The C++ Answer Book

Tony L. Hansen

The C++ Answer Book

Tony L. Hansen
AT&T Bell Laboratories
Lincroft, New Jersey

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

Library of Congress Cataloging-in-Publication Data

```
Hansen, Tony L.
   The C++ answer book.

   Bibliography: p.
   Includes index.
   1. C++ (Computer program language)   I. Title
QA76.73.C15H3685    1990                 005.13'3    88-7515

ISBN 0-201-11497-6
```

This book was typeset in Times Roman and Courier by the author, using a Mergenthaler Linotronic phototypesetter driven by an AT&T 3B2/700 running UNIX® System V.

ABCDEFGHIJ-MA-89

Foreword

A common problem for people wanting to learn C++ is to get hold of a body of C++ code to read. To be useful, such a body of code must be of sufficient quality to make it worth reading and must contain sufficient commentary to make it reasonably obvious what is being done, how it is being done, and why it is being done.

The C++ Answer Book provides such a body of code in the form of heavily annotated solutions to the exercises in *The C++ Programming Language*. The exercises vary in difficulty from something a novice can do in a minute to something an experienced programmer needs weeks to accomplish. The problems and their solutions grow in complexity and sophistication to match the student's increasing grasp of the language and its associated programming techniques.

Tony Hansen's solutions are based on a thorough grasp of the C++ programming language and a solid background in real-life software construction. They are also tested on several types of machines and operating systems (8086 MS-DOS based, 6386 UNIX based, DEC Vax, AT&T 3B2, MIPS 1000 and Amdahl 5870) using several different C++ compilers.

Naturally, some examples are just exercises to test a students grasp of language features but in several cases the solution is a complete library of a quality rarely surpassed in current production code. For example, there are classes for extended-precision and arbitrary precision integer artithmetic and a library of classes for co-routines and event-driven simulation.

In addition to the solution to the exercises, *The C++ Answer Book* contains sections describing the language features introduced into C++ since the publication of my book.

The C++ Answer Book can be an important compendium to *The C++ programming Language*. It adds depth, detail, and realism to the presentation of the language features and programming techniques and provides up-to-date information about the language features found in the latest implementations of C++.

Murray Hill, February 1989 Bjarne Stroustrup

Acknowledgments

I would like to thank Jim DeWolf, senior computer science editor at Addison-Wesley, and John Thompson, his assistant, who gave considerable help and encouragement throughout this project. Most of all, they were very patient. I'd also like to thank all of the people who have reviewed portions or all of this book. In particular, thanks go to Avi Gross, who endured a number of drafts, Glenn Bradford, James Coplien, Margaret Ellis, Ron Hiller, Mark Horton, Andrew Koenig, Chris Macey, Jonathan Shopiro, and Kathy Stark of AT&T, Keith Gorlen and John Shaw of the National Institute of Health, Mark Linton of Stanford University, and Mike Jones of Carnegie Mellon University. Most of all, I would like to thank Bjarne Stroustrup for creating C++, his encouragement while I was writing this book, and his many comments and suggestions.

Contents

Chapter 1

A Tour of The C++ Answer Book

This book is a companion to *The C++ Programming Language* by Bjarne Stroustrup [Stroustrup 1986], hereafter referred to as the C++ book.

Why an Answer Book?

While learning a new computer programming language, one typically goes through many steps before being able to write one's own programs, including

- read the language's manual (the C++ book),
- study existing programs,
- speak with others who have written programs in that language,
- compare solutions with some reference, and
- learn the paradigms, idioms, techniques and style of the language

It is assumed that the reader is in the process of completing the first step of this list, reading the C++ book. Since C++ is such a new language, it is difficult to complete the other steps – there are so few C++ programs available for perusal and so few experienced C++ programmers. Of course, many C programs and programmers are available, and much can be learned from them since C++ is essentially a superset of that language. However, that does not help in learning the new features of C++.

The C++ book includes a large number of exercises at the end of each chapter. This book consists of a discussion of and solution to each of those exercises. It is expected that the reader will work on each exercise before looking at the solution presented here. Once the reader has done that, s/he should read through the author's solution and discussion. After comparing the solutions, the reader should look for reasons for any differences.

