Appendix A
C++ Programming
Style Guide

Good style is like touch typing: it may seem counter-productive at first, but the initial effort will pay enormous dividends. After a little while, the elements of good style will be second nature. Good style will help you to debug your programs more easily, and also make them more easily readable by your instructor and the grader.

A.1 Readability

There are a number of stylistic elements which can make a program more readable, including the use of horizontal and vertical spacing, the conventions used in declarations, etc. Each of these elements will be discussed in turn.

Keep in mind that once a program is written, it is seldom read from top to bottom. While debugging or modifying a program, programmers often skip large blocks of text in order to find what they are looking for. A good analogy can be made to a dictionary. Imagine if the words in a dictionary were written in normal English style, as is this article. What if the dictionary were not alphabetized? What if the words being defined did not appear in boldface?

As a good programmer you should strive to enhance the visual appearance of the code you write. The effort you put in will begin to pay dividends as you debug your code.

A.1.1 Indentation

Indentation is used to enable a reader to determine the nesting level of a statement at a glance. In order to be useful, indentation must be consistent - the number of spaces used per indentation level should be between 3 and 5 - and the same style of indentation should be used throughout the program. Proper indentation makes your program much easier to debug.

A.1.2 Spaces

Normally in programming the standard for the use of spaces is that you follow normal English rules. This means that:

1. Most basic symbols in C++ (e.g., “=”, “+”, etc.) should have at least one space before and one space after them, with the following notable exceptions:

   • No space appears before a comma or a semicolon.

   • No space appears before or after a period.

   • No space appears between unary operators and their operands (e.g. "++").

```
for (counter = 0; counter < length; counter++) {
    friendFile « addressBook[counter].firstName « ' '
               « addressBook[counter].lastName « endl;
    friendFile « '(' « addressBook[counter].areaCode « “)”
               « addressBook[counter].phoneNumber « endl;
    friendFile « setw(2) « addressBook[counter].month « '/'
               « setw(2) « addressBook[counter].day « '/'
               « setw(4) « addressBook[counter].year « endl;
    friendFile « endl;
} // for
```

Figure A.1: An example of good spacing practices.

2. More than one space may be used if you are aligning things on adjacent lines, as is done with the "«"in Figure A.1.

A.1.3 Blank Lines

Blank lines should be used to separate long, logically related blocks of code. Specifically:

1. In the global section of a compilation unit, the `include`, `const`, `typedef`, and variable declaration sections should be separated by at least one blank line.

2. Within a long piece of code, groups of related statements may be separated from other groups by a blank line.

3. To be effective as an element of style, blank lines should be used consistently.

Refer to the example in Section A.6 for an illustration of these guidelines.

A.1.4 Statements

1. Each statement should appear on a separate line.

2. The opening brace following a control statement such as if or while should appear on the same line as the if or while, and the closing brace should appear on its own line, lined up with the left of the control statement. As an example, see the for loop in Figure A.1. In contrast, both the opening and closing braces for a function should appear unindented on lines by themselves.

3. The statements within a {....} pair are indented relative to the braces. Again, see Figure A.1 for an example.

4. Even if only a single statement falls in the body of a compound statement, it is indented on a separate line.

A.1.5 Declarations

1. Variables should be listed *one per line*, with the type of the variable preceding *every* declaration. Do not put blank lines between identifiers being declared. The same rules apply to fields declared within a `struct`. As an example, refer to Figure A.2.

```
struct Entry {
    String15      firstName;   // First name of a friend
    String15      lastName;    // Last name of a friend
    int           areaCode;    // Range 100. .999
    String8       phoneNumber; // Phone number of a friend
    int           month;       // Range 1. .12
    int           day;         // Range 1. .31
    int           year;        // Range 1900. .2100
};

int        length = 0;   // Number of entries in addressBook
Entry      currentEntry; // Current record being entered
char       response;     // Response character from keyboard
ofstream   friendFile;   // Output file of entries
char       fileName[51]; // User-specified file name (max. 50 chars)
```

Figure A.2: An example of variable declarations.

2. Variables should be grouped functionally. In other words, related identifiers should be grouped together in the declaration section.

A.2 Comments

In the real world, both maintenance programmers and other members of a programming team rely on comments to explain the program, function, or code fragment they are reading. If a comment and the code disagree, the comment is presumed to be correct, and the code incorrect. Comments are used primarily to state *what* the code is doing (its purpose), while the code itself describes *how* you are doing it. Thus, it is only common sense that you should write the comment first (i.e., define what you are doing) *before* you write the code.

