xrt's add applied to *p
```

- Let's look at this more closely...

28

## Calling Class Member Functions

- In `x.m`:
  - `x` must be an object of structure or class type.
  - `m` must be a member of that type.
- In `p->m`:
  - `p` must be a pointer to a structure or class type.
  - Again, `m` must be a member of that type.
- `p->m` is equivalent to `(*p).m`.
- All this applies whether `m` is a data member or member function.
- Thus, these are equivalent:

```
p->add(w, n);    // call xrt's add applied to *p
(*p).add(w, n); // ditto
```

29

## [01] Using a Class

- Here now, is "`xr.cpp`" using the `xrt` class:

```
// xr.cpp - a cross-reference generator
~~~

#include "xrt.h"
~~~

int main() {
    size_t const MAX_TOKEN = 64;
    char token[MAX_TOKEN];
    unsigned ln = 1;
    // see the next slide for this part
    return 0;
}
```

30

```

xrt lower, upper;
lower.init();
upper.init();
while (get_token(token, MAX_TOKEN)) {
    if (islower(token[0]) || token[0] == '_') {
        lower.add(token, ln);
    } else if (isupper(token[0])) {
        upper.add(token, ln);
    } else { // if (token[0] == '\n')
        ++ln;
    }
}
lower.sort();
upper.sort();
lower.put();
upper.put();

```

31

## Using a Class

- This applies xrt's init member function to target object lower:

```
lower.init();
```

- Conceptually, it initializes lower.
- This applies xrt's add member function to target object upper, passing additional arguments token and ln:

```
upper.add(token, ln);
```

- Conceptually, it adds token and ln to upper.

32

## Using a Class

- The program remains organized as before, in two source (.cpp) files and one header (.h) file.
- However, in this version:
  - “*xrt.h*” contains a class definition.
  - “*xrt.cpp*” contains the corresponding class member function definitions.

33

## Declarations vs. Definitions

- In C++, as in C:
  - A **declaration** introduces a name into a scope, and specifies attributes for that name.
  - A **definition** is a declaration for an object or function that also reserves storage, or a declaration for a complete type.
- For example,

```
void xrt_put(xrt *t) { // definition
    ~~~
}
```

```
void xrt_put(xrt *t); // non-defining declaration
```

34

## Declarations vs. Definitions

- This is a class **declaration**:

```
class xrt; // non-defining declaration
```

- It declares xrt as an incomplete type.

35

## Declarations vs. Definitions

- This class **definition** contains member function **declarations**:

```
class xrt { // definition
public:
 void add(char *w, unsigned n); // declaration
 void init(); // declaration
 void put(); // declaration
 void sort(); // declaration
private:
 entry *xa[MAX_ENTRIES];
 size_t xm;
};
```

36

## Declarations vs. Definitions

- You typically place class definitions in header files.
- As with any type definition, a class definition doesn't occupy run-time storage.
  - It describes the size, alignment and layout for objects of that type.
- A class definition contains member function declarations.
- The member function definitions (the function bodies) typically appear elsewhere, in a separate source file.
- The definition of a class object allocates storage for the class data members, but doesn't generate new copies of the member functions.

37

## The Scope Resolution Operator

- The same member name can appear in more than one class:

```
class gadget {
public:
 int foo(int);
    ~~~
};

class widget {
public:
    char *foo(char *); // same name; different class
    ~~~
};
```

38

## The Scope Resolution Operator

- Each member function definition must refer to the function by its qualified name.
- A class member's **qualified name** has the form:

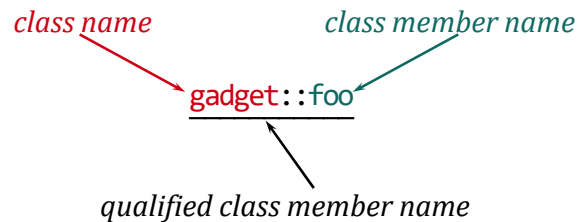
*class-name :: member-name*

- For example:
  - The qualified name for gadget's foo function is ***gadget::foo***.
  - The qualified name for widget's foo function is ***widget::foo***.

39

## The Scope Resolution Operator

- ***gadget::foo*** is not a single identifier.
- It's actually an expression that selects one member from a collection of members, much as `x.m` and `p->m` do.



- Formally, `::` is the **scope resolution operator**.
- Most people call it “colon colon”.

40



## [01] Class Member Function Definitions

- The xrt member function definitions are in “xrt.cpp”:

```
// xrt.cpp - cross-reference table implementation

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

#include "xrt.h"
~~~

void xrt::init() {      // qualified name
    xm = 0;
}
~~~
```

41

## Class Member Function Definitions

- Inside xrt::init, this statement assigns 0 to the xm member of the target xrt object:

```
xm = 0; // assign 0 to target's xm
```

- Thus, for example:

```
lower.init(); // stores 0 into lower.xm
upper.init(); // stores 0 into upper.xm
```

- The remaining xrt functions (in “xrt.cpp”) are the same as they were in the first version of xr that used separate compilation, except for the function headings...

42

## [01] Class Member Function Definitions

```
void xrt::add(char *w, unsigned n) { // qualified name
 size_t i;
 for (i = 0; i < xm; ++i)
 if (strcmp(xa[i]->word, w) == 0)
 break;
 if (i >= xm) {
 // didn't find word
 // add a new word and line number
 } else if (xa[i]->last->number != n) {
 // found word but not line number
 // add a new line number for that word
 }
}
~
```

43

## [01] Class Member Function Definitions

```
void xrt::put() { // qualified name
 ~
}

void xrt::sort() { // qualified name
 ~
}
```

- The compiler understands that xa and xm in each of these functions are members of the target xrt object.

44

## Implementing Class Member Functions

- How do class member function calls work?
- Here's a simple conceptual model...
- Each member function of class T is equivalent to a non-member function with an additional parameter of type T \*.
- That non-member function uses the parameter to locate the target object in each call.
- A member function call is equivalent to a non-member function call that passes the address of the target object as the additional argument.

45

## Implementing Class Member Functions

- Here, again, is the definition for `xrt::init`:

```
void xrt::init() {
 xm = 0;
}
```

- C++ translates it into something equivalent to a C function written as:

```
void xrt__init(xrt *this) {
 this->xm = 0;
}
```

46

## Implementing Class Member Functions

- C compilers and many linkers can't cope with symbols such as `xrt::init`.
- C++ compilers typically transform source names with `::` in them into something that C compilers and linkers can handle.
- `xrt__init` represents such a C- and linker-friendly name.

47

## Implementing Class Member Functions

- In each call to the C function `xrt__init`, the parameter, `this`, points to the target object.
- Here, again, is a call to `xrt::init`:

```
lower.init();
```

- C++ translates it into something akin to:

```
xrt__init(&lower);
```

- In other words, a member function call typically passes the target object by address.

48

## C++ Notation

- The `::` (scope resolution operator), `.` (dot) and `->` (arrow) are closely related, but different.
- `S::m` refers to member `m` in scope `S`.
  - `S` must be a class type (not an object).
- `x.m` refers to member `m` of object `x`.
  - `x` must be an object of a class or structure type.
- `p->m` refers to member `m` of the object addressed by pointer `p`.
  - `p` must be a pointer to an object of a class or structure type.

49

## C++ Notation

- For a class member function `T::f`,
  - A call such as `x.f(a)` compiles as a non-member function call of the form `T__f(&x, a)`.
  - A call such as `p->f(a)` compiles as a non-member function call of the form `T__f(p, a)`.
- That is, both `x.f(a)` and `p->f(a)` call the same function.
- In both cases, the call passes the address of the target object as a hidden argument.
- The only difference is how the call computes the target's address.

50

## The Keyword `this`

- Every class member function actually has one more parameter than it's declared to have.
  - Except for static members, discussed later.
- Every class member function uses the same name for that implicit parameter: the keyword *this*.
- In a member function of class T, `this` has type `T *`, and its value is the address of the target object.

51

## The Keyword `this`

- Every class member function call must have a target object it can access via `this`.
- Thus, the function name in a class member function call must appear either:
  - to the right of either `x.` or `p->` (for some object `x` or pointer `p`), or
  - in the body of a member of the same class...

52

## The Keyword `this`

- A member function `f` can call another member `g` of the same class `T`.
- The call need not mention `this` explicitly:

```
void T::f() {
 g(); // looks like a non-member call, but isn't
}
```

- In this case, the compiler interprets the call `g()` as `this->g()`.
- This example is a special case of the lookup rules that C++ compilers use to decipher names...

53

## Scope Regions and Name Lookup

- When the compiler encounters the declaration of a name, it stores that name and its attributes in a symbol table.
- When the compiler encounters a reference to a name, it looks up the name in the symbol table to find those attributes.
- In C++, declarations can appear at:
  - **local scope**: local to a function definition (including that function's parameter list) or a block nested therein
  - **class scope**: in the brace-enclosed body of a class definition, or in the parameter list or body of a class member function definition.
  - **global scope**: outside of any function, class, structure, or union.

54

## Qualified vs. Unqualified Names

- A name appearing just to the right of a `::`, `.` or `->` is a **qualified name**.
- A name that's not qualified is an **unqualified name**.
- For example:

```
xrt::add // xrt is unqualified
 // add is qualified
lower.init() // lower is unqualified
 // init is qualified
xa[i]->word // xa is unqualified
 // i is unqualified
 // word is qualified
```

- C++ looks up qualified names differently from unqualified names.

55

## Qualified Name Lookup

- Lookup for qualified names is fairly simple:
  - For `T::n`, where `T` is a class type, or
  - for `x.n`, where `x` is a `T` object, or
  - for `p->n`, where `p` is a pointer to `T`:
    - Look for `n` in `T`.

56



## Unqualified Name Lookup

- Name lookup for unqualified names in C++ is an extension of name lookup in C.
- In this example, `m` and `n` are unqualified names appearing in a function at global scope:

```
int m;
void foo(int n) {
 m = n; // m and n are unqualified names
}
```

- For names `m` and `n` appearing in the body of global function `f`, name lookup is the same as in C...

57

## Unqualified Name Lookup

- For an unqualified name `n` appearing in a function `f` at global scope:
  - ***Look in the local scope.***
    - That is, look for `n` in the scope of `f`.
  - ***Look in the global scope.***
    - That is, look for `n` in the scope enclosing `f`.
  - ***Stop as soon as you find any declaration for `n`.***
- Failure to find the name is an error.

58

## Unqualified Name Lookup

- In this example, `m` and `n` are unqualified names appearing in a function that's a member of a class:

```
int m;
class T {
 void f(int n);
};
void T::f(int n) {
 m = n; // m and n are unqualified names
}
```

- Classes add one more place to look...

59

## Unqualified Name Lookup

- For an unqualified name `n` appearing in a function `f` that's a member of class `T`:
  - ***Look in `f`'s local scope.***
  - ***Look in the class scope.***
    - That is, look for `n` in the scope of `T`.
    - If the compiler finds `n` as a class member, it translates the reference to `n` into `this->n`.
  - ***Look in the global scope.***
  - ***Again, stop as soon as you find any declaration for `n`.***

60

## Name Lookup

- The compiler uses the first matching declaration that it finds.
- Thus, a local declaration in a member function with the same name as another class member isn't an error.
  - But it's not good practice.
- The local declaration simply masks the class member declaration.
- For example...

61

## Name Lookup

```
void g(int);

class T {
public:
 int f(int);
 void g(int);
 int m;
};

int T::f(int m) {
 g(m); // short for this->g(m)
 // call T::g passing f's parameter m
 return m; // return f's parameter m
}
```

62

## Name Lookup

- In the previous example, both references to `m` inside `T::f` refer to `T::f`'s parameter `m`.
- If you want `T::f` to return the class member `T::m` instead, you must write either:

```
return T::m; // return member m
return this->m; // return member m
```

- Again, this isn't good practice.
- `T::f` should use a different name for its parameter.

63

## The Unary `::` Operator

- Sometimes, a member function for a class must call a global function with the same name as a class member.
- For example, suppose you have a legacy C library with functions such as:

```
int open(int fd); // open a file
int close(int fd); // close a file
```

64

## The Unary :: Operator

- Suppose also that you want to implement a database class that uses these functions to access files:

```
class database {
public:
 int open();
 int close();
    ~~~
};
```

- How can `database::open` call the global function named `open`?

65

## The Unary :: Operator

- This produces a compile error:

```
int database::open() {
    open(f);        // compile error
    ~~~
}
```

- Inside the body of `database::open`, the compiler looks in class scope before it looks in the global scope.
- It finds `open` as a member of class `database`.
- The call is an error because member `open` accepts no arguments.

66

## The Unary :: Operator

- Inside `database::open`, the call `open(f)` is an error even though there's a global `open` that will accept the given argument.
- The compiler does name lookup before it does any other semantic analysis.
- If the compiler finds a name that's unusable in the current context, the compiler reports the error.
- The compiler doesn't resume name lookup with the hope of finding a usable name.
- Other languages may resume lookup, but C++ doesn't.

67

## The Unary :: Operator

- One way around this is to define the member function as:

```
int database::open() {
 extern int open(int); // refers to global open
 open(f); // calls global open
    ~~~
}
```

- In this case, name lookup finds the local declaration for `open`.
- In C++ as in C, a function declaration at local scope always refers to a function declared in an enclosing scope.
- This call to `open` calls the global `open`.

68

## The Unary :: Operator

- A simpler and cleaner way to refer directly to a name at global scope is to use :: as a unary prefix operator.
- When a compiler sees a name n appearing immediately after a unary ::, it looks for n only in global scope.
- For example,

```
int database::open() {  
    ::open(f);      // calls global open  
}
```

69

## [02] Exercise: Improving Abstraction

- Modify the xrt class so that cross-reference tables always appear sorted to users.
- That is, users should be able to print a table without first calling `xrt::sort()`, yet the table will come out sorted anyway.
- Hint:
  - Don't change the data structures or algorithms.
  - Declare `sort()` as a private member and the rest will follow.

70

## Separating Design Decisions

- Using a class isolates the cross-reference table data structure from the input processing.
- However, most of the program's data structures remain tightly coupled.
- The `xrt` class — more specifically, the `entry` structure — embodies at least three logically distinct design decisions...

71

## Separating Design Decisions

- The design decisions are:
  1. Each word and its associated line number set is stored in a dynamically allocated entry structure.
    - Those structures are collected in an unordered array of `entry *`, which must be sorted before printing.
  2. Each word is stored as an NTCS (a null-terminated character sequence) in a dynamically allocated array of `char`.
  3. Each line number set is stored as a singly linked, null-terminated list of `list_node` structures, with a pointer to the first `list_node` in the list, and a pointer to the last.

72



## Separating Design Decisions

- It's not hard to imagine changing one or more of these decisions over the life of the program.
  - You might decide to use a circular list for the line numbers and eliminate one of the pointers.
  - You might decide to collect the entries in a binary tree or hash table instead of an unordered array.
- Unfortunately, the code that implements each of these decisions runs together.
- The blurring of design decisions is evident in the following code in `xrt::add`, which creates a new entry for word `w` appearing on line number `n`...

73

## Separating Design Decisions

