120++: a C++ Subset Corresponding to "A Project-Based Introduction to C++"

Terence Soule and Clinton Jeffery

University of Idaho

Contents

Chapter 1

Reference

1.1 Variables and Types

Computer programs use variables to store data. A *variable* can be thought of as a box with a name that contains a value – the data. Before a variable is used, it must be declared. The basic variable declaration consists of two parts: the type and the variable name. For example, the command:
`int radius;`
declares a variable of type `int`, which is short for *integer* and whose name is `radius`. It can be thought of as a box named `radius` that can contain an integer number.

The type is important for several reasons.

1. The type determines how big the "box" that holds the data needs to be (e.g., a single character takes up less space in memory than a real number).

2. The type determines how to interpret the data stored in the "box" – all data is fundamentally stored as a binary number (i.e., a series of 1's and 0's), but how the number is interpreted depends on the data's type.

3. The type determines how operations should be applied to the data stored in the "box." For example, if the division operation is applied to two real numbers it returns a real number, but if it is applied to two integers, it truncates the answer and returns an integer.

The most commonly used types in C/C++ are introduced in Table 1.1.

There are several commonly used forms a of variable declaration:

```
int x;
int v1, v2, v3;
int v = 7;
```

The first example simply declares a single variable called `x`. The second example is a multiple declaration that declares three variables, all of type `int`, named `v1`, `v2`, and `v3`. Any number of variables can be declared in one line as long as they all have the same type. The third example declares a variable named `v` and assigns it the initial value 7. It's often a good idea to assign variables an initial value, otherwise the variable's value is undefined, which may cause errors if an attempt is made to use the variable's value without setting it somewhere else in the program.[1]

[1]Many languages require all variables to be assigned an initial value, assign variables a default initial value such

Table 1.1: Common Variable Types in C++. Note that the actual number of bytes used to store a variable, and hence the allowed range of the variable, may vary from system to system.

Type	**Name**	**Size(bytes)**	**Range**	**Description**
`int`	Integer	4	-2147483648 to 2147483647	Variable holding an integer number; a positive or negative number without a decimal.
`short int` (or `short`)	Short integer	2	-32768 to 32767	Variable holding a short integer number; a positive or negative number without a decimal.
`long int` (`long`)	Long integer	4	-2147483648 to 2147483647	Often the same as an integer; may be larger, for example, on a 64-bit machine.
`float`	Floating point	4	$\pm 3.4e^{\pm 38}$ (∼7 digits of precision)	Variable holding a positive or negative real number. Floating-point numbers are stored using an exponent, which allows the decimal place to "float."
`double`	Double-precision floating point	8	$\pm 1.7e^{\pm 308}$ (∼15 digits of precision)	Often the same as a `float`, but may use 16 bytes.
`long double`	Long double-precision floating point	8	$\pm 1.7e^{\pm 308}$ (∼15 digits of precision)	Often the same as a `double`; may be larger depending on the machine it's running on.
`char`	Character	1	0 to 255	Stores a single character ('a', 'b', '4', '$', '+', etc.).
`unsigned`	Unsigned number	Varies	Varies	The modifier `unsigned` can be applied to `int`s (including long and short `int`s). It makes the number unsigned (always positive) and doubles the maximum range of the variable. For example, an unsigned `int` has a range of 0 to 4294967295.

1.1.1 Naming Variables

Variables should be given names that make it easier to read the program. For example, variable names from the first two projects include: `lucky` for the user's lucky number, `favorite` for the user's favorite integer, and `num_objects` for the number of objects remaining in the game of NIM.

The rules for valid variable names are fairly simple. Variable names can use letters, underscores (_), and digits, but cannot start with a digit and must contain at least one letter. They may not be a reserved word, such as `if`, `int`, or `main`. Case matters in variable names: `Lucky` is not the same variable as `lucky`. Legal variable names include: `X`, `a_variable`, and `variable7`, but *not* `7variable` (starts with a digit), `_123` (doesn't contain a letter), or `do` (a reserved word). There are a number of conventions for choosing variable names and software companies often enforce the use of a particular convention. Several conventions are used in this text to demonstrate the range of variable naming styles.

