A Comparison of the Syntax and Semantics of C++ and Java

version 1.3

Russell C. Bjork
Professor of Computer Science
Gordon College
Wenham, MA 01984

Because the designers of Java were very familiar with C++ and based its syntax on that of C++, it appears relatively easy for a person who knows one language to learn the other. However, there are some subtle differences which can be a source of confusion if one naively assumes that similar appearance always implies similar meaning.

This document attempts to address some of these differences. It is intended for students who know Java and wish to learn C++. It may also be useful for readers wishing to go the other way. It does not discuss most areas where the two languages are similar. By way of discussion and example code fragments, it addresses key syntactic and semantic differences related to the following:

1. Paradigm-Related Differences

C++ was developed as an object-oriented extension of C. (In fact, an early version of C++ was called “C with Classes”.) As such, it is a hybrid of two distinct programming paradigms: the procedural paradigm and the object-oriented paradigm. (The object-oriented features were basically added on top of a procedural base.) For the most part a C program conforming to the ANSI standard for C is also a valid C++ program, though the reverse is certainly not true. Java, on the other hand, was designed from the ground up as an object-oriented language, not a hybrid.

One place where this distinction becomes immediately apparent is in the overall structure of a program. In Java everything in a program is part of some class. In C++, it is possible to have “top-level” variables and functions which are declared outside of any class. (Much of the standard library is organized this way.) Further, main() must be top-level.

Example 1 (on the next page) shows two versions of a program having the same functionality, which illustrate this distinction.

- In the C++ version, the functions fib() and main(), and the variable fibCallsCounter, are declared at “top level” outside any class. (Indeed, no classes are declared at all in this program.)

- In the Java version, everything is declared inside the class Demo1.

Note that the C++ version could be rewritten to embed fib() and fibCallsCounter in a class, as in the Java version. However, this would *not* normally be done. In any case, main() would have to be top-level.

2. Data Abstraction; Separation of Interface and Implementation

A key principle that is part of both the procedural and object-oriented paradigms is data abstraction: the separation of the *interface* of a data type/class from its *implementation.* However, the mechanisms for doing this in C++ support a stronger separation than those in Java do.

a. C++ allows the interface of a class to be declared in a separate file from its implementation.

- It is common practice to put the declaration of a class in a separate *header* file. (As an aside - but an important point for people used to Java to note - the *declaration* of a C++ class always ends with a semicolon. Forgetting this can lead to really strange compiler errors!)
- Client files will use a `#include` directive to include the header file. When a header file is included, the effect is the same as if the text of the header file were physically present in the source file, in place of the `#include` directive.
- The implementation goes in an implementation file whose name is typically the same as that of the header, but with a “.cc” suffix. The implementation file also “#include”’s the related header file.
- Note that a header (.h) file can (and often does) include declarations for multiple related classes, in which case the implementation (.cc) file will typically include implementations for each class as well.

C++ does *allow* declaration and implementation to be done in one place (as in Java); but this is *not* normally considered good practice. In particular, if the implementation code is placed in the class declaration, then every time a source file includes the declaration, the compiler will create a new copy of the implementation code.

Example 1: C++ Version

```
/*
 *  This program calculates and prints
 *  a fibonacci number, using a
 *  recursive function to do the
 *  calculation.  It also prints out
 *  the number of times the function
 *  was called.
 */

#include <iostream>
using namespace std;

int fibCallsCounter;

/*  Calculate the nth fibonacci
 *  number. Increment fibCallsCounter
 *  every time it is called.
 */

int fib(int n)
{
    fibCallsCounter ++;
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

int main(int argc, char * argv[])
{
    while (true)
    {
        int N;

    fibCallsCounter = 0;
    cout << "Desired value of N - "
             << "0 to quit: ";
    cin >> N;
    if (N == 0)
        break;

    cout << "The " << N
            << "th fibonacci number is "
         << fib(N) << endl;
        cout << "Function was called "
            << fibCallsCounter
            << " times" << endl;
     }
}
```

Example 1: Java Version

```
/**
 *  This program calculates and prints
 *  a fibonacci number, using a
 *  recursive function to do the
 *  calculation.  It also prints out
 *  the number of times the function
 *  was called.
 */

import java.io.*;

public class Demo1
{
    private static int fibCallsCounter;

    /*  Calculate the nth fibonacci
     *  number. Increment fibCallsCounter
     *  every time it is called.
     *
     *  @param n desired fibonacci number
     *  @return the fibonacci number
     */
    private static int fib(int n)
    {
        fibCallsCounter ++;
        if (n <= 2)
            return 1;
        else
            return fib(n-1) + fib(n-2);
    }

    public static void main(String[]args)
            throws IOException
    {
        while (true)
        {
            int N;

            fibCallsCounter = 0;
            System.out.print(
                "Desired value of N - " +
                "0 to quit: ");
            String input =
                (new BufferedReader(
                    new InputStreamReader(
                System.in))).readLine();
            N = Integer.parseInt(input);
            if (N == 0)
                break;

            System.out.println("The "+ N +
              "th fibonacci number is " +
              fib(N));
            System.out.println(
              "Function was called " +
              fibCallsCounter + " times");
        }
        System.exit(0);
    }
}
```

b. Java does not really support the kind of separation of interface and implementation that C++ does.

- Each publicly accessible Java class must be placed in its own source (.java) file. This source file includes both the declaration and the full implementation of the class. When a java source file is compiled, the resulting .class file contains a representation of both the method declarations and their implementations.
- Client classes will `import` the class. This causes the compiler to extract the declaration information from the compiled .class file.
- The javadoc program does allow the interface portion of this file to be extracted to a separate html documentation file; but this is only useful to a human reader, not client code.

c. Some key distinctions between C++ #include and Java import:

C++ #include:

- `#include` means *textual inclusion* - i.e. the text of the header file is copied at this point. Thus, the source code form of the header must be available.
- Two slightly different forms of the #include directive are used:

`#include <filename>`	-- for standard library headers. **Note:** Traditionally, most library headers had filenames ending in ".h", but in the latest versions of the standard library there is no suffix on the filename.
`#include "filename"`	-- for programmer-written headers. **Note:** User-written header files always have names ending with ".h".

- Programs must explicitly include *each* distinct header file they need; however, C++ header files - especially those for the library - often declare multiple entities. Any library class which is needed must be accessed via a suitable `#include` directive. In particular, if a program uses strings it must include `<string>` and if it does input-output it must include `<iostream>`.
- If program includes a header file that itself includes some other header file, then the latter is included in the program as well (since `#include` means "copy the text of the header file at this point".) This can lead to the same library header file being included twice (if several headers include it, or the main program explicitly includes a header implicitly included through one of its headers.) Normally, this does no harm - standard header files contain a mechanism which causes the second and subsequent inclusions of the same header file to be ignored.
- In the case of the latest versions of the library, a `using` directive or explicit use of `std::` is also necessary to access standard library entities - see the next major topic below, and the various examples.

Java import:

- The source code of the class(es) being imported need not be available. For library classes, `import` in Java does the same thing as `#include` and `using` accomplish together in C++.
- The `import` directive always has the same form, but the package name determines whether the class being imported is a library class or a programmer-written class: the former have a package name beginning with `"java."` or `"javax."`.
- It is possible to use the `".*"` wildcard to import all the classes in a given package, or it is possible to import classes individually. Library classes in the package `java.lang` are implicitly imported into every Java program (e.g. `Integer, String, System`, etc.)
- Importing a class does *not* automatically import classes it imports; each class must *explicitly* import the classes it needs.

d. Examples

In Example 1 (on page 2), note that the C++ program includes `<iostream>` (which declares a number of classes), and specifies that it is `using namespace std`, while the Java program imports `java.io.*`.

Example 2 (on the next page) shows a simple class that represents a Person, declared and implemented as separate header (.h) and implementation (.cc) files in C++, and the corresponding single file required for Java.

- Following the conventions of the language, the C++ files have the same name as that of the class, but beginning with a lower-case letter (person.h, person.cc).
- The C++ implementation (.cc) file explicitly includes the corresponding declaration (.h) file. (Note that it does not need to include the library header `<string>`, because the header file includes it and the implementation file includes the header file!) It also includes the library header `<iostream>`; facilities in this header are needed for the implementation but not for the declaration, so only the implementation file needs to include it.
- Both the header file and the implementation file explicitly specify what library entities they use.
- Any other C++ class which needed to make use of the Person class would contain the following statement:

  ```
  #include "person.h"
  ```
- As required by the Java compiler, the Java file has the same name as the class, case included (Person.java).
- Any other Java class which needed to make use of the Person class would contain the following statement:

  `import` *`package-name(s)-if-needed`*`.Person;`

3. Namespaces and Packages

A key issue faced by the designer of any programming language is how to deal with the possibility of name confusion - i.e. different portions of a large system using the same name to mean different things. A classic example of this problem occurs in Java - the `java.awt` package includes a class called `List` (meaning a visible component that displays a list of items from which the user can choose) and the `java.util` package includes an interface called `List` (meaning a sequence of items.) If a program imports both `java.awt.*` and `java.util.*`, and then uses the name `List` without further qualification, the compiler will complain about an ambiguous name.

