

Unit 1. Transitioning to C++

1.1 C++ Introduced

This module of the course introduces the C++ programming language. An overview of the key features of C++ follows a discussion of its history and development as a programming language. This module also demonstrates compilation and execution of a simple C++ program.

Readings:

- **Required:**

Schildt, chapter 11. Remark: Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.

1.1.1 C++ Background

- [History](#)
- [Key Features](#)

History

C++ is a modern object-oriented programming language that supports many of the same features as Java, including classes, inheritance, polymorphism, and exceptions. As you know and will continue to learn, these features are excellent tools for creating and maintaining large software systems. Together they allow the software's design to follow the shape of the problem closely, which reduces the amount of code that needs to be written while at the same time maximizing the readability of the code.

C++ isn't an entirely new language. It is based on the C programming language. C was designed to be a small, simple language, which programmers could use to produce very fast code. C++ adds to C many of the Java-like features such as classes and inheritance. In doing so, C++ became a much larger language than C, but one better suited for large-scale projects. Because C++ compilers can compile C programs, C++ gained rapid acceptance in the market. Today, there are literally millions of lines of C++ code in use by a wide variety of software applications.

For those interested in the timeline, C was first invented in 1970. The language became an ANSI standard in 1980. The first book about ANSI C was published in 1983. C++, originally called "C with Classes," debuted in 1983. The name C++ was coined in 1983. The first C++ book was published in 1985. The Java programming language, in development for several years, debuted in 1995. The C++ language became an ISO standard in 1998.

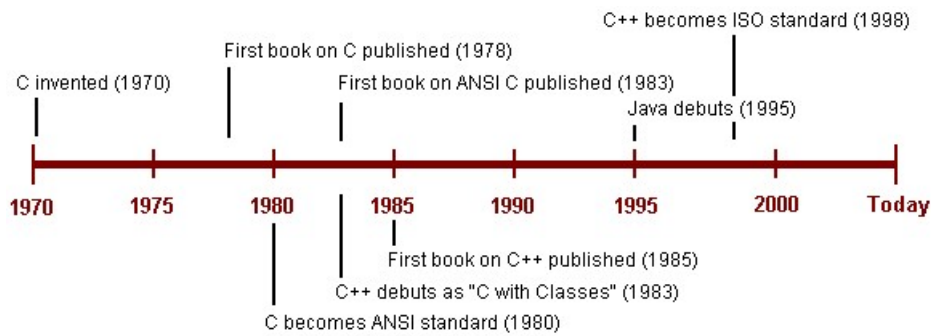


Figure 1 History of C, C++, and Java

Key Features

Here is a list of some of the key features of C++.

- C++ is strongly typed. This simply means that every object must belong to a certain type and that operations such as assignment or comparison are only permitted between objects of the same type.
- C++ has the concept of a class, a type of record that combines data members and the functions that operate on the data. Ignoring various minor differences, classes in C++ are very similar to classes in Java.
- C++ supports parameterized types, or templates. Templates make it possible to define, say, a single vector class that works for Booleans, characters, integers, reals, and so on. This can be done in Java by leveraging inheritance. For instance, a vector of type `java.lang.Object` can operate on other types since all classes eventually inherit from `java.lang.Object`. This Java approach, however, sacrifices static type checking. The real advantage of C++ templates is that their use does not prohibit the compiler from performing static, or compile-time, type checking.
- Similar to Java, C++ supports inheritance, a mechanism that makes it possible to build new classes (called derived classes) on top of an existing class (called the base class) without having to reiterate the base class design for each new class.
- C++ supports polymorphism. In C++, polymorphism is achieved through the use of virtual functions and pointer variables. Together with inheritance, this turns C++ into a full-fledged object-oriented language.
- C++ comes with two libraries known as the Standard Library and the Standard Template Library (STL)—both of which extend the capabilities of the base language. The Standard Library supplies all the old C libraries as well as new input and output facilities. The STL provides a library of container types (types that hold or "contain" collections of objects) as well as a set of attendant algorithms (which are general-purpose algorithms for common data structures). That is, the STL supplements the built-in types of C++ with vectors, linked lists, and other useful types.

- C++ has a very large user base. From all over the world, public and private companies, government agencies, academics, and hobbyists use C++ in all types of interesting ways and applications. A major benefit of this large user base is the wide availability of different tools, libraries, and tutorials, all related to some aspect of the language.

1.1.2 Compiling and Running a C++ Program

- [Hello World!](#)
- [Phases of C++ Program Development](#)
 - Edit
 - Preprocess
 - Compile
 - Link
 - Execute
- [Compiling and Running in Cygwin](#)

Hello World!

To get an idea of what a C++ program looks like, we can look at a very simple example. The Hello World! example remains a popular first program when learning any programming language. What follows is a C++ implementation of Hello World!

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7
8      // Say hello.
9      cout << "Hello world!" << endl;
10
11     /* indicate normal termination */
12     return EXIT_SUCCESS;
13 }
```

It is important to notice that C++ source code resembles Java source code. Many of the keywords in the above listing (`int`, `void`, `return`) have similar meanings in Java. In C++, as in Java, a semi-colon is used to terminate the end of a line of code. Also, curly braces define the beginning and ending of functions much the same way in Java.

Source code comments look the same in C++. Actually, in the above listing, we can see two different types of C++ comments in use. The first, in line 8, uses the double slash style of comment. In line 11, we see a second style that allows a comment to span multiple lines. C++ and Java share many other similarities. In summary, we can say that C++ and Java have similar syntax. The syntax of a programming language describes the words and symbols that are important to a programming language. Syntax also describes how programmers can arrange these words and symbols to achieve some higher-level meaning. The syntax of C++ and Java Variable declaration and initialization, operators, control structures, and function declaration are areas where C++ and Java share other similarities in syntax.

Phases of C++ Program Development

C++ programmers must perform five steps, *edit*, *preprocess*, *compile*, *link*, and *execute*, to produce an executing copy of a program.

Edit

The first step involved in taking a program from source to execution is the creation of a file that contains the source code. The program that is used to create a source code file is called an editor. Editors that programmers use range from simple and generic text editors (such as Notepad in Windows or vi in UNIX) to sophisticated editors that typically come as part of Integrated Development Environments (IDEs). These sophisticated editors are quite powerful since they provide functionality that is geared towards the creation and maintenance of source code. The syntax coloring in Listing 1 is one example of the type of functionality that sophisticated editors provide.

Preprocess

Preprocessing involves the modification of C++ source code files prior to compilation. The first two lines from Listing 1 contain commands called *preprocessor directives*, which inform the *preprocessor* to perform some action. In Listing 1, the first two lines instruct the preprocessor to include the contents of files into the program source code. Preprocessing also involves the text substitution of *macros*. A more detailed discussion of the preprocessor can be found in [1.3.4 The Preprocessor](#).

Compile

Preprocessing usually is performed automatically, just before the compile step. Compiling is a complex process that converts the preprocessed source code into *object code*. Part of the compile process involves verifying that the syntax of the source code is valid. Often, when a program is compiled (especially the first time it is compiled), something is wrong with the syntax. This is referred to as a "compile error." When faced with a compile error, a programmer must return to the first step of this process and edit the source code to remove the error.

The software tool used to compile source code, not surprisingly, is known as a *compiler*. An example of a C++ compiler is the GNU Compiler Collection, or GCC. The GNU Compiler Collection actually compiles many different programming languages, one of which is C++. GCC is also free software. This compiler can be obtained through the Cygwin environment.

Link

Linking is a step that is typically performed by the same tool that compiles a program. The linking process involves combining the object code produced by the compiler with other precompiled library code. The result of the operation of the linker is an executable image. This executable image is a file that contains the compiled and linked object code of the program, stored in persistent storage (the hard drive).

Execute

After the program source code has been edited, preprocessed, compiled, and linked into an executable image, the program is ready to be executed, or run. Errors encountered at this point are known as "runtime errors," and are typically more difficult to correct than compile errors, since they may involve problems in the logic of the program.

Compiling and Running in Cygwin

Quite often, however, the steps preprocess, compile, and link are often informally grouped together and referred to as "compiling." It is easy to see why this is done when we consider that the tools that programmers use often perform these groups of related tasks together. For example, using GCC through Cygwin, we can preprocess, compile, and link all in one step. The following command line assumes the source code file is named `hello.cpp`.

```
$ g++ hello.cpp
```

Example 1 Compiling `hello.cpp`

As a result of the execution of the above command, the source-code file `hello.cpp` will first be preprocessed, then compiled, and then linked to produce an executable image named `a.exe`. The default filename `a.exe` can be overridden using the `-o` compiler option.

```
$ g++ hello.cpp -o hello.exe
```

Example 2 Overriding the default output file name

Additional options (`-ansi` and `-pedantic`) inform the compiler that it should only compile programs that are [ANSI](#) C++ compliant and that it should issue warnings when it encounters code that violates the [ANSI](#) standard. Using these options helps to locate and prevent certain types of errors.

```
$ g++ -ansi -pedantic hello.cpp -o hello.exe
```

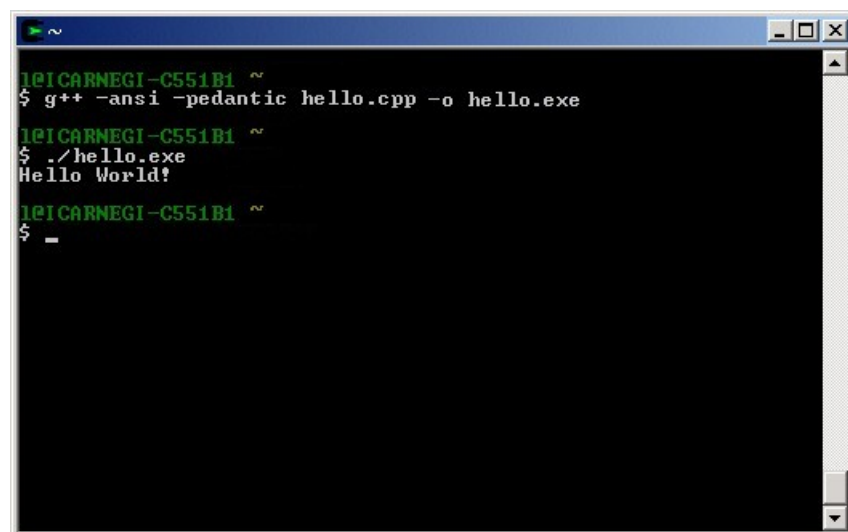
Example 3 Conforming to the ANSI standard

We can run the program now that it has been compiled into an executable. From the command shell, we issue the following command:

```
$ ./hello.exe
```

Example 4 Running the compiled program

The dot and slash preceding the filename may not be necessary. It simply instructs the command shell to look for the file `hello.exe` in the current directory. The following screen shot depicts the results of the compilation and execution of the program.