1.2 Conditionals: `if, if-else, switch`

Conditionals are commands that allow a program to take different actions depending on specific conditions. The most common conditionals in C++ are `if`, `if-else`, and `switch`.

The structure of an `if` statement is

```
if(condition) {
  Execute this block of code
  if the condition is true
}
Always executed
```

If the condition in the `if` statement is, or evaluates to, zero, it is treated as false; otherwise, it is treated as true. Thus, the condition can be anything that C++ can interpret as an integer. However, to make programs readable, the condition should generally be a test of some kind, such as `X < Y`, which is true if `X` is less than `Y` and false otherwise.

The structure of an `if-else` statement is:

```
if(condition) {
  Execute this block of code    if the condition is true
}
else {
  Execute this block of code otherwise
}
Always executed
```

In both conditionals, the curly brackets defining the block after the `if` and the `else` may be omitted if only a single command is used after the `if` or the `else`. For example,

```
if(condition)
      Single command to be executed
Always executed
```

However, it is good programming practice to include the curly braces, even when they are not

as zero, or check that a variable has been assigned a value before the variable can be used. C++ does none of these things, instead leaving it up to the programmer to make sure that variables are assigned reasonable values before they are used.

necessary. It makes the code easier to read and avoids potential errors if additional commands are added to the `if` or `else` clause.

The switch statement allows a program to make multiple decisions based on an integer or character value. The basic structure of a switch statements is:

```
switch(variable of type int or char){
    case 1:
       code block 1
       break;
    case 2:
       code block 2
       break;
    ...
    default:
       code block N
       break;
}
```

The program examines the value stored in the `variable` and jumps to the matching case. For example, if the `variable` has the value 2, then the program jumps to case 2. If none of the cases match, then the program jumps to the default case. (A default case is not required; if one is missing and the `variable` value doesn't match any other case, the `switch` is simply skipped.) Once the program reaches a case, it continues to execute instructions sequentially, including later cases. Thus, the `break` statement is included, which causes the program to jump to the end of the switch statement instead of continuing to the other cases.

Generally the `variable` is an integer, as shown in the example. Alternatively the `variable` could by a `char` in which case the cases should be labeled with characters in single quotes, such as `case: 'a'`.[2]

1.3 Loops: `for`, `while`, `do-while`

Loops allow a program to execute the same block of code repeatedly. Without loops, writing repetitive programs would be extremely tedious (even with cut and paste). The three common types of loops in C++ are `do-while`, `while`, and `for`.

The `do-while` loop is used when the code within the loop should be executed at least once. The basic structure of a `do-while` loop is

```
do{
    Execute this code the first time and
    while the condition is true
}while(condition);
Jump to the code here when the condition is false
```

Because the condition comes at the end of the loop, the code within the loop is executed at least once before the condition is checked.

[2]The `variable` can be an expression that evaluates to either an integer or a character, in which case the program evaluates the expression and jumps to the case matching the result. However, this can make for very confusing code. If an expression is necessary it's better to include it as a separate line of code before the switch statement and store the result of the expression in a variable that's used to control the switch.

The basic structure of a `while` loop is:

```
while(condition){
    Execute this code while the condition is true
}
Jump to the code here when the condition is false
```

The condition comes at the beginning of the loop, so if the condition is false the first time it is tested, the statements within the loop won't be executed.

`For` loops are generally used for "counting" tasks, doing something N times. The basic structure of the `for` loop is

```
for(statement 1; conditional ; statement 32){
    Execute this code while the condition is true
}
Jump to the code here when the condition is false
```

For loops are generally controlled by a *counter* variable that is used to determine how many times the loop will be executed. `Statement 1` is the *initializer*, it is executed when the loop begins and is used to initialize the counter variable for the loop. The `condition` is checked at the beginning of each loop, if it is true the loop code is run and if its false the loop is exited and the program jumps to the code after the loop. `Statement 2` is executed at the end of each loop and is used to increment the counter variable.

The easiest way to understand a `for` loop is by example. For example, a `for` loop to print the numbers from 1 to 100 is:

```
for(int i = 1 ; i <= 100 ; i++){
      cout << i << endl;
}
```

The first statement creates a new integer and sets it equal to 1, the second statement (the conditional) checks whether the counter has reached 100, and the third statement increments the counter.