An important concept in Java is the notion of a *package*. The package to which a class belongs is specified by a `package` statement at the start of the source file, and the package name is part of the full name of a class that is imported, though it can be omitted when referring to the class. (The Java tools require that source and class files be stored in a directory structure that mirrors the package structure of the program.) The ambiguity referred to above can be solved in one of several ways: either by using a fully qualified name (e.g. `java.awt.List` or `java.util.List` instead of the ambiguous name `List`, or by explicitly importing only the classes needed from one of the packages - e.g. if the program needs several things from the awt (including `List`), but only the class `Vector` from `java.util`, then it could import `java.awt.*` and `java.util.Vector`, and the name `List` would be unambiguous. Note that if a class is defined in a file that does not include a `package` statement, it is considered to belong to a single top level anonymous package, to which all such classes belong.

Example 2: C++ Version

file person.h:

```
/* This class represents a Person */
#include <string>
using std::string;
class Person
{
   public:
       // Constructor
       Person(string name, int age);

       // Mutator
       void setAge(int age);

       // Accessors
       string getName() const;
       int getAge() const;
       // Print out full information
       void print() const;

   private:
       string _name;
       int _age;

}; // *** NOTE TO THE READER: THIS
   // SEMICOLON IS _REALLY_ IMPORTANT
```

file person.cc:

```
/* Implementation for class Person */
#include "person.h"
#include <iostream>
using std::cout;
Person::Person(string name, int age)
: _name(name), _age(age)
{ }

void Person::setAge(int age)
{
    _age = age;
}
string Person::getName() const
{
   return _name;
}
int Person::getAge() const
{
   return _age;
}
void Person::print() const
{
   cout << "Name: " << _name
         << "Age: " << _age;
}
```

Example 2: Java Version

file Person.java:

```
/* This class represents a Person */

public class Person
{
   /** Constructor
     *
     *  @param name the person's name
     *  @param age the person's age
     */
   public Person(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

   /** Mutator for age
     *
     *  @param age the new age
     */
   public void setAge(int age)
   {
      this.age = age;
   }

   /** Accessor for name
     *
     *  @return the person's name
     */
   public String getName()
   {
      return name;
   }

   /** Accessor for age
     *
     *  @return the person's age
     */
   public int getAge()
   {
      return age;
   }

   /** Print out full information */
   public void print()
   {
      System.out.print("Name: " + name +
           " Age: " + age);
   }

   private String name;
   private int age;

}  // *** NOTE TO THE READER:
   // NO SEMICOLON HERE
```

C++ has no package concept per se, and many entities are defined at "top-level", rather than inside a class. Historically, this meant that the designers of the standard library had to be careful not to use the same name for two different things, and programmers working on a large system had to exercise similar caution. Traditionally, if a C++ program includes two different header files that both use the same top level name in different ways, there is an unavoidable ambiguity (and, in fact, the compiler will complain even if one of the names is not used in the including code, since `#include` is a textual operation.)

Recent changes to the language have incorporated the notion of namespaces, which allows top level names to be qualified to prevent conflicts, but this notion is not yet fully implemented by many C++ compilers. In brief, it is possible for a header file to specify that the names it defines belong of some specified `namespace`. (As in Java, if a name is defined outside the context of a specific namespace , it is considered to belong to a single top-level anonymous namespace.) Identifiers belonging to the same namespace can be declared in multiple files - e.g. all the names in the various library headers are made part of namespace `std`.

When names defined in a namespace are used outside of the namespace definition, the name has to be qualified by the namespace name. This can be done in one of several ways. (All three of these approaches are illustrated in the various example programs.)

1. Anywhere an identifier can legally occur, it can be preceded by the name of a namespace and `::`. Example: `foo::bar` would refer to the name `bar` defined in namespace `foo`. (See Example Program 3) (The anonymous namespace can be explicitly specified by using `::` with nothing on the left.)
2. A `using` directive can be used to allow an identifier to be implicitly qualified by a namespace. Example: if a program contains `using foo::bar;`, then any place where the program uses the name `bar` all by itself, it will have the same meaning as `foo::bar`. (The qualified form is still legal, but unnecessary.) (See Example Program 2).
3. A `using` declaration can be used to incorporate *all* of the names in some namespace into the top level anonymous namespace. Example: if a program contains `using namespace foo;`, then *every* name belonging to namespace `foo` (and defined in some header included by the program) can be used without being qualified. (Stroustrup recommends against using this approach for new software; but it is a convenient way to transition software originally designed without using namespaces to a new C++ environment, specifically with names in the library namespace `std`.) (See Example Program 1)

Note that it is common practice in C++ to group declarations for closely-related classes into a single header file, which a program can then include. This contrasts with the Java approach of requiring each class to be in a file of its own. Thus, for example, the following two statements that appear in Example Program 1 perform similar roles in the C++ and Java versions (though the classes they access are quite different - the former allows use of all classes defined in the header file `iostream`, while the latter allows use of all classes containing `package java.io` and residing in multiple files in the directory `java/io` in the library.)

```
#include <iostream>                                   import java.io.*;
using namespace std;
```

As a final note, standard practice for newer versions of C++ is to have two versions of the library header files - one version without ".h" at the end of the filename and defining all names in namespace `std`, and one (transitional) version with ".h" at the end of the filename and defining all names without a namespace. If the former form of the header file name is used, then qualified names or `using` is also necessary; but if the latter form of the header file name is used, no qualification of names is needed (but a recent compiler will issue a warning about using deprecated headers.) The intention is for the latter form of header files to eventually disappear.

4. Standard Libraries/API

Both Java and C++ are specified in terms of a language and a set of standard libraries. In the case of Java, the language and API are defined by Sun Microsystems. In the case of C++, the language and libraries are an ANSI standard. Both the Java API and the C++ standard libraries continue to evolve over time. Though the two libraries are quite different, there is significant overlap of functionality between them.

In the case of Java, the API is consistent across all platforms implementing a given version of the specification (e.g. JDK 1.1, Java 2, etc.) However, for a variety of reasons, most C++ implementations are not 100% compliant with the library standard. Variations between implementations mean that some fine-tuning of code is often needed when moving a C++ program from one platform to another.

As noted above, the handling of namespace issues for the library is in transition, and varies from platform to platform.

5. Support for Console and GUI Input/Output

One place where the Java and C++ standard libraries differ greatly is in the way they support input-output. C++ is derived from C, a language that was developed when the dominant mode of input-output was via text-based terminals displaying a (typically) 24 x 80 screen of characters. Java was developed after the graphical user interface had become the dominant mode of input-output.

Accordingly, the C++ standard libraries provide strong support for textual input-output to the standard input and standard output (normally the command line) on the host platform. They provide *no* support whatsoever for GUI input-output. To write GUI programs, one must use platform-specific GUI libraries - e.g. X-Windows on Unix platforms. This means that GUI programs will require extensive rewriting when being ported from one platform to another.

The Java API, on the other hand, defines a platform-independent set of GUI tools through the packages java.awt and javax.swing. GUI programs written in Java can run without modification on any system supporting the Java API. However, while the Java API provides reasonable support for output to the console (via System.out) it provides limited support for input from the console. Something of the difference in levels of support can be seen by comparing two segments from Example Program 1 - the code used to read a value of n from the user:

C++

```
int n;
cin >> n;
```

Java

```
String input =
      (new BufferedReader(
         new InputStreamReader(
            System.in))). readLine();
int n = Integer.parseInt(input);
```

C++ provides a simple extractor operator (>>) for extracting a string of characters from a stream and parsing them in accordance of the data type of a variable (here n, declared of type int). Java requires the standard input stream (System.in) to be “wrapped” in an InputStreamReader object, which is in turn “wrapped” in a BufferedReader. That object is then asked to extract a line of text via its readLine() method; finally, the parseInt method of class Integer is used to convert the string to an integer. (A considerably more complex approach would have to be used if it were desired to input two or more values from the same line!)

The C++ console input-output facility is part of a large package of iostream facilities, including support for text file input-output.   The basic facilities are accessed via

```
#include <iostream>
using namespace std;      (or explicitly qualify istream, ostream, cin, cout,
                           cerr, etc., with std:)
```

This file, among other things, declares:

- classes istream and ostream (input and output streams)
- variables cin, cout, and cerr - the standard input, output, and error streams, respectively.
- the extractor operator (>>) which can be used to extract a value from an input stream and parse it into a variable of primitive or string type.
- the inserter operator (<<) which can be used to insert a formatted value into an output stream, from a variable or expression of primitive or string type.
- The manipulator endl which can be used to insert a newline into an output stream.