The reader must realize that for each exercise, there is no *one right* answer. If the reader's solution is significantly different from the author's, the reader should look closely at the differences and then try to understand how both solutions work.

By working on the exercises and looking at the solutions presented here, the reader will be helped in the remaining steps of learning the exciting new language of C++. The end result for the reader is to become a better C++ programmer more quickly.

The Exercises and Solutions

Each exercise begins with a copy of the exercise text from the C++ book, followed immediately by the solution to that exercise. Also included is the estimate (from the C++ book) of the difficulty of the exercise. The scale is exponential – it is expected that a (*1) exercise might be solvable in 5 minutes, a (*2) exercise in an hour, a (*3) exercise in a day, a (*4) exercise in a week, and a (*5) exercise in a month. Of course, the reader's mileage may vary, depending on prior experience and knowledge.

Each of the solutions uses only the features of the language that have been introduced up to that point. For example, the solutions prior to Chapter 7 do not make use of derived classes. Improvements are sometimes suggested that make use of features introduced in a subsequent chapter.

All section and page references are relative to the C++ book – for example, §4.3.2. Sections in the Reference Manual, found at the end of the C++ book, are denoted by a lowercase *r* – for example, §r.2.4.1. Because of the wide variety of exercises, some short and others considerably longer, there is a wide variety of solutions. The shorter exercises are generally meant to illustrate one or two points, while the longer exercises are generally meant to give practical experience writing real or almost-real modules. Consequently, the solutions to the shorter exercises contain more discussion than code, while the solutions to the longer exercises include extensive examples of working code.

Attention is paid to the efficiency of various alternative solutions, as well as their portability and machine independence. Some discussion is also given regarding when it is appropriate to think about efficiency, and pointers are given to further texts on the subject.

All solutions have been tested using compilers based on the AT&T cfront Translator, releases 1.2 and 2.0,[1] and run on a variety of machines running UNIX® System V releases 2 and 3, and MS®-DOS. Changes necessary to run the programs on other systems, such as the Berkeley UNIX systems, are noted where necessary.

While it is almost impossible to guarantee that the test cases have covered all possibilities, every attempt has been made to ensure that the code shown here compiles and runs correctly. To make certain that no mistakes were introduced while moving the code into the manuscript, the code for the programs was electronically included directly from the program text, with no manual copying performed.

1. At publication, tests were being performed on the Zortech C++ Compiler, release 1.51.

Dependencies on C

C++ depends on a C library for most of its supporting functionality. The related support libraries, however, differ from system to system. Unless otherwise indicated, all of the solutions presented here use only those functions listed in the proposed ANSI C Standard [ANSI 1988][2] and the draft IEEE Portable Operating System Environment (also known as POSIX) [IEEE 1988]. Notable differences from other major versions of the C language and its library are mentioned where appropriate.

The Evolution of C++

Since C++ is an evolving language, several modifications have been made to the language since the publication of the C++ book. Appendix A introduces the new features added to the language up to the publication date of this book. Some of these features are due to additions to the language and others are due to a better definition of the C language becoming available. With only a few minor exceptions, C++ is now considered a superset of ANSI standard C (see Appendix A for details).

Although the author uses a compiler for a newer version of the language than that described in the C++ book, none of the newer features of the language are used for the initial answers to the exercises. Where appropriate, comments are made about the use of the newer features of the language, or how newer features may affect the solution presented. Sometimes a solution is presented again using some of the newer features.

The Appendices

As just mentioned, Appendix A covers the additions made to the language since the C++ book was written. Appendix B describes a few header files introduced and used in the book, such as `<swap.h>`.

2. Although the ANSI C Standard is still a proposed standard, it is sufficiently close to publication that the ANSI C features discussed in this book are unlikely to change.

Chapter 2

Declarations and Constants

Exercise 2.1 (*1)

Get the "Hello, world" program (§1.1.1) to run.