```
1@ICARNEGI-C551B1 ~
$ g++ -ansi -pedantic hello.cpp -o hello.exe
1@ICARNEGI-C551B1 ~
$ ./hello.exe
Hello World!
1@ICARNEGI-C551B1 ~
$ _
```

Figure 1 After compilation and execution

The compiling and linking of programs that consist of many files can be simplified using a makefile. A makefile is a text file that defines the relationships and dependencies between the files that are used to create a project. The main advantage of using a makefile is that they allow a programmer to compile only those source code files that have changed since the last compile. This can be very time saving when working with projects that consist of many source code files. Understanding the syntax and rules available in makefiles is beyond the scope of this course. It is important, though, to know that once a makefile is defined, the utility make is used to build the project associated with the makefile. From a command-line prompt, a programmer issues the command make to build the makefile in the current directory. As an example, you can use this `Makefile` to build the `Hello world!` project.

1.2 Data Structures and Algorithms in C++

With this module, the course introduces some of the basic concepts of data structures and algorithms.

1.2.1 What are Data Structures and Algorithms?

- [Data Structures](#)
- [Algorithms](#)

Data Structures

In just about every aspect of today's modern world, we encounter information in all shapes and sizes. The amount of information we encounter is sometimes so large that without an effective method of storage and representation, the information is rendered useless. Think of all the books contained in your local library. Without the storage system implemented by the librarians, it would be virtually impossible to find a specific book.

Even when faced with just a small amount of information, a structured representation can prove to be invaluable. For instance, think of the problems that could arise if people did not stand in line while waiting to purchase tickets at a movie theater. The line prevents the people waiting from just standing around arguing over whose turn is next. In a sense, the line serves as a way to store information.

A data structure, in the simplest terms, is a structured representation of information. Both the system used to store books in the library and the line of people at the movie theater are real world instances of data structures. Considering a larger context, data structures typically represent, or store, data to facilitate solving some problem. In the library example, the problem that the data structure helps solve is locating a specific book. At the movie theater, people wait in a line so it is easy to determine whose turn is next.

A data structure can be composed of simple pieces of data. Consider your address as a data structure. There are several items in this data structure, all of which are simple pieces of data. First, there is the street address, which is usually a number followed by a street name. Add to that the city name, the state or province abbreviation, and the postal code, and you have a very useful data structure composed simply of numbers and words.

Data structures can also be complex in that they can contain other data structures. For example, a library contains many bookcases, which in turn contain many books. Sitting on the shelf of one of these bookcases could be a cardboard box that also contains a few books. In this example, the bookcase is a data structure composed of books and boxes of books. The boxes are also data

structures, since they store books as well. Here we have a data structure (a bookcase) that is composed of another data structure (a box). Both data structures provide a way to store information (the books) to help solve some problem (finding a specific book). A filing cabinet full of folders of papers is another good example of how data structures can be composed of other data structures. In this case, the cabinet as a whole is a data structure. The cabinet is composed of cabinet drawers, which are composed of folders, which are composed of files.

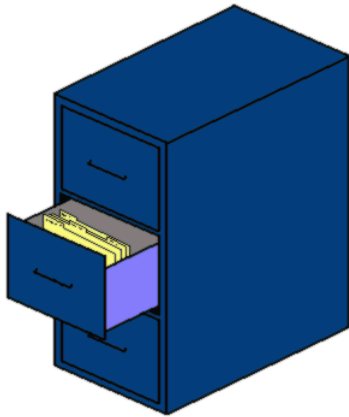


Figure 1 A filing cabinet as a data structure

Algorithms

In addition to having a data structure that represents information, some sequence of actions, or series of steps, needs to be taken to solve a problem. The series of steps that manipulate and uses a data structure to solve a problem is called an algorithm. Let's think back to the example of the line at the movie theater. Here, the problem at hand is the management of a group of people waiting to buy tickets. The data structure is a line of people. The algorithm involves removing a person from the beginning of the line (until the line is empty) when the ticket cashier becomes available.

1. Wait for the cashier to become available.
2. Remove the customer that is at the beginning of the line.
3. Let the cashier help this removed customer.
4. Go to step 1.

The steps contained in an algorithm are performed in a mechanical fashion. From the example above, first we do step 1, then we do step 2, then we do step 3, then we repeat with step 1. Algorithms can involve this type of repetition, and they can also involve simple branching or decision making (if "this," then do "that"). Since computers can perform these tasks very rapidly, it should be no surprise that algorithms play such a prominent role in computers and computer science.

A recipe for baking a cake is another good example of an algorithm. In a recipe for a cake there are clearly defined steps that are followed. First, the oven has to be preheated. Then we have to mix the ingredients. The cake batter is then poured into a pan, which is then placed into the oven, and so on. The algorithm involved in baking a cake has clearly defined inputs (the ingredients) and a clearly defined output (the cake).

1.2.2 Problem Solving with Data Structures and Algorithms

- [Multiple Solutions](#)
- [Multiple Representations](#)
- [Refining the Representation](#)
- [Decomposing the Representation](#)

Multiple Solutions

One of the prevailing questions we address in this course is how do we use data structures and algorithms to solve problems? With so many different types of problems, and so many different possible representations and algorithms, finding a solution for a problem may seem like a daunting task. One fact that is on our side is that typically there is not just one solution for any given problem. Often there are many solutions to a problem, each with advantages and disadvantages. Sometimes enough solutions exist that the real problem is determining which approach to take based on these trade-offs.

A library contains an example of multiple approaches to solving a problem, and the advantages and disadvantages of each approach. Here we are faced with the problem of storing a large number of books in a way that allows users of the library to find specific books easily.

One solution to this problem involves using a hierarchical system to store together books on related topics. For example, this approach stores all books on "Politics" together. It further separates and groups books into sub-topics of "Politics." One shelf would contain the books on "History of Politics" while another shelf would contain the books on "Modern Politics." The representation continues to group the books into more detailed categories. Perhaps it divides the books in "History of Politics" into "History of Spanish Politics," "History of Brazilian Politics," "History of Italian Politics," and so on. The algorithm to find a specific book involves a person walking first to the "Politics" bookshelf, then to the "History of Politics" shelf, and so on until they find the book they seek.

Another representation involves storing books alphabetically by author. In this system, perhaps only top-level categories exist. Within these categories, the representation arranges books alphabetically by the name of the book's author. When we find the book we are looking for in this system, we would also find all the books written by the same author next to our book on the same shelf. After we found a book using the first representation, we would find the other books on the same specific topic, and not the books written by the same author. This difference could be an advantage or disadvantage, depending on the reason for locating the book in the first place. For instance, when researching a specific topic it would be helpful to see all the other books on the topic.

Multiple Representations

We can sometimes solve two (or more) very different looking problems using the same data structure and algorithm. Consider the list of the following problems and corresponding solutions:

Problem	Solution
Remembering which groceries to buy	A grocery list
Tracking inventory by category and sub-category	A category list of sub-categories
Grading students over an entire semester	A list of students each with a list of grades
Managing your work tasks	A list of tasks and priorities
Performing the pre-flight procedure for an airplane	A check- list

Table 1 Different problems with similar solutions

Each of the problems in the table above essentially is solved using some type of list to represent the data. The algorithms involved in solving the problems may vary slightly, but essentially the algorithm traverses each list and performs some action (updating a grade, marking an item as complete).

Refining the Representation

The refinement of an initial representation can often lead to a more elegant solution. Consider the problem of garbage removal from an office building. We'll add a restriction that recyclable materials (we will just consider paper) must be separated from the non-recyclable garbage.

One way to represent the problem is to simply provide a "recycle" dumpster and a garbage dumpster outside the building. The algorithm to solve the problem involves each employee carrying their garbage (with their hands!) outside to these dumpsters, where they then separate the paper from the rest of the garbage. This works, and it solves the problem, but it would probably lead to unhappy employees who will grow tired of the tedious process of walking outside and back each time they want to throw something away. We can optimize our representation slightly by placing trashcans on each floor of the office building. The algorithm for this representation requires each employee to carry their garbage and paper to the trashcan on their floor. Then, at the end of the day, someone carries the trashcans from each floor to the dumpsters outside, where they separate the paper from the garbage. This again solves the problem, and it probably will make the employees happier, but it still requires employees to carry their garbage to a central trashcan. It also is a lot of work for the person who has to empty and sort all the floor trashcans. Let's refine the representation even further by placing trashcans in each office. We'll also add wheels to the trashcans that reside on each floor. Now, each employee throws their garbage and paper into their own office trashcan and at the end of the day, one person wheels the floor trashcan from office to office, emptying the office trashcans. They then separate this garbage outside into the two dumpsters. We've reached an improved solution, but one that still involves a good deal of sorting and separating of the garbage and recyclable paper.

What if we optimized by placing, in addition to the garbage can, a recycle basket in each office? We will also attach a recycling basket to the trashcan on wheels. This basket collects the paper thrown into the office recycle baskets. In this representation, each employee throws their unneeded paper into their recycle basket and their garbage into their garbage can. At the end of the day, one person walks around each floor with the trashcan on wheels, emptying both the garbage can and the recycle basket from each office. They then empty the can and the basket into the appropriate dumpsters outside. This solution keeps the employees happy, and reduces the work of the person who takes the garbage and paper outside. Actually, this refined representation eliminates the separating of garbage altogether. This is really an optimal solution!

Decomposing the Representation

After a representation of a problem has been established, the next step that can be taken involves identifying the key entities, or objects, that make up the representation. Just as important as identifying these objects, we will try to understand the relationship between these objects. This process of identifying objects and relationships is known as decomposition. Decomposition, in the basic sense, involves the "breaking down" of a problem representation into its core components.

Looking back at our refined representation of the garbage collection problem, let's try to identify some of the key objects. There are definitely a lot of office garbage cans and recycle baskets, since every office has one of each. We also have two cans on wheels for each floor (one for regular garbage, and one for recyclable paper), and a recycle dumpster and a trash dumpster

outside the office building. Listing these objects, we can see we basically have different types of things that store garbage.

- outside garbage bin
- outside recycle bin
- garbage can on wheels
- attached recycle basket
- office garbage can
- office recycle basket

All of these objects are just bins of some sort since they essentially store materials. A bin that stores garbage is just a specific type of a bin. Likewise, a bin that stores recyclable paper is another type of bin (even though we call it a "basket"). From our list above, we can get more specific with the types of bins that our representation uses:

- GarbageBin is a type of Bin
- RecycleBin is a type of Bin
- GarbageDumpster is a type of GarbageBin
- RecycleDumpster is a type of RecycleBin
- OfficeGarbageCan is a type of GarbageBin
- OfficeRecycleBasket is a type of RecycleBin
- FloorTrashCan is a type of Bin, it contains a GarbageBin and a RecycleBin

Our decomposition now illustrates the relationships between the objects. This is an important step in solving any problem since understanding relationships helps allows us to better design a solution.

1.3 Basic C++ Programming

With this module, the course introduces some of the basic concepts and features of the C++ programming language. Whenever possible, comparisons are drawn between Java and C++ to introduce and illustrate key differences and similarities across the languages.

Readings:

- **Required:**

Weiss, chapter 2. Remark: Remember that this book supplements the course's online material. You will be asked questions based on this material.

- **Required:**

Schildt, chapters 12, 19 - 21. Remark: Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.

1.3.1 Data Types

- [Fundamental Data Types](#)
- [Strings](#)
- [Arrays](#)
- [Vectors](#)
- [Creating New Data Type Names](#)

Fundamental Data Types

As in most other programming languages including Java, C++ categorizes data objects into different types. These different data types describe not only the fundamental operations that the language performs on the data, but also the range of values that the data types accept. Since different data types have different allowable values, the language can check to ensure that a programmer only assigns appropriate values to a data object. An error occurs if a programmer assigns an inappropriate value, such as a number that is too large or too small, to a data object. This mechanism is known as type checking. C++ is considered a strongly typed language since it is very strict about checking data types and their corresponding values. In the end, this is good for the C++ programmer, since strongly typed languages can detect errors that other languages may not detect.

The fundamental data types that C++ supports are similar to the primitive data types of Java. The following table lists each of the fundamental data types of C++ and the corresponding Java primitive data type.

C++ type	Java type
<code>bool</code>	<code>boolean</code>
<code>char</code>	<code>char</code>
<code>int</code>	<code>int</code>
<code>short</code>	<code>short</code>
<code>long</code>	<code>long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>

Table 1 C++ and Java data types

The space required to store variables differs across languages. In C++, the storage space requirements are left to the discretion of the compiler implementers. Unlike Java, these requirements are not specifically stated by the language standard. Therefore, on one platform a C++ `int` may be two bytes long, while on another it may be four bytes long. But, in Java, a variable of type `int` is guaranteed to require four bytes of memory to store. Usually, the size in bytes of a variable is unimportant, except that it dictates the range of values that a variable can accept. As the number of bytes that a language implementation uses to store a variable increases, so does the range of values that the variable accepts. The following example program uses the C++ `sizeof` operator to display the size, in bytes, of the fundamental data types of the particular C++ implementation used in compilation.