Note, too, that both extractor and inserter operations can be chained. See the main() method in Example Program 1 for an example of chaining inserters.

6. Primitive Data Types

Most programming languages have a notion of *primitive data types* - data types that correspond to types that are directly supported by the underlying hardware. The set of primitive types in Java is defined in terms of the Java Virtual Machine, rather than in terms of a particular physical CPU; therefore, the Java primitives are identical on all platforms. The C++ primitive types are meant to map directly to hardware-supported data types on the underlying physical CPU, and thus are not necessarily identical across all platforms. Moreover, the primitive types of C++ (which are largely derived from those of C) mirror a time when 16 bit CPU's were still in wide use and 32 bit CPU's were the state of the art, while Java emerged in a time when 32 bit CPU's were the norm and 64 bit CPU's were beginning to emerge. Finally, the character type in C++ reflects a time when the English language dominated computing, while the Java character type was designed to allow for supporting a wide range of world languages.

The following table summarizes the primitive types in C++ and their relationship to the Java types. Where a name is given in parentheses after a C++ type name, it is a commonly used abbreviation for the type name.   Note that the type structure of the two languages is similar.

C++ Type name	Typical Meaning	Closest Java type(s)
`(char in some contexts)`	8 bit integer	`byte`
`short int (short)`	16 bit integer	`short`
`long int (long)`	32 bit integer	`int`
`(none)`	64 bit integer	`long`
`int`	16 or 32 bit integer (platform specific)	`short or int (depending on platform)`
`char`	8 bit character	`(byte in some contexts)`
`wchar_t`	16 bit character	`char`
`bool`	false or true	`boolean`
`float`	32 bit real	`float`
`double`	64 bit real	`double`
`long double`	96 or 128 bit real	`(none)`

An important distinction between C++ and Java is that C++ regards the type `char` as an integer type. Thus, in some contexts the C++ type `char` can be used as a one-byte integer, equivalent to the Java type `byte`.

Also, C++ allows all integer types to be explicitly declared as `signed` or `unsigned`. Types `short`, `int`, and `long` are signed by default. Whether type `char` is signed or unsigned by default is platform-dependent. The relevance of the distinction is that an unsigned type can represent about twice as large a value as the corresponding signed type - e.g. a short can represent the values –32768 to 37267, while an unsigned short can represent 0 to 65535. Also note that C++ allows the word `unsigned` to be used by itself as an abbreviation for `unsigned int`.

Finally, the type `bool` is a relatively recent addition to C++. Historically, the C++ comparison operators (like `==`, `<`, etc.) returned an integer value (0 for `false`, 1 for `true`); the boolean operators (`&&`, `||`, and `!`) operated on integer values, and statements like `if ()` and `while()` expected an integer expression, with 0 being interpreted as false and any nonzero value as true. It is still legal to use integers where a boolean value is expected, though this can result in cryptic code. (The bool values `false` and `true` are functionally equivalent to the integers 0 and 1.)

7. Class Types

Both C++ and Java allow programs to declare classes using the keyword `class`. Though the syntax for class declaration is similar in the two languages, there are also some key differences (beyond the separation of interface/implementation issues noted earlier). Most of these are illustrated in Example 2 on page 5.

a. Accessibility specification

A Java class may be declared as having `public` accessibility or package accessibility (keyword `public` omitted from the declaration.) This distinction does not exist in C++.

Both Java and C++ allow class members to have `public`, `protected`, or `private` accessibility. Java also uses default (package) accessibility when no other accessibility is specified. The meaning of `public` and `private` is essentially the same in the two languages. Because C++ does not have a package structure like Java does, there is no package accessibility in C++, and the meaning of `protected` is somewhat different. Java `protected` members can be accessed both from subclasses *and* from other classes in the same package; in C++ only the first sort of access is relevant.

Java and C++ also use slightly different syntax for specifying access to members. Java requires *each* member to have the access specified as part of its declaration - if no access is specified, the default (package) access is used. C++ breaks the declarations of members up into groups, proceeded by an access specifier followed by a colon. That access applies to all members declared until another access is specified. Thus, in Example Program 2, the constructor and the methods setAge(), getName() getAge() and print() are public, while the instance variables _name and _age are private. (If members are declared before any access specifier, they are private by default.).

(Not illustrated in the example programs.) C++ also allows a class to declare some other class or function as a `friend`. The friend class or function can access any member of the class without restriction - even private ones. Thus, suppose we have the following set of C++ declarations:

```
class A
{
    friend class D;

    public:  int x;
    protected:    int y;
    private: int z;
    ...
}
```

```
class B : public A
{ ...
}

class C
{ ...
}

class D
{ ...
}
```

Any method of class A can access the members x, y, and z. Methods of class B (a subclass of A) can access the members x and y, but not z. Methods of class C (not related to class A) can access only x. Methods of class D (a friend of A) can access the members x, y, and z.

b. Instance Variable Names

In Example 2 above, note that in the Java version the instance variables and the corresponding parameters to the constructor and mutator methods have the same names. The code for these methods uses “this” to explicitly distinguish between the two variables of the same name.

In the C++ version of the same program, the instance variables and method parameters have different names (e.g. `_age` and `age`). This is preferred practice in C++ because of the way field names are used in the initializer portion of a constructor. Note that there are two general ways this could be achieved: instance variables could be given “conventional names”, and distinct names could be chosen for parameters (e.g. the parameter of `setAge` might be called `newAge`); or names beginning with “_” could be used for instance variables, as in the example. The latter solution is a common (though not universal) practice in the C++ programming community, and is the one that will be followed here. Using names for instance variables that begin with “_” has the advantage that it is easy to recognize an instance variable whenever it occurs in code.

c. Initializers

Note also the special syntax used in the C++ constructor (`Person::Person()`) to initialize the instance variables. The syntax used in the C++ version can only be used in constructors. Although the “initialization by assignment” approach used in Java is also usually permissible in C++, there are cases where the special constructor syntax is necessary (e.g. when initializing const instance variables or instance variables of reference type), and there are also cases where both syntax's are permissible, but the special constructor syntax yields more efficient code. For these reason, the syntax given here should be used in C++ constructors.

d. Member functions declared const in C++

Note that the accessor methods (`getName(), getAge(),` and `print()`) have the word `const` at the end of their prototype, both in the class declaration and in their implementations. When used in this context, the word `const` means that the method will not alter the object to which the method is applied. (Given this declaration, the compiler will not permit one of these methods to alter the instance variables `_name` or `_age`). The modifier `const` is therefore appropriate on accessor methods, but not on constructors or mutators. There is no equivalent to this in Java.

e. Don't forget the semicolon after a C++ class declaration, but don‘t use one in Java!

8. Inheritance

Like all object-oriented languages, both C++ and Java support inheritance. C++ allows *multiple inheritance* - a class may inherit from any number of base classes. Java specifies that a class inherits from exactly one base class. (If no base class is specified, the base class is the class `Object`, which thus serves as a root class for the entire inheritance hierarchy. There is no analogous common root base class for all classes in C++.)

To partially compensate for its lack of support for multiple inheritance, Java also allows a class to implement any number of interfaces. There is no C++ analogue to the Java interface.

The syntax used for inheritance and related issues differs in the two languages. Most of the key differences can be seen in Example 3 on the next page - a class Student which inherits from the class Person developed for Example 2. In both languages, a Student object inherits the accessor and mutator methods of a Person object, to which it adds the accessor and mutator for the major field.

Note how the base class is specified - by using the syntax "` : public Person`" in C++ and "`extends Person`" in Java. The C++ keyword "`public`" stipulates that all of the public methods of Person become public methods of Student, and all the protected methods of Person become protected methods of Student. (Protected and private inheritance is also possible in C++ , but rarely used.) Java specifies inheritance by using the keyword "`extends`". Also note how the name and age parameters to the constructor are "passed up" to the superclass constructor. In C++ the syntax "`Person(name, age)`" is used in the initializer list. In Java, the syntax "`super(name, age)`" is used at the start of the constructor body. (With single inheritance, the class referred to by `super` is obvious in Java. The word `super` is <u>not</u> a reserved word in C++)

Example 3: C++ Version

file student.h:

```
/* This class represents a Student */

#include "person.h"

class Student : public Person
{
   public:

       // Constructor

       Student(string name, int age,
               string major);

       // Mutator

       void setMajor(string major);

       // Accessors

       string getMajor() const;
       virtual void print() const;

   private:

       string _major;

};
```

file student.cc:

```
/* Implementation for class Student */

#include "student.h"
#include <iostream>

Student::Student(string name, int age,
                string major)
: Person(name, age), _major(major)
{ }

void Student::setMajor(string major)
{
    _major = major;
}

string Student::getMajor() const
{
   return _major;
}

void Student::print() const
{
   Person::print();
   std::cout << " Major: " << _major;
}
```