```

if (i >= xm) {
    assert(i < MAX_ENTRIES);
    ++xm;
    xa[i] = new entry;
    xa[i]->word = new char [strlen(w) + 1];
    strcpy(xa[i]->word, w);
    xa[i]->first = new list_node;
    xa[i]->last = xa[i]->first;
    xa[i]->last->number = n;
    xa[i]->last->next = NULL;
}
  
```

The diagram illustrates three distinct design decisions in the code block above, each highlighted in a different color and pointed to by a numbered circle:

- 1 (Yellow box):** `assert(i < MAX_ENTRIES);` and `++xm;` (Resource management and safety checks).
- 2 (Cyan box):** `xa[i] = new entry;` and `xa[i]->word = new char [strlen(w) + 1]; strcpy(xa[i]->word, w);` (Memory allocation for the word and copying).
- 3 (Pink box):** `xa[i]->first = new list_node;` and `xa[i]->last = xa[i]->first; xa[i]->last->number = n; xa[i]->last->next = NULL;` (List structure management and linking).

74

## Separating Design Decisions

- Why is this a problem?
  - If you change the implementation for one design decision, you risk changing the code for another.
  - Who knows what you might break?
- It should be that:
  - to change the implementation for one decision, you need look only at the code for that decision.
- The code for each design decision should be clearly separate from the code for all others.

75

## Separating Design Decisions

- How do we decouple these design decisions?
- Even before object-oriented techniques were in vogue, Parnas [1972] suggested a simple guideline...
  - ✓ *Each encapsulation unit should hide one design decision.*
- In C++, the encapsulation unit is a class.
  - ✓ *Hide each design decision in a separate class.*
- We've already encapsulated the first decision (the unordered array) as class `xrt`.
- We should repackage each of the other two decisions — the representations of words and line number sets — as separate classes.

76

## Separating Design Decisions

- Physically, an entry has three data members:

```
struct entry {  
    char *word;           // 1  
    list_node *first, *last; // 2 and 3  
};
```

- But the first is merely the low-level implementation for a ***variable-length character string***:

```
char *word;
```

77

## Separating Design Decisions

- We could use `string` as a class representing such variable-length strings.
- Then we could declare the data member as:

```
string word;
```

- This is more abstract.
- There's little need to invent a string class because the C++ Standard Library provides one.

78

## Separating Design Decisions

- Similarly, the second and third members are merely the low-level implementation for a *line number set*:

```
list_node *first, *last;
```

- We could invent *lns* as a class representing such sets.
- Then we could replace both data members with:

```
lns lines;
```

- This is also more abstract.

79

## Separating Design Decisions

- Applying these abstractions to entry we get:

```
struct entry {
    string word;
    lns lines;
};
```

- This declaration matches the original design quite well.
- It says “an entry (in the cross-reference table) is a string and a corresponding line number set”.

80

### [03] Exercise: Separating Design Decisions

- Design and implement class `lns` for line number sets.
- Use it to define entry as:

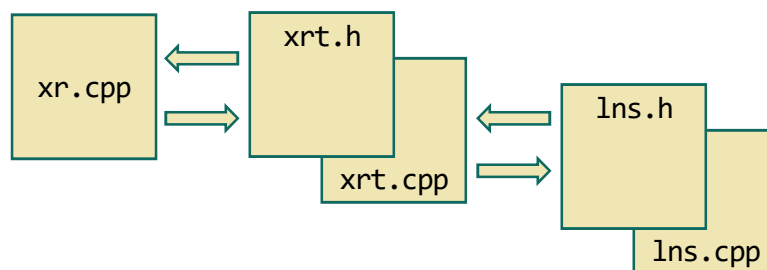
```
struct entry {
    char *word;
    lns lines;
};
```

- Don't try to use a string class yet.
- More...

81

### [03] Exercise: Separating Design Decisions

- Implement the `lns` class in its own header and source file so that the `xr` program will be organized as:



- Hide the `lns` implementation as well as you can from the `xrt` class.
- More...

82

## [03] Exercise: Separating Design Decisions

- Hint:
  - For the most part, you can implement the `Ins` class by moving code from `"xrt.h"` and `"xrt.cpp"` to `"Ins.h"` and `"Ins.cpp"`, respectively.
  - In fact, if you find yourself writing much new code other than declarations, you're probably doing too much work.

83

## Classes and Headers

- As in C, C++ doesn't have any rules about what must appear in headers and what must appear in source files.
- C++ simply requires that:
  - You must declare a name before you use it.
  - You must define certain names before you use them in certain ways.

84

## Classes and Headers

- Placing code in headers is simply a convenient way to package declarations used in more than one source file.
- To reduce coupling between a class and its clients (code that uses the class), you typically implement each class in a separate source file.
- In some cases, you may even split the class member definitions among several source files.
- All of these separate source files need to work from a common class definition.

85

## Classes and Headers

- ✓ *As a general rule, define classes in headers.*
- A class definition is effectively a contract between a producer and consumers.
- Using a common header insures that all source files are working from the same “contract”.
- Exception:
  - When you use a class only in a single source file, you need not define the class in a header.
  - You can define it in the lone source file that uses it.

86

## Classes and Headers

- One class can depend on another.
- The classes may be in separate headers.
- Thus, one header may depend on another.
- For example:
  - Class `xrt` refers to structure `entry` which, in turn, refers to class `Ins`.
  - The definitions for `xrt` and `entry` appear in `"xrt.h"`, but the definition for `Ins` appears in `"Ins.h"`.
  - Thus, `"xrt.h"` depends on `"Ins.h"`.
- How should you organize the program to satisfy such dependencies?

87

## Classes and Headers

- One way to satisfy the dependency is to require code that uses `"xrt.h"` to include `"Ins.h"` first, as in:

```
// xr.cpp - a cross-referencer
~~~
#include "Ins.h"
#include "xrt.h"
```

- Unfortunately, this makes `xrt` users aware that `xrt` is implemented using `Ins`.
- This is unnecessary coupling.
- `xrt` users shouldn't know that `xrt` employs `Ins`...

88



## Classes and Headers

- ✓ *Don't write headers that depend on other headers being included first.*
- A better approach is to include *"lns.h"* in any header or source file that actually needs it.
- In particular, include it in *"xrt.h"*:

```
// xrt.h - cross-reference table interface
~~~  
#include "lns.h"
```

89

## Classes and Headers

- What happens if an *xrt* user actually needs *lns* as well?
- Since *xrt* users don't know (or at least should act as if they don't know) that *"xrt.h"* includes *"lns.h"*, code that needs both headers should include both headers directly.

```
#include "lns.h"  
#include "xrt.h"    // may include lns.h again
```

90

## Classes and Headers

- As in C, C++ doesn't allow definitions to appear more than once in the same scope.
- Thus, including "lms.h" a second time might cause compilation errors.
- ✓ *Design headers so that users can include just the ones they need, in any order that's convenient for them.*
- You can prevent a header from being included more than once by using an #if-#endif wrapper around the header...

91

## Classes and Headers

- For example,

```
// lms.h - line number set interface
```

```
#ifndef LMS_H_INCLUDED
```

```
#define LMS_H_INCLUDED
```

```
~~~ contents of lms.h ~~~
```

```
#endif
```

- Many programmers call this kind of #if-#endif wrapper an ***include guard***.

92

## An Object-Speak Lexicon

- A **class** is a user-defined data type.
- An **object** is an instance of a class.
- Data and functions declared within a class are called **members** of that class.
- Data members specify the **state** of an object.
- Some languages refer to member functions as **methods**.
- Public members specify the permissible **operations** on an object.
- Classes provide **data abstraction** (or **encapsulation**):
  - by gathering the state of an object and the operations on that object into a single unit, and
  - by using **private access** to deny outsiders direct access to that state.

93

## Classes vs. Structures

- The default access specifier for classes is **private**.
- For example:

```
class widget {
 int i; // i is private by default
public:
 int f(); // f is explicitly public
};
```

94

## Classes vs. Structs

- In C++, a structure is merely a class whose default access specifier is `public`.
- In C++, but not in C, a structure can have member functions and access specifiers.
- For example, these are equivalent:

```
struct X {
 int i;
private:
 int f();
};
```

```
class X {
public:
 int i;
private:
 int f();
};
```

```
class X {
 int f();
public:
 int i;
};
```

95

## Bibliography

- Parnas [1972]. David Parnas. "On the Criteria for Decomposing Programs into Modules", *The Communications of the ACM*, December 1972.

96

## [5] Language Features for Libraries

1

### Saying “Hello”

- Here, again, the classic “Hello, world” program in Standard C:

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

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

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

- The C++ Standard Library provides an alternate input/output library in a header called `<iostream>`...

2

## [01] 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.

3

## What’s 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");
```

4

## Why Use a Different I/O Library?

- `<iostream>` is an alternative, not a replacement for `<stdio.h>`.
- C functions such as `printf` and `scanf` are still available in C++.
- 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?

5

## Why Use a Different I/O Library?

- 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.

6

## 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, suppose you're writing an application that processes financial transactions.
- The program must handle data representing currency, calendar dates, clock times, and so on.
- In C++, you'd implement them as classes:

```
class clock_time {
    ~~~
};
~~~
clock_time t;
```

7

## User-Defined Types

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

```
printf("The value is %d", i);
```

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

```
printf("The time is %t", t);
```

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

8



## 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.
  - At least, you can't do it easily.
- What would you do?
- Probably add a member function to the class.
- But then printing a `clock_time` doesn't look at all like other output operations:

```
printf("The time is ");
t.print();
putchar('\n');
```

9

## User-Defined Types

- C++ lets you define new types that can look 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`.
- In particular, you can define a function named `operator<<` so you can display a `clock_time t` using:

```
std::cout << "The time is "; // print an NTCS
std::cout << t; // print a clock_time
```

- C++ lets you use the same notation for operands of different types.
- You'll see the mechanics later.

10

## Why Use a Different I/O Library?

- In short, the `<iostream>` components let you use a uniform notation for input and output operations on user-defined types as well as built-in types.
- The `<stdio.h>` components don't.

11

## A Little Bit of History

- As in C, C++ programs access library facilities by including headers.
- Every C Standard Library header name ends in `“.h”`.
- For compatibility, the C headers have the same names in the C++ Standard Library.
- The *Annotated C++ Reference Manual* (Ellis and Stroustrup [1990]) hardly mentioned any library headers.
- When it did, it also used header names with `“.h”` suffixes, such as `<iostream.h>` and `<new.h>`.
- Not all compiler vendors took the hint.
- Some early C++ compilers had headers with names ending in `“.hpp”` or `“.hxx”`.

12

## A Little Bit of History

- The C++ Standard established a new convention for header names.
- The C++ Standard Library headers, other than those from the C Standard Library, now have names with no suffix at all.
- For example, the header that was once *<iostream.h>* is now *<iostream>*.
- For each C header, C++ has an additional header whose name begins with “*c*” and has no “.*h*” suffix.
- For example, *<cstring>* effectively replaces *<string.h>*.
- The headers whose names end in “.*h*” have been **deprecated**.
  - They may disappear from the C++ Standard in the future.

13

## A Little Bit of History

- In most C++ implementations, each header name maps into a file containing declarations.
- However, the spelling of the file name, and whether such a file even exists, depends on the implementation.
- If *<iostream>* maps into a file, the file name could be:
  - *iostream*,
  - *iostream.h*,
  - *iostream.hpp*,
  - *iostream.hxx*,
  - or some other variation on the name *iostream*.

14

## [01] “Hello, world” (Again)

- Here (again) is the “Hello, world” program:

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

#include <iostream>

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

15

## Namespaces

- The C++ “Hello, world” program writes “Hello, world\n” to the standard output stream `std::cout` using:

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

- For years, the output stream name was simply `cout`.
- However, C++ now has a facility called ***namespaces***.
- The Standard Library places nearly all components, such as `cout`, inside a namespace called `std`.
- Names declared in a namespace are ***members*** of that namespace.
- In this example, `cout` is a member of `std`.

16

## Namespaces

- As it does with classes...
- C++ uses the `::` (scope resolution operator) with namespaces to form qualified names for namespace members.
- For example, `std::cout` is the qualified name for member `cout` in namespace `std`.
- Unlike classes...
  - Namespaces aren't types.
  - There are no namespace "objects".
  - You never use the `.` (dot) or `->` (arrow) operators with namespace names.

17

## Namespaces

- Namespaces help avoid naming conflicts that can arise in C++ programs that include headers from various sources.
- Headers from different sources may inadvertently use the same name for different purposes...

18

## Namespaces

- For example, one header might define status as:

```
// gadget.h - from ABC Corp.
~~~  
enum status { busy, idle, error };  
~~~
```

- Another header might define it as:

```
// widget.h - from XYZ Ltd.
~~~  
typedef int status;  
~~~
```

19

## Namespaces

- Suppose a translation unit includes both of these headers:

```
#include "gadget.h"
#include "widget.h" // error: 'status'
 // declared twice
```

- The declarations for status conflict with each other, producing a compile-time error.

20

## Namespace Members

- A header can place the names it declares inside a namespace:

```
// gadget.h - from ABC Corp.

namespace ABC {
    ~~~
    enum status { busy, idle, error };
    ~~~
}
```

- Names status, busy, idle and error are members of namespace ABC.

21

## Namespace Members

- Namespace members are not global names:

```
#include "gadget.h"
~~~
status s = idle;    // error: 'status' and
                   // 'idle' undeclared
```

- Code appearing outside the namespace must refer to namespace members by their qualified names, as in:

```
#include "gadget.h"
~~~
ABC::status s = ABC::idle; // OK
```

22

## Namespace Members

- The status declared in “*widget.h*” is still global.
- The status declared in “*gadget.h*” is now a namespace member.
- Thus, you can include both headers in the same translation unit:

```
#include "gadget.h" // declares 'ABC::status'
#include "widget.h" // declares global 'status'
```

- The two declarations for status no longer conflict.

23

## Namespace Members

- Of course, “*widget.h*” should place the names it declares in a different namespace, say XYZ, as in:

```
// widget.h – from XYZ Ltd.
```

```
namespace XYZ {
    ~~~  
    typedef int status;  
    ~~~  
}
```

24



## Namespace Scope

- Each namespace defines a new scope region.
- Names declared in a namespace have ***namespace scope***.
- In most ways, the global scope in C++ is just another namespace scope.
- In fact, the C++ Standard says that global names have ***global namespace scope***.
- When the Standard speaks of names declared at namespace scope, it means global names as well.

25

## Namespace Scope

- The global namespace is “just another namespace”.
- That is, the rules for declarations in the global namespace are the same for all namespaces.
- For example, the linkage and storage allocation of a name is the same whether its declaration is global or in a namespace:

```
int n; // as if declared "extern"
int const level = 42; // as if declared "static"

namespace demo {
 int n; // as if declared "extern"
 int const level = 42; // as if declared "static"
}
```

26

## Namespace Nesting

- A namespace definition is itself a declaration.
- Thus, you can define one namespace within another, as in:

```
namespace N { // fully-qualified names:
 namespace S { // N::S
 int i = 0; // N::S::i
 }
 void f() { // N::f
        ~~~
    }
}
```

- However, you can't define a namespace within a function or a class.

27

## Namespace Nesting

- Here, namespace S is a member of namespace N.
- Code inside N can refer to S as just S.
- Code outside N must refer to S as N::S.

```
namespace N {
    namespace S {
        int i = 0;
    }
    void f() {
        S::i = 1;
    }
}
int j = N::S::i;
```

The diagram illustrates the scope resolution for the nested namespace example. It features two annotations with arrows pointing to specific parts of the code:

- A teal arrow points from the text *these are all the same S* to the `S` in `namespace S`, the `S` in `S::i`, and the `S` in `N::S::i`.
- A red arrow points from the text *these are all the same i* to the `i` in `int i = 0`, the `i` in `S::i`, and the `i` in `N::S::i`.

28

## Namespace Nesting

- Namespaces were designed to encourage library vendors to wrap their libraries inside namespaces.
- A vendor who sells more than one distinct library can:
  - place each library in its own namespace, and
  - place each namespace within the vendor's entire namespace.

29

## Namespaces and Name Lookup

- The name lookup rules presented earlier didn't account for namespaces.
- Here are more accurate rules, with changes shown in *red italics*...

30

## Namespaces and Name Lookup

- In C++, declarations can appear at:
  - **block scope**: local to a function definition (including that function's parameter list) or a block nested therein
  - **class scope**: in the brace-enclosed body of a class definition, or in the parameter list or body of a class member function definition.
  - **namespace scope**: outside of any function, class, structure, or union, *whether global or in some other namespace*.