```
1 #include <cstdlib>
```

```

2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7
8      cout << "  bool: " << sizeof(bool) << endl;
9      cout << "  char: " << sizeof(char) << endl;
10     cout << " short: " << sizeof(short) << endl;
11     cout << "   int: " << sizeof(int) << endl;
12     cout << "  long: " << sizeof(long) << endl;
13     cout << " float: " << sizeof(float) << endl;
14     cout << "double: " << sizeof(double) << endl;
15
16     return EXIT_SUCCESS;
17 }

```

Listing 1 [Data types and the sizeof operator](#)

Like Java, C++ contains a mechanism to create "read-only" variables. C++ uses the keyword `const` to allow programmers to create "read-only" variables. This keyword signals that the variable being declared cannot be modified after it is initialized. The keyword `const` of C++ is analogous to the keyword `final` in Java. Listing 2 shows the declaration and initialization of some "read-only" variables, otherwise simply known as "constant variables" or just "constants."

```

1  const int BOILING_POINT = 100;
2  const int FREEZING_POINT = 0;
3  const float PI = 3.14159;

```

Listing 2 Constant variables

In the above listing, notice that the variable names of the constants appear in uppercase. This is not required, but is a standard that professional C++ programmers tend to follow. Naming constants in all uppercase allows a programmer to easily recognize and recall that an identifier is a constant variable.

Strings

In C++, the `string` data type provides the necessary abstraction to allow C++ programmers to work with character strings. The Java counterpart is actually two separate classes. Class `java.lang.String` and class `java.lang.StringBuffer` provide Java programmers with character string support, or what we typically just refer to as "strings." These classes are part of the core `java.lang` package, and thus are available to all Java programs by default. Unlike its Java counterparts, the C++ `string` type is not available to all programs by default. If a C++ program requires the `string` type, the programmer must refer to the library that defines this type. The following listing illustrates the C++ `string` data type in action.

```

1  #include <iostream>
2  #include <string>
3  #include <cstdlib>
4
5  using namespace std;
6
7  int main(int argc, char* argv[]) {
8

```

```

9   string s1 = "first";    // Initialization
10  string s2;
11
12  s1 += " string";        // Concatenation
13  s2 = s1;                // Assignment
14
15  cout << s1 << endl;     // Stream output
16  cout << s1.length() << endl; // Length
17
18  return EXIT_SUCCESS;
19 }

```

Listing 3 [string variables in use](#)

The inclusion of the preprocessor directive found in line 2 in the above listing is necessary to allow the program access to the `string` data type. Notice also the various methods and functionality of the `string` variables that the example demonstrates. In line 9, a string literal initializes a string object. Lines 12 and 13 demonstrate concatenation and assignment. To display the contents of a string variable, we can use basic console output as shown in line 15. This listing demonstrates only some basic functionality of the `string` data type. Many other useful functions exist.

Arrays

C++ provides basic support for a sequence of homogeneous data objects through arrays. Both similarities and differences exist between Java arrays and arrays in C++. Let us first consider the syntax for declaration and initialization of arrays. In Java, we can declare and initialize an array of ten `int`s using the following.

```

1 // declare and create two arrays of integers
2 int[] javaArray1 = new int[10];
3 int javaArray2[] = new int[10];

```

Listing 4 Arrays in Java

The C++ syntax equivalent to the above Java code looks similar, except for the use of the keyword `new`. The keyword `new`, in Java, creates an instance of an object. In C++, the keyword `new` has a different meaning, one that we address in [1.4.3 Dynamic Memory Management](#). The following listing demonstrates the declaration of an `int` array of size ten in C++.

```

1 // declare and create an array of integers
2 int cpp_array[10];

```

Listing 5 An array in C++

The double bracket (`[]`) in the declaration indicates that the line declares an array. Notice the placement of the double bracket (`[]`) in each of the examples. In Java, the double bracket can be placed after the name of the data type or after the name of the variable. In C++, it can only be placed after the name of the variable.

Accessing the elements stored in an array is done the same way in C++ as it is in Java. Both languages use the bracket operator (`[]`). A programmer encloses in brackets an index number of the element they wish to access. Indexing of arrays in C++, as in Java, begins with zero. This means that the first element of an array is accessed at index `0`, the second element at index `1`,

the third element at index 2, and so on. The following listing demonstrates accessing the elements of a C++ array.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7
8      int arr[25];
9
10     for (int i = 0; i < 25; i++) {
11         arr[i] = i;
12     }
13
14     cout << "The first element equals: " << arr[0] << endl;
15     cout << "The second element equals: " << arr[1] << endl;
16     cout << "The last element equals: " << arr[24] << endl;
17
18     return EXIT_SUCCESS;
19 }
```

Listing 6 [Accessing elements of a C++ array](#)

One inherent danger that exists in using C++ arrays is the lack of bounds checking. In Java, if we attempt to access an index of an array that is outside the bounds of the array, we generate an exception. This is because the language actively checks our attempts at array indexing to ensure that we access only valid array elements. This is not the case in C++, since boundary checking of arrays is not supported. If we have an array of ten elements in C++, we can attempt to access the 12th, or 20th, or even 100th index of the array. Depending on a few different things, the program may or may not "crash" as a result of our out-of-bounds access. We can be sure, however, that the data we obtain from an out of bounds access will not be meaningful. The following listing shows an out-of-bounds array access in C++.

```
1  int arr[10];
2  cout << arr[11] << endl;
```

Listing 7 Out of bounds access

The code in the above listing, in the context of an actual C++ program, definitely compiles, and it may even run without error. What value does line 2 display? It depends on a few things, but typically, that value will be the value of another variable in the program. This is clearly a dangerous practice, and one that we will always avoid.

Vectors

The `vector` data type provides a much safer alternative to a basic C++ array. In C++, as in Java, vectors exist as feature-rich array. For example, unlike an array in C++ a `vector` has a built in function that returns the size of the vector. Vectors also provide bounds checking support, and unlike arrays, they automatically increase in size when the need arises. Page [2.2.2 Using the STL vector Container](#) contains a complete discussion of type `vector`.

Creating New Data Type Names

It is possible in C++ for a programmer to create additional names for existing data types. Creating another name uses the keyword `typedef`. The syntax to create a new name is as follows.

```
1 | typedef type-expression new-name;
```

Example 1 Usage of typedef

The following listing contains a few examples of the use of the keyword `typedef`.

```
1 | #include <iostream>
2 | #include <cstdlib>
3 |
4 | using namespace std;
5 |
6 | typedef int my_int;
7 | typedef my_int* my_int_ptr;
8 |
9 | int main(int argc, char* argv[]) {
10 |
11 |     my_int i = 10;
12 |     my_int_ptr ptr = &i;
13 |     cout << *ptr << endl;
14 |
15 |     return EXIT_SUCCESS;
16 | }
```

Listing 8 A sample use of typedef

1.3.2 Specifying Classes

- [Basic Syntax](#)
- [Constructors](#)
- [The Destructor](#)
- [Declaration vs. Definition](#)

Basic Syntax

The class is the basic unit of abstraction in C++. As in Java, we can use classes to specify and then instantiate objects. The basic syntax involved in specifying a class and in instantiating an object differs between Java and C++. Let's look first at a simple class specified in Java, and then the corresponding version in C++.

```
1 | /**
2 |  * The Java BankAccount class
3 |  */
4 | public class BankAccount {
5 |
6 |     private double sum;
7 |     private String name;
8 |
9 |     public BankAccount(String nm) {
10 |
```

```

11         name = nm;
12         sum = 0;
13     }
14
15     public double balance() { return sum;}
16     public void deposit(double amount) {sum += amount;}
17     public void withdraw(double amount) {sum -= amount;}
18     public String getName() { return name;}
19 }

```

Listing 1 [A Java BankAccount class](#)

The above listing declares a rather basic Java class. The class represents a bank account, and provides some basic bank account related operations. The following listing shows the equivalent C++ version of class BankAccount.

```

1  class BankAccount {
2
3  private:
4      double sum;
5      string name;
6
7  public:
8      BankAccount(string nm) : name(nm), sum(0) {}
9
10     double balance() { return sum;}
11     void deposit(double amount) {sum += amount;}
12     void withdraw(double amount) {sum -= amount;}
13     string getName() { return name;}
14 };

```

Listing 2 A C++ BankAccount class

A few key differences distinguish class specification in C++ and in Java. First, notice the different use of access modifiers. The Java example above repeats an access modifier for each data member and each member function. Access modifiers in C++ do not repeat for each data member. Instead, C++ uses one access modifier to delimit a section of the class definition. All data members within that section share the access level of the delimiting modifier. In the above listing, everything defined below line 3 and up until the next access modifier in line 7 has `private` access.

Unlike Java, there is no notion of a "public" or "private" class in C++. A C++ program can use any class as long as the class declaration is included in the program. We cover the various ways of including classes in C++ in page [1.3.4 The Preprocessor](#).

Another key difference in class specification is that C++ class declarations must end with a semicolon. This semicolon appears in line 14 of the above example. This is a very subtle difference from Java and one that is often the source of many cryptic compiler error messages.

Constructors

Constructors are the methods of a class that define what actions to take when creating an object. A C++ class can have multiple constructors. This allows variation in object instantiation since different numbers and types of parameters can exist in each constructor. The following listing is a modified version of the C++ `BankAccount` class. This modified version includes an additional constructor.


```

1  class BankAccount {
2
3  private:
4      double sum;
5      string name;
6
7  public:
8      BankAccount(string nm) : name(nm), sum(0) {}
9      BankAccount(string nm, double bal) :
10         name(nm), sum(bal) {}
11
12     double balance() { return sum;}
13     void deposit(double amount) {sum += amount;}
14     void withdraw(double amount) {sum -= amount;}
15     string getName() { return name;}
16 };

```

Listing 3 Initializer lists and multiple constructors

The use of initializer lists in constructors is the preferred way to specify initial values for class data members. Initializer lists are comma-separated variable initializations that appear prior to the body of a constructor. An example of initializer lists appears in Listing 3. Everything in line 8 following the colon and preceding the empty curly-braces comprises the initializer list. This initializer list sets the initial value of the private data member `name` equal to the value of parameter `nm`. It also sets the initial value of the data member `sum` equal to zero.