Example 3: Java Version

file Student.java:

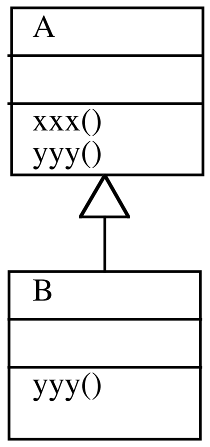
```
/* This class represents a Student */

import Person;

public class Student extends Person
{
   /** Constructor
    *
    *  @param name the student's name
    *  @param age the student's age
    *  @param major the student's major
    */

   public Student(String name, int age,
                  String major)
   {
      super(name, age);
      this.major = major;
   }

   /** Mutator for major
    *
    *  @param major the new major
    */

   public void setMajor(String major)
   {
      this.major = major;
   }

   /** Accessor for major
    *
    *  @return the student's major
    */

   public String getMajor()
   {
      return major;
   }

   /** Print out full information */

   public void print()
   {
      super.print();
      System.out.print(" Major: " +
                       major);
   }

   private String major;

}
```

9. Polymorphism; Dynamic and Static binding; Superclass Method Use Syntax

A key distinction between the two languages arises in terms of how they handle polymorphism.

a. Polymorphic References

Consider the simple inheritance hierarchy represented by the following UML class diagram:



Because class B inherits from class A, every "B" object is also an "A" object. Moreover, by virtue of inheritance, every B object inherits the method `xxx()` from A. However, B overrides the method `yyy()`.

In Java, a variable that is declared as referring to an object of class A can actually be made to refer to a B object at run-time. In C++, a variable that is declared to point or reference an object of class A can point to or reference a B object at run-time. (We will discuss pointers and references shortly.)

b. Static and Dynamic Binding of Methods

Now consider the following scenario:

C++

```
A * a = new B();
a -> yyy();
```

Java

```
A a = new B();
a.yyy();
```

Which version of `yyy()` is called? The one declared in class A, or the overridden version declared in class B?

In the Java version of the program, the answer is clearly the version declared in B. In the C++ version, the answer depends on how the method `yyy()` was *declared* in A. If `yyy()` were declared in the normal way, the A version would be called. This is because ordinarily C++ uses the *declaration* of the variable to determine which version of a method to call. Since the variable "a" is *declared* to refer to an object of class A, the class A version of `yyy()` would be used, regardless of the actual type of object the variable "a" currently points to. This approach is called *static binding*.

If the programmer's intention were to use the *actual* type of the object to determine which version of the method to call, the declaration for yyy() in A would have to declare it "virtual":

```
virtual void yyy() ...
```

This causes the compiler to use dynamic binding for calls to this method; in particular for class A and its subclasses, the compiler will construct a virtual method table with pointers to the appropriate version of each virtual method to call for that class. When a method is applied to an object of class A through a polymorphic reference, the run-time code will use this table to decide which version of the method should be called. That is, when overriding of inherited methods is possible, the virtual form of the C++ declaration is functionally equivalent to an ordinary Java method declaration. In the case of final Java methods (which cannot be overridden), the non–virtual form of the C++ method declaration is functionally equivalent. on this point - though the non-virtual form of the C++ method can be overridden in a subclass, where as the Java final form cannot be.

C++	**Java**
`void xxx();`	`final void xxx();`
`virtual void yyy();`	`void yyy();`

In Example 3 on page 12, note that the method `print()` in class Student is explicitly declared as `virtual`, because we want to ensure that any variable that refers to a Student object uses the version that prints the major, regardless of how the variable that refers to it is declared.

c. Abstract Methods and Classes

Recall that Java allows a method to be declared "abstract" - meaning it has no implementation in the class in which it is declared, but must be implemented in a subclass for that subclass to be concrete. In C++, only virtual methods can be abstract, and the corresponding syntax uses `= 0` after a virtual method declaration, which specifies that there should be a null entry in the virtual method for that class. That is, the following are equivalent:

C++	**Java**
`virtual void foo() = 0;`	`abstract foo();`

In Java, a class must be *declared* abstract if it contains any abstract methods. In C++, a class is automatically considered abstract if it contains at least one abstract method; there is no way to explicitly declare a class abstract.

d. Use of Superclass Version of Method

Sometimes, an overriding method needs to *add to* functionality of a superclass method, rather than replace it. In that case the overriding method needs to explicitly call the superclass method. This is accomplished by using the reserved word "super" in Java and the class name followed by "::" in C++. In the example based on the UML diagram above, the `yyy()` method in B could use the `yyy()` method in A, as follows:

C++	**Java**
`A::yyy();`	`super.yyy();`

In Example 3 on page 10, note how the implementation of `print()` in class Student explicitly makes use of the superclass version declared in Person.

e. Final methods in Java

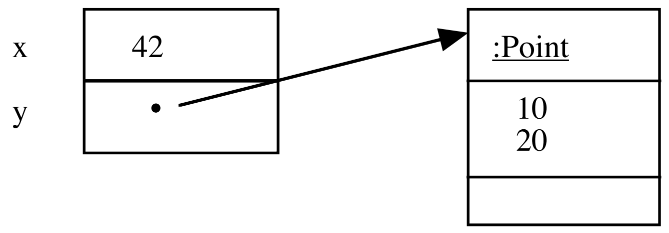
A method declared `final` in Java <u>cannot</u> be overridden in a subclass. There is no equivalent to this in C++.

10. Value and Pointer Types

The memory location specified by a variable can actually refer to an entity in one of two distinct ways: by *value*, or by *reference*. If variable names a memory location holding a value, then the memory location actually contains the entity. If variable names a memory location holding a reference, then the memory location contains the address of another memory location that actually contains the entity. This distinction can be seen in the diagram below, based on the following Java code fragment:

```
int x = 42;
Point y = new java.awt.Point(10, 20);
```

The variable x names a memory location that holds the value 42; the variable y names a memory location that holds a reference to another memory location that contains the point (10, 20).



In Java, the distinction as to whether a variable specifies a value or a references is entirely determined by the variable's *declared type*. If the variable is declared to be of a primitive type (`boolean, char, byte, short, int, long, float,` or `double`), then it always specifies a value. If the variable is declared to be of an array or object type, then it always specifies a reference. (For this reason, array and object types are called *reference types* in the Java documentation.)

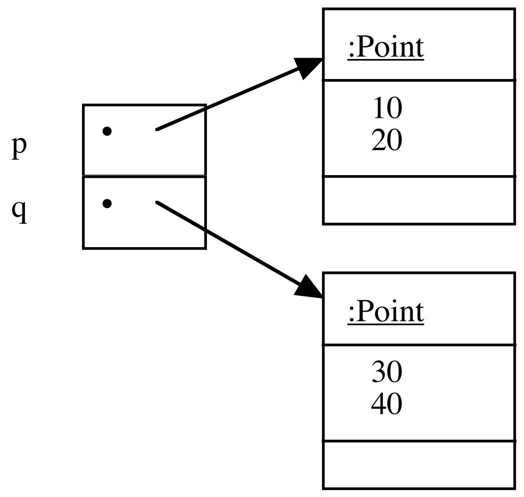
In C++, the distinction is made by the way the variable is declared. Any type (primitive or structured) can be specified by value or by reference. Moreover, C++ has two distinct ways of specifying a reference - pointers and references. Pointers are carried over from C; references are new with C++. We will focus on C++ pointers now, and discuss C++ references later.

a. Pointers in C++

A C++ pointer variable is specified by using a "*" in the declaration. (Note carefully how `p` is declared in the example below.) The variable name, by itself, stands for a memory address. To refer to the entity it point to, the variable name is preceded by * - e.g. (continuing the above example) `p` stands for a memory address and `* p` stands for the entity `p` points to. This distinction becomes most obvious when we consider the meaning of the assignment operator. Consider the following code fragment

```
// Assume we have declared a C++ class Point analogous to java.awt.Point

Point * p = new Point(10, 20);
Point * q = new Point(30, 40);
```

The above statements result in the following situation in memory:



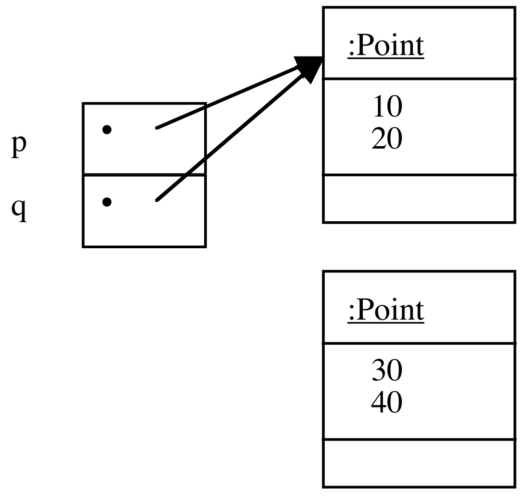
Now consider the following two similar-appearing, but by no means identical statements. (Assume we execute just <u>one</u> of the two statements.)

```
q = p;          // Statement 1
* q = * p;          // Statement 2
```

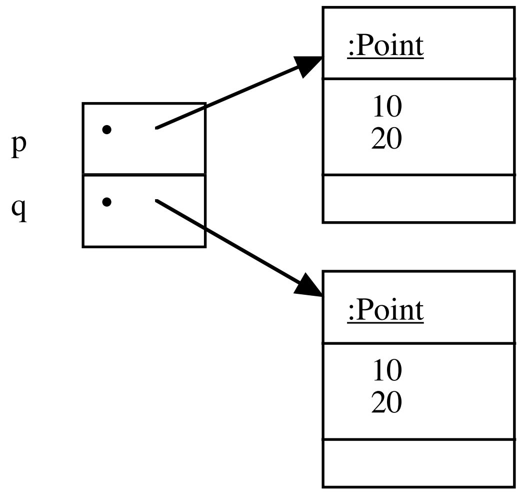
Statement 1 alters the value of the variable `q`. It makes `q` point to the same object `p` points to (leaving nothing pointing to the object `q` pointed to earlier.)

