Glossary of C++ Terms found in "Classic Data Structures in C++"

Timothy A. Budd
Oregon State University

July 16, 1993

- **Abstraction** The technique in problem solving in which details are grouped into a single common concept. This concept can then be viewed as a single entity, and inessential information ignored.

  Abstraction is introduced in Chapter 1 of the text.

- **Abstract Superclass** A class that is not used to make direct instances, but rather is used only as a base from which other classes inherit. In C++, the term is often reserved for classes that contain at least one *pure virtual method*.

  Abstract superclasses are introduced in Chapter 7.

- **Accessor Method** A member function designed to provide access to the internal data values maintained by instances of a class. Such functions permit the data values to be read but not modified by users of the software abstraction.

  Accessor methods are discussed in Chapter 2.

- **Argument Signature** An internal encoding of a list of argument types; the argument signature is used to disambiguate overloaded function invocations, with the function body being selected that matches most closely the signature of the function call.

  Overloading of function names and the concept of argument signatures is introduced in Chapter 2.

- **Automatic Variable** A variable that is allocated space automatically when a procedure is entered. This is the default behavior in C++. Contrast to *dynamic memory allocation*, in which space must be allocated by the user.

The distinction between automatic and dynamic memory allocation is discussed in Chapter 4.

- **Base Class** Syn. *ancestor type, superclass.* A class from which another class is derived. Subclasses, or child classes, inherit data fields and behavior from their base class.

  Inheritance is introduced in Chapters 6 and 7, and used throughout the text.

- **Child Class** Syn. *subclass, derived class.* A class that is defined, using inheritance, as a specialization or an extension of another class. The original class is known as the *parent* or *base* class.

  Inheritance is introduced in Chapters 6 and 7, and used throughout the text.

- **Class** An abstract description of the data and behavior of a collection of similar objects. The representatives of the collection are called *instances* of the class. Classes provide a mechanism for encapsulation and instance generation.

  Classes are introduced in general form in Chapter 1, and the syntax of class formation in C++ is introduced in Chapter 2.

- **Cohesion** The degree to which components of a single class are tied together. That is, the degree to which the functions and data fields in a class description are unified by a single purpose. A good design attempts to maximize class cohesion.

  Contrast with *coupling*

  Cohesion and coupling are introduced in Chapter 1.

- **Composition** A software reuse mechanism. A new class being constructed is said to use composition if it declares as a data field a value with a type defined by an existing class. The facilities of the older class can then be used to simplify the creation of the new component.

  Composition is one of the major techniques for software reuse, the other mechanism being inheritance.

  Composition is introduced in Chapter 6.

- **Constructor** A method having the same name as the class in which it is defined. A constructor is invoked implicitly by the run-time system when a declaration statement is processed, or when dynamic memory allocation is performed. The purpose of a constructor is to make sure each newly created object is properly initialized.

  There are a variety of special constructors. See *default constructors*, *copy constructors*, and *conversion operators*.

Constructors are introduced in Chapter 1.

- **Copy Constructor** A constructor that takes a single argument, which is a pass-by-constant-reference value of the same type as the class in which the constructor is defined. Copy constructors are usually invoked implicitly by the run-time system when it is necessary to form a duplicate copy of a value. This occurs when an instance of a class is sent to a pass-by-value parameter, for example.

  Copy constructors and the different parameter passing mechanisms are discussed in Chapter 2.

- **Coupling** The degree to which one class relies for its proper performance on the data fields or services provided by another class. Good software design attempts to reduce the coupling between classes to a minimum, since coupling tends to inhibit the ability to reuse software components in new projects.

  Contrast with *cohesion*.

  Coupling and cohesion are introduced in Chapter 1.

- **Data Member** Syn. *instance variable, data fields.* The data fields maintained by instances of a class, as opposed to *function members*, which are functions that provide the behavior associated with a class.

  Data members are introduced in Chapter 2.

- **Default Constructors** A form of constructor with no parameter values. A default constructor, if defined, is invoked in response to a declaration statement that does not list arguments.

  Default constructors are introduced in Chapter 2.

- **Destructor** A function method that is invoked by the run-time system immediately before memory is released for an object. The destructor can perform any actions required for the management of the object. Destructors are almost never directly invoked by the programmer. The name of the destructor is formed by a tilde being prepended to the name of the class.