As is usual in C++, the curly brackets defining the block after a loop statement can be omitted if only a single statement is used in the body of the loop.

1.4 Mathematical Operators

Mathematical operators are simply used to perform calculations within a program. Table 1.2 presents the common C++ mathematical operators. C++ uses the standard rules for order of operation and precedence when evaluating complex expressions. For example, multiplication and division are applied before addition and subtraction, and addition and subtraction are applied from left to right. However, there are several reasons why it is a good idea to use parentheses in complex mathematical expressions. First, many of the operators available in C++ are unusual (`++`, `%`, `+=`, etc.) and their order of operation is not immediately clear. Using parentheses guarantees that expressions are evaluated in the desired order. Second, well-placed parentheses, and spacing, make complex expressions easier to read and understand.

In mathematical expressions with variables or literals of different types C++ will often automatically convert compatible numeric types. Generally, C++ will convert lower precision types (e.g., `int`) in to higher precision types (e.g., `double`). This is known as *implicit conversion*, *implicit casting*, or *promotion*. For example, the expression `9.0/5` will return the value 1.8 - the integer 5

Table 1.2: Common C++ Mathematical Operators.

Operation	Symbol	Description
Addition	`+`	Addition
Subtraction	`-`	Subtraction
Multiplication	`*`	Multiplication
Division	`/`	Division; division of two integers results in an integer, the decimal is truncated
Modulus	`%`	Performs division and returns the remainder
Increment	`++`	Increments a variable by 1; note that `x++` and `++x` have different orders of operation in compound expressions.
Decrement	`--`	Decrements a variable by 1; note that `x--` and `--x` have different orders of operation in compound expressions.
Addition assignment	`+=`	Increases a variable by the given amount; e.g., `x += 7` increases `x` by 7.
Subtraction assignment	`-=`	Decreases a variable by the given amount; e.g., `x -= 7` decreases `x` by 7.
Multiplication assignment	`*=`	Multiplies a variable by the given amount; e.g., `x *= 7` increases `x` by a factor of 7 and stores the result in `x`.
Division assignment	`/=`	Divides a variable by the given amount; e.g., `x /= 7` divides `x` by 7 and stores the result in `x`.
Modulus assignment	`%=`	Divides a variable by the given amount and takes the remainder; e.g., `x %= 7` divides `x` by 7 and stores the *remainder* in `x`.

has been promoted to a `double` to match the type of the 9.0.

The programmer can also force C++ to *temporarily* convert a type for a single operation. For example,

```
int x = 7;
cout << double(x)/14;
```

will print 0.5 because the `x` is temporarily treated as a `double`. An equivlelent

Caution has to be used when using implicit conversion. The expression `9/5` will return the value 1 - both values are integers so no conversion takes place and the answer is *truncated* to return an integer. More subtly the expression `1.5 * (5/9)` will return the value 0 because the operation `(5/9)` is performed first and it returns 0 (no conversion is done because 5 and 9 are both integers). Thus, it's a good idea to use explicit conversion

Table 1.3: Common C++ Comparisons and Boolean Operators.

Operation	Symbol	Description
Less than	<	Is true if the left operand is less than the right operand
Greater than	>	Is true if the left operand is greater than the right operand
Less than or equal	<=	Is true if the left operand is less than or equal to the right operand
Greater or equal	>=	Is true if the left operand is greater than or equal to the right operand
Equal	==	Is true if both operands are equal
Not equal	!=	Is true if the operands are not equal
AND	&&	Boolean AND; is true only if both operands are true
OR	||	Boolean OR; is true if either operand is true
NOT	!	Boolean NOT; is true if the operand is false

1.5 Comparison and Boolean Operators

Comparison and Boolean operators are used to compare values and to construct logical expressions. They are almost always used to define the conditions in conditional statements and loops. Table 1.3 lists the common comparison and Boolean operators used in C++. Boolean operators always return a 1 representing `true` or a 0 representing `false`. Thus, a comparison such as $7 < 9$ has the value 1, which is treated as true in conditionals. So,

```
cout << 7 < 9 << "\n";
```

would print the value 1 (not `true`), but $7 < 9$ is treated as true in Boolean operations.