Comments fall into one of the following groups:

1. Function prologue comments
2. Program prologue comments
3. Declaration comments
4. Sidebar comments
5. In-line comments

A.2.1 Function Prologue

See Section A.4 for a discussion of function prologues.

A.2.2 Program Prologue

The major function of a program prologue is to explain the purpose of the program. A program prologue is similar to a function prologue and includes the following sections, following the name of the file (the first three are particular to student projects):

1. **Your name**.
2. **Date** (or semester and year).
3. **Class** and **professor's name**.
4. **Purpose**: an explanation of what the program does.
5. **Algorithm**: a general description or outline of the processing done.

6. **Program input**.

7. **Program output**.

8. **A description of the data structures used**. This section is optional, depending on need.

9. **Limitations or restrictions**: what assumptions are made about the input data; under what conditions the program or unit fails to operate properly, etc.

10. **Modification history**: who has modified the program, when, and why. This section is normally started once the program goes into production; thus it seldom appears in student programs.

See Figure A.3 for an example of a program prologue.

```
//*****************************************************
// friends.cpp
//   Author:    Joe Student
//   Date:      November 15, 2000
//   Class:     CS 161, Professor Royden
//   Purpose:   This program creates an address book.
//   Input:     (from standard input) first names, last
//              names, phone numbers, and birth dates of a
//              group of people
//   Output:    (to an output file) an alphabetical listing
//              of address book entries
//*****************************************************
```

Figure A.3: An example of a program prologue.

A.2.3 Declaration Comments

1. Constants and variables are *always* commented with short, precise comments stating their purpose. These comments normally follow the declaration on the same line and only rarely take more than a line or two. See Figure A.2 for some examples.

2. It is usually not necessary to comment types, whether global or local (although they are commented in our example program). However, fields within a struct are always

commented.

A.2.4 Sidebar Comments

A sidebar comment is one which explains a single statement and should follow the statement on the same line. The comment should be brief, accurate and *precise*. See Figure A.4 for some examples. Such comments are common in assembly code.

```
void Insert( ... )
{
   list[index] = item;    // Insert item
   length++;              // Increment length of list
}  // Insert
```

Figure A.4: Examples of sidebar comments.

A sidebar comment should always be used after a closing brace to *uniquely* identify the compound statement which is being ended, and is essential in finding the matching opening brace. In the case of a function, the comment contains the name of the function.

A.2.5 In-line Comments

In-line comments explain a block of code. Such a comment should precede the code itself and should be indented the same as the block it describes. A blank line should be placed before the in-line comment, between the in-line comment and the block of code, and after the block of code to separate it from the next block of code. See Figure A.5 for an example of when to use in-line comments.

```
// Get a friend's first name

cout << "Enter person's first name." << endl;
cin.get(theEntry.firstName, 16);
cin.ignore(100, '\n');

// Get a friend's last name

cout << "Enter person's last name." << endl;
cin.get(theEntry.lastName, 16);
cin.ignore(100, '\n');
```

Figure A.5: An example of in-line comments.

Sidebar comments and in-line comments should be used sparingly. Before adding such comments you should first attempt to make the code itself more understandable by improving the identifier names used, replacing groups of statements with function calls, reducing the control complexity of the code, etc.

A.3 Naming

When choosing names for items in your program, *order* is very important. As an example, suppose a variable `state` has been declared of type `StateType`. This is fine as long as only one variable of this type is used. The naming problem starts when a second variable of the same type is needed. First shot: `state2`. This is certainly better than naming it `s`, *but* it would have been *even better* to pick really good names like `currentState`, `lastState`, `normalState`, `errorState`, etc. for the variables and just name the type `State`.

It is clear from this example that the *type names must be chosen first*. (This is no surprise when looking at object oriented programs or abstract data types.) After having chosen the type names one can start to name the functions, variables, and constants. The remainder of this section discusses the structure of names for different entities in the order they should be named.

A.3.1 Type Names

The simplest and shortest names should be reserved for type identifiers. Therefore they must be chosen before any other name, especially before the names for variables of this type. Types should be named with short, generic nouns that reflect their contents. Type names

should start with a capital letter and, if the name consists of several words concatenated together, each successive word should also be capitalized.

Examples: `Entry`, `Table`, `Name`, `Node`, `StateTable`, `FileName`, `TableIndex`.

A.3.2 Function Names

See Section A.4 for a discussion of function naming.