31

## Qualified vs. Unqualified Names (Again)

- A name appearing just to the right of a `::`, `.` or `->` is a **qualified name**.
- A name that's not qualified is an **unqualified name**.
- For example:

```
xrt::add      // xrt   is unqualified
              // add   is qualified
lower.init()  // lower is unqualified
              // init  is qualified
xa[i]->word   // xa    is unqualified
              // i     is unqualified
              // word  is qualified
```

- C++ looks up qualified names differently from unqualified names.

32

## Qualified Name Lookup

- Lookup for qualified names is fairly simple:
  - *For  $S::n$ , where  $S$  is a namespace:*
    - *Look for  $n$  in  $S$ .*
  - For  $T::n$ , where  $T$  is a class type, or
  - for  $x.n$ , where  $x$  is a  $T$  object, or
  - for  $p \rightarrow n$ , where  $p$  is a pointer to  $T$ :
    - Look for  $n$  in  $T$ .

33

## Unqualified Name Lookup

- Name lookup for unqualified names in C++ is an extension of name lookup in C.
- In this example,  $m$  and  $n$  are unqualified names appearing in a function at namespace scope:

```
namespace S {
    int m;
    void f(int n) {
        m = n;           // m and n are unqualified names
    }
}
```

- For names  $m$  and  $n$  appearing in the body of function  $S::f$ , *namespaces add more places to look than* in C...

34

## Unqualified Name Lookup

- For an unqualified name *n* appearing in a function *f* at global scope:
  - ***Look in the local scope.***
    - That is, look for *n* in the scope of *f*.
  - ***Look in the namespace scope(s).***
    - *That is, look for *n* in the namespace scope(s) enclosing *f*.*
    - *Start in the namespace immediately enclosing *f*.*
    - *Work outward to the global scope.*
  - ***Stop as soon as you find any declaration for *n*.***

35

## Unqualified Name Lookup

- In this example, *m* and *n* are unqualified names appearing in a function that's a member of a class *within a namespace*:

```
namespace S {
    int m;
    class T {
        void f(int n);
    };
    void T::f(int n) {
        m = n;           // m and n are unqualified names
    }
}
```

- Classes add one more place to look...

36

## Unqualified Name Lookup

- For an unqualified name *n* appearing in a function *f* that's a member of class *T* within a namespace *S*:
  - ***Look in *f*'s local scope.***
  - ***Look in the class scope.***
    - That is, look for *n* in the scope of *T*.
    - If the compiler finds *n* as a class member, it translates the reference to *n* into `this->n`.
  - ***Look in the namespace scope(s).***
    - *That is, look for *n* in *S*, the namespace scope enclosing *f*.*
    - *Search outward through enclosing namespaces to the global scope.*
  - ***Again, stop as soon as you find any declaration for *n*.***

37

## Using-Directives

- Writing the qualified names for the namespace members can get tedious.
- This is especially so when a particular source file uses names from only one namespace.
- This is a ***using-directive***:
 

```
using namespace N;
```
- It makes the names from namespace *N* appear as if they had been declared in an outer scope.
- It reduces the need to write qualified names.

38

## Using-Directives

- For example, before we had this:

```
#include "gadget.h"
~~~
ABC::status s = ABC::idle; // OK
```

- We can now write it as this:

```
#include "gadget.h"
using namespace ABC; // using-directive
~~~
status s = idle;           // OK
```

39

## Using-Directives

- This using-directive makes the names from namespace ABC appear to be global:

```
using namespace ABC;
```

- Thereafter, the compiler can find the unqualified names `status` and `idle` inside namespace ABC.

40



## Using-Directives

- A using-directive can appear at either namespace scope or block scope.
- It can't appear in class scope.
- It remains in effect to the end of the scope in which it appears.

41

## [01] Using-Directives

- Here's the classic "Hello, world" program presented earlier:

```
// "Hello, world" in Standard C++  
  
#include <iostream>  
  
int main() {  
    std::cout << "Hello, world\n";  
    return 0;  
}
```

42

## [02] Using-Directives

- Here it is with a using-directive at global scope:

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

#include <iostream>

using namespace std;

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

43

## [03] Using-Directives

- Yet another way to write a “Hello, world” program in C++ is:

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

#include <iostream>

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

44

## Using-Directives and Name Lookup

- Using-directives affect name lookup:  
`using namespace N;`
- This makes N's members appear to be members of the smallest namespace that encloses both N and the scope of the using-directive.
- For example...

45

## Using-Directives and Name Lookup

- The using-directive in f is in effect until the brace that closes f.
- While in effect, that directive makes the members of namespace S appear to be members of namespace N:

```
namespace N {
    namespace S {
        ~~~
 }
 void f() {
 using namespace S; // make members of S...
        ~~~                // appear to be members of N
    }
}
```

46

## Headers from Standard C

- The C++ Standard Library provides all of the C Standard Library headers.
- For each Standard C header named `<header.h>`, Standard C++ provides a header named `<cheader>`.
- For example, the C++ Standard Library provides a header named `<cctype>` as well as `<ctype.h>`.
- The headers have essentially the same contents.

47

## Headers from Standard C

- The difference between `<header.h>` and `<cheader>` is that:
  - The non-macro names declared in `<header.h>` are global.
  - The non-macro names declared in `<cheader>` are members of namespace `std`.
    - They might be global as well.
- Macro names are always global.
  - Actually, they're worse than global because they ignore nested scopes.

48

## Headers from Standard C

- The C++ Standard tries to set a good example.
  - It declares nearly all would-be global names as members of namespace `std`.
- The C++ Standard also deprecates all the `<header.h>` headers.
  - This is the Standard's way of discouraging the use of `<header.h>` headers.

49

## [04] Exercise: Using `<chheader>` Headers

- Rewrite the `xr` program, replacing each `<header.h>` with `<chheader>`.
- Add `std::` qualifiers or using-directives as needed.
- You might not need any.

50

## Using-Directives in Headers

- User-defined headers often refer to names declared in other (standard or third-party) headers, as in:

```
// xrt.h - cross-reference table interface
```

```
class xrt {
    ~~~
 size_t xm; // error: size_t undeclared
};
```

- The code in the header won't compile unless a declaration for `size_t` is available...

51

## Using-Directives in Headers

- “`xrt.h`” should include an appropriate header:

```
// xrt.h - cross-reference table interface
```

```
#include <cstdint> // for size_t
```

```
~~~~
class xrt {
    ~~~
 size_t xm; // error: size_t still undeclared
};
```

- It still doesn't compile because...

52

## Using-Directives in Headers

- When using headers of the form `<header>`, `size_t` is a member of namespace `std`.
- You could add a `using-directive` to the header, as in:

```
// xrt.h - cross-reference table interface

#include <cstddef> // for size_t
using namespace std; // not recommended
~~~~

class xrt {
    ~~~~
 size_t xm; // OK, but problematic...
};
```

53

## Using-Directives in Headers

- A `using-directive` might hoist tens or possibly hundreds of names from the library into the global scope.
- Users of “*xrt.h*” might not appreciate the name conflicts the `using-directive` might cause.
- ✓ *Avoid using-directives at global scope in headers.*
- Rather...

54

## Using-Directives in Headers

- In headers, refer to namespace members by their fully-qualified names, as in:

```
// xrt.h - cross-reference table interface

#include <cstdint> // for size_t

class xrt {
    ~~~
    std::size_t xm;    // jolly good
};
```

55

## Stream I/O Basics

- The `<iostream>` components support input as well as output.
- Here's a variation on "Hello, world" that illustrates both input and output...

56



## [05] Stream I/O Basics

```
// ask "Who are you?" and say "Hello"

#include <iostream>

using namespace std;

int main() {
    char name[20];
    cout << "Who are you? ";
    cin >> name;
    cout << "Hello, " << name << '\n';
    return 0;
}
```

57

## Stream I/O Basics

- `<iostream>` provides the following facilities:
  - `cin` is the standard input stream.
  - `>>` is the input operator, often called a stream *extractor*.
  - `cout` is the standard output stream.
  - `<<` is the output operator, often called a stream *inserter*.
- All are members of namespace `std`.

58

## Stream I/O Basics

- Here, `name` is an “array of char”:

```
cin >> name;
```

- This expression discards leading space characters from `cin` and reads the next sequence of non-space characters into `name`.
- The effect is essentially the same as calling:

```
scanf("%s", name);
```

- If the input sequence is longer than the array `name`, `scanf` will write merrily past the end of the array.
- So will `>>`.

59

## Stream I/O Basics

- A single expression can chain several inserters together.
- This writes "Hello, ", followed by the value of `name`, followed by a newline, to `cout`.

```
cout << "Hello, " << name << '\n';
```

- The effect is essentially the same as calling:

```
printf("Hello, %s\n", name);
```

- The C++ Standard Library provides format controls for `<<` and `>>` in another header called `<iomanip>`, described later.

60

## More about Stream I/O

- Normally, << writes using a default format.

- For example,

```
char name[20];
int age;
~~~
cout << name << " is " << age << "years old";
```

- When using << for output:
  - The default format for a character array is equivalent to printf's %s format.
  - The default format for an integer is equivalent to printf's %d format.

61

## More about Stream I/O

- The Standard C++ header <iomanip> defines format controls called **manipulators**.
- setw is a manipulator that specifies field width, as in:

```
int age;
~~~
cout << setw(4) << age;
```

- This statement writes `age` right-justified with leading spaces in a field 4 characters wide.
- `age` has type `int`, so this is equivalent to:

```
printf("%4d", age);
```

62

## More about Stream I/O

- `<iomanip>` provides other manipulators that control numeric precision, left- and right-justification, fill characters, and so on.
- Whereas `setw` requires an argument, most manipulators do not.
- For instance, the following uses the `hex` manipulator to display numeric value `age` in hexadecimal format:

```
cout << hex << age;
```

63

## More about Stream I/O

- `<iostream>` declares a few manipulators, too.
- One of them is `endl`.
- These expressions have almost the same effect:

```
cout << '\n';    // insert a newline
```

```
cout << endl;    // insert a newline and flush
```

- They both insert `'\n'` (a newline) into the output stream.
- The difference is that inserting `endl` also flushes any buffered output to the stream's output device.

64

## More about Stream I/O

- Flushing the output after writing every newline character makes the program run slower, but often not enough to notice.
- However, flushing the output after every newline ensures that every complete line written to standard output will actually appear in the output, even if the program aborts prematurely.
- If the program didn't use `endl`, some output might be lost.

65

## [06] Exercise: Stream Output

- Until now, *xr* used `<stdio.h>` for all its input and output.
- Rewrite *xr* so that the output goes to `std::cout` using the components in `<iostream>` and `<iomanip>`.
- Leave the input function, `get_token`, alone for now.

66

## Reference Types

- Reference types are another “Better C” feature of C++.
- References provide an alternative to pointers as a way of associating names with objects.
- C++ libraries often use references instead of pointers as function parameters and return types.

67

## Reference Types

- A reference declaration looks like a pointer declaration.
- It uses the & operator instead of the \* operator, as in:

```
int i;  
~~~  
int &ri = i;
```

- The last line above:
  - defines `ri` with type “reference to `int`”, and
  - initializes `ri` to refer to `i`.
- Reference `ri` is an *alias* for `i`.

68

## Reference Types

- A reference is essentially a pointer that’s automatically dereferenced each time it’s used.
- You can always rewrite code that uses references as code that uses pointers, as in:

| reference notation            | equivalent pointer notation     |
|-------------------------------|---------------------------------|
| <code>int &amp;ri = i;</code> | <code>int *cpi = &amp;i;</code> |
| <code>ri = 4;</code>          | <code>*cpi = 4;</code>          |
| <code>int j = ri + 2;</code>  | <code>int j = *cpi + 2;</code>  |

69

## Reference Semantics

- Once you define a reference, you can’t change it to refer to something else.
- Since you can’t change it after you define it, you must give it a value at the time you define it:

```
int foo() {
 int &r; // error: missing initializer
    ~~~~~  
}
```

70

## Reference Binding

- Initializing a reference is different from assigning to a reference.
- **Initializing** a reference associates the reference with an object.
  - Initializing a reference is also known as **binding**.
- **Assigning** to a reference stores through the reference and into the referenced object.
- For instance,

```
int &ri = i;    // binds reference to object
~~~
ri = 3; // assigns to referenced object
```

71

## Reference Binding

- You can bind references to class objects:

```
class widget {
public:
 T m;
    ~~~
};
widget w;
T x, y;
~~~
widget &rw = w; // binds rw to w
rw.m = x; // assigns x to w.m
widget *pw = &rw; // initializes pw to point to w
pw->m = y; // assigns y to w.m
```

72



## References and Operators

- An operator applied to a reference applies to the referenced object, not to the reference itself:

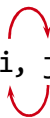
```
int &ri = i;
++ri; // increments i, not ri itself
int *p = &ri; // p points to i, not ri itself
```

- The `sizeof` a reference is:
  - the `sizeof` the type to which it refers.
  - not the `sizeof` the pointer that secretly implements the reference.
- Again, a reference is an alias for the object to which it refers.

73

## Reference Parameters

- What good are references?
- Consider a function named `swap` that you can use as follows:

 `swap(i, j);`

- Calling `swap(i, j)` is supposed to place:
  - the value that was in `i` into `j`, and
  - the value that was in `j` into `i`.

74

## Reference Parameters

- You can try writing the function as:

```
void swap(int v1, int v2) {
 int tmp = v1;
 v1 = v2;
 v2 = tmp;
}
```

- However, calling `swap(i, j)` doesn't swap `i` and `j`:
  - By default, C++ passes arguments by value, just as C does.
  - Thus, the call passes a copy of `i` as `v1` and a copy of `j` as `v2`.
  - The function swaps the copies and discards them as it returns.
  - The original `i` and `j` remain unchanged.

75

## [07] Reference Parameters

- In C, you would implement `swap` by using pointers as parameters:

```
void swap(int *v1, int *v2) {
 int tmp = *v1;
 *v1 = *v2;
 *v2 = tmp;
}
```

- You would also pass the addresses of `i` and `j` as the arguments:

```
swap(&i, &j); // swap values of i and j
```

76

## [08] Reference Parameters

- In C++, you can use references as parameters:

```
void swap(int &v1, int &v2) {
 int tmp = v1;
 v1 = v2;
 v2 = tmp;
}
```

- Using reference parameters, you need not (in fact, can't) pass the arguments by their addresses:

```
swap(i, j); // swap values of i and j
```

77

## Reference Parameters

- At the time of the call, each function parameter bind to its corresponding argument as if by doing this:

```
int &v1 = i;
int &v2 = j;
```

- Whether you use pointer or reference parameters, the compiler generates essentially identical code for the function body as well as the calls.
- Many of the `<iostream>` functions in the Standard Library use reference parameters...

78

## Stream Input

- For example, using `<stdio.h>` the `get_token` function in `xr` gets the next character from standard input using:

```
int c;
while ((c = getchar()) != EOF) {
    ~~~
}
```

- Using `<iostream>` the code looks like:

```
char c;
while (cin.get(c)) {
    ~~~
}
```

79

## Stream Input

- In C++, `cin` is the standard input stream.
- It's an object of class type `istream`.
- Like most of the Standard Library, `cin` and `istream` are members of namespace `std`.
- The `istream` class has a member `get` declared somewhat like:

```
class istream {
public:
 bool get(char &c);
    ~~~
};
```

- It's actually a little different, but this is close enough for now.

80

## Stream Input

- Calling `cin.get(c)` gets a character from `istream cin` and places it in `c`.
- If `get` fails to get a character, it returns a value that compares equal to zero.
- One big advantage of this interface is that it's safer than using `getchar...`

81

## Stream Input

- `getchar` uses a single return value of type `int` to return both:
  - the end-of-file indicator and
  - the value of the char read, if any.
- Using `getchar` often requires casting, as in:

```
int c;
char *s;
~~~
while ((c = getchar()) != EOF) {
 *s++ = (char)c;
}
```

82

## Stream Input

- In contrast, `istream::get` keeps the end-of-file indicator separate from the returned character.
- You don't need a cast to pull them apart:

```
char c;
char *s;
~~~
while (cin.get(c)) {
    *s++ = c;
}
```

83

## Stream Input

- Arguably, `istream::get` could use a pointer parameter instead of a reference, as in:

```
char c;
char *s;
~~~
while (cin.get(&c)) {
 *s++ = c;
}
```

- But, a function parameter of type “pointer to char” will accept either a pointer passed by value or a character passed by address...