Statement 2 alters the value of the object `q` points to. It makes this object contain the same coordinates as the object `p` points to.

The varying effects of these two statements can be seen by the following diagrams:



Effect of Statement 1



Effect of Statement 2

In both cases, `q` ends up pointing to an object that contains the coordinates (10, 20). But in the first case, this is the *same* object `p` points to; and the object that contains (30, 40) continues to exist though nothing points to it. In the second case, there are *two* objects that contain the coordinates (10, 20), and *no* object contains (30, 40).

Note that Java reference assignment corresponds to Statement 1; there is no Java analogue to Statement 2.

b. Pointers and Equality testing in C++

A similar situation pertains with regard to *testing* for equality. Suppose we have declared a class Foo, and two pointers to Foo objects, declared like this:

```
Foo * p, * q;
```

Now consider the effect of the following two statements:

```
... Code that assigns values to p and q and does things with them
if (p == q) ...          // Statement 1
if (* p == * q) ... // Statement 2
```

Statement 1 tests whether p and q refer to the *same object*. Statement 2 tests whether p and q refer to objects *having the same value*. (Note: A statement like Statement 2 is legal only if the kind of object pointed to supports comparison for equality. Assume, in this case, that Foo does.) Obviously, if Statement 1 is true, then so is Statement 2; but the reverse is not necessarily the case.

Once again, Statement 1 corresponds to the use of == on references in Java. There is no direct analogue to Statement 2 in Java; however, many classes provide an equals method that accomplishes the same thing - e.g.

```
Foo p, q;           // Java code roughly equivalent to C++ declaration
...
if (p.equals(q))    // Java code equivalent to Statement 2 (if Foo defines
                    // equals())
```

c. Pointers to Structures in C++

When a pointer is used to refer to an object that has individual members, the "." can be used with the "*" to reference an individual member of the object pointed to. For example, suppose we have a class Foo with a method named bar(). Suppose p is declared as follows:

```
Foo * p;
```

We might want to refer to the bar() method of the object p points to. However, the following syntax will not work, due to operator precedence:

```
* p . bar(); // Incorrect syntax
```

Instead, one must use parentheses:

```
(* p).bar();
```

An alternative syntax which has the same meaning (and is a lot more readable) is:

```
p -> bar();
```

This same syntax can also be used with instance variables - e.g. if Foo has an instance variable named baz, and p is declared as above, then we have:

```
... * p . baz ...   // Incorrect syntax
... (* p).baz ...   // Correct but awkward
... p -> baz ...    // Preferred
```

d. The implicit (this) pointer in C++

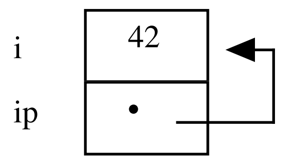
As in Java, any instance method of a class has an implicit "`this`" parameter. In C++, the this parameter is always a pointer to the object whose method is being executed - e.g. you may see syntax like `* this` or `this ->` but *never* `this .`.

e. The Address Of (&) operator in C++

For any variable, the & operator can be used to create a pointer to the memory location where it is stored. Thus, the following is possible:

```
int i = 42;         // The memory location named by i contains the integer 42
int * ip = & i;     // The memory location named by ip contains the address of i
```

The following diagram shows the state of memory after these statements:



The resultant pointer can be used like any other pointer, but since it refers to the same memory location as the variable the & was applied to, operations on it can affect the original variable as well - e.g. the following assignment would set the value of both i and * ip to 43:

```
(* ip) ++;
```

11. Array Types in C++

Another place where the C heritage of C++ shows up is in the handling of arrays, which is the same as in C. In contrast to Java, which requires separate steps for declaring an array and allocating its storage, C++ normally does both at the same time. For example, the following declares and allocates storage for an array of 10 integers:

```
int a[10];
```

In many contexts, an array and a pointer to the type of its elements can be used interchangeably. E.g., given the above declaration, we could now declare:

```
int * p = a;
```

and we could then refer to the ith element of the array a using *either* of the following:

```
   a[i]
```
or
```
   p[i]
```

In particular, when declaring the type of a parameter to a function, the following are equivalent:

```
   void foo(int a[])
```
or
```
   void foo(int * a)
```

12. Storage Allocation Model

All programming languages use variables to refer to memory locations. The establishment of a connection between a variable and the memory location it refers is technically called *binding* a storage allocation to a variable. Both Java and C++ accomplish this binding in one of three ways:

- *Static (permanent) allocation*: The binding between a variable and a particular location in memory lasts during the entire time the program is running. C++ uses this for all top-level variables (declared outside any class or function), and also for variables explicitly declared `static` either inside a class or in a function. Java uses static allocation only for variables that are explicitly declared static in a class; it does not have have top-level variables and does not allow local variables of a function to be declared static.

  In both versions of Example Program 1 (page 2), the variable `fibCallsCounter` uses static allocation.

- *Automatic (stack) allocation:* The binding between a local variable in a function/method and a particular location in memory is established when the declaration for the variable is reached during the execution of the function. It remains in effect until the block in which the variable is declared exits.

  In both versions of Example Program 1 (page 2), the parameter of `fib()` (`n`) and local variable(s) of `main()` (`n` in both programs; `input` in the Java program) use automatic allocation.

- *Dynamic (heap) allocation:* The binding of storage for an entity is accomplished by an explicit use of the `new` operator, which always returns a *pointer* to the newly allocated storage. Both Java and C++ use `new` in similar ways.

  However, they handle freeing up unneeded storage differently. In Java, the storage remains allocated until there are no more references to the entity, in which case the storage is reclaimed by *garbage collection*. In C++, the programmer is responsible for explicitly freeing storage allocated using new by using the corresponding operator `delete`. This approach is more time-efficient, since garbage collection is time consuming; however, it is also a source of programming errors when a programmer either forgets to free up storage when it is no longer needed (resulting in *storage leaks*) or frees up storage prematurely (resulting in *dangling references)* or multiple times (usually resulting in serious corruption of the storage management subsystem.) The following illustrates the difference between C++ and Java as regards the explicit deallocation of dynamic storage:

C++

```
void someRoutine()
{
   SomeClass * p = new SomeClass();

   ... Code that uses p

   ... Now suppose the object we created
   ... is no longer being used

   // Free up the space allocated by new

   delete p;
}
```

Java

```
void someRoutine()
{
   SomeClass p = new SomeClass();

   ... Code that uses p

   ... Now p is no longer needed

   // No need to worry about freeing
   // up space allocated by new - the
   // garbage collector will do it for us
}
```

Note that storage deallocation was relatively straightforward in the above case, because the object allocated is used only in one function. More typically, dynamically-allocated objects have lifetimes that span many functions, often making it nontrivial to know when storage can be freed by `delete`.

The following illustrates various kinds of variables allocated using static, automatic, and dynamic allocation, and shows the Java equivalents where they exist. The diagram below shows the state of memory at the end of `bar()`, after all the declarations/assignments have been executed.