Creating Your First Program

The first step is to create the file that contains the program. Many excellent books are available which show how to create, edit and manipulate files on the various operating systems. The reference list shows some of the books on the market for the UNIX Operating System. The file would be named something like *hello.c*, *hello.C*, *hello.cc*, *hello.cpp* or *hello.cxx*, depending on the conventions chosen by the compiler that is being used.[1]

The file hello.c should look like:

```
#include <stream.h>
main()
{
    cout << "Hello, world\n";
}
```

On the author's UNIX system, one would type:

```
$ CC hello.c -o hello
```

CC is the command to invoke the C++ compiler. `-o hello` says to stored the compiled program into a file named `hello`. To run the program, one would then type:[2]

1. Naming C++ files with .c and .C is the convention most commonly used on UNIX Systems. Naming files with .cpp or .cxx is the convention chosen by two different MS-DOS compilers. The reader should consult the documentation for his/her compiler to find the proper convention. Similarly, header files are conventionally named using .h, .hpp and .hxx suffixes; this book will only use the .h suffix.
2. Depending on the setting of the $PATH environmental variable, one may have to type:

```
    $ ./hello
```

```
$ hello
Hello, world
$
```

Improvements

An improvement to the preceding program would be to add a `return 0;` statement at the end of `main()` so that the program doesn't return a random value to the invoking environment. An equivalent alternative is to call `exit(0)` at the end of the program.[3] The C language convention is that a zero return value indicates success, while a non-zero return value indicates a failure of some type. [Kernighan/Ritchie 1978] All further programs in this book include such a return statement.

Another improvement is to indicate the arguments with which `main()` is called: the first argument is an integer, indicating the number of parameters with which the program was invoked; the second argument is a pointer to an array of strings, which are the parameters with which the program was invoked. Even when the arguments are not used, it is still better to declare the arguments of functions. If the types of function arguments are declared without specifying names for the argument, the C++ compiler must still verify the types of the arguments wherever the function is invoked. All further programs shown in this book always declare the types of the arguments to `main()`.[4] For further discussions on style, see Exercise 3.17. Here is the preceding program again with these two additions:

```
#include <stream.h>
int main(int, char**)
{
    cout << "Hello, world\n";
    return 0;
}
```

□

Exercise 2.2 (*1)

For each of the declarations in §2.1, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.

3. There is at least one commercially available C compiler for micro-computers which incorrectly ignores all return values from `main()`, assuming that the value should be zero. Later releases of that compiler have fixed this problem.

4. Some invoking environments pass a third argument to `main()`. This argument will not be declared or used in this book.

Definition: ch

The definition

```
char ch;
```

would have a declaration such as

```
extern char ch,
```

Definition: count

The definition

```
int count = 1;
```

would have a declaration such as

```
extern int count;
```

Definition: name

The definition

```
char *name = "Bjarne";
```

would have a declaration such as

```
extern char *name;
```

Definition: complex

The definition

```
struct complex { float re, im; };
```

defines the members of the structure. The program cannot declare variables of type `complex` until the definition is seen, but it can declare pointers to those types by first using a declaration of the structure:

```
struct complex;
complex *x;
```

It is in this fashion that circular dependencies are handled within two structure definitions. The second structure is declared before the definition of the first structure so that the first structure can have a pointer to the second:

```
struct second;
struct first { second *p2; };
struct second { first *p2; };
```

C++ needs the first declaration in oder to know that the symbol `second` is a type specification.

Definition: cvar

The definition

```
    complex cvar;
```

would have a declaration such as

```
    extern complex cvar;
```

Declaration: sqrt

The lack of a body in[5]

```
    extern complex sqrt(complex);
```

indicates that this is a declaration. A definition of the function would look like

```
    #include <complex.h>
    #include <math.h>
    // determine sqrt of x + yi
    complex sqrt(complex z)
    {
        double x = real(z);
        double y = imag(z);

        // the easy ones: y == 0
        if (y == 0.0)
            if (x < 0.0)
                return complex(0.0, sqrt(-x));

            else
                return complex(sqrt(x), 0.0);

        // almost as easy: x == 0
        if (x == 0.0)
            if (y < 0.0)
                {
                double x = sqrt(-y / 2);
                return complex(x, -x);
                }

            else
                {
                double x = sqrt(y / 2);
                return complex(x, x);
                }
```