Similarly, a comparison such as $9 < 7$ has the value 0, which is treated as false in conditionals, and the statement `cout << 9 < 7 << "\n";` would print 0. More generally, any nonzero value is treated as `true` and a zero value is treated as `false` within conditionals. Thus, a statement such as `if(6)` is treated as `if(true)` because 6 is not zero; and `if(6 - 3*2)` is treated as `if(false)` because `6 - 3*2` is zero.

There are a few common mistakes to avoid when using these operators. First, in addition to the Boolean operators && and || C++ has operators & and |. The & and | operators perform *binary* AND and OR operations, which are very different from the *Boolean* operations. If a compound conditional is not behaving as expected check that it has && or || and not & or |.

A similar, common, mistake is to use a single equals sign = instead of a double equals sign == in a condition. The double equal sign compares two values and returns *true* (1) if they are the same and *false* (0) if they are different. The single equal sign is an assignment. Thus, the (incorrect) statement:

```
if(x = 7){
  Execute this block of code if true
}
```

assigns `x` the value 7 and always executes the code within the `if` (because `x = 7` has the value 7, which is treated as true).

1.6 The `String` Class

The `string` class defines `string` objects, which are used to store strings of characters. Strings are useful for storing names, addresses, and other pieces of text longer than a single character. Table 1.4 lists some of the more useful functions that are defined as part of the `string` class. Keep in mind that a `string` object is different from a C-style string (see Interlude 5), but that the member function `c_str()` can be used to return a C-style string from a `string` object.

1.7 `iostream` Operations

The iostream library defines classes, objects, and functions related to input and output. Table 1.5 lists some of the functions that can be applied to `iostream` objects. Most of these can be applied to either to `cin` or to an input file or to `cout` or an output file.

1.8 Libraries

Many useful libraries have been created to extend C++. If you are starting a project that will require lots of specialized code, for example, complex mathematical functions, the ability to manipulate vectors, or complex data structures, it's a good idea to see if an appropriate library already exists. It's rare that a library will contain code to do exactly what you want your program to do, but there are often libraries that will provide the basic code for your program, making it much easier to write.

Table 1.6 lists some of the more useful and commonly used libraries. Many, but not all, of these are used in the text. More information on the functions defined within these libraries can be found on-line.

Table 1.4: Useful Functions Related to the `string` Class.

The examples use the following definitions:
`char c_str[]`
`string str1, str2`
`int N`		

Operation	**Example**	**Description**
`=`	`str1 = "some characters";`	Assigns a string of characters to a `string` object.
`c_str()`	`str1.c_str()`	Returns the string of characters as a C-style string.
`length()`	`str1.length()`	Returns the length of the string.
`size()`	`str1.size()`	Returns the length of the string.
`empty()`	`str1.empty()`	Returns true if the string is empty (length is 0); otherwise, it returns false.
`[int]`	`str1[N]`	Returns the character at location `N` in the string. The first position in the string is 0, not 1.
`at(int)`	`str1.at(N)`	Returns the character at location `N` in the string. The first position in the string is 0, not 1. Unlike the `([])` operator, `at()` performs bounds checking and will return an error if out of bounds.
`find(string)`	`str1.find(str2)`	Returns the location of `str2` in `str1`.
`+=`	`str1+= str2`	Appends `str2` onto `str1`.
`+`	`str1 = str2 + " " + str3`	Concatenates `str3` onto `str2` and stores the result in `str1`.
`getline(istream,string)`	`getline(cin,str1)`	Gets a string of characters (including spaces and tabs) up to a new line character from the given input stream (`cin` in the example) and stores them in the given string object (`str1` in the example).
`==, !=, <,` `<=, >, >=`	`str1 == str2`	Boolean operators that compare two strings lexigraphically.

Table 1.5: Useful Functions of the `iostream` and `fstream` classes. Most, but not all, of these functions can be applied to `cin` or an input file or `cout` or an output file. Many of them have variants to control the number of characters read, format of output, delimiting characters, etc.