84

## Stream Input

- If `istream::get` used a pointer parameter, you might accidentally write:

```
char c;
char *s;
~~~  
while (cin.get(s)) {  
    *s++ = c;  
}
```

- The compiler won't catch this.
- This can't happen with a reference parameter.
- A "reference to char" won't bind to a "pointer to char".
- It will bind only to a char.

85

## [09] Exercise: Stream Input

- Rewrite the `get_token` function in *xr*, using components from `<iostream>` in place of `<stdio.h>`.
- In addition to `cin.get(c)`, you will need to use:
  - `cin.eof()` returns true if `cin` has reached end-of-file; otherwise it returns false.
  - `cin.putback(c)` puts character `c` back into `cin` so that the next call to `cin.get(c)` returns that character.

86

## Bibliography

- Ellis and Stroustrup [1990]. Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley.



## [6] Function Name Overloading

---

1

### Declaring Overloaded Functions

---

- Many C libraries contain at least one group of functions with very similar names.
- Each function in such a group performs essentially the same operation but has a different parameter list or return type.
- For example, the C Standard Library has a group of “put” functions including:

```
int fputc(int c, FILE *f);  
int fputs(char const *s, FILE *f);  
int putchar(int c);  
int puts(char const *s);
```

2

## Declaring Overloaded Functions

- Even though the functions in the group have different names, programmers often refer to the functions as if they all had the same name.
  - I've never heard a programmer say "the program *fputc-ed* a character."
  - I have heard programmers say "the program *put* a character."
- C++ lets you actually give all those functions the same name.
- Two or more functions are **overloaded** when they're declared with the same name in the same scope.

3

## Declaring Overloaded Functions

- For example, in C++ you can implement the Standard C "put" functions as overloaded functions actually named put:

```
int put(int c);                // putchar
int put(int c, FILE *f);       // fputc
int put(char const *s);        // puts
int put(char const *s, FILE *f); // fputs
```

- C doesn't allow overloading.
  - If you tried to compile the above declarations as C, the compiler would reject all the declarations after the first.
- In C++, you can overload member functions as well as non-member functions.

4

## Overload Resolution

- When the compiler encounters a call to an overloaded function, it selects the appropriate function to call.
- The selection process is called ***overload resolution***.
- The compiler matches the type(s) of the argument(s) in the call against the type(s) of the parameter(s) in the function declarations.
- For example,

```
                // calls...  
put('a', stdout); // int put(int c, FILE *f);  
put("Hello\n");   // int put(char const *s);
```

5

## Overload Resolution

- If the compiler can't find a function that accepts the specified arguments, it issues an error message.
- For example, this is an error:

```
put("n = ", n, stdout);    // error: too many arguments
```

- The number of parameters in the matching function declaration must be the same as the number of arguments in the call.
- However, the type of each parameter need not be exactly the same as the type of its corresponding argument.

6

## Overload Resolution

- C++ compilers apply some conversions in an attempt to find a match.
- For example, each of these functions writes a single character to a file:

```
int put(int c);  
int put(int c, FILE *f);
```

- In each function, parameter `c` has type `int` rather than `char`.
- This follows the convention of the C Standard Library.

7

## Overload Resolution

- Even though these functions use `int` as the parameter type, you can call them with `char` as the argument type.
- For example,

```
char c;  
~~~  
put(c); // calls int put(int c);
put(c, stdout); // calls int put(int c, FILE *f);
```

- In each case, the compiler promotes `char` to `int` to achieve the match.

8

## Best Matches and Ambiguities

- More than one function in a group of overloaded functions might be capable of satisfying a particular function call.
- For example, only the first three of these functions is capable of satisfying a call to `f(0)`:

```
int f(int i); // f(0)? maybe
long int f(long int li); // f(0)? maybe
char *f(char *p); // f(0)? maybe
int f(double d, int i); // f(0)? definitely not
```

- Clearly, the fourth function can't satisfy the call:
  - `f(0)` passes only one argument, while the fourth function requires two.

9

## Best Matches and Ambiguities

- We're left with three viable functions:

```
int f(int i);
long int f(long int li);
char *f(char *p);
```

- The call `f(0)` can pass `0`:
  - as an `int`,
  - as a `long int`, or even
  - as a `char *` whose value is the null pointer.

10

## Best Matches and Ambiguities

- When confronted with a choice of functions:
  - Overload resolution uses a **ranking** of the conversions from the argument type into the parameter type
  - The ranking helps determine which function, if any, is the **best match**.
- Thus, overload resolution depends on the exact type of each function call argument as well as the type of its corresponding parameter.
- In the case of calling `f(0)`, it depends on the exact type of argument `0`.
- Obviously, `0` is an `int`.
- But is it signed or unsigned?

11

## The Types of Literals

- The literal `0` is a plain `int`, which is signed by default.
- In general:
  - Any decimal integer literal whose value can be represented as an `int` has type `int`.
  - Otherwise, if the literal's value can be represented as a `long int`, it has type `long int`.
  - Otherwise, it has type `long long int`.

12

## The Types of Literals

- For example:
  - The literals 12 and 437 always have type `int`.
  - The literal 95000 may be either an `int` or a `long int` depending on the maximum value for `int` on the target platform.
- If a literal has a value that's too big to fit into an allowed type, the compiler flags the literal as an error.
- You can specify the type of an integer literal by adding a suffix:
  - U or u for unsigned
  - L or l for long
  - F or f for float
- See Saks [2000a] and [2000b] for more details on the types of literals in C and C++.

13

## Best Matches and Ambiguities

- Once again, the question is: Which function is the best match for the call `f(0)`?

```
int f(int i);
long int f(long int li);
char *f(char *p);
```

- Since 0 is an `int`, calling `f(0)` with the third function requires a ***pointer conversion*** from `int` to `char *`:

```
char *f(char *p);
```

14

## Best Matches and Ambiguities

- Calling `f(0)` with this function requires an **integral conversion** from `int` to `long int`:

```
long int f(long int li);
```

- Calling `f(0)` with this function requires no conversion at all:

```
int f(int i); // the winner for f(0)!
```

- The argument is an **exact match** for the argument type.
- An exact match is always preferable to a conversion.

15

## Best Matches and Ambiguities

- Now, suppose we take away the exact match:

```
int f(int i);
long int f(long int li);
char *f(char *p);
~~~  
f(0);          // compile error
```

- The integral conversion (from `0` to `long int`) and the pointer conversion (from `0` to `char *`) have the same rank.
- The compiler won't choose one over the other.

16



## Best Matches and Ambiguities

- When the compiler can't find at least one function that satisfies a particular call, it typically complains that there's "no matching function".
- When the compiler finds more than one function that satisfies the call, but no unique best match, it typically complains that the call is "ambiguous".

17

## Best Matches and Ambiguities

- By the way, the literal `0` is the only integer with an implicit conversion to a pointer type.
- `f(x)` is ambiguous only when `x` is the integer literal `0`:

```
long int f(long int li);
char *f(char *p);
int n;
~~~
f(n); // calls long int f(long int li)
f(1); // calls long int f(long int li)
```

- See Lippman and Lajoie [1998] for more details on overload resolution and ambiguities.

18

## nullptr

- Using `nullptr` provides more intuitive behavior for overload resolution:

```
int f(int i);
char *f(char *p);
~~~  
f(NULL);           // surprising: calls f(int);  
f(nullptr);        // unsurprising: calls f(char *);
```

- `nullptr` is a new keyword available in C++11...

19

## nullptr

- `nullptr` is a null pointer constant of type `std::nullptr_t`.
  - It can be converted implicitly or compared to any pointer type.
  - It can't be implicitly converted or compared to any integral type, except for `bool`.
- `std::nullptr_t` is defined in `<cstdlib>`.
  - You need not include any header to use `nullptr`.
  - Again, `nullptr` is a keyword.
- `0`, `0L`, and `NULL` remain valid null pointer constants.
  - They're no longer fashionable.

20

### [01] Exercise: Using `nullptr`

- Rewrite *xr*, replacing all uses of `NULL` with `nullptr`.

21

### Function Signatures

- Overloaded functions can appear at global scope and have external linkage.
- This means that a C++ program can define an overloaded function in one translation unit and call it from another.
- A linker for C++ must be able to resolve calls to overloaded functions across translation units.

22

## Function Signatures

- In a C program, no two functions with external linkage can have the same name.
  - Therefore, a name is all it takes to uniquely identify a C function.
- In a C++ program, overloaded functions share the same name.
  - Thus, it takes more than just a name to identify a particular C++ function.
- In C++, an overloaded function must have a parameter list that's sufficiently different from all other functions with the same name.
- Otherwise, the compiler and linker can't tell the functions apart.

23

## Function Signatures

- For example, these declarations declare three different functions named `abs`:  
  

```
int abs(int i);  
long int abs(long int li);  
double abs(double d);
```
- Saying “the expression `abs(x)` calls the `abs` function” doesn't tell you which `abs` function it calls.
- If, for example, `x` has type `double`, it's more precise to say:
  - “It calls the `abs` function with a single parameter of type `double` and a return type of `double`”.

24

## Function Signatures

- As in C, a C++ program can declare a function more than once.
- A function declaration whose name appeared in a previous declaration doesn't necessarily introduce another overloaded function.
- It might just be a *redeclaration*:

```
int put(int i);  
int put(int n);          // a redeclaration
```

- Since the parameter name is optional, this is yet another declaration for the same function:

```
int put(int);            // another redeclaration
```

25

## Function Signatures

- The C++ Standard defines the *signature* of a function as:
  - “the information about a function that participates in overload resolution”.
- A function's signature is primarily the sequence of types in its parameter list.
- For example, consider:

```
char *get(char *s, int n, FILE *f);
```

- Here, the signature is the type sequence (char \*, int, FILE \*).

26

## Function Signatures

- The concept of function signature doesn't exist in C.
- Many of the following examples behave the same in C and C++.
- However, the C Standard doesn't use the term ***signature*** to describe the behavior.

27

## Function Signatures

- The signature of this function is just the type (FILE \*):  
  
`int get(FILE *f);        // signature = (FILE *)`
- Parameter types are part of the signature, but parameter names aren't.
- Thus, both of these functions have the same signature as well:

```
int get(FILE *stream); // signature = (FILE *)  
int get(FILE *);       // signature = (FILE *)
```

28

## Different Declarations, Same Signature

- Two function declarations that differ in more than their parameter names can still have the same signature.
- For example, these types are equivalent:

```
long  
long int  
signed long int
```

- Thus, these functions have the same signature:

```
long    abs(    long    n);  
long int abs(    long int n); // same  
signed long int abs(signed long int n); // same
```

29

## Different Declarations, Different Signatures

- Although (plain) `int` and `long int` have the same size on *many* machines, there's no guarantee that they have the same size on *all* machines.
- Thus, C++ treats `int` and `long int` as distinct types.
- These are overloaded functions with distinct signatures:

```
int abs(    int n);  
long int abs(long int n);    // a different function
```

30

## Different Declarations, Same Signature

- In C++, a parameter declared with an array type actually has a pointer type.
- Thus, for example, these functions have the same signature, namely, (char \*):

```
char *get(char *s);    // signature = (char *)  
char *get(char s[]);  // same function
```

- C++ ignores array dimensions in parameter declarations.
- Hence, the signature of this function is also just (char \*):

```
char *get(char s[N]);  // still the same function
```

31

## Different Declarations, Same Signature

- A typedef name is not a distinct type.
  - It's just an alias for another type.
- C++ turns each typedef name into its underlying type when determining signatures...

32



## Different Declarations, Same Signature

- For example, the standard type `size_t` is often defined by a typedef declaration such as:

```
typedef unsigned int size_t;
```

- In that case, these functions have the same signature:

```
size_t f(size_t n);  
unsigned int f(unsigned int n); // the same function
```

- They're not overloaded.

33

## Different Declarations, Different Signatures

- On the other hand, each class, structure or union is a distinct type, even if they have exactly the same members.
- For example, these are distinct types:

```
struct foo { int m; };  
struct bar { int m; }; // a different type
```

- Therefore, these are distinct overloaded functions:

```
int f(foo x);  
int f(bar x); // a different function
```

34

## Different Declarations, Same Signature

- Finally, these functions have the same signature in C++:

```
int get(void); // in C++: an empty parameter list
int get();     // in C++: same thing
```

- Both declare `get` as a function with an empty parameter list.

35

## Signatures and Return Types

- The return type of a function is not part of its signature.
- C++ doesn't let you overload functions by writing declarations that differ only in their return types.
- For example, you can't declare these in the same scope:

```
int f(int);
char *f(int); // no: compile error
```

- Overloading on return types would place an undue burden on the compiler to determine the programmer's intent...

36

## Signatures and Return Types

- For example, suppose you could declare two functions named `f` as on the previous slide.
- How should the compiler interpret the call to `f` in:

```
int n;  
~~~~  
if (f(n)) { // works whether f returns int or char *
    ~~~~~  
}
```

- Either return type could be the type of the controlling expression in an if-statement.
- On what basis should the compiler prefer one function to the other?

37

## Name Mangling

- Most linkers can't handle data type information directly.
- Therefore, most C++ compilers use a technique called ***name mangling*** to encode each function's signature with its name to form a single longer name.
- The compiler places these long "mangled" names into the external symbol dictionaries of the object modules it produces.
- The mangled names are what the linker uses.
- The *ARM* (Ellis and Stroustrup [1990]) describes one such encoding scheme...

38

## Name Mangling

```
char *get(char *s, int n, FILE *f);
```

- For this function, the scheme encodes:
  - the name `get` and the fact that it's a function as `get__F`,
  - the parameter type `char *` as `Pc`
  - `int` as `i`, and
  - `FILE *` as `P4FILE`.
- The entire mangled name looks like:

```
get__FPciP4FILE
```

- This is the name that the linker sees as the external symbol.

39

## Name Mangling

- Some C++ implementations actually display the mangled names in error messages for unresolved or multiply-defined externals, forcing you to demangle the names yourself.
- To make matters worse, there's no standard encoding scheme.
- Different compilers use different schemes...

40

## Name Mangling

- Again, the function is:

```
char *get(char *s, int n, FILE *f);
```

- The Edison Design Group (EDG) C++ front end uses an encoding scheme similar to, but not quite the same as, the ARM's:

```
_get__FPciP6_iobuf
```

- The name `_iobuf` appears in the mangled name in place of `FILE` because, in the header `<stdio.h>` for this compiler, `FILE` is a typedef name that's an alias for `_iobuf`.

41

## Name Mangling

- Microsoft Visual C++ does a real number on the name, mangling it as:

```
?get@@YAPADPADHPAU_iobuf@@@Z
```

- This is the way the name appears in both link maps and assembly listings.

42

## Name Mangling

- Borland C++ generates link maps that display function names and signatures much as they appear in the source code.
- The function appears in a Borland link map as:

```
get(char*,int,FILE*)
```

- However, assembly listings display the name mangled as:

```
@get$qp4FILE
```

43

## Bibliography

- Ellis and Stroustrup [1990]. Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley.
- Lippman and Lajoie [1998]. Stanley B. Lippman and Josée Lajoie, *C++ Primer, 3<sup>rd</sup>. ed.* Addison-Wesley.
- Saks [2000a]. Dan Saks, "Numeric Literals", *Embedded Systems Programming*. September, 2000.
- Saks [2000b]. Dan Saks, "Character and String Literals", *Embedded Systems Programming*. October, 2000.