In C++, objects are created, or instantiated, using a syntax similar to regular variable declaration. Unlike Java, C++ does not rely on the use of the keyword `new` to handle object instantiation. C++ can use the keyword `new` to instantiate objects, but this has a different effect, one that we explore in page [1.4.3 Dynamic Memory Management](#). Programmers declare and instantiate objects in C++ using syntax identical to the declaration of fundamental data types. Listing 4 demonstrates instantiation of class `BankAccount`.

```

1  BankAccount account1("checking");
2  BankAccount account2("savings", 200);
3
4  account2.withdraw(100);
5  account1.deposit(100);

```

Listing 4 Object instantiation

C++ instantiates an object when the line of code containing the object declaration executes. Object instantiation involves the execution of a class constructor. Listing 4 declares two different `BankAccount` objects. Instantiation occurs when the code contained in lines 1 and 2 executes.

The Destructor

A destructor is a special member function of a C++ class called when an object's lifetime ends. Like a copy constructor, only one destructor can exist for a class. Since they execute when an object's lifetime ends, destructors typically define the actions necessary to release any resources that an object may be using. For example, consider an object that opens a connection to a database. When this object's lifetime ends, the destructor could close the database connection.

We examine more important uses of destructors in page [1.4.3 Dynamic Memory Management](#) when we discuss dynamic memory management. Until then, we can look at a listing to see at least what the definition of a destructor looks like in C++.

```

1 ~BankAccount() {
2     if (balance() < 0) {
3         cout << "warning: negative balance!" << endl;
4     }
5 }

```

Listing 5 The destructor

The difference between the definition of a destructor and a constructor is very subtle. Notice in line 1 of Listing 5 that a tilde (~) exists in front of the name of the class. This signifies that this member function is the destructor for the class.

Declaration vs. Definition

In this discussion on the specification of classes in C++, the term "definition" has been used regarding functions. When we "define" a function, we dictate the function's behavior through the code that exists within the curly braces. The "declaration" of a function, on the other hand, only specifies the interface of the function. This interface includes the function name, the return type, and the list of parameters and their types. The following listing shows both a declaration and definition of the function `average`.

```

1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 // function declaration
7 double average(int, int);
8
9 int main(int argc, char* argv[]) {
10
11     cout << average(10, 2) << endl;
12     return EXIT_SUCCESS;
13 }
14
15 // function definition
16 double average(int total, int count) {
17     return (total / count);
18 }

```

Listing 6 [Declaration vs. definition](#)

1.3.3 Input and Output

- [Streams](#)
- [Using the Standard Streams](#)
- [File Input and Output](#)
- [Some Common Pitfalls](#)

Streams

Input and output in C++ is based on the concept of a stream. A stream is a sequence of bytes that flow from something to something else. The process of output involves moving bytes, one at a time, from a program to a device. This device could be a monitor, a printer, or even a file on a hard drive. Input is the opposite. Input involves the flow bytes from a device (a keyboard, a file, a

network connection) into the program.

```
1  #include <string>
2  #include <cstdlib>
3  #include <iostream>
4
5  using namespace std;
6
7  int main(int argc, char* argv[]) {
8
9      cout << "Enter your name: ";
10
11     string name;
12     cin >> name;
13
14     cout << "Hello " << name;
15
16     return EXIT_SUCCESS;
17 }
18
19
```

Listing 1 Stream based output

Listing 1 is a simple example of stream based input and output. The above code first streams data (the characters in the text string "Enter your name: ") from the program to a device (the console) in line 9. Stream based output operations use the `<<` operator to indicate the data to write to the stream. The stream used in this line is the output stream referenced by object `cout`. This object is of type `ostream`, which is short for "output stream."

The program in Listing 1 then streams data from the keyboard into the program, storing the user entered text in the variable `name`. Stream input operations use the `>>` operator to specify the variable where the program should place the data it reads from the stream. The stream used in this line is the input stream referenced by object `cin`. The `cin` object is of type `istream`, which is short for "input stream." The listing then again streams output to the console. In line 14, we see that we can place more than one piece of data into the stream in one statement.

We can open and use streams to read and write data to and from many devices besides the console. For example, a program can use a file output stream to write data to the file system. Network communication through sockets is also stream based.

Using the Standard Streams

Three specific streams are always available throughout the lifetime of every C++ program. These are the standard input, standard output, and standard error streams. Each of these standard streams has a specific use. The standard input stream reads data from the console, the standard output stream writes data to the console, and the standard error stream displays error messages to the console.

Programmers access the standard streams through a set of objects. Objects `cin` and `cout` provide access to the standard input and output streams, respectively. We have seen their use in previous examples. Object `cerr` provides access to the standard error stream. Listing 2 demonstrates use of the three standard streams.

```

1  cout << "Enter your name and age: ";
2
3  string name;
4  int age;
5
6  cin >> name >> age;
7
8  if (age < 0) {
9      cerr << "\nInvalid age entered";
10 }
11 else {
12     cout << "\n" << name << " is " << age;
13 }

```

Listing 2 Standard output and the << operator

Programmers do not have to explicitly open and close the three standard streams. These streams are automatically available for use (through their respective objects) when a program begins execution. Programmer's must explicitly open and close all other input and output streams.

C++ programmers can define how the classes they create interact with streams using the << and >> operators. This is called operator overloading. Remember, stream classes define the insertion (<<) and extraction (>>) to operate in a special way, for many different data types. Whether used with integers, floating point numbers, or strings, these operators output and format data accordingly. We can also define the behavior of these operators for classes that we create. This allows input and output code for user-defined classes to resemble input and output for built-in types.

```

1  class Person {
2  private:
3      string first_name;
4      string last_name;
5      string job;
6
7  public:
8      Person (string f, string l, string j) :
9          first_name(f), last_name(l), job(j) {}
10
11     friend ostream& operator<<(ostream& os, Person const& r);
12
13 };
14
15 ostream& operator<<(ostream& os, Person const& r) {
16     os << r.first_name << " " << r.last_name;
17     os << " works as a " << r.job;
18     return os;
19 }
20
21

```

Listing 3 A class that overloads <<

Listing 3 defines class `Person`. Since we would like to output objects of class `Person` the same way that we output integers or strings, we overload the `<<` operator. Declaring this function as a "friend" function in line 11 allows the function to access private data members of class `Person`. This is necessary since the overloaded operator function is not a member of class `Person`.

```
1 Person p("Stan", "Dardeviation", "Math Teacher");
2 cout << p << endl;
```

Listing 4 [Using class Person](#)

The output of line 2 in Listing 4 follows.

```
Stan Dardeviation works as a Math Teacher
```

Example 1 Output of Listing 4

File Input and Output

File based input and output is similar to the mechanisms for keyboard and screen I/O. The main difference is that programmers must explicitly open and close files. In pseudocode, a generic program that reads input from a file might look like this.

```
1 open input file
2
3 while( there is input left ) {
4
5   read next input item
6   process it
7 }
8
9 close input file
```

Example 2 Pseudocode to read input from a file

Listing 5 contains a typical way to open and read a file of integers.

```
1 #include <fstream>
2 #include <iostream>
3 #include <cstdlib>
4
5 using namespace std;
6
7 int main(int argc, char* argv[]) {
8
9   ifstream inf;
10  inf.open("somefile.txt");
11
12  if (! inf) { // check to see if file opened
13      cerr << "Could not open file!" << endl;
14      return EXIT_FAILURE;
15  }
16
17  int x;
18
19  // while input remains, read an integer.
20  while (inf >> x) {
```

```

21     cout << x << endl;
22 }
23
24 inf.close(); // Close the input file
25
26 return EXIT_SUCCESS;
27 }

```

Listing 5 [File input](#)

The object used in file input is of type `ifstream`. Since this class is not part of the C++ language by default, we must include its library in our program. This is done in line 1 in the above listing. Line 9 declares an object of type `ifstream`, and line 10 calls the member function `open` to open a file. It is good programming practice to check whether an attempt to open a file actually succeeded. Also, note the use of the extraction operator in the conditional of the while-loop in line 20. As long as the extraction attempt succeeds, `true` is returned (and the value read from the file is assigned to `x`). A failure to read another integer is signaled by a value of `false`. This terminates the while-loop.

File output resembles file input. We still need to include a reference to the `fstream` library, but we use an `ofstream` object instead of an `ifstream` object. Here is an example.

```

1  ofstream onf;
2  onf.open("output.txt");
3
4  if (! onf) { // Check to see if file opened
5      cerr << "Could not open file!" << endl;
6      return EXIT_FAILURE;
7  }
8
9  for (int i = 1; i <= 10; i++) {
10     onf << "This is line " << i << endl;
11 }
12
13 onf.close(); // Close the output file

```

Listing 6 [File output](#)

1.3.4 The Preprocessor

- [Text Substitution](#)
- [File Inclusion](#)
- [Macro Substitution](#)
- [Conditional Compilation](#)
- [An Example: Assumption Verification](#)

Text Substitution

The preprocessor is a tool that C++ programmers use to manipulate the contents of source code files prior to compilation. In the most general sense, the preprocessor performs text substitution and text modification. Higher-level features emerge when we consider the overall effect of these rather basic manipulations. File inclusion, macro substitution, and conditional compilation are three higher-level features the preprocessor provides to a programmer.

Source code files, as authored by programmers, typically need to be modified in various ways before compilation can take place. Because programmers rely on the preprocessor to perform these modifications, knowledge of the basic use of the preprocessor is essential. Since C++ programs consist entirely of text, a programmer must use the preprocessor to include the declarations of external classes or functions. This is known as file inclusion. The use of other preprocessor features, such as macro substitution, is not necessarily required. Macros exist as a convenience to the programmer. They also provide backward compatibility with C programs.

A programmer interacts with the preprocessor through commands called preprocessor directives. Beginning with the number sign (`#`), preprocessor directives are single-line commands a programmer places into a source code file. Since preprocessor directives are not C++ code, they do not follow the language's scoping rules and therefore can appear on any line in a source code file. The appearance of a preprocessor directive in a source code file instructs the preprocessor to perform some action. The action the preprocessor takes depends on the directive. For some directives, the preprocessor makes exactly one modification in the source code file. An example of this is file inclusion where the preprocessor includes the contents of another file into the file being processed. Preprocessor directives used to define other tasks, such as a macro substitution, can cause the preprocessor to make several modifications in a source code file.

The Java language does not have a tool similar to the C++ preprocessor. Instead, Java provides language mechanisms that accomplish the same tasks that the C++ processor performs. One example is the `import` statement. Using `import` statements, a Java programmer specifies the external classes and packages a program requires.

File Inclusion

File inclusion is a feature of the C++ preprocessor that allows a source code file to use shared code. We consider shared code to be classes or functions declared in other files. In C++, we can only access a shared class or function by including its declaration into our program. This must be done since C++, unlike Java, shares code textually. Imagine having to include manually the declaration for every function and class used in a program. That would be quite a cumbersome task. Luckily, the preprocessor automates this through file inclusion.