C++

```
class Foo
{
   public:
        Foo(char ch, int code)
        :_ch(ch), _code(code)
        { }
   ...
   private:
        char _ch;
        int _code;
};

int a = 1;

int * b = new int;

int c[3];


void bar()
{
   * b = 2;

   c[2] = 3;

   int * d = new int[4];

   d[1] = 4;

   Foo e('A', 65);

   Foo * f = new Foo('B', 66);

   // STATE OF MEMORY SHOWN HERE
}
```

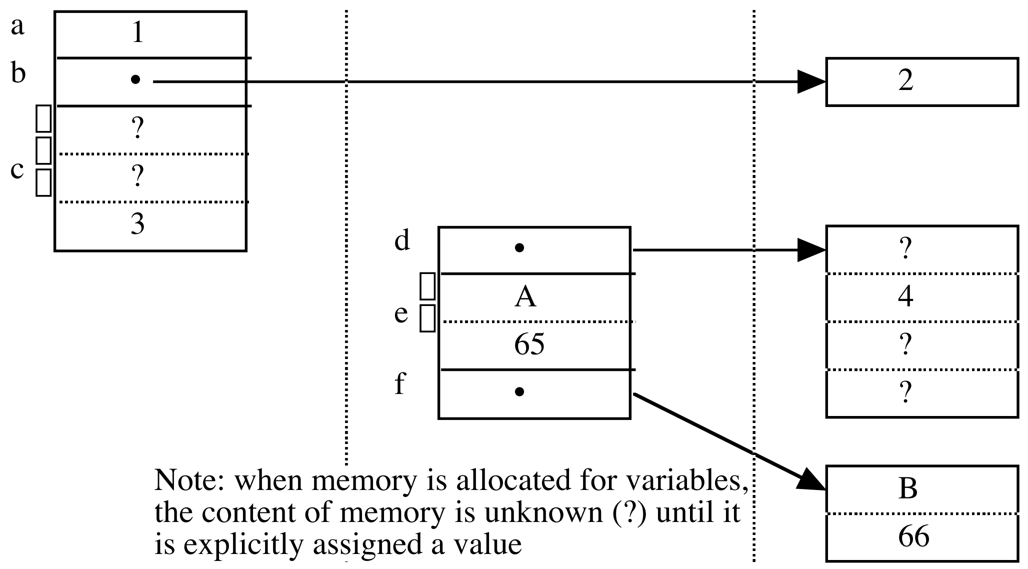
Java

```
class Foo
{
   public Foo(char ch, int code)
   {    this.ch = ch;
        this.code = code;
   }
   ...
   private char ch;
   private int code;
}
class Bar
{
   static int a = 1;

   // No Java equivalent

   int [] c = new int[3]; // Close, but
                          // not identical

   public static void bar()
   {
        // Java couldn't declare b

        c[2] = 3;

        int [] d = new int[4];

        d[1] = 4;

        // No java equivalent

        Foo f = new Foo('B', 66);
   }
}
```

Statically-allocated storage - will be freed when the program exits

Automatically-allocate (stack) storage - will be freed when `bar()` exits

Dynamically-allocated (heap) storage - must be freed by operator delete, or will be freed when the program exits.



Note: when memory is allocated for variables, the content of memory is unknown (?) until it is explicitly assigned a value

13. Reference Types in C++

When we introduced the distinction between value and pointer types in C++, we noted that C++ actually has two kinds of types that are analogous to Java reference types: pointers (inherited from C) and references (new with C++). Since we have already discussed pointers, we will now discuss C++ references.

a. Reference Variables in C++

A C++ reference variable is basically an *alias* for another variable. A reference is specified by using a "&" in the declaration. (We have already seen that & is also used as an operator to convert a variable into a pointer - a very different use.)

Example: The following declarations make `r` a reference to (an alias for) the variable `i`:

```
int i;
int & r = i; // Declare r and initialize it to refer to variable named i
```

The essential semantic distinction between pointers and references is that C++ pointers are *mutable,* while C++ references are *immutable*. A pointer variable can point to different memory locations (or none at all) at different times during the execution of a program. A reference variable must be initialized to refer to some other variable when it is created, and from there on out always refers to that same location memory. Assignment to the reference variable, or looking at its value, is the same as assignment to/looking at the variable it is an alias for.

Now, given the above declarations, consider the impact of the following statements:

```
i = 42;            // Statement 1
r = 42;            // Statement 2
```

What is the difference in meaning between these two statements? *Absolutely no difference!* Since r was initialized as a reference to (alias for) i, any operation on r has the same meaning as if were done directly on i.

Again, consider the following statement, which, given the above declarations, is necessarily always true:

```
if (i == r) ...    // Must always be true
```

To make life confusing when moving between C++ and Java, note that the Java reference resembles the C++ pointer in some ways and the C++ reference in others. The *semantics* (meaning) of the Java reference is most like the C++ pointer; but the *syntax* of the Java reference is most like the C++ reference. To put it differently, Java has nothing that *looks like* the C++ pointer (syntactically), but has nothing that *behaves like* the C++ reference (semantically.)

b. Value, Pointer, and Reference Parameters to Functions/Methods in C++

At this point, you might be wondering why reference variables were introduced into C++, since having an immutable alias to some other variable does not seem like it would be useful in too many cases. It turns out that one of the chief uses for references is with parameters of methods.

Sometimes, it is necessary for a method to *change* an entity passed to it as a parameter. For example, we might want to define a function that increments (adds one to) its parameter. This has always been possible in both C and C++ by using pointers, as the following pair of program fragments illustrate. (Note that both are valid code both in C and C++):

C/C++ - using value parameter (won't work as intended):

```
void addOne(int x) // Add one to parameter
{                  // Doesn't work!
    x ++;
}
...

int i = 42;
addOne(i);
cout << i << endl;
```

C/C++ - using pointer parameter: (works correctly)

```
void addOne(int * x) // Add one to
                     // parameter
{                    // This one works!
    (* x) ++;
}
...

int i = 42;
addOne(& i);
cout << i << endl;
```

The first version of the program would print 42 - not was intended - because the addOne() method operates on a value - that is, a *copy* of the value of i. It increments *the copy* to 43, but has no effect on the *actual* value of i in the main program. The second version of the program does print 43 as apparently intended, but at the cost of rather confusing syntax! (And it gets much worse when C++ operator overloading is utilized!).

One major use for references into C++ is to allow the writing of programs like the second example, but without syntactic ugliness. (However, some C++ library routines inherited from C still use pointers when passing parameters that the called routine must be able to change.)

Consider the following three program fragments - one written in C++ using an ordinary (value) parameter,one written in C++ using a reference parameter, and one written in Java. Note that the formal parameter is an `int` (what Java calls a value type).

C++ - value parameter:

```
void foo(int x)
{
     x ++;
}
...
int i = 42;
foo(i);
cout << i << endl;
```

C++ - reference parameter:

```
void foo(int & x)
{
     x ++;
}
...
int i = 42;
foo(i);
cout << i << endl;
```

Java:

```
void foo(int x)
{
     x ++;
}
...
int i = 42;
foo(i);
System.out.println(i);
```

What output would these program fragments produce? The first C++ program and the Java program would print 42, because the formal parameter x is passed by value, so changing it has no effect on the actual parameter `i`. The second C++ program would print 43, because the formal parameter x becomes a reference to (alias for) i for the duration of the execution of the function, so any operation done on x does affect the actual parameter `i`.

In the above example, the Java code behaved like the C++ code using a value parameter.

Now consider the following similar program fragments, this time using a mutable character string. (Note that the C++ `string` class is equivalent to *both* the Java `String` class and the Java `StringBuffer` class, because a C++ `string` is mutable.) In this case the formal parameter is an object (which Java calls a reference type). Therefore, the Java program will behave like the second C++ example, using call by reference. Accordingly, the output of the Java and second C++ program fragments will be "`Hello world!`", while the first C++ program fragment will simply output "`Hello`".

C++ - value parameter:

```
void foo(string x)
{
    x += " world!");
}
...

string s = "Hello";

foo(s);
cout << s << endl;
```

C++ - reference parameter:

```
void foo(string & x)
{
    x += " world!";
}
...

string s = "Hello";

foo(s);
cout << s << endl;
```

Java:

```
void foo(StringBuffer x)
{
    x.append(" world!");
}
...

StringBuffer s = new
  StringBuffer("Hello");
foo(s);
System.out.println(s);
```

In addition to allowing a method to change the actual parameter, another reason for using reference parameters is efficiency.

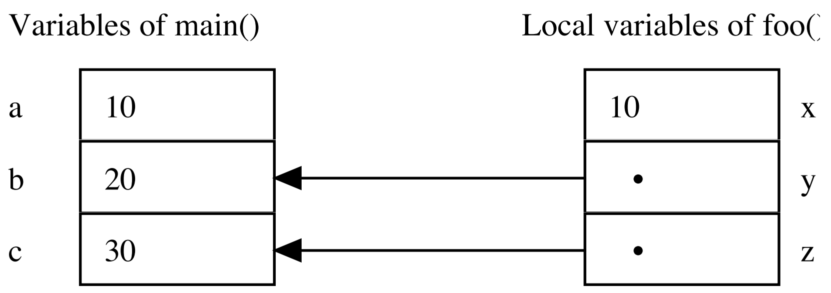
- When a parameter is passed by value, the actual parameter is *copied* into the formal parameter. In the case of large objects, this can mean utilizing considerable time and space for the copy. (This is one reason why C and later C++ specified that arrays are *always* passed by reference.)

- When a parameter is passed by reference, the formal parameter holds the memory address of the actual parameter - typically comparable in size (and copying time) to an `int`.

