

Programming in C++ (for C Programmers)

Seminar Goals

© 2001, 2005 Rex Jaeschke. All rights reserved.

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how to use the basic I/O facilities to write to the screen and read from the keyboard, using simple formatting via manipulators.

2. [Skip] Looping, Testing, and Branching

3. [Skip] Arrays

4. Functions

Functions are the backbone of C++ programs. A typical program is made up of one or more user-written functions and calls to functions provided in the standard and other libraries.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how to define default arguments for function calls.
- Be able to use function overloading and function templates, and have a basic understanding of function inlining.

5. [Skip] Storage Classes

6. Pointers and Addresses

So far, we have learned how to do things in C++ that we already know how to do in some other language. In this chapter, we will be exposed to those capabilities of C++ that make it different from higher-level languages.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how to allocate and free memory at runtime, under the programmer's control.

7. Structures, Bit-Fields, and Unions

In this chapter, we will learn how to deal with a set of objects as a group. We will also see how to redefine an area of memory so that it can be used for multiple purposes, one at a time. An introductory discussion of linked lists is also provided.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Have a basic understanding of how I/O can be defined for structure types.

8. [Skip] Introducing the Standard Class Libraries

9. The string Class

In this chapter, we look at the standard class string, a class designed to provide storage and manipulation of text strings. This class can be used instead of, or in conjunction with, C-style strings.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Construct C++-style strings from a variety of sources, including C-style strings.
- Access individual characters within a C++-style string.
- Assign to C++-style strings from a variety of sources, including C-style strings.
- Convert a C++-style string to a C-style string.
- Compare C++-style strings with each other and with C-style strings.
- Insert one string into another.
- Concatenate two strings.
- Perform search and replacement operations in strings.
- Extract substrings.
- Perform I/O on C++-style strings.

10. Input and Output

We have learned how to perform basic I/O via the objects cin and cout, and their corresponding operators >> and <<. We have also used manipulators, both with and without arguments. In this chapter, we will get a

more formal introduction to these facilities. We'll also see how to do file I/O and how to do formatted I/O to and from strings.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Use manipulators.
- Save and restore the state flags of a stream.
- Have a basic understanding of the console stream I/O member functions.
- Know how to perform basic I/O using files.
- Know how to encode values into C++-style strings and how to decode them from such strings.
- Have a basic understanding of how to define manipulators.

11. Classes and Objects

Classes are the main concept underlying the object-oriented nature of C++. Simply stated, a *class* is a special kind of structure.

In this chapter, we will learn about data hiding and encapsulation. *Data hiding* is the process whereby implementation details of a class can be hidden from general access. *Encapsulation* involves the syntactic association of data and the functions that have permission to operate on it.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Understand the purpose of data hiding and encapsulation.
- Be able to use effectively the access specifiers `public` and `private`.
- Understand the purpose of a `const`-qualified member function.
- Know that the way in which an object is represented internally is not necessarily related to the way in which programmers see it externally.
- Know the type of `this`, and what it refers to.
- Know how to use effectively inlining, default arguments, and overloading with member functions.
- Understand the difference between instance and class data.
- Know the advantages and disadvantages of using `friend`.
- Be able to define simple user-defined types using object-oriented constructs.

12. Object Creation and Destruction

It can be very useful to have a function called automatically each time an object of some class is created. C++ provides such a capability via a special member function called a *constructor*. The main purpose of such a function is to ensure that the object being created is initialized with a predictable value. It can also be useful to have another function called when that object's memory is released. Such a function is called a *destructor*.

In this chapter, we will see how to define and use constructors and destructors for our own classes. We will also see how they help make dynamic memory allocation fit more smoothly into the language.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how to guarantee that an object of a user-defined type starts life with a predictable and sensible state.
- Understand why a class might need a destructor.
- Be able to say exactly when constructors and destructors are called for objects (including arrays and nested objects) that are allocated automatically, statically, and dynamically.
- Know how and when to create temporary unnamed objects.
- Understand the need for copy constructors, and how to define them.
- Know that constructors can act as implicit conversion tools, and how this can be disabled using explicit.

13. Operator Overloading

The addition of classes is a step toward integrating user-defined types into the language. Once a class has been defined, we can create objects of that type by using the same notation as C++'s built-in types. However, to make these classes really fit in, we need to be able to operate on class objects using the same easy notation allowed by the built-in operators. In this chapter, we will see how most of the built-in operators can be given meaning in the context of class objects.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know which operators can and cannot be overloaded.
- Understand when it is and isn't useful to overload operators.
- Understand that a thorough knowledge of a built-in operator must be gained before it can be overloaded in an effective and complete manner.
- Know when the simple assignment operator need be overloaded.
- Be able to overload most unary and binary operators, doing so efficiently.

14. Inheritance

In this chapter, we'll learn about single and multiple inheritance, polymorphism, abstract classes, and virtual base classes.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how and why one would want to use inheritance.
- Understand the difference between containment and inheritance.
- At the design stage, be able to study a set of class descriptions and determine if those classes are unrelated, related by inheritance, or related by containment.
- Understand the purpose of virtual functions.
- Know when and how to use the protected access specifier.
- Understand the difference between public and private inheritance.
- Know how and why a class should be made abstract.
- Have a basic understanding of multiple inheritance.

- Realize the potential problems of inheriting from a class that was not intended to be a base class.

15. Exception Handling

An *exception* is some unusual condition in that it is outside the ordinary expected behavior. Generally, we view exceptions as errors, requiring special handling. In this chapter, we'll look at C++'s support for such handling.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Understand how the traditional approaches to dealing with extraordinary errors are limited and messy in general and even more so in an object-oriented environment.
- Know how to catch system exceptions using try and catch blocks, and how to recover from such exceptions where possible.
- Be able to throw an exception of a given type.
- Understand that an exception type could be a full-blown class using all the facilities we've learned so far.
- Understand the utility of, and issues relating to, a family of derived exception types.
- Know how the I/O (and other) standard class machinery recovers when exceptions are thrown.
- Be able to encapsulate dynamic memory allocations using `auto_ptr`, so cleanup is performed automatically when exceptions are thrown.

16. Class Templates

In this chapter, we will see how to implement families of classes using templates.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Apply abstraction to a class, so specialized versions of it can be created using templates.
- Understand that template classes typically require that their source code reside in a header rather than an implementation file.
- Know how to explicitly specialize a template class.

17. Name Spaces

In this chapter, we'll look at how the same name can be used to mean different things in the same program scope. Along the way, we'll introduce the new keywords `namespace` and `using`.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how to import names from a namespace.
- Understand how namespace pollution is a problem in projects involving multiple subsystems and development groups.
- Know how the standard library uses namespaces to avoid name clashes.
- Define namespaces and partition source code accordingly.

18. [Skip] The Preprocessor

19. Sundry Issues

In this chapter, we look at a number of specialized and/or esoteric topics that don't fit into any of the other chapters.

Goals of this Chapter

Once the reader has read the information provided in this chapter, and solved the related programming exercises, they should be able to do the following:

- Know how and when to use the mutable keyword in a class.
- Be able to obtain type information at runtime.