The examples use the following definitions:
`char char_str[]` // a c_style array of characters
`string str` // a string object
`ifstream infile`
`ofstream outfile`
`int N`		

Operation	**Examples**	**Description**

`open()`	`infile.open(char_str)`
	`outfile.open(char_str)`
	`infile.open(str.c_str())`
	`outfile.open(str.c_str())`
		Opens a file stream (input or output) with the given name. Note that the argument must be C-style array of characters (`char [] or char *`). Not used with `cin` or `cout`.

`close()`	`infile.close()` `outfile.close()`	Closes a file stream (input or output). Not used with `cin` or `cout`.
`>>`	`infile >> str` `cin >> str`	Gets a string of characters (up to the first whitespace character) from the input stream. The string object `str` can be replaced with other standard types (`int`, `char`, `double`, etc.).
`<<`	`outfile << str` `cout << str`	Sends the string of characters from the given `string` object to the output stream. The string object `str` can be replaced with other standard types (`int`, `char`, `double`, etc.).
`get()`	`infile.get()` `cin.get()`	Gets and returns a character from the input stream. Variants allow the programmer to store the character, get multiple characters, get characters up to a delimiter, etc.
`ignore()`	`infile.ignore()` `cin.ignore()`	Reads and discards the next character from the input stream. Variants allow the programmer to ignore up to N characters and/or characters up to a specific character.
`getline()`	`infile.getline(char_str,N)` `cin.getline(char_str,N)`	Reads characters from the input stream, up to `N-1` characters or a new line character, whichever comes first, and stores them in a C-style array of characters. Variants allow the programmer to specify the delimiting character.
`eof()`	`infile.eof()`	Returns true if the input stream's end of file bit has been set and false otherwise. Only regularly used with files.
`peek()`	`infile.peek()` `cin.peek()`	Returns the next character without removing it from the input stream.

Table 1.6: Some Useful Libraries.

Library	**Description**
`cstdlib`	Contains many general purpose-functions: conversion from C-style strings to numbers, generation of random numbers, and dynamic memory management.
`cmath`	Contains useful mathematical functions: trigonometric, hyperbolic, exponential, rounding, etc.
`ctime`	Contains useful functions relating to time: get the current time, calculate the difference between times, convert times to strings, etc.
`cstring`	Contains useful functions for manipulating C-style strings: concatenate strings, compare strings, search strings, etc.
`fstream`	Defines the `fstream` classes (file stream) used in C++.
`iostream`	Defines the `iostream` classes used in C++.
`string`	Defines the `string` class for C++.
`iomanip`	Defines stream manipulators used in C++. These are used for formatting output to streams, setting width, precision, scientific notation, etc.

Chapter 2

Lexical

This chapter discusses lexical rules inferred from the text of "A Project-Based Introduction to C++". Wherever this chapter says "lexical category" you can mentally substitute "integer code with which I am going to identify to the parser what kind of word we are dealing with here".

2.1 Identifiers

A single lexical category is utilized for all kinds of names. Variable names can use letters, underscores and digits, but cannot start with a digit and hence, must contain at least one letter. Case matters.

Note that the Soule text claims _123 is not a legal variable name ("does not contain a letter"), while the C/C++ languages allow it (underscore is considered to be a letter). When the Soule text is actually incorrect, you should ignore it and implement C++ correctly.

2.2 Operators and Punctuation

You can expect to use a separate lexical category for each operator or punctuation mark; this makes for a whole lot of lexical categories, and drives up the size of tables that the parser will use. If two or more operators are used interchangeably, a compiler writer can assign them to a single, shared lexical category, run them through syntax analysis identically, and look at their other lexical attributes during code generation in order to generate the right code. This may save space in the parser.

```
( ) { } [ ]
= + - * / %
< <= > >= && || ! == !=
+= -= ++ --
<< >> & . -> : :: ;
```

2.3 Literal Values

120++ has integer, float, bool, character and string literals. Bool literals are true and false. Real number literals have digits and a decimal; 120++ does not use C/C++ scientific literals such as

5.3e20. 120++ supports a number of escapes in character and string literals:

Name	Interpretation (ASCII)
\n	newline (10)
\t	tab (9)
\'	apostrophe (39)
\\	backslash (92)
\"	doublequote (34)
\0	nul (0)

2.4 OOP Features

120++ introduces classes but not inheritance.

```
class public private ~
```

2.5 Comments

120++ uses both C and C++ style comments. Comments do not appear in the language syntax, they are removed during lexical analysis. Despite their simplicty, C comments are famously difficult to write a regular expression for, yet it is provably possible to do so and a correct regular expression can easily be obtained by internet search. By why are they difficult?

```
/* */
//
```

2.6 includes

120++ uses only a few system includes, inside less than and greater than marks, for which only a tiny subset of their content is necessary. 120++ allows application includes inside doublequotes, only for .h files in the current working directory. Application includes may only contain class declarations, constant declarations, and (non-circular) #includes at most two-levels deep,

```
#include <iostream> - if present it allows cout/cin
#include <fstream> - if present it allows ifstream
#include <string> - if present it allows string
#include <cstdlib> - if present it allows rand(), srand()
#include <ctime> - if present it allows time()
#include <cmath> - if present it allows sin()
#include "file.h" - application include
```

2.7 namespaces

The only namespace used in 120++ is

```
using namespace std;
```

In fact, this namespace can be required of all 120++ programs, but that would be done in either the syntax or semantic analyzers. In any case, `using` and `namespace` are reserved words. The identifier `std` is predefined to be a namespace name, meaning that when your lexical analyzer sees `std` it should return the integer code for `NAMESPACE_NAME` instead of `IDENTIFIER`.

2.8 Built-in classes, functions, operators

See Soule's reference section.

```
cout << cin >> endl cin.ignore() string ifstream ofstream fstream
```

2.9 built-in types

See section 1 of this document.

```
int void double char bool
```

120++ does include float, short, long, and unsigned, but all types are to be implemented as either one-byte (char) or eight-byte (everything else) signed numbers. For example, a float is allowed and is implemented as a double.

2.10 reserved words

These generally all need their own lexical category since they appear in different places in the syntax.

```
if return while else do case break switch for
using namespace new delete
```

Chapter 3

Syntax

This chapter discusses lexical rules inferred from the text of "A Project-Based Introduction to C++". You are to implement a subset of C++ to include at least the following syntax constructs.

3.1 Namespaces

You are allowed, but not required, to require the declaration

```
using namespace std;
```

The preferred handling would be to require it if system includes are present.

3.2 Function prototypes

120++ allows zero or more parameters which consist of a type followed by optional square-brackets or ampersand. Examples:

```
type name();
type name(type,type);
type name(type []);
type name(type &);
type name(type *); /* doublecheck */
```

3.3 Function bodies

120++ functions have a return type, a function name with optional classname:: prefix, zero or more parameters, declarations and statements inside appropriate parentheses and curly braces. Examples:

```
type name() { declarations statements }
type name::name() { declarations statements }
type name(type name,type name) { declarations statements }
```

3.4 Classes

Although the Soule text mentions the existence of structs and typedefs in C, it does not use them and 120++ will not include them. 120++ does include simple classes, with no inheritance.

```
class name { fields methods } ;
```

3.5 Variable Declarations

120++ includes common forms of C++ variable declarations. Note that variable declarations are all global or at the beginning of their respective scope blocks, with an allowance for variables declared spontaneously inside for-loops. This is a restriction on standard C++ which allows variable declarations at any point within any statement block.

```
type name;
type name1, name2, name3;
type name4 = value;
const type name4 = value;
type name[size];
type *name;
type *name[size1][size2];
```

Note that although I saw in one place in the Soule text some code that suggested the pattern

```
type [size] name;
```

I cannot confirm this as legal C++ and it is not in 120++

3.6 Statements

Statements in 120++ include the following, along with void function calls. Statements are often terminated with semi-colons.

```
if (expr) statement
if (expr) { statements }
if (expr) { statements } else { statements }
if (expr) { statements } else statement
switch (expr) { statements }
case intvalue: statements break
case charvalue: statements break
case (value): statements break
case val1: case val2: statements break
default : statements
break
do { statements } while (expr);
while (expr) { statements }
for (type var = val ; expr ; expr) { statements }
for (var = val ; expr ; expr) { statements }
```