44

## [7] Resource Management

---

1

### Variable-Length Arrays

---

- The most recent version of the *xr* program defines entry as:

```
struct entry {  
    char *word;  
    lns lines;  
};
```

- Though it's declared as a pointer, *word* is conceptually a variable-length character string:

```
string word;
```

2

## Variable-Length Arrays

- The C++ Standard Library provides a string class, which should be adequate for many applications.
- You'll be able to use a string class more effectively if you understand the work that goes into writing one.
- Moreover, the issues that arise in designing a string class provide valuable insights into many features of C++.
- Rather than study just string classes, we'll look at a broader class of related data structures — variable-length arrays.

3

## Variable-Length Arrays

- A **variable-length array** is an array whose dimension need not be determined until run time.
- If variable-length arrays were native to C++, you could write code such as:

```
void f(size_t n) {  
    float fa[n];           // what we want  
    ~~~  
 // use fa to do something productive
    ~~~  
}
```

4



## Variable-Length Arrays

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

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

- In C++, an array object must have dimensions that can be determined at compile time.
- That is, `n` above must be a *constant-expression*.
- C++ will reject this declaration unless `n` is constant.

5

## Variable-Length Arrays

- An array dimension may be an expression composed of multiple operators and operands.
- However, all the operands must be constant.
- For example,

```
#define M 10  
enum { N = 15 };  
int const K = 1;  
~~~  
float x[M + N - K]; // OK, 24 elements
```

6

## Variable-Length Arrays

- The following array declaration is not valid, even if `MAX` is an integer constant:

```
int n = MAX; // n is a non-constant object
~~~
long d[n];            // error: non-constant dimension
```

- A clever compiler might be able to determine the dimension at compile time, but that's not good enough here.
- A non-constant object is not a constant expression.

7

## Variable-Length Arrays

- Note: Although C++ doesn't provide built-in support for variable-length arrays, C does.
- C99 includes built-in support for a form of variable-length arrays.
- However, they never really caught on.
- C11 made them optional.
- In any case, variable-length arrays in C are not as flexible as variable-length arrays you can build for yourself in C++.

8

## Variable-Length Arrays

- In C++, if a variable-length array is what you really want, you can approximate it.
- Use a pointer to the first element of a dynamically allocated array, as in:

```
size_t n;
T *x;
~~~
// do something to determine the value of n
~~~
x = new T [n];
```

- In an array new-expression, the array dimension need not be a compile-time constant.

9

## Variable-Length Arrays

- In C++, as in C, you can apply the `[]` operator to a pointer.
- Thus, if `x` points to the initial element of an array, you can refer to the *i*-th element of that array as `x[i]`.
  - This is good.
- However, when you're done with the array, you must remember to discard the storage using:

```
delete [] x;
```

- Unfortunately, this is easy to forget.
- Forgetting leads to memory leaks.
  - This is not so good.

10

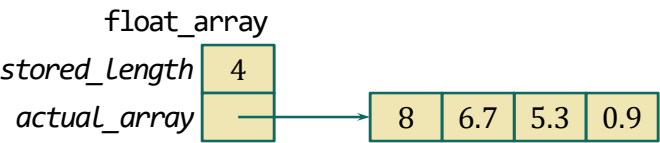
## Variable-Length Arrays

- If C++ doesn't have what you want built in, you can often build a class that will come pretty close.
- Let's build a class `float_array` that implements a variable-length array of `float` as an abstract type.
- `float_array`'s public member functions should provide operations that make `float_array` act like other array types.
- Each `float_array` object contains a pointer to a dynamically allocated array of `float`, but it hides that pointer as private data.
- More precisely...

11

## Variable-Length Arrays

- You can implement a `float_array` as a pair of values:
  1. `stored_length`: the number of elements in the array
  2. `actual_array`: a pointer to the first element of the actual array allocated from the free store
- For example, the storage for a `float_array` with four elements looks like:



12

## Variable-Length Arrays

- Remember that C++ goes to *great* lengths to let you fashion user-defined types that look and act very much as if they were built into the language.
- Our first version of `float_array` won't look and act much like a built-in array type.
  - Give it time.
- We'll gradually make `float_array` look more built-in as we apply more C++ features.
- Here's some terminology that will help describe the behavior of the `float_array` class...

13

## Deep vs. Shallow Classes

- `float_array` is a ***class with deep structure***, or more simply, a ***deep class***.
- A class is deep if it has at least one data member referring to separately allocated resources managed by the class.
- The most common deep classes are those with pointers to dynamically allocated memory.
- `float_array` is deep because it has a member that points to a dynamically allocated array of `float`.
- The `xrt` and `lns` classes are also deep:
  - `xrt` contains an array of pointers to dynamically allocated entry structures.
  - `lns` contains two pointers to dynamically allocated `list_node` structures.

14

## Deep vs. Shallow Classes

- A deep class need not have any pointer members.
- A class with a member of any type that designates a separately allocated, managed resource has deep structure.
- For example, files are typically managed resources.
  - You must open them before you use them.
  - You should close them when you're done with them.
- A class with an integer member designating a file can be a deep class.

15

## Deep vs. Shallow Classes

- Obviously, not all classes have deep structure.
- For example, a class representing complex numbers typically has just two data members of some floating-point type, as in:

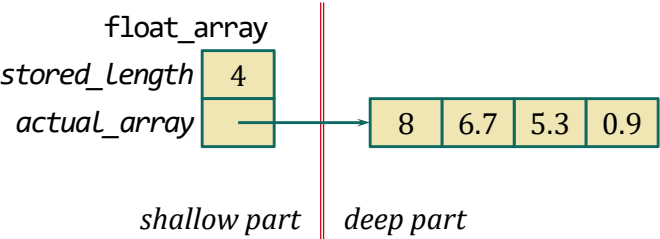
```
class complex {  
    ~~~  
private:
 double real, imaginary;
};
```

- This class uses no resources beyond its data members.
- It's a ***class with shallow structure***, or simply, a ***shallow class***.

16

## Shallow vs. Deep Parts

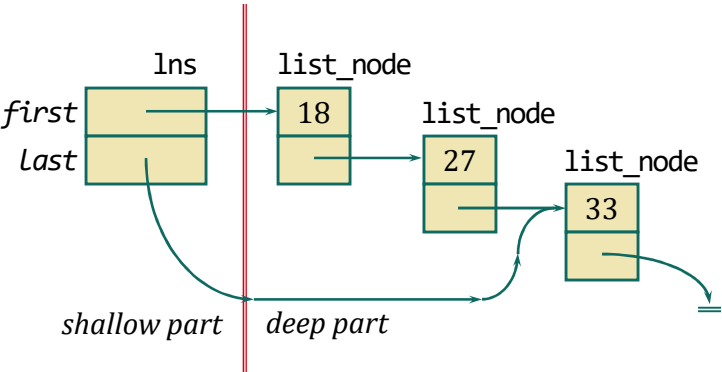
- The **shallow part** of an object is the storage that contains the object's data members.
- The **deep part** of an object is any storage used to represent the object's state beyond the shallow part.
- For example, here are the deep and shallow parts of a float\_array object:



17

## Shallow vs. Deep Parts

- Here are the deep and shallow parts of an `Ins`:



18

## Shallow vs. Deep Parts

- Objects with shallow structure have no deep part.
- The `sizeof` operator applied to a class object yields the number of bytes in the object's shallow part:

```
widget w;
~~~
size_t n = sizeof(w);    // # of bytes in shallow part
```

- The `sizeof` operator applied to a class yields the number of bytes in the shallow part of any object of that class.

```
if (sizeof(w) == sizeof(widget))    // always true
    ~~~
```

19

## Array-Like Operations

- Ultimately, `float_array` objects should look and act much like other arrays.
- For example, if `fa` is a `float_array`, you should be able to access its elements using the `[]` operator, as in:

```
x = fa[i];
fa[i] = 0;
```

- We'll start by implementing something simple that does the job, even if it's not as convenient as we'd like it.
- Then we'll improve the packaging.

20



## Array-Like Operations

- Instead of the `[]` operator, our first implementation of the `float_array` class provides two member functions:

```
float get(std::size_t i);
void set(std::size_t i, float f);
```

- `get` can fetch the *i*-th element of `float_array fa`, as in:

```
x = fa.get(i); // think of x = fa[i]
```

- `set` can store into the *i*-th element of `fa`, as in:

```
fa.set(i, x); // think of fa[i] = x
```

21

## Array-Like Operations

- Each `float_array` keeps track of the number of elements it contains.
- `float_array` provides a member function that returns that value:

```
std::size_t length();
```

- For example, you can write all the values in `float_array fa` to `cout` using:

```
size_t i;
for (i = 0; i < fa.length(); ++i) {
 cout << fa.get(i) << '\n';
}
```

22

## Implementing float\_array

- Here's a definition for a `float_array` class with the members seen thus far:

```
class float_array {
public:
 float get(std::size_t i);
 void set(std::size_t i, float f);
 std::size_t length();
private:
 std::size_t stored_length;
 float *actual_array;
};
```

23

## Where's the Rest of Me?

- We don't yet have member functions to manage the deep part of a `float_array`.
- Consider what happens when you declare a `float_array` local to a function, as in:

```
void foo() {
 float f;
 float_array fa; // declare fa and...
 // allocate its shallow part
    ~~~
    f = fa.get(0);  // where's its deep part?
    ~~~
}
```

24

## Where's the Rest of Me?

- Objects declared local to a function without an `extern` or `static` specifier have ***automatic storage***.
- When it calls a function, the program automatically allocates storage for the function's automatic objects.
- That storage typically resides on a stack.
- Such automatic allocation acquires only the shallow parts.
- The deep parts must be acquired some other way.
- The `float_array` class should provide a member function that "allocates" or "creates" the deep part.
- Each `float_array` can have a different length, so that function should accept the length as an argument.

25

## Where's the Rest of Me?

- Thus, it takes two statements to declare a `float_array` and allocate all the storage it needs, as in:

```
void foo() {
 float f;
 float_array fa; // declare fa and...
 // allocate its shallow part
 fa.create(4); // allocate its deep part
    ~~~
    f = fa.get(0);  // OK
    ~~~
}
```

26

## Avoiding Leaks

- When a function returns, the program automatically deallocates the storage for the function's automatic objects.
- The automatic deallocation of an object discards only its shallow part.
- If the shallow part contains the only handle to the deep part, that handle will be lost.
- If the function doesn't deallocate the deep part explicitly, the deep part becomes a **leak** — an allocated resource that can never be deallocated.
- The `float_array` class should provide a member function that "deallocates" or "destroys" the deep part.

27

## Avoiding Leaks

- A function that uses a `float_array` should explicitly discard the `float_array`'s deep part before returning, as in:

```
void foo() {
 float f;
 float_array fa; // declare fa and...
 // allocate its shallow part
 fa.create(4); // allocate its deep part
    ~~~  
    f = fa.get(0);  // OK  
    ~~~  
 fa.destroy(); // deallocate fa's deep part
} // deallocate its shallow part
```

28

## [01] Implementing float\_array

- Here's the header that defines the float\_array class:

```
// fa.h - float_array interface

#ifndef FA_H_INCLUDED
#define FA_H_INCLUDED

#include <cstddef>

class float_array {
 // see the next slide
}

#endif
```

29

## [01] Implementing float\_array

- Here's the complete class definition:

```
class float_array {
public:
 void create(std::size_t n);
 void destroy();
 float get(std::size_t i);
 void set(std::size_t i, float f);
 std::size_t length();
private:
 std::size_t stored_length;
 float *actual_array;
};
```

30

## [01] Testing float\_array

- Here's a simple program for testing float\_array:

```
// testfa.cpp - test float_array
```

```
#include <cstdint>
#include <iostream>
```

```
#include "fa.h"
```

```
using namespace std;
~~~
```

31

## [01] Testing float\_array

- The test program contains a function that displays a float\_array:

```
void display(char *s, float_array &fa) {
    size_t i;
    cout << s << " =";
    for (i = 0; i < fa.length(); ++i) {
        cout << ' ' << fa.get(i);
    }
    cout << endl;
}
```

- Notice that display's second parameter is a reference, but the first parameter is a pointer...

32

## Testing float\_array

- If `float_array fa` contains five elements with values 0 through 4, calling:

```
display("fa", fa);
```

displays:

```
fa = 0 1 2 3 4
```

- `fa` is a `float_array`, which is a class.
- Passing class objects by reference is very conventional in C++.
- `"fa"` is a string literal.
- C++, like C, passes string literals as pointers.

33

## Non-Member Functions Still Have a Place

- Not every function in a C++ program needs to be a member of some class.
- In the test program, `display` is not a member function.
- It doesn't need access to `float_array`'s private members to do its job.

34

## Non-Member Functions Still Have a Place

- Not every operation on an object of class type needs to be a member function.
- In general, a class should provide members with just enough functionality so that applications can do what they need to do clearly, safely, and efficiently.
- Building too much functionality into a class can actually reduce its utility.

35

## [01] Testing float\_array

- Here's the remainder of the test program:

```
int main() {  
    size_t length;  
    cout << "length? ";  
    cin >> length;  
    float_array fa;  
    fa.create(length);  
    float_array fb;  
    fb.create(0);  
    display("fa", fa);  
    display("fb", fb);  
    size_t i;  
    ~~~
```

36



## [01] Testing float\_array

```
for (i = 0; i < fa.length(); ++i) {
 fa.set(i, static_cast<float>(i));
}
display("fa", fa);
float_array fc;
fc.create(fa.length());
for (i = 0; i < fc.length(); ++i) {
 fc.set(i, static_cast<float>(i * i));
}
display("fc", fc);
fa.destroy();
fb.destroy();
fc.destroy();
return 0;
}
```

37

## Testing float\_array

- When the input is 5, the program output is:

```
length? 5
fa = 0 0 0 0 0
fb =
fa = 0 1 2 3 4
fc = 0 1 4 9 16
```

38

## [01] Implementing float\_array

- Here's an implementation of float\_array's create function:

```
void float_array::create(size_t n) {
 stored_length = n;
 if (stored_length == 0) {
 actual_array = nullptr;
 } else {
 actual_array = new float [stored_length];
 }
}
```

39

## Implementing float\_array

- The create function fills in the data members so that the float\_array represents an array of float with the specified dimension.
- In particular, create stores the number of elements into the private data member stored\_length.
- If that number is zero, create allocates nothing and stores a null pointer into the private data member actual\_array.
- Otherwise, it allocates an array of float and stores the address of the initial array element into actual\_array.

40

## Implementing float\_array

- Suppose you omit this part of create:

```
if (stored_length == 0) {
 actual_array = nullptr;
} else {
```

- In that case, the program might execute this array new-expression with stored\_length equal to zero:

```
actual_array = new float [stored_length];
```

- This is not an error.

41

## [01] Implementing float\_array

- An array new-expression of the form `new T [n]`, where `n` has the value zero, returns a non-null pointer.
- Dereferencing that pointer yields undefined behavior.
  - This is no more nor less restrictive than using a null pointer.
  - Dereferencing a null pointer has undefined behavior, too.
- Thus, the program can't access the allocated storage.
- However, it can delete that storage.
- Thus, you can simplify the create function to just:

```
void float_array::create(size_t n) {
 stored_length = n;
 actual_array = new float [n];
}
```

42

## [01] Exercise: Implementing float\_array

- Implement the remaining float\_array member functions:
  - destroy
  - get
  - set
  - length
- Test your implementation using the test program provided.
- Hint: You can implement each function with just a single statement.

43

## Simplifying Resource Management

- Once again, we'd like to be able to write code like this:

```
void f(size_t n) {
 float fa[n]; // what we want
    ~~~  
    // use fa to do something productive  
    ~~~  
}
```

- The float\_array class is not nearly so simple to use...