5. The `extern` keyword is optional in both function declarations and definitions. It is considered good style to include the `extern` keyword in function declarations and to leave it out in function definitions. If the function were declared `static` within the definition, then the `static` keyword should also be used in the declaration instead of `extern`.

```
        // convert to polar and take the root
        //
        //          2     2  1/2
        // r = ( x   + y )
        //
        // theta = θ = arc tan (y / x)
        //
        //  1/2     1/2
        // z     = r    (cos θ/2 + i sin θ/2)
        double root_r = sqrt(sqrt(x * x + y * y));
        double half_t = atan2(y, x) / 2.0;
        return complex(root_r * cos(half_t),
                       root_r * sin(half_t));
    }
```

Declaration: error_number

The declaration

```
extern int error_number;
```

would have a definition of the integer indicated as

```
int error_number;
```

Definition: point

The `typedef` definition

```
typedef complex point;
```

does not have a separate declaration syntax. All type names declared through `typedef` require a definition of what the type consists of; one cannot declare a `typedef` without such a definition.

Definition: real

The function definition

```
float real(complex *p) { return p->re; }
```

would be declared as

```
extern float real(complex *p);
```

Definition: pi

The `const` definition

```
const double pi = 3.14159265358979323 85;
```

does not have a separate declaration syntax. All constant declarations are by default given static linkage and are required to include an initialization, making them definitions. If, however, the variable had been defined using

```
extern const double pi = 3.14159265358979323 85;
```

then the variable can be referred to in another file by using the following declaration:

```
extern const double pi;
```

Declaration: user

The declaration

```
struct user;
```

does not specify the members of the structure. A complete definition might be

```
struct user { char *name; int uid, gid; };
```

□

Exercise 2.3 (*1)

Write declarations for the following: a pointer to a character; a vector of 10 integers; a reference to a vector of 10 integers; a pointer to a vector of character strings; a pointer to a pointer to a character; a constant integer; a pointer to a constant integer; and a constant pointer to an integer. Initialize each one.

Declarations are best created from the inside out while reading the description from left to right.[6]

Declaring a Pointer to a Character

A pointer to a character starts with a pointer:

```
*pc
```

then the type to which it points is added:

```
char *pc
```

To initialize the pointer, the address of a character is needed:

```
char c;          // character
char *pc = &c;   // pointer to a character
```

Declaring a Vector of 10 Integers

A vector of 10 integers starts with the vector:

```
iv[]
```

then adds the dimension:

6. There is a public domain program named `c++decl` which will convert between an English representation of a C++ declaration and the equivalent C++ syntax. For example, given the command, *declare x as pointer to char*, `c++decl` will respond with *char *x*, and given the command *explain char *x*, `c++decl` will respond with *declare x as pointer to char*.

```
    iv[10]
```

and finally what the vector contains:

```
    int iv[10]
```

To initialize the vector requires a list of elements:[7]

```
    // vector of 10 int
    int iv[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Declaring a Reference to a Vector of 10 Integers

A reference to a vector of 10 integers starts with the reference:

```
    &riv
```

then adds the vector:

```
    &riv[]
```

and the dimension:

```
    &riv[10]
```

and finally the type of vector:

```
    int &riv[10]
```

To initialize this reference requires the name of another variable of the same type without the reference:

```
    int &riv[10] = iv;  // reference to vector of 10 ints
```

Declaring a Pointer to a Vector of Character Strings

There are two possible interpretations of this declaration. The first treats the vector as before, declaring it with a size. The second treats the vector as a dynamic, unbounded object. In C++, an unbounded vector is created using a pointer to the first element of the vector; it is allocated using the `new` operator.

(Interpretation 1) A pointer to a vector of character strings starts with the pointer:

```
    *psv
```

It then adds the array:

```
    (*psv) []
```

(The extra parentheses are necessary because of the precedence rules.) A size is required, so let us choose 4. Add its size:

7. Any elements not listed will be initialized to zero. See §r.8.6 and §r.8.6.1 for further information on initialization of external and static variables.