3.7 Expressions

```
cout << string ;
cout << string1 << expr << string2 ;
cin >> name ;
cin >> name >> name2 >> name3 ;
&x
*p
p->field
NULL
new type
new type(args)
(type)(expr)
delete p
name1 = expr ;
name1.name2 = expr;
return expr;
expr op expr
expr++
expr--
(expr)
f()
f(expr,expr)
type(expr)
name1.name2
```

Chapter 4

Semantics

This chapter presents various aspects of the 120++ language as they pertain to the semantic analysis of the input.

4.1 Scope Rules

120++ implements three levels of scoping: global, class, and local. Classes inside classes are not supported. Nested local scopes inside (compound) statements are treatd as part of the enclosing local scope.

4.1.1 Storage Classifiers

Storage classifiers do not appear in 120++. Some (such as register) might be ignored if found on an input source file. 120++ uses const only in global and local variable declarations; it does not use them in parameters.

120++ does not use extern. It can be treated as a semantic error, ignored (for prototypes), or implemented.

120++ does not use static. A static outside any function could be reported as a semantic error, treated as a global, or implemented. A static inside a function could be reported as a semantic error, treated as a global with name-mangling, or implemented.

4.2 Type Checking

120++ would report errors for any illegal combination of types, such as trying to add an object to a real number.

4.2.1 Type Promotion

120++ only supports two sizes: byte (1 byte, used by char and bool) and word (8 bytes, used by all other types). It does not discuss type promotion or compatibility. 120++ is free to require that types be exact matches, with the exception of the common interchangeability of pointers and arrays.

4.2.2 Parameter Types

120++ does use pointers, references, and base types as parameters.

4.2.3 Pointers

120++ does not do pointer arithmetic. It does use new, delete, &, , and $->$ operators.

4.3 Control Structure Constraints

Although the grammar may allow interesting things, 120++ does not allow anything that C++ does not allow, and may be further constrained as follows.

4.3.1 No Gotos or Labels (Other Than Switch Cases)

This goes without saying.

4.3.2 No Cases Outside of Switches

Although one would think the C++ compiler would complain, some might not.

4.3.3 No Breaks Outside of Loops or Switches

The keyword break must always have an obvious enclosing control structure.

4.4 Function and Operator Overloading

Although the Soule text mentions the existence of multiple constructors, it never uses this feature, or function overloading in general.

Operator overloading does not appear in 120++.

4.5 Object Oriented Features

120++ supports simple classes, without inheritance, but with member variables that are direct instances of other classes, as well as member variable pointers to the same or other classes, such as those used for link lists.

4.5.1 Constructors and Destructors

120++ mentions the concept of function overloading as it pertains to class constructors, but never utilizes it in a live example. It does utilize constructors that have parameters as well as ones that do not. It does contain a class (world, in chapter 6) that has no constructor and is therefore using a default constructor. There is something ironic about world having a set_up() method instead of a consructor. It would be useful for it to have a constructor that calls set_up(), to illustrate a common implementation pattern which is to have an object that can be reset to its start state.

4.5.2 Virtual

The keyword virtual does not appear in 120++.

4.5.3 New and Delete

120++ uses new and delete expressions, albeit exceedingly sparingly. delete is used in an actual program example only once (delete temp in listing 7.5). New is used more aggressively, in contexts such as:

```
p = new int;
p = new node;
p = new robot(1);
```

4.6 "Built-in" Classes

If the correct include and "using namespace" declarations are present, 120++ supports at least the following classes as built-ins.

```
class ifstream {
   public:
      void close();
      bool eof();
      void ignore();
      void open(char *);
   }
class ofstream {
   public:
      void close();
      void open(char *);
   }
class string {
   public:
   char *c_str();
   }
```

The main operations on the stream built-ins are of course << and >>. There is also a function getline(ifstream, string&).

4.7 Miscellaneous

There are neither friend functions nor friend classes in 120++. There are no templates.