44

## Simplifying Resource Management

- To use a `float_array` properly, you have to know that it has a deep part.
- You don't need to know exactly what's in the deep part, but you have to remember to use the `create` and `destroy` members:

```
void f(size_t n) {
 float_array fa;
 fa.create(n);
    ~~~
    // use fa to do something productive
    ~~~
 fa.destroy();
}
```

45

## Simplifying Resource Management

- For `float_array` to look and act like a built-in array type, each `float_array` object should acquire and release its resources with no more programming effort than required for any fixed-length array.
- With remarkably little effort, we can define `float_array` so that we can write:

```
void f(size_t n) {
 float_array fa (n); // what we can have
    ~~~
    // use fa to do something productive
    ~~~
}
```

46

## Simplifying Resource Management

- We can modify the class so that a definition such as automatically allocates the deep as well as the shallow part.

```
float_array fa (n);
```

- Moreover, returning from the function will automatically deallocate the deep part as well as the shallow part.
- C++ provides the tools you need to do this:
  - Constructors provide automatic resource allocation and object initialization.
  - Destructors provide automatic resource deallocation.

47

## Constructors

- A **constructor** is a special member function that initializes an object after that object's shallow part has been allocated.
- Initialization may include allocating the deep part.
- In C++, a constructor is a member function with the same name as its class:

```
class float_array {
public:
 float_array(std::size_t n); // constructor
    ~~~~~
};
```

48

## [02] Constructors

- `float_array`'s `create` member function already does what a constructor should do.
- You can turn `create` into a constructor by simply changing its declaration:

```
class float_array {
public:
    float_array
    void create(std::size_t n);
    ~~~
};
```

49

## [02] Constructors

- You must also change the heading in the function definition:

```
float_array
void float_array::create(std::size_t n) {
 stored_length = n;
 actual_array = new float [n];
}
```

- A constructor can have parameters.
- It can't have a return type — not even `void`.

50

## Constructors

- If a class T has a constructor, then every definition for a T object invokes that constructor automatically.
- If the constructor has a non-empty parameter list, then every definition for a T object *must* provide constructor arguments:

```
float_array fa (100); // OK: pass 100 to constructor
```

- This definition allocates storage for fa's shallow part, the same as if there were no constructor.
- It then applies the float\_array constructor to fa, passing 100 as the constructor argument.
- That initialization includes allocating an "array of 100 elements of float" as the object's deep part.

51

## Constructors

- A missing argument produces a compile error:

```
float_array fb; // error: missing argument
```

- You can't call a constructor to operate on an object the same way you can call other member functions.
- That is, if fa is a float\_array, you can't call:

```
fa.float_array(n); // no can do
```

52



## Brace Initializers in C++11

- C++11 lets you enclose constructor arguments in braces as well as in parentheses.
- For example,

```
float_array fa (100); // OK in C++03 and C++11
float_array fb {200}; // OK in C++11, but not C++03
```

- Enclosing an initializer in braces has an added benefit:
  - Using { } around the initializer prohibits narrowing conversions from the argument type to the parameter type.

53

## Destructors

- Many class objects acquire resources (such as memory or files) that should be released when the object goes away.
- A **destructor** is a special member function that releases the resources used by an object just as the object reaches the end of its lifetime.

54

## Destructors

- A destructor for a class has the same name as the class, prefixed with a ~:

```
class float_array {
public:
 float_array(std::size_t n); // constructor
 ~float_array(); // destructor
    ~~~
};
```

- A destructor can have neither parameters nor a return type.

55

## [02] Destructors

- float\_array's destroy member function already does what a destructor should do.
- You can turn destroy into a destructor by simply changing its declaration:

```
class float_array {
public:
    ~float_array
    void destroy();
    ~~~
};
```

56

## [02] Destructors

- You must also change the heading in the function definition:

```

~float_array
void float_array::destroy() {
 delete [] actual_array;
}

```

- Notice where the ~ is in the destructor's fully qualified name:

```
float_array::~~float_array // yes!
```

```
~float_array::float_array // no
```

57

## Destructors

- The compiler automatically generates a call to an object's destructor (if any) at the point where the object's lifetime ends:

```

void f(size_t n) {
 float_array fa (n);
    ~~~
    // ~float_array() implicitly applied to fa here
}

```

58

## Declarations as Statements

- Since declarations may invoke constructors, C++ treats them as executable statements.
- That is, you can interleave declarations with other statements.
- As always, you must declare each name before you use it.
- Other than that, you can place declarations anywhere inside a block of code, not just at the beginning:

```
void f() {
    size_t n;           // declaration
    cin >> n;           // expression statement
    float_array fa (n); // declaration
    ~~~
}
```

59

## Declarations as Statements

- If C++ required that all declarations precede the first statement, as C once did, then you would have to write `f` as something like:

```
void f() {
 size_t n;
 float_array fa (0); // (1)
 cin >> n;
 fa.resize(n); // (2)
    ~~~
}
```

(1) declares `fa` with an arbitrary size, 0, and then

(2) resizes `fa` after it knows the actual size that `fa` should be.

60

## Declarations as Statements

- You don't have to declare an object with phony initial values just to satisfy an overly restrictive rule.
- C++ lets you delay the declaration until you know enough to construct the object properly.
- C99 lets you do this, too.
- ✓ *Delay the declaration of local objects until just before the point of first use.*

61

## Declarations as Statements

- C++ extends the for-statement syntax to permit the declaration of the control variable in the initializing expression.
- For example, you can combine this:

```
int i;  
for (i = 0; i < n; ++i)
```

into this:

```
for (int i = 0; i < n; ++i)
```

- There is, however, a subtle difference in behavior...

62

## Declarations as Statements

- Here, the scope of `i` extends after the for-statement to the end of the enclosing block:

```
{
    int i;
    for (i = 0; i < n; ++i) {
        // loop body
    }
    // i is still in scope
}
```

- In Standard C++, the scope of `i` is just the for-statement, including its loop body...

63

## Declarations as Statements

```
for (int i = 0; i < n; ++i) {
    // loop body
}
```

- The above loop is equivalent to:

```
{
    int i;
    for (i = 0; i < n; ++i) {
        // loop body
    }
}
// i is no longer in scope
```

64

## Declarations as Statements

- Thus, you can reuse the name of a loop counter in a later loop in the same function body, as in:

```
for (int i = 0; i < n; ++i) {  
    // loop body  
}  
for (int i = 0; i < n; ++i) {    // OK: different i  
    // another loop body  
}  
for (i = 0; i < n; ++i) {        // error: i undeclared  
    // yet another loop body  
}
```

- C99 behaves this way, too.

65

## Constructor Execution Order

- Again, declarations are statements.
- They execute along with all the other surrounding statements in the order that they appear:
  - from top to bottom, and
  - from left to right.

66

## Constructor Execution Order

- For example, if you compile this function using the most recent version of `float_array` (with a constructor), ...

```
void f(size_t n) {
    size_t m;
    cin >> m;
    float_array fa (n), fb (m);
    size_t i = m + n;
    ~~~
}
```

67

## Constructor Execution Order

- ...it generates code equivalent to this code using the previous version of `float_array` (with a `create` member function):

```
void f(size_t n) {
 size_t m;
 float_array fa, fb;
 size_t i;
 cin >> m;
 fa.create(n);
 fb.create(m);
 i = m + n;
    ~~~
}
```

} *local objects without initializers*

68



## Storage Allocation

- Again, when it calls a function, the program automatically allocates storage for the shallow parts of the function's automatic objects.
- Compilers typically compute the total number of bytes needed for all the automatic objects in a function, and allocate them all at once.
- The allocation usually requires just a single instruction to add to or subtract from the stack pointer.

69

## Storage Allocation

- For example, on a typical 32-bit processor, `sizeof(size_t)` and `sizeof(any pointer type)` are both 4.
- `float_array` has two data members: a `size_t` and a pointer.
- Thus, `sizeof(float_array)` is 8.
- The compiler computes the stack space for automatic objects by totaling their sizes, as in:

```
void f(size_t n) {
    size_t m;           // 4
    float_array fa, fb; // + 8 + 8
    size_t i;           // + 4
    ~~~                 // = 24
}
```

70

## Destructor Execution Order

- If a function has more than one local object with destructors, it invokes the destructors for those objects in the reverse order that their constructors executed:

```
void f(size_t n) {
 float_array fa (n), fb (n);
    ~~~
    // ~float_array() applied to fb, then ...
    // ~float_array() applied to fa
}
```

- Why in reverse?

71

## Destructor Execution Order

- A constructor may accept another object as an argument, as in:

```
gadget g (10);
widget w (&g); // initialize w using g
```

- The program will construct g and use g to construct w.
- w's state depends on g, but g's state doesn't depend on w.
- Destroying g before destroying w leaves w in an uncertain state.
- Destroying w has no impact on g's state.
- Thus, destroying objects in reverse order of construction is appropriate.

72

## Non-Local Static Objects

- Objects declared at namespace scope have static storage duration, which is allocated:
  - when the program loads into memory, or
  - by the startup code executed as the program starts running.
- Many compilers execute constructors for non-local static objects at program startup.
- However, the C++ Standard lets programs delay initialization of a non-local static object until the first use of any static object in the same translation unit.
- Destructors for non-local static objects execute during program termination (after exiting `main`).

73

## Overloading Constructors

- When you define an object of a built-in type, such as `int`, you can initialize the object in different ways, such as:

```
int i;      // (1) no initialization
int j = 3;  // (2) initialization with a constant
int k = j;  // (3) initialization by copying
```

- If (1) occurs at block scope, it leaves `i` uninitialized.
- (2) initializes `j` with a constant value, namely 3.
- (3) initializes `k` by copying the value of `j`.

74

## Overloading Constructors

- It's often appropriate to offer similar choices for objects of class type.
- For example, when you define a `float_array` you might want to initialize it:
  - with no elements at all
  - with a fixed number of elements
  - by copying another `float_array`
- You can offer these choices by overloading the constructor...

75

## Overloading Constructors

- Here's class `float_array` with two more constructors:

```
class float_array {
public:
    float_array();                // (1) this is new
    float_array(size_t n);       // (2)
    float_array(float_array &fa); // (3) this is new
    ~float_array();              // (4)
    ~~~
};
```

- (2) is the original constructor and (4) is the destructor.
- (1) initializes a `float_array` so that it has no elements.
- (3) initializes a `float_array` to be a copy of another.

76

## Overload Resolution

- Overload resolution for constructors is the same as it is for other functions.
- For each definition whose object type has more than one constructor, the compiler selects (at compile time) the constructor whose formal parameter list is the best match for the actual argument list in the object definition.
- For example...

77

## Overload Resolution

- This definition passes 10 as the single argument to the constructor:  
  
`float_array fa (10);`
- Clearly, the compiler can't call `float_array()` because this constructor won't accept any arguments.
- Each of these constructors accepts one argument:

```
float_array(size_t);
float_array(float_array &);
```

- The compiler uses overload resolution to choose between them...

78

## Overload Resolution

- The object definition has the argument `10`, a signed int:
- The first constructor has a parameter of type `size_t`, an unsigned integer type.
- It's not an exact match for the unsigned type `size_t`.
- However, there is a conversion from signed int to `size_t`.
- The second constructor has a parameter of type "reference to `float_array`".
- A reference acts like the type it references.
- There's no conversion from signed int to `float_array`.
- Thus, `float_array(size_t)` is the better match.

79

## Overload Resolution

- Here, the constructor argument `fa` has type `float_array`:  
  
`float_array fb (fa);`
- There's no conversion from `float_array` to `size_t`.
- A `float_array` argument matches a "reference to `float_array`" parameter.
- In fact, it's considered to be an exact match.
- For a constructor argument of type `float_array`, overload resolution selects `float_array(float_array &)`.

80

## Default Constructors

- A constructor that you can call with no arguments is a **default constructor**:

```
class float_array {
public:
 float_array(); // default
 float_array(size_t n);
 float_array(float_array &fa);
 ~float_array();
    ~~~
};
```

81

## Default Constructors

- This doesn't declare a float\_array object using the default constructor:

```
float_array fc(());    // surprise! fc(void) is a function
```

- It declares a function that accepts no arguments and returns a float\_array.
- To define a float\_array object using the default constructor, you must omit the parenthesized argument list entirely, or...
- In C++11, you can use empty braces:

```
float_array fc;        // no parentheses at all, or
```

```
float_array fc {};    // empty braces (in C++11 only)
```

82

## Ambiguities

- As with other function calls...
- If the compiler can't find a unique best argument match, the constructor call is *ambiguous*.
- An ambiguous call is a compile error...

83

## Argument Conversions and Ambiguities

- For example, suppose class T is:

```
class T {  
public:  
    T(float);  
    T(double);  
    ~~~  
};
~~~  
T x (100);    // ambiguous
```

- The literal `100` is a signed int.
- The conversions `int`  $\Rightarrow$  `float` and `int`  $\Rightarrow$  `double` have the same rank.

84



## Copy Constructors

- The `float_array` class has a constructor that initializes one `float_array` with a copy of another:

```
float_array(float_array &fa);
```

- As you'll see, constructors that copy objects are a fundamental mechanism in C++.
- C++ has a special name for this kind of constructor.
- It's called a ***copy constructor***.

85

## Copy Constructors

- If you insist, you could implement a comparable constructor using a pointer parameter:

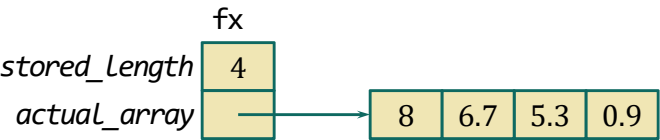
```
float_array(float_array *fa);
```

- This is ill-advised.
- Copy constructors with reference parameters are very much ingrained in C++.
- You should go with the flow.

86

## Shallow vs. Deep Copy

- There are two basic ways to copy objects:
  - A **shallow copy** copies only the shallow part of the source object to the shallow part of the destination object.
  - A **deep copy** copies the deep part as well as the shallow part.
    - As you'll see shortly, it's actually a bit more complicated than that.
- To illustrate the effect of each kind of copy, suppose float\_array fx looks like:



87

## Shallow vs. Deep Copy

- A shallow copy copies each data member of the source object to the corresponding data member of the destination object.
- For example, if the float\_array copy constructor makes a shallow copy, then:

```
float_array fy (fx);
```

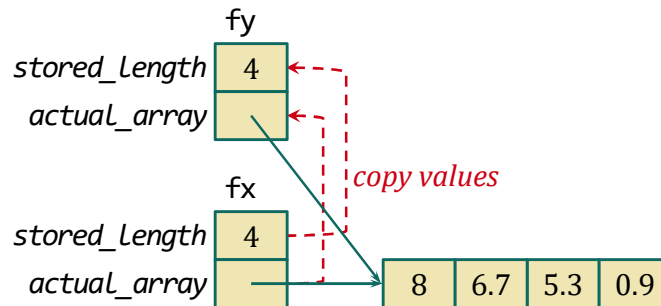
allocates storage for the shallow part of `fy` and initializes its members as if it executed these assignments:

```
fy.stored_length = fx.stored_length;  
fy.actual_array = fx.actual_array;
```

88

## Shallow vs. Deep Copy

- A shallow copy leaves `fx` and `fy` sharing the same dynamically allocated array:



- This raises serious questions about who really owns the array...

89

## Shallow vs. Deep Copy

```
void f(size_t n) {
    float_array fx (n);
    ~~~
 float_array fy (fx); // who owns the array?
    ~~~
    // ~float_array() applied to fy
    // ~float_array() applied to fx -> danger
}
```

- Exiting function `f` destroys `fy` and `fx`, in that order.
- Destroying `fy` deletes the array that `fx` is also using.
- Destroying `fx` deletes that array again. Ouch!

90

## Shallow vs. Deep Copy

- A properly written deep copy avoids the undefined behaviors that a shallow copy can cause.
- A deep copy replicates the entire structure of an object, the deep as well as the shallow part.
- For float\_array, a deep copy:
  - copies the stored\_length member from the source object to the destination object,
  - allocates a new array and places the array's address into the destination object's actual\_array, and
  - copies the source object's array elements to the new destination object's array.

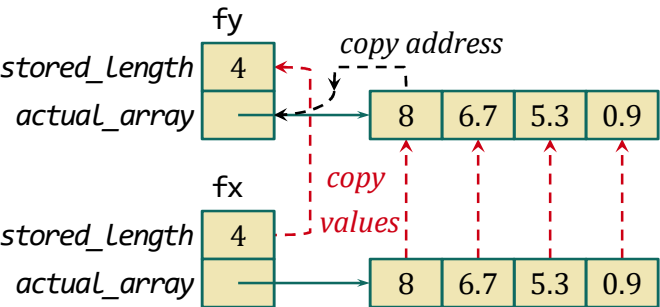
91

## Shallow vs. Deep Copy

- If the float\_array constructor makes a deep copy, then this:

float\_array fy (fx);

initializes fy with its own copy of the contents of fx:



92

### [03] Exercise: Overloading Constructors

- Add these constructors to class `float_array`:
  - `float_array()` — Initialize a `float_array` so that it's empty.
    - An empty `float_array` uses no free store memory.
  - `float_array(float_array &fa)` — Initialize a `float_array` by copying `fa`.
    - Use a deep copy rather than a shallow copy.
- Extend the test program to use all three constructors.
- Insert output statements into the constructors and destructor to verify that your program calls each constructor as well as the destructor.
- Can you demonstrate that the program destroys the `float_arrays` in the reverse of the order of construction?

93

### The Meaning of Privacy

- The copy constructor copies from one `float_array` to another:  

```
float_array(float_array &fa);
```
- Obviously, the constructor can't construct the target (`*this`) without accessing its private members.
- But, can it also access the private members of `fa`?
- In general:
  - Can a member function of class `T` access the private members of `T` objects other than the target?
  - Must it use other `T` public member functions to access the private members of other `T` objects?

94

## The Meaning of Privacy

- For example, the `float_array` constructor can initialize the `stored_length` member (of the target object) using:

```
stored_length = fa.length();      // (1)
```

- Can it use this instead?