A programmer issues the `#include` preprocessor directive to instruct the preprocessor to perform file inclusion. This directive takes a file or library name as a parameter. When processing an `#include` directive, the preprocessor replaces the line containing the directive with the contents of the file that the directive specifies. Listing 1 demonstrates file inclusion.

```
1  #include <string>
2  #include <cstdlib>
3  #include <iostream>
4  #include <fstream>
5
6  #include "my_functions.h"
7  #include "my_class.h"
8
9  #include "..\another_file1.h"
10 #include "directory\sub\another_file2.h"
11
12 using namespace std;
13
14 int main(int argc, char* argv[]) {
15
16     // Rest of program...
```

```
17  
18
```

Listing 1 The `#include` directive

The `#include` directive accepts two different forms of its parameter. The above example demonstrates use of the first form in lines 1 through 4. In this form, angle brackets surround the parameter in the `#include` directive. This signifies that the preprocessor should search for the specified file in an implementation dependent set of places. Typically, this set of places includes system and library directories. Double quotes surround the parameter in the second form of the `#include` directive. In this form, the `#include` directive instructs the preprocessor to look for the specified file in the same directory where the file being preprocessed exists. Lines 6 through 10 of Listing 1 illustrate this form of the directive. C++ programmers typically use the first form to include libraries and the second form to include files they have created.

Macro Substitution

The C++ preprocessor can perform a programmer defined text substitution throughout an entire source code file. This is known as macro substitution. Programmers define a macro using the `#define` preprocessor directive, which can take the following form.

```
1 | #define identifier replacement-text
```

Example 1 General form of a `#define` directive

Using the `#define` directive, a programmer declares an identifier and specifies its replacement text. Macro substitution in a source code file involves the preprocessor replacing every occurrence of the identifier with the replacement-text. Listing 2 illustrates macro definition and usage.

```
1 | #include <iostream>  
2 | #include <cstdlib>  
3 |  
4 | #define MAXIMUM 20  
5 |  
6 | using namespace std;  
7 |  
8 | int main(int argc, char* argv[]) {  
9 |  
10 |     for (int i = 0; i < MAXIMUM; i++) {  
11 |         cout << i << endl;  
12 |     }  
13 |  
14 |     return EXIT_SUCCESS;  
15 | }
```

Listing 2 [Macro substitution](#)

We can use macro substitution to implement a constant variable. In the above listing, `#define` creates an identifier named `MAXIMUM`, and associates with it the replacement text `20`. Anywhere in the program source code that the preprocessor finds `MAXIMUM`, it replaces with `20`. Macro substitution, in this case, allows the identifier `MAXIMUM` to function as a constant variable.

C++ programmers should use the keyword `const` instead of macro substitution to create constant variables. Because the keyword `const` is part of the C++ language (and not a preprocessor feature), constants created with it support type checking better than constants created using macro substitution. Constants created with macro substitution exist in C++ to provide backward compatibility with C programs.

The C++ preprocessor also supports parameterized macros. The use of a parameterized macro looks much like a normal C++ function. The preprocessor replaces the apparent function call with the macro replacement text. A parameterized macro definition takes the following form.

```
1 | #define identifier(identifier, identifier, ...) replacement-text
```

Example 2 General form of a parameterized macro

The following listing demonstrates the definition and use of a parameterized macro.

```
1 | #include <iostream>
2 | #include <cstdlib>
3 |
4 | #define max(x,y)    ( ( (x)>(y) ) ? (x):(y) )
5 |
6 | using namespace std;
7 |
8 | int main(int argc, char* argv[]) {
9 |
10 |     int i = 4;
11 |     int j = 3;
12 |
13 |     cout << max(i, j) << endl;
14 |
15 |     return EXIT_SUCCESS;
16 | }
```

Listing 3 [Parameterized macro](#)

In Listing 3, the preprocessor replaces the identifier `max` with the text "`(((x)>(y)) ? (x):(y))`". During the replacement, the preprocessor substitutes into the replacement text the text given as parameters. In line 13, the parameters given are "i" and "j". The preprocessor substitutes this text for the parameters `x` and `y` in the replacement text.

Conditional Compilation

Beyond macro substitution, a more important reason to use `#define` is to support conditional compilation. Using `#define`, and some other preprocessor directives, we can instruct the compiler to compile only certain sections of our source code. This is useful in many circumstances, one of which is for inserting debugging code that can be easily enabled and disabled. Below we see an example that uses the `#define`, `#if`, and `#endif` directives.

```
1 | #include <iostream>
2 | #include <cstdlib>
3 |
4 | #define DEBUG
5 |
6 | using namespace std;
7 |
8 | int main(int argc, char* argv[]) {
```

```

9
10 #if defined(DEBUG)
11     cerr << "Debugging enabled" << endl;
12 #endif
13
14 int arr[10];
15 for (int i = 0; i < 10; i++) {
16     arr[i] = i;
17
18 #if defined(DEBUG)
19     cerr << "i = " << i << endl;
20     cerr << "arr[i] = " << arr[i] << endl;
21 #endif
22
23 }
24 return EXIT_SUCCESS;
25 }

```

Listing 4 [Conditional compilation](#)

The `#if` preprocessor directive works similar to a regular `if`-statement, except that it has to be paired with an `#endif` directive. These two directives partition a section of source code that can be conditionally compiled. The preprocessor evaluates the value that follows the `#if`. If this value evaluates to true (non-zero), the preprocessor includes the source code block. If it evaluates to false, the preprocessor omits the source code block. In the above listing, `defined(DEBUG)` follows the `#if` directives. The preprocessor evaluates this to true only if we have defined an identifier named `DEBUG`. Since we have defined `DEBUG` in line 4, the source code blocks partitioned by the `#if` and `#endif` pairs will be compiled. The power of this technique is apparent when we realize all we have to do to disable the debugging code found throughout the program is remove the definition of `DEBUG` from the program. This causes the preprocessor to omit the debugging code.

Conditional compilation is also often used to prevent multiple definitions of classes and functions that are contained in header files. Including a header file more than once in a program can cause class and function redefinition problems. We can prevent this with a technique that uses conditional compilation. Below we see in line 1 the `#if` directive used to check if the program has defined the identifier `_PERSON_H_`. If it has not been defined, then the rest of the source code in the example is processed. If it has been defined, the source code is skipped by the preprocessor. The key to this technique is in line 2, where the program defines `_PERSON_H_`. If we had a program that had several source code files that all included the following header file, the conditional compilation would ensure that the content of the file is included only once. The first time the file was included would result in the definition of `_PERSON_H_`, which would then prevent the inclusion of the contents of the file a second time.

```

1 #if !defined(_PERSON_H_)
2 #define _PERSON_H_
3
4 class Person {
5
6     // Class declaration...
7 };
8
9 #endif

```

Listing 5 Preventing multiple declarations

Shortcut conditional compilation constructs exist that we can use in place of the `defined` operator. The directive `#ifdef identifier` is equivalent to `#if defined(identifier)`. Likewise, the directive `#ifndef identifier` is equivalent to `#if !defined(identifier)`.

An Example: Assumption Verification

Verifying assumptions using assertions is an example of a common use of the preprocessor and its features. An assertion is a statement placed in source code to verify an assumption. Usually, programmers place assertions at the beginning of a function definition to verify assumptions they made when designing the function. If at run-time the assumption proves to be incorrect, the `assert` statement displays a notification message and stops the execution of the program. Used in this manner, assertions are an excellent tool for error detection.

All kinds of assumptions are made in programs about the data contained in variables, especially those found in parameters being passed to a function. When we design and code a function, we expect the parameters to contain valid data. If the parameters do not contain valid data, this could signify that an error exists in some other area of the program. Coding around the invalid data only serves to hide the error, whereas using an assertion can detect and point out the existence of that error.

Consider the function `calculate_average` that calculates the average of a series of values. We assume the caller of the function passes two non-zero integer parameters to the function. In the context of a larger program, if the second parameter were passed as zero, a run-time error would occur as a result of the divide-by-zero. How can we handle this invalid data? One way, seen in Listing 6, involves coding defensively to detect the invalid data case.

```
1 double calculate_average(int total, int count) {
2
3     // avoid divide by zero error
4     if (count != 0) {
5         return total / count;
6     }
7     else {
8         return 0;
9     }
10 }
```

Listing 6 [Defensive coding](#)

The above version of `calculate_average` works in that it prevents the divide-by-zero error. It does not take into consideration that a zero `count` could mean that an error occurred in another part of the program. Perhaps a bug exists in the code that reads the values from the user. Or, maybe some other code erroneously overwrote the value of `count`. We really do not know, but using this version of `calculate_average` will not help us detect and locate this error.

The following version of `calculate_average` takes a different approach. Here, the assumption of valid data is verified using an assertion. If the caller of the function passes invalid data (that is, `count` equals zero) to the function, the assertion displays an error message and stops program execution. The programmer can then find the error that caused the passing of invalid data to function `calculate_average`.

```

1 double calculate_average(int total, int count) {
2
3     // assume we are given valid data
4     assert (count != 0);
5
6     return total / count;
7 }

```

Listing 7 [Verifying an assumption using an assertion](#)

The `assert` statement actually is a macro. Contained in library `<cassert>`, this macro definition is a little complex, but worth examining since it incorporates a few different uses of the preprocessor. The following example lists the definition of the `assert` macro from a GNU C++ compiler.

```

1  /*
2   assert.h
3   */
4
5  #ifdef __cplusplus
6  extern "C" {
7  #endif
8
9  #include "_ansi.h"
10
11 #undef assert
12
13 #ifdef NDEBUG           /* required by ANSI standard */
14 #define assert(p)      ((void)0)
15 #else
16
17 #ifdef __STDC__
18 #define assert(e)      ((e) ? (void)0 : __assert(__FILE__, __LINE__,
19 #e))
20 #else /* PCC */
21 #define assert(e)      ((e) ? (void)0 : __assert(__FILE__, __LINE__,
22 "e"))
23 #endif
24
25 #endif /* NDEBUG */
26
27 void _EXFUN(__assert, (const char *, int, const char *));
28
29 #ifdef __cplusplus
30 }
31 #endif

```

Listing 8 The `assert` macro definition

Notice the use of conditional compilation in the definition of the `assert` macro. Including a definition of `NDEBUG` into a program would disable all the assertion checks. When releasing production versions of software, programmers typically remove assertions.

1.3.5 A Side-By-Side Example

We have seen a lot of similarities and differences between C++ and Java. Until this point in the course, C++ and Java are more similar than they are different. In [1.4 Memory Management](#), we focus on some of the major differences in the languages. Before we delve into those topics, now is probably a good time to look at a full-length Java program and an equivalent C++ version.

- Java version
 - [BankAccount.java](#)
 - [BankAccountDriver.java](#)
- C++ version
 - [bankaccount.h](#)
 - [bankaccount.cpp](#)
 - [main.cpp](#)
 - [makefile](#)

1.4 Memory Management

With this module, the course introduces some of the basic concepts and features of the C++ programming language that are related to memory management. Whenever possible, comparisons are drawn between Java and C++ to introduce and illustrate key differences and similarities across the languages.

Readings:

- Required:

Weiss, chapter 1. Remark: Remember that this book supplements the course's online material. You will be asked questions based on this material.

- Required:

Schildt, chapters 13 through 15. Remark: Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.