As a final example, consider the following (admittedly contrived) C++ code, in which three `int` parameters are passed to a function by value, pointer, and reference, respectively. (Note that we must use the & operator with the *actual* parameter `b` in `main()`, because `foo()` expects a *pointer* as its second parameter.)

```
void foo(int x, int * y, int & z)
{
     x ++;
     (* y) ++;      // y ++ would mean something rather surprising!
     z ++;
}

int main(int argc, char * argv[])
{
     int a = 10, b = 20, c = 30;
     cout << a << " " << b << " " << c << endl;
     foo(a, & b, c);
     cout << a << " " << b << " " << c << endl;
}
```

Its output will be `10 20 30 (newline) 10 21 31`. To see why, consider the following diagram of the state of memory when `foo()` is first entered:



c. Value, Pointer, and Reference Return Values from Functions/Methods in C++

We have seen that there are two good reasons for using a reference parameter to a method:

- When it is necessary for the method to alter the actual parameter passed to it.

- For efficiency - to avoid the need to copy a large value.

Similar reasons sometimes lead to methods that *return* a value by reference.

For example, C++ allows *operator overloading* - the possibility of giving standard operators meaning when used with a user-written class. Suppose we wish to define a class `Date` that represents calendar dates. We might want to overload the operator "+" to allow us to add an integer to a date, giving us another object representing a date that number of days in the future. (E.g. if we took the date object that represented July 4, 1776 and added 10 to it, we would get July 14, 1776.) If we define "+" in that way, we might also want to define "+=". However, in this case we want to get back *the same* date object we started with, but altered by the addition of some number of days. Note well the distinction: for "+" we want to get back a different object, and leave the original object unaltered; for "+=" we want to get back the same object, appropriately altered.

This could be accomplished by code like the following - assuming that our `Date` class represents a date value internally as an integer called `_dayNumber` (perhaps the number of days that have elapsed since some base date.) The example gives just the code for implementing the methods - obviously they would need to be declared in the class declaration, along with the `_dayNumber` field, etc.

```
Date Date::operator + (int offset) const
{
    return Date(_dayNumber + offset);      // Return a different object by value
}

Date & Date::operator += (int offset)
{
    _dayNumber += offset;                  // Modify this object
    return * this;                         // Return this object by reference
                                           //  Note that this is a pointer to the
                                           //  object to which the method is
                                           //  applied, so * this is the object
                                           //  and we return a reference to it
}
```

As another example, suppose we wanted to create a method to search a database of information about people and return the `Person` object for the person having a given name. For efficiency, we would want to avoid creating a new copy of the `Person` object. We could accomplish this by returning either a pointer or a reference to the `Person` object (depending on which is more convenient to use in subsequent code.) Thus, our method would have one or the other of the following signatures:

```
Person * lookup(string name)
```

or

```
Person & lookup(string name)
```

14. Const/final variables

a. Constant variables

Both Java and C++ allow a variable declaration to specify that the variable is really a constant - i.e. its value cannot be changed after it is declared. C++ uses the word "`const`" for this, while Java uses "`final`". The following, therefore, are equivalent:

C++	Java
`const int bjorksFavoriteNumber = 42;`	`final int bjorksFavoriteNumber = 42;`

b. Pointers and References to Constants in C++

In C++ it is also possible to use const with pointers and references to specify that the *object pointed or referred to* cannot be modified through the pointer or reference, as the case may be. E.g. the following does not say that p is a constant that must always point to the same object, but rather that the object that p points to (whatever it is) cannot be modified through p.

```
const Foo * p = ...;
```

Likewise, the following does *not* say that r is a reference that must always refer to the same object (which is true in any case), but rather that the object that r refers to cannot be modified through r.

```
const Foo & r = ...;
```

Given the above declarations, if `Foo` were a class having a mutator method `bar()` that potentially changed some attribute of the object, then Statement 1 is legal, while Statements 2, 3, and 4 would result in an error message from the compiler.

```
p = someNewPointer; // Statement 1 - legal given above declarations
p -> bar();             // Statement 2 - not legal given above declaration -
                        //  bar() could change the object p points to
r = someNewValue;       // Statement 3 - not legal given above declaration -
                        //  attempt to assign a new value to object r refers to
r.bar();                // Statement 4 - not legal given above declarations -
                        //  bar() could change the object r refers to
```

If we wanted to say that `p` itself could not be changed, the form of the declaration for `p` would be:

```
Foo * const p;
```

After such a declaration, Statement 1 above would be illegal, but Statement 2 would be fine!

c. Const and final are NOT equivalent when applied to methods:

One place where the C++ const and Java final have very different meanings occurs when they are applied to methods. The following are not at all equivalent:

C++	Java
`const Foo * bar()`	`final Foo bar()`

The C++ form says that `bar()` returns a pointer to a `Foo`, which cannot be used to modify the `Foo` object to which it points; the Java form says that `bar()` cannot be overridden in any subclass of the current class.

d. Constant Methods and Parameters to Functions/Methods in C++

It is also possible to use const with parameters to a function/method. This is most useful with pointer or reference parameters when the reason for using a pointer or reference is efficiency (to avoid copying a large object), but the method doesn't need to actually change the parameter.

Consider the Date class example used above. Suppose we wanted to define operator "-" to allow us to subtract to dates and get back an integer - the number of days between them. (E.g. if we subtracted January 1, 2003 from January 1, 2004 we would get 365.) In this case, we might want to pass the second date by reference (to avoid copying it), but wouldn't need to change it. The following code would accomplish this:

```
int Date::operator - (const Date & other) const
{
      return _dayNumber - other._dayNumber;
}
```

Note that *both* the parameter and the method are declared `const`. The former indicates that the method will not alter the "`other`" date; the latter ensures that "`this`" date will not be altered either - as discussed above in section 7d. (In effect, a `const` declaration at the end of a method prototype makes the implicit "this" parameter of the method a pointer to `const`).

There is a correlation between `const` methods and methods with `const` parameters in the following sense: if a method has a `const` parameter, it may only invoke `const` methods of that parameter, and if a method is `const`, it can only call const methods on its object, and can only pass its object (`this`) as a parameter to `const` formal parameters of other methods. Note that there is never any harm in applying a `const` method to a non-`const` variable; the `const` declaration is simply a promise that the method won't alter its object.

15. C-Style Character Strings in C++

One issue that any programming language library must face is support for character strings. The string type poses problems for programming language designers because:

- On the one hand, the type string has many similarities to primitive types. It is widely used in programs, it is desirable to be able to have constants of string type, and it is desirable to be able to apply various operators (e.g. +) to strings.

- On the other hand, strings have widely varying lengths, so it is not possible to assign a fixed size allocation of memory to store a string as it is for primitive types.

We have seen that both C++ and Java resolve this by providing a string class (`string` in C++, `java.lang.String` in Java). In addition, the Java compiler accords special treatment to this class, recognizing `String` constants. No other class is accorded this sort of treatment by the compiler.

The C++ `string` class is one of two ways of addressing this need in C++. C++ also uses "C-style" strings, inherited from C. The C language represents a string by an array of characters, terminated with the null character, `'\0'`. Because of the equivalence of arrays and pointers, this leads to strings being represented by the type `char *`. (Such strings are often declared as <u>`const char *`</u> when there is no need to manipulate the contents of the string.) The C library includes a variety of functions for working with such null-terminated array of char strings. These are declared in the header file `<cstring>`. Of particular significance is that when the compiler encounters as quoted string, it represents it by a C-style `char *` string, <u>not</u> a C++ `string` class object.

For most purposes, the C++ `string` class is superior to C `char *` strings. Because the latter are really arrays, they are prone to all sorts of memory-allocation related problems. In fact, vulnerabilities arising from the use of C style strings in certain Unix system programs has been exploited by a number of worm and "cracker" programs. This handout confines itself to using the C++ `string` class; but if you continue to work with C++, you will no doubt encounter a context in which you must use C style strings for compatibility with existing software. Note that the `string` class includes a constructor that converts C-strings to C++ strings, and a method `c_str()` for going the other way.

16. Error-handling and Exceptions

Both Java and C++ provide support for throwing and catching exceptions, using very similar syntax. A major difference is that, in Java, all exceptions must be members of subclasses of the class `java.lang.Exception`, which is, in turn, a subclass of `java.lang.Throwable`. In C++, an exception can be of any class.

Another important difference is that the Java API was designed from the beginning to use exceptions. A method that fails almost always signals failure by throwing an exception. The exception facility in C++ is a relatively recent addition in the history of the language, having been added after much of the core of what is now the standard library was defined. Thus, library routines typically report failure in some other way . The exception facility is primarily used to support user-created exceptions - though newer versions of the library are starting to use exceptions for reporting failure of library routines as well.