```
stored_length = fa.stored_length; // (2)
```

- Another way to ask the question is:
  - Is privacy with respect to individual objects or entire classes?

95

## The Meaning of Privacy

- In C++, privacy is with respect to entire classes.
- A member function of class `T` can access the private members of `T` objects other than `*this`.
- Thus, you can use either:

```
stored_length = fa.Length();      // (1)
```

```
stored_length = fa.stored_length; // (2)
```

- Is there a reason to prefer one to the other?

96

## The Meaning of Privacy

- Some programmers argue that (1) is more robust:
  - There's one less change to make if you ever change the representation of `float_array`'s length.
- In fact, (1) doesn't really make the program more robust:
  - If the representation of the length changes, you still have to rewrite the statement anyway.
- (2) has the advantage that it shows very clearly what the constructor is doing:
  - It's initializing the `stored_length` member by copying the corresponding member of another object of the same type.
  - (1) obscures this.

97

## Subscripting Errors

- `float_array`'s `get` and `set` functions don't verify that the subscript (the function parameter) is in the range of indices for the current object.
- How should they react to a subscript range error?
- Ignore it (as they do now)?
  - This is fast, but not very safe.
- Other possibilities are...

98

## Subscripting Errors

- Terminate the program, with an explanation?
  - It's often better to notify the user that the program's output is questionable rather than blithely act as if everything's okay.
  - However, some real-time systems (such as life-support systems or nuclear power plants) can't just shut down.
- Throw an exception (once you learn how to do it)?
  - This is often a good approach, provided the program catches the exception somewhere.
  - Otherwise, the program just halts.
- Enlarge the array?
  - This is appropriate for variable-length character strings.
  - It's the basis for array operations in languages such as APL.

99

## Subscripting Errors

- There may be times when each of these responses is appropriate.
- It's your job as class designer to make the class do the "right" thing.

100



## Assigning vs. Initializing

- Let's look at ways to extend the utility of `float_array` by adding another fundamental operation:
  - Copying one object to another of the same type.
- For example, the call below copies `fx` to `fy`:

```
float_array fx, fy;
~~~
fy.copy(fx); // think of this as fy = fx
```

101

## Assigning vs. Initializing

- `copy` is a member of class `float_array` declared as:
 

```
void copy(float_array &fa);
```
- It's very similar to the copy constructor:
 

```
float_array(float_array &fa);
```
- However, `copy` is a distinct operation:
  - `copy` **replaces** the value of a previously constructed object with a new value.
  - A constructor **initializes** an object that has no prior value.

102

## Shallow vs. Deep Copy, Again

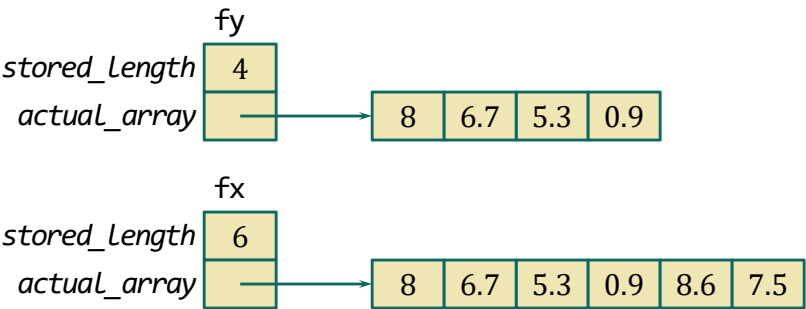
- As with the constructor, `copy` can employ a shallow copy or a deep copy.
- Here's the shallow version:

```
void float_array::copy(float_array &fa) {
 stored_length = fa.stored_length;
 actual_array = fa.actual_array;
}
```

- Again, a shallow copy can leave two or more `float_array` objects competing for the same dynamic storage.
- Subsequent destructor calls may corrupt the free store.

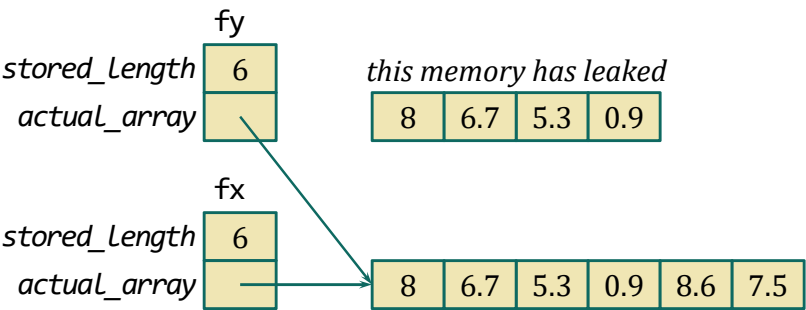
## Shallow vs. Deep Copy, Again

- This shallow copy also causes a memory leak.
- For example, suppose the copy operands look like:



## Shallow vs. Deep Copy, Again

- After a shallow copy from fx to fy, the objects look like:



105

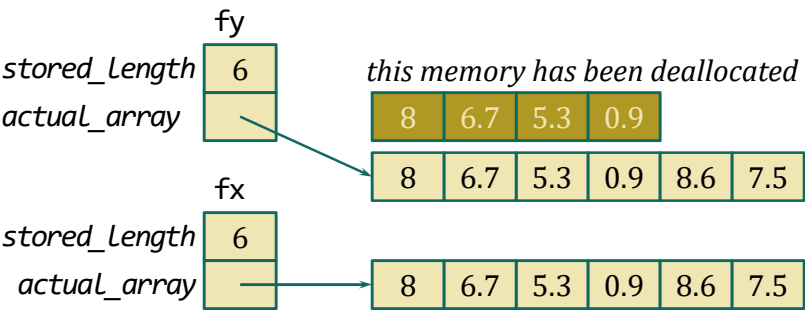
## Shallow vs. Deep Copy, Again

- The array originally addressed by `fy.actual_array` hasn't been returned to the free store, yet no pointer points to it any more.
- Thus, it can never be returned to the free store.
- It's a memory leak.
- Again, you can avoid these problems by using a deep copy instead of a shallow copy.
- A deep copy replicates the entire structure of an object...

106

### Shallow vs. Deep Copy, Again

- After a deep copy from fx to fy, the objects look like:



107

### [04] Exercise: Copying Objects

- Add the following member function to the `float_array` class :

```
void copy(float_array &fa);
```

so that this copies the contents of `fx` to `fy` using a deep copy:

```
fy.copy(fx);
```

- After making the copy, `fx` and `fy` should own different actual arrays.
- Those arrays should have the same number of elements and the same values in corresponding elements.
- More...

108

## [04] Exercise: Copying Objects

- Of course, copy should not cause any memory leaks.
- Think about what happens when you copy a `float_array` to itself, as in:

```
fx.copy(fx);
```

- Make sure that your copy function is well behaved in this case.

109

## Guaranteed Initialization

- The primary reason that C++ supports constructors is that:
- ***Constructors provide guaranteed initialization.***
- That is, C++ guarantees (within reason) that if a class has any constructors, then programs must apply one of those constructors whenever they create an object of that class.

110

## Guaranteed Initialization

- We have already seen that a program may call a constructor:
  - when it creates an automatically allocated object as it enters a function, and
  - when it creates a statically allocated object at (or near) program startup.
- A program might also call a constructor:
  - when it creates an object via a new-expression,
  - when it creates an object as an array element, and
  - when it creates an object as a data member of a larger object.
- Let's examine these last three situations in detail...

111

## New-Expressions and Constructors

- Suppose T is a class type with a default constructor:

```
class T {
public:
 T();
    ~~~  
};
```

112

## New-Expressions and Constructors

- The new-expression below now does more than allocate storage for a T object:

```
p = new T;
```

- It also initializes that storage using T's default constructor.
- In contrast, calling malloc merely allocates storage:

```
p = static_cast<T*>(malloc(sizeof(T)));
```

- Using malloc and a cast is one way to defeat the initialization guarantee.

113

## New-Expressions and Constructors

- Now, suppose T has a constructor, but not a default constructor:

```
class T {  
public:  
    T(int);    // non-default constructor  
    ~~~~~  
};
```

114

## New-Expressions and Constructors

- The new-expression below is an error:

```
p = new T; // error: no default constructor
```

- T's constructor requires an argument, which the new-expression doesn't provide.
- In this case, you must provide a parenthesized argument list after the type name in the new-expression, as in:

```
p = new T (v); // v is the constructor argument
```

- This new-expression passes v as the argument to the constructor for the newly allocated T object.

115

## New-Expressions and Constructors

- In C++11, the initializer can be enclosed in braces:

```
p = new T {v}; // in C++11, but not in C++03
```

- As in a class object definition, the compiler uses overload resolution:
  - It selects the constructor whose parameters are the best match for the arguments in a new-expression.
  - If there's no unique best match, the compiler rejects the new-expression.
- For example...

116



## New-Expressions and Constructors

```
class T {
public:
 T(); // default constructor
 T(int i, double d);
    ~~~  
};  
  
int i;  
~~~  
p = new T; // OK: uses T()
q = new T (i, 4.5); // OK: uses T(int, double)
r = new T (i); // compile error: no match
```

117

## Constructors and Arrays

- Once again, if T is a class with a constructor, then every time a program creates a T object, it must apply a constructor to that object.
- This array definition creates N objects of type T:

```
T x[N]; // applies default constructor to each element
```

- Thus, executing the definition calls a constructor for each array element.
- The array definition doesn't supply constructor arguments, so the definition calls the default constructor for each element.

118

## Constructors and Arrays

- Similarly, the array new-expression in:

```
p = new T [N];
```

does both:

- allocates storage for an array of T with N elements, and
  - applies T's default constructor to each element.
- The construction of an array invokes the constructors for its elements (if any) in order from the first (0-th) element to the last.

119

## [05] Constructors and Arrays

- When you define an array object with elements of class type, you can supply constructor arguments in a brace-enclosed initializer list, as in:

```
class T {
public:
 T(int i);
    ~~~
};
~~~
```

```
T v[] = { 1, 4, 9 };
```

120

## [05] Constructors and Arrays

- This defines `v` as an array of three elements of type `T`:

```
T v[] = { 1, 4, 9 };
```

- `v[0]` is initialized by calling the constructor `T(int)` with 1 as the argument.
- `v[1]` is initialized by calling the constructor `T(int)` with 4 as the argument.
- `v[2]` is initialized by calling the constructor `T(int)` with 9 as the argument.

121

## Constructors and Arrays

- There's no notation for supplying a single constructor argument list to be used for all elements in one array.
- Thus, try as you might, these won't compile:

```
T x[N] (v); // syntax error
```

or:

```
p = new T [N] (v); // syntax error
```

122

## [05] Constructors and Arrays

- In C++03, an array definition without a brace initializer can initialize array elements of class type only by calling a default constructor.
- Similarly, an array new-expression can initialize array elements of class type only by calling a default constructor.
- C++11 provides a new syntax for initializing arrays which overcomes some of these limitations.
- For example, an array-new-expression in C++11 can have a brace initializer, as in:

```
T *p = new T [4] { 16, 25, 36, 49 };
```

123

## Constructors and Member Objects

- A class can have data members of (other) class types.
- For example, here's a class for little objects:

```
class little {
public:
 little();
    ~~~  
private:  
    int i;  
    char *p;  
};
```

124

## Constructors and Member Objects

- And here's a class for **big** objects containing **little** objects:

```
class big {
public:
    big(int k);
    ~
private:
    little m;    // big has a little data member
    int n;       // it also has an int data member
};
```

- Every time a program constructs a **big** object, it must construct the **little** object inside.

125

## Constructors and Member Objects

- Suppose the definition for **big**'s constructor looks like:

```
big::big(int k) {
    n = k;
}
```

- This constructor initializes member **n** but doesn't mention member **m** (of type **little**) at all.
- Even though it appears that **big**'s constructor doesn't initialize **m**, in reality it calls **little**'s default constructor applied to **m**.
- That's why the default constructor is so named.
- It's the one that the program calls by default.

126

## Constructors and Member Objects

- In general, every one of the constructors for a class T calls a constructor for each of T's member objects whose type has a constructor.
- What happens if class T has no constructors?
- If a class T has members with constructors, but T itself declares no constructors, then the compiler generates a default constructor for T that invokes the default constructor for each of T's members with constructors.

127

## Generated Default Constructors

- The way the previous rule is phrased suggests that some types don't have constructors.
- Which types?
  - Scalar (arithmetic, enumeration or pointer) types don't have constructors.
  - Neither do classes with members of only scalar types and no explicitly declared constructors.

128

## Generated Default Constructors

- Actually, you get a simpler statement of the rules for constructors if you assume that all types, including scalar types, have constructors...
- Scalar types have *trivial default constructors*, which do nothing.
- Then it follows that...

129

## Generated Default Constructors

- If a class T doesn't declare any constructors, then the compiler generates a default constructor for T that invokes the default constructors for T's members.
- A constructor that does nothing except invoke trivial constructors is itself trivial.
- The compiler may "optimize away" trivial constructors.
- That is, a compiler may elect to generate no code for a call to a trivial constructor.
  - A good optimizing compiler will do just that.

130

## Member Initializers

- Now, suppose `little`'s only constructor requires arguments that initialize `little`'s members:

```
class little {  
public:  
    little(int, char *);    // non-default constructor  
    ~~~~~  
private:
 int i;
 char *p;
};
```

- In this case, the compiler doesn't generate a default constructor.

131

## Member Initializers

- Class `big` still has a `little` member:

```
class big {
    ~~~~~  
private:  
    little m;  
    int n;  
};
```

- If you don't write any constructors for `big`, the compiler tries to generate a default constructor for `big` that applies `little`'s default constructor to member `m`.
- However...

132



## Member Initializers

- `little` doesn't have a default constructor, so the program fails to compile.
- You must write a `big` constructor, which tells the compiler to initialize `big`'s member `m` using a `little` constructor other than the default constructor.
- For that, you need an additional language construct called a ***member initializer***...