  Destructors are introduced in Chapter 4.

- **Dynamic Memory Allocation** A process whereby the programmer explicitly requests a space for a variable from the run-time system. Such space must then be managed by the programmer; that is, it will not be automatically released when a procedure is exited, as are *automatic variables*. The management of dynamic memory is usually hidden within a class, and can be facilitated through the use of features such as *destructors*.

Dynamic memory management is introduced in Chapter 4 for strings, and is used extensively by the vector abstraction (Chapter 6), the List datat type (chapter 8) and trees (chapter 10).

- **Friends** A *friend* is a class or function that is explicitly identified by a class. Friends are allowed to break the normal variable scoping rules, by being permitted access to the private and protected portions of a class description.

  Friends are introduced in conjunction with the linked-list abstraction in Chapter 6.

- **Information Hiding** The principle that users of a software component (such as a class) need to know only the essential details of how to initialize and access the component, and do not need to know the details of the implementation. By reducing the degree of interconnectedness between separate elements of a software system, the principle of information hiding helps in the development of reliable software.

  Information hiding is introduced in Chapter 1.

- **Inheritance** The property of classes through which instances of a class can have access to data and method definitions contained in a previously defined class, without those definitions being restated.

  Inheritance is one of two major mechanisms for software reuse, the other being composition.

  Inheritance is introduced in Chapters 6 and 7, and used extensively throughout the remainder of the text.

- **Inline Function** An inline function is declared using the keyword **inline**. The definition of such functions is given in the interface file, not the implementation. Such a function is expanded internally by the compiler as code at the point of call, thus avoiding the overhead of a subroutine call and return sequence. Since the code for an inline function can be duplicated many times, the feature is used mainly for functions that generate very little code, such as accessor methods.

  Inline functions are mentioned in Chapter 4.

- **Message Passing** Syn. *Member Function Lookup*. The process of determining which actual function to execute in response to a message passing request. Message passing differs from regular procedure calls in that there is always a designated *receiver* for a message, and the function executed depends upon the type of the receiver.

  Message passing is introduced in Chapter 2.

- **Method** Syn. *Member Function.* A function defined as part of a class structure. A method, or member function, is invoked as a *message* sent to a specific instance of the class in which the function is defined.

  Methods are introduced in Chapter 2.

- **Object Oriented Programming** A style of program design that is centered around the delegation of responsibilities to independent interacting agents, and a style of programming characterized by the use of message passing and classes organized into one or more inheritance hierarchies.

  Object oriented design and programming is developed throughout the text.

- **Override** The action that occurs when a method in a subclass with the same name as a method in a parent class takes precedence over the method in the superclass.

  Overriding methods are introduced along with inheritance in Chapter 7.

- **Prototype** A description of a function or class that describes the function name, argument types, and return type but does not include the function definition. Prototypes are typically found in interface files.

  Prototypes are introduced in Chapter 2.

- **Pure Virtual Method** A method provided with a prototype in a class description, but not implementation. The absence of an implementation is indicated by an assignment symbol which appears to assign the function the value zero. A class containing such a method must be subclassed before instances can be created.

  Pure virtual methods are introduced in Chapter 7.

- **Receiver** The instance of a class to which a message is sent. In C++ the receiver is the object to the left of the field qualifier (the period) in a message passing expression. Within a method, the current receiver is accessible through a pointer variable named *this*.

  Message passing expressions are introduced in Chapter 2.

- **Reference Variable** A variable or field declared using the address-of (&) modifier. Such a variable points to another value, and is an alias for this value. Changes to the reference variable will be reflected in changes to the object to which the reference has been assigned.

  The two main uses of reference variables are in the pass-by-reference parameter passing mechanism, and in the creation of classes that maintain, as part of the data values, a reference to another object.

  References are introduced in Chapter 2.

- **The Slicing Problem** The process by which a value of a derived type is assigned to a variable declared as a base type. In effect, the data fields and methods of the derived type are sliced off from the value.

  This problem is described in Chapter 7.

- **Virtual Method** A method declared with the **virtual** keyword. Such a method that will be called using late binding. The method body to be invoked cannot be determined at compile time, and thus a run-time search is performed to determine the method to execute. The method executed will be the one associated with the dynamic type of the receiver, not the static (declared) type.

  Virtual methods are introduced in Chapter 7.