1.4.1 Pointers

- [Pointers and Indirection](#)
- Basic Operations
 - [Declaration and Initialization](#)
 - [Dereference](#)
 - [Pointer Arithmetic](#)

Pointers and Indirection

A pointer is a variable that stores the memory address of another variable. We have seen already in C++ that data types dictate the range and type of values a variable can store. Variables of data types that we have examined so far store values such as integer numbers, floating-point numbers, and character strings. A pointer variable is unique in that it stores the memory address of another variable. A memory address is the specific location in main memory where a variable exists during program execution.

Programmers use pointers to indirectly access and manipulate other variables. This access and manipulation is considered "indirect" since it is accomplished using a pointer instead of the actual variable being modified. Indirection allows the creation of complex data structures and powerful algorithms. For instance, without pointers and indirection it would not be possible to create a linked list data structure.

Basic Operations

Declaration and Initialization

The declaration of a pointer variable requires the use of some unfamiliar syntax. A pointer declaration must prefix its variable name with an asterisk (*). This signifies to the compiler that the variable declared is a pointer. Listing 1 demonstrates the declaration of a few pointer variables.

1.4.2 Parameter Passing Mechanisms

- [Pass by Value](#)
- [Pass by Reference](#)

Pass by Value

The Java language creates copies of variables passed to functions. Even for objects, a copy of the reference to the object is passed to the function. This parameter passing mechanism is known as pass by value since effectively, via the copy, the "value" of the parameter is passed to a function.

Pass by value is the default parameter passing mechanism in C++. Just like Java, when a parameter is passed by value to a function, a copy of the parameter is created and given to the function. This is important, since if we make a change to a parameter that is passed by value, the original variable will remain unchanged. Our change is made to a copy of the original variable. Listing 1 illustrates this point.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  void increment(int x) {
7      x++; // Increment x by 1
8  }
9
10 int main(int argc, char* argv[]) {
11
12     int y = 10;
13     increment(y);
14
15     // variable y remains unchanged.
16     cout << y << endl;
```

```

17
18     return EXIT_SUCCESS;
19 }

```

Listing 1 [Pass by value in C++](#)

Since parameter to function `increment` in Listing 1 is passed by value, a copy of variable `y` is created and given to the function. This copy is incremented to the value of `11`. Once function `increment` returns, the lifetime of the variable (`x`) that now stores the value of `11` ends. The variable `y` remains unchanged and the program outputs the value `10`.

C++ can also pass objects by value. The following listing defines a simple class and passes an object of that class by value.

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4  #include <cassert>
5
6  using namespace std;
7
8  class Person {
9  private:
10     string name;
11     int age;
12
13 public:
14     Person() : name(""), age(0) {}
15
16     void set_age(int age) {this->age = age;}
17     int get_age() { return age;}
18     void set_name(string name) {this->name = name;}
19     string get_name() { return name;}
20 };
21
22 void increment_age(Person p) {
23     p.set_age(p.get_age() + 1);
24 }
25
26 int main(int argc, char* argv[]) {
27
28     Person person;
29     person.set_name("John Doe");
30     person.set_age(30);
31
32     increment_age(person);
33
34     // age remains unchanged
35     cout << person.get_age() << endl;
36
37     return EXIT_SUCCESS;
38 }

```

Listing 2 [Passing an object by value](#)

The `main` routine in Listing 2 passes an instance of class `Person` by value to the function `increment_age`. Inspecting the output of the example yields the result we expect to see. The code in function `increment_age` does not change the state of the object `person` in function `main`, since the object was passed by value.

Pass by Reference

C++ also supports the pass by reference parameter passing mechanism. Unlike pass by value, copies are not made of variables that are passed by reference. Instead, a called function receives a reference, or alias, to the actual parameter supplied by the calling function. For this reason, pass by reference is used to build functions that can modify the variables in the calling function. Even when a function does not need to modify the variables in the calling function, pass by reference is sometimes used to avoid the overhead of pass by value.

One common use of pass by reference is to create functions that can modify the variables passed to them by a calling function.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  void increment(int& x) {
7      x++; // Increment x by 1
8  }
9
10 int main(int argc, char* argv[]) {
11
12     int y = 10;
13     increment(y);
14
15     // variable y is changed.
16     cout << y << endl;
17
18     return EXIT_SUCCESS;
19 }
```

Listing 3 [Pass by reference](#)

Listing 3 is the same program presented in Listing 1. The only difference is in line 6 where the parameter `x` is declared as a reference parameter using the syntax `int& x`. Do not confuse this use of the ampersand (&) with the address-of operator. Here the ampersand signals to the compiler that this parameter is to be passed by reference. Since it is passed by reference, the increment operation in line 7 affects the original variable `y` found in `main`. Therefore, this program outputs the value `11`.

Passing a parameter by reference is also used as a mechanism to pass large objects to functions. When objects are large, pass by value can result in time-consuming copy operations. Pass by reference is more efficient because it does not involve copying. Even when a function does not intend to modify one of its parameters, pass by reference should be used when the parameter is a large object. We should declare parameters passed by reference that a function should not modify as constants. This is a good practice since it provides protection against accidental modification. Listing 4 demonstrates how to pass an object by reference while still preserving the safety of pass by value.


```

1  #include <iostream>
2  #include <string>
3  #include <cstdlib>
4
5  using namespace std;
6
7  void display_letters(const string& data) {
8
9      for (int i = 0; i < data.length(); i++) {
10         cout << data[i] << "\n";
11     }
12 }
13
14 int main(int argc, char* argv[]) {
15
16     string s = "This is a demonstration";
17     display_letters(s);
18
19     return EXIT_SUCCESS;
20 }

```

Listing 4 [A constant reference](#)

It is sometimes useful to pass a pointer by reference. This is done when a function needs to change the pointer's stored memory address. In other words, a pointer is passed to a function by reference when the function needs to reposition the pointer.

```

1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  void find_first_greater_than(int*& ptr, int threshold) {
7
8      while (*ptr <= threshold) {
9          ptr++;
10     }
11 }
12
13 int main(int argc, char* argv[]) {
14
15     int exam_scores[] = {74, 94, 64, 77, 68,
16                          99, 58, 89, 74, 88,
17                          100, 95, 71, 81, 89,
18                          54, 76, 83, 88, 67};
19
20     int* score = &exam_scores[0];
21
22     cout << *score << endl;
23     find_first_greater_than(score, 98);
24     cout << *score << endl;
25
26     return EXIT_SUCCESS;
27 }

```

Listing 5 [Passing a pointer by reference](#)

1.4.3 Dynamic Memory Management

- [The Free Store](#)
- [Memory Allocation](#)
- [Memory Deallocation](#)
- [Copy Constructors](#)
- [Some Common Pitfalls](#)
 - Memory Leaks
 - Overwrites
 - Using Deallocated Memory
 - Deallocating Memory Twice

The Free Store

Every C++ program has what is called the "free store." The free store, which is sometimes called "the heap," is an area of a program's memory that is used dynamically. Using memory dynamically means that the amount of memory needed for some task is specified at run-time, rather than at compile time. For example, imagine a program that stores in an array a list of numbers input from the user. If the maximum size of the list of numbers is known ahead of time, there is little difficulty involved in declaring an array of suitable size. But, what if the size of the list is unknown? In this situation, we could prompt the user to enter first the size of the list of numbers. Then, using dynamic memory, we can create an array of equal size.

Variables created in the free store have dynamic extent. The extent of a variable describes how long a variable stays around in a program. Another term commonly used in place of extent is lifetime. Local variables in functions have local extent; they are created when the function is called and they are destroyed when the function returns. Global variables, which have static extent, are created and available throughout the entire lifetime of a program. A variable with dynamic extent has its lifetime specified explicitly by the programmer. The programmer issues a statement to create the variable and a statement to destroy the variable. This provides a lot of flexibility in the type of solutions that programmers can create. Think back to the list of numbers example where we prompted the user for the size of the list. There is a better solution that takes advantage of the dynamic extent of variables created from the free store. Instead of prompting the user to enter the size of the list, we assume the size of the list will not exceed one hundred elements. Using dynamic memory, we create an array of one hundred elements. If, during the input process, a user enters a one hundred and first number, we dynamically create an array of two hundred elements. We copy the first one hundred elements from the first array to the new array, and then insert the one hundred and first number. After this is complete, the first array is no longer needed and we return its memory to the free store. We repeat this process of dynamically allocated memory each time the capacity of our array is exceeded.

Memory Allocation

The process of obtaining memory from the free store is called memory allocation. The operator `new` is used in C++ to allocate memory dynamically.

```
1 // Allocate a single integer
2 int* ptr = new int;
```

Listing 1 [The new operator](#)

The `new` operator always returns a memory address. Remember, pointers store memory addresses, so we must store the return value of the `new` operator in a pointer. Using a pointer, we can indirectly access and modify the variable that we just created.

```
1 // Allocate a single integer
2 int* ptr = new int;
3 *ptr = 10;
4
5 cout << "Address: " << ptr << endl;
6 cout << "Value: " << *ptr << endl;
```

Listing 2 Using a variable from the free store

The `new` operator works for all data types. We can dynamically allocate integers, floats, strings, and other user created classes. Listing 3 shows the allocation of several different data types.

```
1 int* i_ptr = new int;
2 char* c_ptr = new char;
3 bool* b_ptr = new bool;
4 float* f_ptr = new float;
5 double* d_ptr = new double;
6 string* str_ptr = new string;
```

Listing 3 Allocating different data types

Arrays can also be dynamically allocated.

```
1 // Dynamically allocate an array of size 100
2 float* ptr1 = new float[100];
3
4 // Prompt the user for the size of the second array
5 int size = 0;
6 cin >> size;
7 float* ptr2 = new float[size];
```

Listing 4 [Allocating arrays](#)

Objects can also be dynamically allocated. The `new` operator, in addition to allocating the memory for an object, will call a constructor for the object. Listing 5 shows objects allocated using `new`.

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 class my_class {
7 private:
8     int x;
9 public:
10    my_class() : x(0) {}
11    my_class(int p) : x(p) {}
12    int value() { return x;}
13 };
14
15 int main(int argc, char* argv[]) {
```

```

16
17 // Allocate a single object
18 my_class* ptr1 = new my_class(4);
19
20 // Allocate an array of objects
21 my_class* ptr2 = new my_class[10];
22
23 cout << ptr1->value() << endl;
24 cout << ptr2->value() << endl;
25
26 return EXIT_SUCCESS;
27 }

```

Listing 5 Allocating objects

Listing 5 defines a simple class and illustrates allocation of both a single instance and an array of objects of the class. In line 18, the single instance is allocated. After allocating memory for the object, the `new` operator invokes the single parameter constructor. In line 21, the `new` operator allocates an array of ten elements of type `my_class`. In this case, the default constructor is called for each of the ten objects in the array.

Memory Deallocation

The `delete` operator deallocates memory that is allocated using the `new` operator.

To release, or deallocate, memory, the `delete` operator needs to know what location in memory we want to deallocate. To this end, we supply it with a pointer, which is really just the address that we obtained from the `new` operator.

```

1 // Dynamically allocate a variable.
2 double* ptr1 = new double;
3
4 // ... use the variable ...
5
6 // The variable is no longer needed,
7 // so we return its memory to the Free Store.
8 delete ptr1;

```

Listing 6 The `delete` operator

A special syntax exists for the `delete` operator for use in deallocating arrays. The keyword `delete` is followed by the double bracket (`[]`) operator. This signals to the run-time environment that what `delete` needs to deallocate is actually an array, and not just a variable.

```

1 // Allocate two arrays.
2 int* ptr1 = new int[100];
3
4 int n = 0;
5 cin >> n;
6 int* ptr2 = new int[n];
7
8 // when they are no longer needed,
9 // we deallocate them.
10 delete [] ptr1;
11 delete [] ptr2;

```

It might be surprising that the size of the array does not have to be specified when using `delete []`. This is not necessary since the run-time environment automatically maintains the size of allocated arrays.