133

## Member Initializers

- For example, this constructor for class `big` has a member initializer:

```
big::big(int k):
    m (0, nullptr) {
    n = k;
}
```

*parameter list* points to `(int k)`

*member initializer* points to `m (0, nullptr)`

*function body* points to `{ n = k; }`

- The member initializer `m (0, nullptr)` specifies that this `big` constructor initializes member `m` using a `little` constructor that accepts `0` and `nullptr` as arguments.

134

## Member Initializers

- A member initializer can appear only between the parameter list and the function body in a constructor definition.
- A : (colon) separates the member initializer from the constructor's parameter list.
- A constructor for a class with more than one member can specify more than one member initializer, separated by commas.
- Once again, C++ strives to treat built-in types and user-defined types the same.
- Thus, you can initialize members of non-class types in a member initializer list.
  - In fact, this is often good practice.

135

## Member Initializers

- For example, given the previous definition for class `big`, these two constructors are equivalent:

|                                                                              |                                                                      |
|------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <pre>big::big(int k):     m (0, nullptr) {         <b>n = k;</b>     }</pre> | <pre>big::big(int k):     m (0, nullptr), <b>n (k)</b> {     }</pre> |
| <p><i>These are equivalent</i></p>                                           |                                                                      |

- The constructor on the right has two member initializers separated by a comma.

136

## Member Initializers

- The member initializer, `n (k)`, initializes member `n` with value `k`.
- The compiler recognizes that `n` is just an `int`, so initializing `n` with `k` simply means “assign `k` to `n`”.
- It generates the same code as the assignment `n = k` in the body of the constructor.

137

## Member Initializers

- You can think of the member initializer `n (k)` as specifying a call to a constructor for `n`, passing the argument `k`.
- This extends the notion that scalar types, not just class types, have constructors.
- In general, you can program *as if*:
  - Every scalar type has a trivial default constructor that leaves the target object uninitialized.
  - Every scalar type has a constructor with a single parameter of the same scalar type; it assigns the parameter value to the target object.

138

## Member Initializers

- In fact, the notion that “scalars have constructors” extends to let you rewrite a declaration such as:

```
int n = v;
```

as:

```
int n (v);
```

- This may be carrying the generality a little far, but it is C++.

139

## Again with the Brace Initializers

- In C++11, you can use braces to enclose initializers in constructors, as in:

```
big::big(int k):  
    m {0, nullptr}, n {k} {  
}
```

- You can also use braces to initialize scalars:

```
int n {v};
```

- Again, using braces prohibits narrowing conversions.

140

## Member Initializers

- One of the earlier rules regarding constructors was that:
  - Every one of the constructors for a class T calls a constructor for each of T's member objects whose type has a constructor.
- Here's a more accurate and complete statement that incorporates that notion that even members of non-class types have "constructors":
  - Every one of the constructors for a class T calls a constructor for each of T's member objects.
  - The constructor invoked for a member with a member initializer is the constructor whose parameters best match the arguments in that member initializer.
  - The constructor invoked for a member without a member initializer is its default constructor.

141

## Member Initializers

- And here's something that may surprise you...
- The constructors for the members of a class execute in the order in which the members are declared from top to bottom and left to right in the class definition.
- That is, the compiler doesn't necessarily execute the member constructors in the order in which the member initializers appear in a constructor.

142

## Member Initializers

- For example, recall this constructor:

```
big::big(int k):  
    m (0, nullptr), n (k) {  
}
```

- `m`'s member initializer appears before `n`'s member initializer.
- This doesn't mean that `m` will be initialized before `n`.
- Why is that?

143

## Member Initializers

- A class `T` can have many constructors, but only one destructor.
- The destructor destroys `T`'s members in the reverse order that they were constructed.
- Thus, every constructor for `T` must construct `T`'s members in the same order.
- The compiler can't rely on the order of the member initializers, because each of `T`'s constructors can specify the member initializers in a different order.
- Thus, the compiler uses the declaration order of the members in the class definition as the canonical order for constructing the members.

144

## Member Initializers

- Class `big` declares member `m` before member `n`:

```
class big {  
    ~~~  
private:
 little m;
 int n;
};
```

- Thus, every `big` constructor will construct its `m` member before constructing its `n` member.
- The order of the member initializers in each constructor doesn't matter.

145

## [06] Exercise: Member Initializers

- Go back to the last version of `xr` (the cross-reference generator) and replace the explicit initialization functions `xrt::init` and `lns::init` with constructors.
- ***DO NOT change any other behavior of `xrt` or `lns`.***
- For example, when you initialize an `lns` using `lns::init`, you must pass the value of the first number in the set.
  - `lns` doesn't allow an empty set.
- ***Thus, the `lns` constructor should also require that you pass it a first line number.***
- More...

146

## [06] Exercise: Member Initializers

- Hint:
  - `lns` has only one constructor, and that constructor has one parameter.
  - `lns` objects appear only as members of `entry` objects.
  - Therefore the `entry` class needs a constructor whose initializer list initializes the `lns` member of the `entry`.
  - Remember, you can treat `entry` like a class even though its declaration uses the keyword `struct`.
  - A `struct` can have constructors, just like a class.
  - If you prefer, replace the keyword `struct` with `class`.

147

## [07] Exercise: More Member Initializers

- If you didn't do so as part of the previous exercise, add a constructor to the `list_node` structure.
- Use member initializers in that constructor to initial the structure's members.

148



## Member Initialization

- Consider a class `widget` defined as:

```
class widget {
public:
 widget();
 widget(int v);
 widget(int v, int w);
private:
 int m;
 int n;
 int p;
};
```

- Suppose you want every constructor to initialize `p` to 0...

149

## Member Initialization

- In C++03, every `widget` constructor must have a member initializer that initializes `p` to 0, as in:

```
widget::widget():
 m (0), n (0), p (0) {
}

widget::widget(int v):
 m (v), n (0), p (0) {
}

widget::widget(int v, int w):
 m (v), n (w), p (0) {
}
```

150

## Member Initialization

- In C++11, you can put an initializer in the declaration of member p, as in:

```
class widget {
public:
    ~~~
private:
    int m;
    int n;
    int p = 0;      // alternatively: int p {0};
};
```

- This value becomes the default initial value for member p...

151

## Member Initialization

- Any constructor that lacks a member initializer for p will initialize p with the value specified in p's declaration:

```
widget::widget():
    m (0), n (0) {          // p initialized to 0
}

widget::widget(int v):
    m (v), n (0) {          // p initialized to 0
}

widget::widget(int v, int w):
    m (v), n (w) {          // p initialized to 0
}
```

152

## In-Class Member Initializers

- You can use an initializer with every member, as in:

```
class widget {
public:
    ~~~
private:
 int m = 0; // or: int m {0};
 int n = 0; // or: int n {0};
 int p = 0; // or: int p {0};
};
```

- This can simplify all the constructors...

153

## In-Class Member Initializers

- Any constructor that lacks a member initializer for p will initialize p with the value specified in p's declaration:

```
widget::widget() { // same as: m (0), n (0), p (0)
}

widget::widget(int v):
 m (v) { // same as: m (v), n (0), p (0)
}

widget::widget(int v, int w):
 m (v), n (w) { // same as: m (v), n (w), p (0)
}
```

154

## [08] Exercise: In-Class Member Initializers

- Rewrite the *xr* program to use in-class member initialization where appropriate.
- Hint: Look in the `xrt` and `list_node` classes.

155

## Static Data Members

- Suppose you want to count the number of `Ins` objects that your program creates.
- Constructors provide a hook that lets you do this pretty easily.
- Simply add a statement to the `Ins` constructor to increment a counter:

```
Ins::Ins(unsigned n) {
    ~~~  
    ++counter;  
}
```

156

## Static Data Members

- Thus far, class `Ins` doesn't have a destructor, but you can easily define one that decrements the counter:

```
Ins::~~Ins() {  
    --counter;  
}
```

- As such, the counter accurately indicates the number of `Ins` objects that the program has created, but not yet destroyed.

157

## Static Data Members

- Where do you declare the counter?
- It can't be an ordinary `Ins` data member, because then you'd have a counter in every `Ins` object.
- It could be a global variable (Boo! Hiss!).
  - We try to avoid them because they introduce implicit coupling.

158

## Static Data Members

- Rather, the counter should be a **static data member**, as in:

```
class lns {  
public:  
    lns(unsigned n);  
    void add(unsigned n);  
    void put();  
    static unsigned counter;  
private:  
    list_node *first, *last;  
};
```

159

## Static Data Members

- For a class T with static data member m, there's only one copy of m and all the T objects in the program share it.
- Static members have external linkage.
- Like ordinary members, static members are subject to access control.
- That is, static members can be public or private.

160

## Static Data Members

- A static member is in the scope of its class.
- Inside a member function, you can refer to a static data member by its unqualified name.
  - Just as you do with ordinary members.
- Outside a member function, you must qualify a static member with its class name, as in:

```
if (lns::counter > 0) {
}
```

- In this example, `lns::counter` is public, so it can appear outside the class.
- If it were private, this would be an access violation.

161

## Static Data Members

- If you have an `lns` object handy, say `s`, then you can also access `counter` as if it were an ordinary (non-static) data member, as in:

```
if (s.counter > 0) {
}
```

- In this case, the compiler doesn't do anything with `s` other than look up `counter` in the class of `s`.

162

## Static Data Members

- The declaration of a static data member within a class is just a declaration:

```
class Ins {  
public:  
    ~~~  
 static unsigned counter; // not a definition
    ~~~  
};
```

- The definition must appear elsewhere...

163

## Static Data Members

- Each static data member must be initialized somewhere outside the class definition with a definition such as:

```
unsigned Ins::counter = 0;
```

- A good place to put this definition is in a source file (not a header) near the definition(s) for the constructor(s) or destructor for the member's class.

164



## Static Member Functions

- There's no harm in letting code outside `Ins` inspect the counter.
- But you don't want outsiders changing the counter at will.
- Therefore, you should declare counter private and provide access to it through a public member function `how_many`:

```
class Ins {  
public:  
    ~~~  
 unsigned how_many();
private:
    ~~~  
    static unsigned counter;  
};
```

165

## Static Member Functions

- Even a private static data member must be initialized somewhere outside the class definition with a definition such as:

```
unsigned Ins::counter = 0;
```

- This definition doesn't violate the encapsulation of the class any more than the definition of any private member function appearing outside the class.

166

## Static Member Functions

- If you declare `how_many` as a member function, you can call it only when it appears:
  - as a qualified name (to the right of `x.` or `p->` where `x` is an `Ins` or `p` is an `Ins *`), or
  - inside a member of class `Ins` (in which case the compiler interprets `how_many` as `this->how_many`)
- In either case, you need an `Ins` object handy just to ask if there aren't any `Ins` objects.
- For example:

```
Ins dummy;
~~~
if (dummy.how_many() == 0) { // a silly question
```

167

## Static Member Functions

- The way to avoid this silliness is to declare `how_many` as a ***static member function***, as in:

```
class Ins {
public:
 Ins(unsigned n);
 void add(unsigned n);
 void put();
 static unsigned how_many();
private:
 list_node *first, *last;
 static unsigned counter;
};
```

168

## Static Member Functions

- Then define `lms::how_many` in the usual way:

```
unsigned lms::how_many() {
 return counter;
}
```

- Outside the scope of class `lms`, you typically refer to `how_many` by its qualified name, as in:

```
if (lms::how_many() == 0) { // more like it
    ~~~
}
```

169

## Static Member Functions

- If you have an `lms` object handy, say `s`, then you can also call `how_many` as if it were an ordinary (non-static) member, as in:

```
if (s.how_many() > 0) {
    ~~~
}
```

- The compiler doesn't do anything with `s` other than look up `how_many` in the class of `s`.
- It doesn't pass `&s` as `this`.

170

## Static Member Functions

- Unlike ordinary (non-static) member functions, static member functions:
  - don't have a target object and
  - can't refer to the keyword `this`.
- However, a static member function is a member function in every other respect.
- A static member function of class `T`:
  - can refer to `T`'s static members (functions and data, public and private) without qualification.
  - can refer to any non-static class member `m` of `T`, but only via an expression of the form `x.m` or `p->m` (where `x` or `*p` is a `T`).

171

## [09] Exercise: Static Members

- To the cross-reference program, `xr`, add code that counts the number of objects that the program creates.
- Do this for all class types, including those defined as structures.
- In particular, add a counter to each class declared as a private static data member.
- Add statement(s) to the constructors to increment and display the counters.
- Don't worry about decrementing the counters in destructors.
- That's coming up.

172

## Perspectives on Destructors

- Thus far, the cross-referencing program, *xr*, allocates memory from the free store but never releases it.
- The memory remains allocated until the program terminates.
- *xr* can get away with this because it terminates as soon as it's done with that memory.
- At termination, the operating system recovers the free store memory along with all the other memory (code and data) allocated to the program.

173

## Perspectives on Destructors

- On the other hand, if *xr* did other work after it finished with the cross-reference table, it should release the memory it no longer needs.
- In general, each object should release resources it no longer needs so that other objects can reuse those resources.
- The right place to do this is in destructors.

174

## Destructors and Member Objects

- Let's define a destructor for class `entry` to complement its constructor.
- First, declare the destructor in the class definition:

```
struct entry {
 entry(char *w, unsigned n);
 ~entry();
 char *word;
 lns lines;
 static unsigned counter;
};
```

175

## Destructors and Member Objects

- Then write the destructor definition.
- Consider what happens if that definition is just:

```
entry::~~entry() {
 --counter;
} // lns::~~lines() called implicitly
```

- Once again, `entry` has a member object `lines` of type `lns`.
- If `lns` has a destructor, you don't have to do anything to invoke the `lns` destructor for `lines`.
- The compiler automatically generates a call to that destructor from inside the `entry` destructor.

176

## Destructors and Member Objects

- In general, when you write a destructor for any class T, you don't explicitly call the destructors for T's non-static data members.
- The compiler generates those calls for you.
- The destructors for T's non-static data members execute after the body of T's destructor.
- Those member's destructors execute in reverse order of the member's declaration within T.

177

## Generated Destructors

- C++ has rules for generating destructors that are very similar to the rules for generating constructors.
- In truth, scalar types don't have destructors, but the rules are simpler if we presume they do.
- Those rules for generating destructors are...

178

## Generated Destructors

- Scalar types have *trivial destructors*, which do nothing.
- A compiler can “optimize away” code for trivial destructors.
- If a class T doesn’t explicitly declare a destructor, then the compiler generates one.
- The destructor invokes the destructor for each of T’s non-static data members.
- A destructor that does nothing except invoke trivial destructors is itself trivial.

179

## Generated Destructors

- Pointer types are scalar types.
- Pointer objects have trivial destructors, which do nothing.
- The most common reason for writing a destructor for a class T is to delete T’s pointer members.
- For example, class `float_array` has a destructor defined as:

```
float_array::~float_array() {
 delete [] actual_array;
}
```

- Without this destructor, the dynamic storage used by `float_array` objects would leak.

180



## Delete-Expressions and Destructors

- Just as a new-expression may call a constructor, a delete-expression may call a destructor.
- For example, if class T has a destructor and p is a pointer to a T object, then this invokes T's destructor:

```
delete p;
```

- It calls the destructor just before it deallocates the storage addressed by p.

181

## Delete-Expressions and Destructors

- If p points to an array of T objects acquired by a new-expression such as:

```
p = new T [n];
```

then:

```
delete [] p;
```

invokes T's destructor for each array element before returning the array's storage to the free store.

182

## Delete-Expressions and Destructors

- Generated destructor calls always execute in the reverse order of the corresponding constructor calls.
- Therefore,

```
delete [] p;
```

destroys *p*'s elements in descending order, from the element with the highest subscript to the element with the lowest.

183

## [10] Exercise: Releasing Resources

- Add destructors to the classes in *xr* so that the program returns all the allocated storage to the free store before termination.
- Use the previous version of *xr* that uses static data members to count the number of objects created.
- Add statements to each destructor to decrement (and display) the corresponding counter.
- You should see all counters return to zero by the end of program execution.

184