The place where the programmer most often must deal with errors from library routines is in connection with input-output, where problems can arise either from user error or failure of a physical device. The C++ io stream facility handles input output errors as follows: The io stream facility associates with each stream the notion of a "state". A stream can be in the `good`, `bad`, `fail`, or `eof` state. A stream is normally in the `good` state, and a stream in any state other than `good` ignores attempted operations on it. An operation that fails due to some problem like input format (e.g. inputting "A" when a stream is trying to read an integer) puts the stream in the `fail` state. An operation that fails due to some system problem (e.g. an irrecoverable read error) puts the stream in the `bad` state. An input operation that fails due to end of file puts the stream in the `eof` state. If the programmer attempts an operation that could fail, the programmer is responsible for testing the state of the stream afterward to see if it succeeded. There are several methods of a stream object for doing this - shown here as applied to `cin`, but they could be applied to any stream:

`cin.good()` - return true just if the stream is in the good state
`cin.fail()` - returns true just if the stream is in the fail or bad state
`cin.bad()` - returns true just if the stream is in the bad state
`cin.eof()` - returns true just if the stream is in the eof state

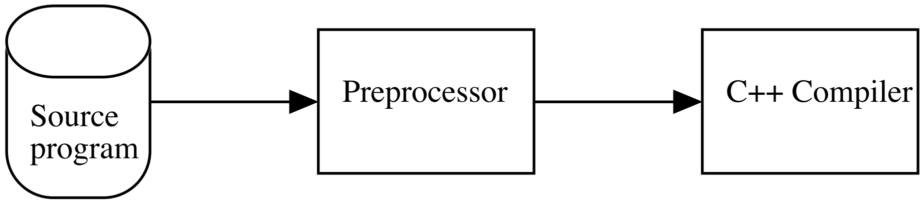
Alternately, the operator "!", when applied to a stream, succeeds just when the stream is <u>not</u> in the `good` state - e.g. code like the following could be used to test the success of an input operation:

```
cin >> someVariable;
if (! cin)
{
    take appropriate action for dealing with an error
...
```

17. Features of C++ for which there is no Analogue in Java (at least at the present time):

a. Preprocessor

One of the features that C++ inherits from C is the use of a preprocessor. When a C++ program is compiled, the source program is automatically run through a preprocessor before it is actually compiled. This is illustrated by the following diagram:



Source program lines that begin with a # in column 1 are recognized by the preprocessor as directives, and are handled by the preprocessor. Based on these directives, the preprocessor performs transformations on the program text, and the transformed text is sent to the compiler proper. The following are some of the more important preprocessor directives:

`#include <filename>`
`#include "filename"`
	These directives cause the contents of the specified file to be inserted into the program text at this point, replacing the directive. The form using < > searches for the file in the system library directories, and the " " form searches the file in the current working directory.

`#define symbol replacement`	Causes the specified symbol to be replaced by the replacement wherever it occurs in the remainder of the program text. This directive can be used to define constants - e.g.
	`#define PI 3.14159`
	(However, in C++, it is preferred to use a const variable instead, though - e.g.
	`const double PI = 3.14159; )`

`#if expression`
`#else`
`#endif`
	Causes the code that follows, up to the `#endif` directive, to be passed on to the compiler only if the expression evaluates to true. If `#else` intervenes between the `#if` and the `#endif`, the code following the `#else` is passed on to the compiler only if the expression evaluates to *false*. Note that this is all handled at compile-time; therefore, the expression can only involve values known at compile time.

b. Enumeration Types

Sometimes, it is useful to define a set of symbolic names to be used for some purpose. For example, the Java class `java.awt.Label` defines the names `LEFT`, `CENTER`, and `RIGHT` that can be passed as a parameter to the constructor to specify how the label should be aligned in its available space. Java uses `static final int` for this purpose. C and C++ have an enumeration facility that accomplishes the same thing in a safer way. This can be seen by comparing the actual source code from the Java library with C++ code that would do the same thing. Note that it would be possible for the C++ compiler to ensure that the alignment parameter is a valid alignment value, not an arbitrary int. The Java compiler cannot ensure this, so the Java code must test the value to be sure it is one of the three permitted ones.

C++

```
enum Alignment
      { LEFT, CENTER, RIGHT };

...
Label(string text,
      Alignment alignment);
...
```

Java

```
public static final int LEFT   = 0;
public static final int CENTER = 1;
public static final int RIGHT  = 2;
...
public Label(String text,
             int alignment)
...
```

c. Templates

One important facility in C++ is the template facility. It is possible to define a class or a function as a template, with one or more data types left unspecified, and then instantiate it one or more times for different data types. One place where this is used is the Standard Template Library (STL), a recent addition to the standard C++ library. The STL defines various container class templates (e.g. Lists, Stacks, Queues, Sets, Maps) that can be instantiated to contain elements of specific type). This is a more type-safe and efficient way of accomplishing what the Java collections framework does.

For example, suppose it is desired to create a list of strings. The following Java and C++ code accomplishes this. Note that the Java code would allow any kind of object to be inserted into the list - it cannot check to be sure that the object is a string. Likewise, in the Java code, it is necessary to cast the object removed from the list to `String`, since the list can contain objects of any type. (If an incorrect type object were inserted into the list, this cast would throw an exception.) The C++ version instantiates the list template for the specific class `string`, and the compiler will reject any code that attempts to insert some other kind of object into it.

C++

```
// Code to declare the list

list <string> stringList;
...

// Code to insert a string s at
// the end of the list.  The
// compiler will ensure that s
// is a string

stringList.push_back(s);

// Code to put the first item
// into variable f

string f = stringList.front();
```

Java

```
// Code to declare the list

List stringList = new ArrayList();
...

// Code to insert a string s at
// the end of the list.  The
// compiler will allow s to be any
// type of object

stringList.add(s);

// Code to put the first item
// into variable f

String f = (String) stringList.get(0);
```

d. Operator Overloading

In both Java and C++, the standard operators (+, -, <, &&, etc.) are defined for appropriate primitive types. With the sole exception of the + operator for for strings, they cannot be used with any other types. (E.g. it is not possible to use + with the class Integer, or with a user-defined class for complex numbers.)

In C++, it is possible to *overload* the standard operators in a class definition, allowing them to be used with objects of that class. For example, the following set of overloaded operator definitions might be appropriate in a class that represents calendar dates - allowing one to perform addition or subtraction between a date and an integer to compute some number of days in the future or the past, using the standard operators +=, -=, +, or -; to subtract two dates to get the number of days in between, using the standard operator -; or to compare two dates, using the standard comparisons <, <=, ==, !=, >, or >=. (Since overloaded operators are considered a special form of method, these declarations would appear with the other methods in the class declaration.)

```
// offset or difference is # of days

Date & operator += (int offset);
Date & operator -= (int offset);

Date operator + (int offset) const;
Date operator - (int offset) const;
int operator - (const Date & other) const;

bool operator <  (const Date & other) const;
bool operator <= (const Date & other) const;
bool operator == (const Date & other) const;
bool operator != (const Date & other) const;
bool operator >= (const Date & other) const;
bool operator >  (const Date & other) const;
```

Each of the operators must be implemented in the implementation file, just like any other method. We have already seen examples above of how these operators might be implemented. The following code shows how the subtraction operator might be used to compute a charge for an interval between a startDate and a finish date, given a dailyRate:

```
dailyRate * (finishDate - startDate)
```

e. Named (typedef) types

In Java, a data type name is either the name of a primitive type, or the name of a class. In C++, it is also possible to give a user-defined type a name using typedef. For example, the following gives an array of ten ints the name TenInts, and then defines a couple of variables of this type:

```
typedef int [10] TenInts;

TenInts a;
TenInts b;
```

f. structs and unions

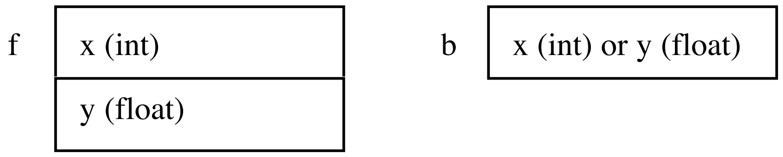
The ancestor of the C++ class is the C struct, which is still available in C++. A struct may be thought of as a class, less inheritance, encapsulation, polymorphism, and methods. (Actually, C++ structs can have methods, but C structs cannot.) The following two definitions are equivalent. Both define an entity that has two fields that are completely accessible.

```
struct Foo
{
    int a;
    float b;
};
```

```
class Foo
{
    public:
        int a;
        float b;
};
```

Both C and C++ also have the union - which is declared like a struct, except all the fields share the same storage - hence only one field can actually exist at a time. The union was an earlier way of providing for variant data structures now more securely done with variant subclasses of a base class. For example, contrast the following union with the previously defined struct in terms of how storage is actually allocated:

```
union Bar
{
    int x;
    float y;
};

Foo f;
Bar b;
```


f
x (int)
y (float)
b
x (int) or y (float)