We have already seen that the `new` operator invokes a constructor when used to allocate objects. Similarly, when deallocating an object the `delete` operator calls the object's destructor.

Copy Constructors

A copy constructor defines the actions that need to be taken to create a copy of an object. Unlike regular constructors, a class can contain only one copy constructor. If a C++ class does not define a copy constructor, the language provides to the class a default copy constructor. This default copy constructor makes a byte-by-byte copy of the object.

Copy constructors are invoked whenever a copy of an object has to be made. There are three situations when copies of objects are made.

1. During declaration that involves initialization
2. When objects are passed by value
3. When objects are returned by value

Using the default copy constructor provided by the runtime system can be dangerous in a program that uses dynamic memory. Situations can arise where two or more objects incorrectly maintain pointers to the same data. Consider the following listing.

```

1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  class Array {
7
8  private:
9      int *ptr;
10
11 public:
12     Array(void) : ptr(new int[10]) {}
13     ~Array(void) {delete ptr;}
14     void display_ptr(void) {cout << ptr << endl;}
15 };
16
17 int main(int argc, char* argv[]) {
18
19     // Create two Array objects. The
20     // second should be a copy of the
21     // data of the first.
22     Array arr1;
23     Array arr2 = arr1;
24
25     arr1.display_ptr();
26     arr2.display_ptr();
27
28     return EXIT_SUCCESS;
29 }
```

Listing 8 [Dynamic memory and the need for a copy constructor](#)

The intent of function `main` in the above listing is to create two objects of type `Array`. Each of these objects is to have ten elements. The ten elements in the second object (`arr2`) should contain copies of the values of the ten elements from the first object (`arr1`). This copy takes place when `arr1` is assigned to `arr2`. In reality, only a copy of the pointer variable `ptr` is made during this assignment. Since class `Array` does not define a copy constructor, a byte-by-byte copy of the object is made. This byte-by-byte copy of the object results in each `Array` object having the same value for `ptr`. This is not a true copy of the object since the elements in the array, since they exist in the heap, are not copied at all. This type of copy is known as a shallow copy. The opposite of a shallow copy is a deep copy.

```

1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  class Array {
7
8  private:
9      int *ptr;
10
11 public:
12     Array(void) : ptr(new int[10]) {}
13
14     Array(const Array& src) {
15
16         cout << "Copy Constructor Invoked!\n";
17
18         ptr = new int[10];
19         for (int i = 0; i < 10; i++) {
20             ptr[i] = src.ptr[i];
21         }
22     }
23
24     ~Array(void) {delete ptr;}
25     void display_ptr(void) {cout << ptr << endl;}
26 };
27
28 int main(int argc, char* argv[]) {
29
30     // Create two Array objects. The
31     // second should be a copy of the
32     // data of the first.
33     Array arr1;
34     Array arr2 = arr1;
35
36     arr1.display_ptr();
37     arr2.display_ptr();
38
39     return EXIT_SUCCESS;
40 }

```

Listing 9 [Class Array copy constructor](#)

A copy constructor can be used to ensure the creation of a deep copy of an object. The above listing contains a copy constructor for class `Array`. This copy constructor allocates a new array in memory and copies into it the values from the source array.

Some Common Pitfalls

The basic rule for dealing with dynamic allocation and deallocation is straightforward: for each call to `new` (which consumes memory), there must be a corresponding call to `delete` (which releases the memory). Moreover, if at all possible, the matching calls should be close to each other, say, in the same function. Following this principle, allocations are made in the first few lines, and all the corresponding deallocations are performed in the last few lines of the function body. In practice, however, it is often not possible to match up closely allocations and deallocations in code. But, it is always important to think carefully ahead of time about where and when memory should be allocated and deallocated. Do not allocate memory at random throughout your code or deep inside nested function calls, because doing so makes it hard to keep track of those allocations. In general, errors relating to dynamic memory allocation are hard to deal with, and it is best to avoid them from the start.

1.5 Mechanisms for Code Reuse and Abstraction

This module of the course presents the mechanisms available in the C++ language that facilitate code reuse and abstraction.

Readings:

- **Required:**

Weiss, chapters 3 and 4.

- **Remark:**

Remember that this book supplements the course's online material. You will be asked questions based on this material.

- **Required:**

Schildt, chapters 16 through 18.

- **Remark:**

Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.

1.5.1 Inheritance

- [A Mechanism for Abstraction and Code Reuse](#)
- [C++ Syntax](#)

A Mechanism for Abstraction and Code Reuse

Inheritance is a mechanism in C++ (as well as in Java) that facilitates abstraction and code reuse. Inheritance establishes the "is-a" relationships between the classes contained in a program. Using inheritance, new classes can be built based on old classes, allowing child classes to share data members and functions of the parent class. It is through these relationships that the advantages of data abstraction (generalization and specialization) emerge.

C++ Syntax

The C++ language syntax for specifying inheritance resembles that of Java. In Java, the first line of a class declaration can specify a class to inherit. This is the same in C++, with one exception. Instead of using the keyword `extends` to denote the parent/child relationship, C++ uses a colon (`:`).

```
1 class Employee { /* declaration of parent */ };
2 class Manager : Employee { /* declaration of child */ };
```

Listing 1 C++ inheritance syntax

The above listing declares class `Employee` and class `Manager`. Programmers and literature often use the terms "parent" and "child" to refer to the classes involved in the "is-a" relationship that inheritance models. Other terms used to describe the classes are "base" and "derived," respectively. As a standard, we use the terms "parent" and "child" throughout this course.

C++ and Java differ slightly in the data members that child classes inherit from their parents. In C++, a child class inherits all non-private data members including constructors. A child class in Java, however, inherits from its parent class all non-private data members except constructors. In both languages, child classes inherit all the non-private data members. In C++, this includes public and protected data members. In Java, this includes data members declared using public, protected, and default access. Unlike Java, C++ does not have a separate, default access. In C++, data members declared without an access modifier default to use private access.

```
1 class BankAccount {
2
3     protected:
4         double sum;
5         string name;
6
7     public:
8
9         BankAccount(string nm) : name(nm), sum(0) {}
10
11         double balance() { return sum; }
12         void deposit(double amount) { sum += amount; }
13         void withdraw(double amount) { sum -= amount; }
14         string get_name() { return name; }
15 };
16
17 class SavingsAccount: public BankAccount {
18     protected:
19         double rate;
20
21     public:
22         SavingsAccount(string nm)
23             : BankAccount(nm), // Call base class constructor
```



```

24     rate(0.055) {}
25
26     void add_interest() {sum *= (1 + rate);}
27     double get_rate() { return rate;}
28 };

```

Listing 2 Invoking parent class methods

Listing 2 contains an example of C++ inheritance that demonstrates how to invoke methods of a parent class. In this listing, we have defined a class `SavingsAccount` that inherits from class `BankAccount`. Our `SavingsAccount` class provides a way to add interest to the money present in an account. Line 23 is interesting in that it invokes the parent class constructor. In Java, we could accomplish this same task using the keyword `super()`. In C++, we instead use the name of the parent class followed by the constructor's parameters in parentheses.

C++ contains three types, or levels, of inheritance: public, private, and protected. Public inheritance is the most common type of inheritance used in C++. The examples we have seen so far all use public inheritance to model the "is-a" relationship of two classes. Private and protected inheritance model a different type of relationship, namely the "uses-a" relationship. For example, to model that a car uses an engine, we could privately inherit from a class `Engine` when defining a class `Car`. The more appropriate way to model this relationship, however, would be to have our class `Car` contain an instance of class `Engine`. Modeling the "uses-a" relationship in this manner is known as composition. All uses of inheritance in this course focus on modeling "is-a" relationships. Therefore, we do not use private or protected inheritance.

1.5.2 Polymorphism

- [A Mechanism for Abstraction](#)
- [Polymorphism in Java](#)
- [Polymorphism in C++](#)

A Mechanism for Abstraction

A fundamental feature of object-oriented programming languages, polymorphism is the ability of an object to take on several different forms. Put another way, polymorphism allows a programmer to refer to an object of one class as an object of another class. This has two primary uses. First, we can create collections of heterogeneous objects. We can operate on the individual objects in these collections as if they were all of the same type, without the objects losing their real identities. Second, we can code algorithms that make only minimal assumptions about the objects they manipulate. This can allow an algorithm to continue to function correctly even when a programmer introduces new child classes into the system. Both of these uses help create more maintainable solutions.

Polymorphism in Java

Polymorphism in Java is achieved by referring to an overridden method of an object via a reference of an ancestor type. This allows simple code to replace what could be an ugly looking switch-statement. The following listing demonstrates polymorphism in Java.

```

1 Account[] accounts =
2 {
3     new CheckingAccount("Fred", 500, 100),
4     new SavingsAccount("Wilma", 1000, 0.03),
5     new CheckingAccount("Barney", 200, 100),
6     new BondAccount("Betty", 2000, 0.07)
7 };
8
9 for (int i = 0; i < 4; ++i) {
10     system.out.println(accounts[i].showBalance());
11 }

```

Listing 1 Polymorphism in Java

The above listing assumes a definition of class `Account` that has immediate child classes of type `CheckingAccount` and `SavingsAccount`. Class `BondAccount` is a child class of class `SavingsAccount`. Polymorphism exists in the for-loop in the above listing where the overridden method `showBalance` is called for each object. Even though the array `accounts` is of type `Account`, the method `showBalance` that is invoked is the `showBalance` method for the individual child classes. Without polymorphism, the single call to `showBalance` in the for-loop would have to be replaced by a switch-statement that casts the `Account` object to the correct child class.

Polymorphism in C++

In C++, polymorphism yields the same benefit as it does in Java: programmers can refer to objects in a way that facilitates elegant solutions. The approach taken to achieve this goal in C++ is the same as in Java. Programmers invoke overridden methods of a child class via an ancestor class. In Java, this is accomplished by defining a class hierarchy and using parent class references with instances of child classes. In C++, we use virtual functions in conjunction with pointers to access objects polymorphically.

To illustrate what virtual functions are and why we need to use pointers in conjunction with them, let's step through an extended example. Suppose we wish to implement a graphics system that can display various geometric shapes on the screen. It is natural to start with a small hierarchy of graphics objects.

```

1 class Shape { /* ... */ };
2 class Circle: public Shape { /* ... */ };
3 class Rectangle: public Shape { /* ... */ };

```

Listing 2 A class hierarchy

We can assume that in our application we need to be able to keep track of a collection of such shape objects. Suppose we want to maintain an array of shapes of either kind. We could try an array of type `Shape`. This is reasonable, since it is legitimate to make an assignment from a child class to a variable of the base class.

```

1 Circle C(3); // radius 3
2 Shape S[10];
3 S[0] = C; // syntactically correct, but ...

```

Listing 3 The slicing problem

Unfortunately, the additional parts of the child class are simply stripped off during the assignment. This is known as the slicing problem. We lose all the additional data members of the child class. Slicing, however, does not occur when we deal with pointers. Hence, we can salvage our project by using an array of pointers to shapes.

```
1 Shape *layout[10];
2 layout[0] = new Circle(3);           // radius 3
3 layout[1] = new Rectangle(2, 4);     // width 2, height 4
```

Listing 4 Using pointers

Shapes have several methods associated with them. For example, we want be able to calculate the area a shape takes up on the screen. The `Shape` class, therefore, would have to contain an `area()` method if we wish to invoke `area()` from an array of `Shape*`. Since `Shape` is a "generic" class, we could define its `area()` method as follows.

```
1 class Shape {
2     private:
3         /* ... */
4
5     public:
6         float area(void) { return 0;}
7         /* ... */
8     };
```

Listing 5 More of the `Shape` class

Next we need class `Circle` and `Rectangle` to override the inherited `area()` method of class `Shape` to provide specific details on how to calculate their respective areas. Once this is done, we can try the following.

```
1 cout << layout[0]->area() << endl;    // prints 0
2 cout << layout[1]->area() << endl;    // prints 0
```

Listing 6 Calling the wrong `area()` function

This implementation doesn't work as intended. The problem is that the static type of `layout[0]` is `Shape*`, and, therefore, the `area()` method belonging to class `Shape` is invoked, rather than the specific `area()` method for each of those array objects. What we need is a mechanism that checks the dynamic type of `layout[0]`, for example, determine that it is `Circle*`, and then call the `area()` method from class `Circle`. This is accomplished in C++ using virtual functions. Below is the proper redefinition of the `Shape` class, particularly its `area()` method. The `area()` method in the child classes need not be redefined (though it is good style to attach the keyword `virtual` to those methods as well).

```
1 class Shape {
2     private:
3         /* ... */
4
5     public:
6         virtual float area(void) { return 0;}
7         /* ... */
8     };
```

Listing 7 A virtual function

Assuming that `Circle` and `Rectangle` provide their own `area()` method overriding the one in `Shape`, we obtain the correct output.

```
1 cout << layout[0]->area() << endl; // prints 28.2743
2 cout << layout[1]->area() << endl; // prints 8
```

Listing 8 Calling the correct `area()` function

We could also have modified `Shape` by making `area()` not just virtual but also totally undefined. A function of this sort is called a pure virtual function.

```
1 class Shape {
2 private:
3     /* ... */
4
5 public:
6     virtual float area(void) = 0; // totally undefined
7     /* ... */
8 };
```

Listing 9 A pure virtual function

A class that contains a pure virtual function is known as an abstract class. Implementations in C++ only use abstract classes in conjunction with inheritance. Put another way, programmers never create instances of abstract classes. They exist merely to specify the common interface of child classes and to access these child classes polymorphically.

By way of summary, in order to obtain polymorphic behavior, we need to:

- deal with pointers rather than direct objects because of slicing
- declare member functions to be virtual

1.5.3 Templates

- [Template Functions](#)
- [Template Classes](#)

Template Functions

Template functions allow a programmer to apply the logic of a function to more than one data type. An ordinary C++ function declaration dictates the data types of its parameters. There are situations, however, when the logical structure of a function makes sense for many different types. For example, to compute the maximum of two values, we perform the same type of calculation regardless of whether the values are integers, floats, or strings. Template functions allow the programmer to create functions independent of the data types of their parameters.

```
1 int max(int x, int y) { return x < y ? y : x;}
2 float max(float x, float y) { return x < y ? y : x;}
3 string max(string& x, string& y) { return x < y ? y : x;}
```

Listing 1 Function overloading

Listing 1 contains a valid but undesirable approach to creating a family of functions that accomplish the same logical task, yet work on different data types. In this listing, the implementations for each version of function `max` are identical. The only differences between the three function definitions are the data types of the parameters and the data types of the return values. This approach is undesirable since it is tedious and leads to serious maintenance problems. If we ever need to change the function, we have to edit every single copy by hand. If we would like the function to support additional data types other than `int`, `float`, or `string`, we have to create those versions manually.

Using a template function, a C++ programmer creates the logic of a function around a generic data type. The compiler then creates versions of the function for specific data types. Listing 2 shows a template version of function `max`.

```
1  template <class T>
2  T my_max(T x, T y) {
3      return x < y ? y : x;
4  }
```

Listing 2 A template function

Template functions require a new and unique syntax. In line 1 of the above listing, the keyword `template` specifies that the function is a template function. It is followed in the angle brackets by the keyword `class` and a generic data type name `T`. The generic data type name `T` is then used throughout the implementation of the function `max`. The compiler replaces this generic data type name with the specific data types used by the program.

Instead of the keyword `class`, C++ also accepts the keyword `typename`. To avoid confusion, this course consistently uses the keyword `class` when defining templates.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4
5  using namespace std;
6
7  template <class T>
8  const T& my_max(const T& x, const T& y) {
9      return x < y ? y : x;
10 }
11
12 int main(int argc, char* argv[]) {
13
14     int i1 = 1;
15     int i2 = 2;
16     cout << "Maximum is: " << my_max(i1, i2) << endl;
17
18     float f1 = 34.4;
19     float f2 = 24.2;
20     cout << "Maximum is: " << my_max(f1, f2) << endl;
21
22     string s1 = "test1";
23     string s2 = "test2";
24     cout << "Maximum is: " << my_max(s1, s2) << endl;
25
26     return EXIT_SUCCESS;
27 }
```

Listing 3 [Creating and using a template function](#)

We refer to a template function as a "template" because the compiler actually creates multiple versions of the function based on the generic or "template" definition. To know which versions it needs to create, the compiler looks through the source code of a program, examining the data types of the actual parameters passed to the template function. When compiling the above example program, the compiler creates three versions of function `my_max`, one for type `int`, one for type `float`, and one for type `string`. If we included a call to function `my_max` that used variables of type `double`, the compiler would create a version of function `my_max` that supported type `double`.

When defining and using template functions, a programmer must consider that the actual data types supplied to a template function must support the operators and data members used with the generic data type. Even though the `my_max` function from Listing 3 is a template function, it will not support all possible data types. Function `my_max`, because of its implementation in line 9, can only be used with data types that define the `<` operator. This is an important principle programmers must follow when designing and using template functions. Listing 4 illustrates a violation of this principle.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4
5  using namespace std;
6
7  template <class GenericType>
8  void display_length(const GenericType& x) {
9      cout << "Length is: " << x.length();
10 }
11
12 int main(int argc, char* argv[]) {
13
14     string s = "string";
15     int i = 0;
16
17     display_length(s);
18
19     // int does not support function length()
20     // which results in compile error!
21     display_length(i);
22
23     return EXIT_SUCCESS;
24 }
```

Listing 4 [Incorrectly using a template function](#)

Template functions can utilize more than one generic data type. Listing 5 shows a template function, `some_function`, that has two generic types. This function takes two parameters. The first parameter is of the generic type `X`, and the second is of the generic type `Y`.

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <class X, class Y>
```

```

7 void output(X x, Y y) {
8     cout << x << " " << y << endl;
9 }
10
11 int main(int argc, char* argv[]) {
12
13     output(2, "test");
14     output(4.5, true);
15     output("test", 3);
16
17     return EXIT_SUCCESS;
18 }

```

Listing 5 Multiple generic types

Template Classes

In addition to template functions, programmers can use C++ to create template classes. A template class in C++ is a class whose definition is independent of a specific data type. For example, we could define a template class `Array` that works for integers, floats, characters, strings, and user defined classes.

The syntax involved in a template class definition is similar to that of a template function.

```

1 template <class ElementType>
2 class Array {
3
4 private:
5     ElementType arr[SIZE]; // fixed size array
6
7 public:
8     // Constructor
9     Array(ElementType e) {
10         for (int i = 0; i < SIZE; i++) {
11             arr[i] = e;
12         }
13     }
14
15     // Element access
16     ElementType& operator[](int i) {
17         assert (0 <= i && i < SIZE);
18         return arr[i];
19     }
20 };

```

Listing 6 [A template class](#)

Looking at both Listing 5 and Listing 6, we can see that the syntax of a template class definition resembles the syntax of a template function definition. In both cases, a line containing the keyword `template` and a list of the generic data type names precedes the definition. Listing 7 demonstrates instantiation of template classes. Notice the programmer specifies the specific type of the generic class inside angle brackets.

1.5.4 Exception Handling

- [C++ and Java Exception Handling](#)
- [C++ Exception Handling Fundamentals](#)
- [The C++ Standard Exception Hierarchy](#)
- [Using Intermediate Handlers](#)
- [Summary of Best Practices](#)

C++ and Java Exception Handling

In many respects, exception handling in C++ closely resembles exception handling in Java. First of all, C++ uses the same `try-catch` block mechanism used in Java. Also, both languages also provide an exception class hierarchy that programmers can use and extend. There are a few differences, however, in C++ and Java exception handling. In this page, we examine the fundamentals of C++ exception handling, paying careful attention to the differences from Java exception handling. Also, we examine how explicit memory management complicates handling exceptions.

C++ Exception Handling Fundamentals

C++ exception handling centers around the use of the `try` keyword and the `catch` keyword. A programmer encloses code that may trigger an exception within a `try` block. One or more `catch` blocks immediately follow the `try` block. When an exception occurs in a `try` block, the program execution point is transferred to a `catch` block corresponding to the type of exception thrown. The `catch` block performs the necessary tasks to recover from the exceptional condition. In other words, it "handles" the exception. Listing 1 demonstrates the use of `try` and `catch` blocks in C++.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <stdexcept>
4
5  using namespace std;
6
7  void calculate_fibonacci(int);
8
9  int main(int argc, char* argv[]) {
10
11     try {
12
13         if (argc != 2) {
14             cerr << "Usage: " << argv[0] << " num" << endl;
15             return EXIT_FAILURE;
16         }
17         int number_fib = atoi(argv[1]);
18         calculate_fibonacci(number_fib);
19
20         return EXIT_SUCCESS;
21     }
22
23     catch (exception& e) {
24         cerr << e.what() << endl;
25     }
26 }
```



```

27     return EXIT_FAILURE;
28 }
29
30 void calculate_fibonacci(int number_fib) {
31
32     int* array = new int[number_fib];
33     array[0] = 1;
34     array[1] = 1;
35
36     // populate the elements with Fibonacci numbers
37     for (int i = 2; i < number_fib; i++) {
38         array[i] = array[i - 1] + array[i - 2];
39     }
40
41     delete [] array;
42 }

```

Listing 1 [The try and catch blocks](#)

C++ programmers trigger exceptions using the `throw` statement. In a `throw` statement, a programmer specifies the exception to trigger following the keyword `throw`. Listing 2 randomly triggers one of two exceptions using `throw` statements.

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <stdexcept>
4  #include <ctime>
5
6  using namespace std;
7
8  int main(int argc, char* argv[]) {
9
10     try {
11
12         srand(time(0));
13         if (rand() % 2 == 0) {
14             throw out_of_range("out of range");
15         }
16         else {
17             throw length_error("length error");
18         }
19
20         return EXIT_SUCCESS;
21     }
22     catch (out_of_range e) {
23         cerr << "caught out_of_range exception" << endl;
24     }
25     catch (length_error e) {
26         cerr << "caught length_error exception" << endl;
27     }
28
29     return EXIT_FAILURE;
30 }

```

Listing 2 [Triggering exceptions](#)