A.3.3 Variable Names

In strongly typed languages, a variable is of a particular type. Therefore the structure "adjective + type name" for variable names is an obvious suggestion. (The name does not have to contain an adjective; alternatively, a name can take on the more general form "qualified type name".) Note that this naming convention is not always appropriate however, especially when variables have standard types. For instance, if your program contains an integer variable which stores the length of a list, it would be much better to choose listLength or just length over lengthInt! The first letter of variable names should *not* be capitalized, but each successive word in the name is capitalized.

Examples: `response`, `length`, `currentEntry`, `headPointer`, `currentSymbol`.

A.3.4 Constants

All non-trivial constant values in a program should be assigned names. Constants often describe a limit within a program. In these cases it is appropriate to use the prefix MAX or MIN in conjunction with the type name. Otherwise, name constants like variable names. Names for constants should be in all capital letters with underscores between words.

Examples: `MAX_FRIENDS`, `PI`, `BLANK`, `MAX_LINE_LENGTH`, `TAX_RATE`, `LOWER_LIMIT`.

A.3.5 General Hints on Naming

The most important criterion when choosing a name is: how easily can *another programmer* (not just yourself) understand the program? If understanding a name was not important we could just name variables `a`, `b`, etc. Here are some additional pointers on how to choose names in a program.

- Names must be pronounceable. You should opt to use untruncated, long names over using names that are not pronounceable. As a "rule of mouth", if you cannot read the name out loud, it is not a good name.

  Examples: `groupID` instead of `grplD`, `nameLength` instead of `namln`, `powersOfTwo` instead of `pwrsOf2`.

- Abbreviate with care. Abbreviations always carry the risk of being misunderstood. For example, does `termProcess` mean `terminateProcess` or `terminalprocess`? Abbreviations are usually also hard to pronounce (for example, `nxtGrp`). Use only commonly known abbreviations, like the `ID` in `processID`. As a general rule of thumb, you should only abbreviate a name if it saves more than three characters.

  Examples: `error` instead of `err`, `name` instead of `nam`, but `maxLineLength` is probably better than `maximumLineLength`.

- Do not use names whose only difference is capitalization. C++ is case sensitive, so the name `groupID` is different from the name `groupId`. If two names in the same program only differ in capitalization, typographical mistakes can create errors that are very difficult to track down.

- Boolean variable and function names should state a fact that can be true or false. This is easy to achieve with the inclusion of "is" in the name.

  Examples: `printerIsReady`, `queueIsEmpty`, or simply `done`. Note how naturally this reads:

  ```
  if (queueIsEmpty) Insert(item);
  ```

- The more important (read *global)* an object is, the more care should go into choosing its name. In a short function, a variable like `ok` is probably fine since "what is OK" is probably easily determined from the context. However, this is most likely *not* the case with a global variable. Thus, the most care should be taken when naming global variables in a program, followed by field names within a record, and finally variables in a function.

A.4 Functions

In this section, we consolidate all the style guidelines relevant to function declarations. An

example illustrating good formatting practices is shown in Figure A.6.

```
//*******************************************************
// void WriteEntries ( const Entry addressBook[], int length,
//                       ofstream& friendFile )
// Purpose:        Writes all entries to the file friendFile
// Output:         (to an output file) all addressBook entries
// Precondition:   length <= MAX_FRIENDS
//                 && addressBook[0. length-1] are assigned
// Postcondition:  Contents of addressBook[0..length-1]
//                 have been output to friendFile
//*******************************************************

void WriteEntries ( const Entry addressBook[], // Array of entries
                    int         length,        // Number of entries
                    ofstream&   friendFile )   // File receiving list
{
     :
     :
} // WriteEntries
```

Figure A.6: An example of a well formatted function declaration.

A.4.l Function Prologues

The major reason for a function prologue is to explain the purpose of the function. A function prologue should appear just before the implementation of the function and include the following sections:

1. **Function name and parameter list**: just as it appears later in the actual code.

2. **Purpose**: what the function does.

3. **Algorithm**: how the function does what it does. If a standard algorithm such as Quicksort is used, a reference rather than an explanation is preferred.

4. **Input and Output**: what the function will expect from the user and what the user will see on the screen.

5. **Precondition**: what assumptions the routine makes about its data; under what conditions the routine fails to operate properly, etc.

6. **Postcondition**: what should be true after the routine is finished. An explanation of the return value, if any, should also be included.

Any of the these items except the first two may be left out if they are inappropriate. For example, if the function does not expect any input, leave out the input section. See Figure A.6 for an example of a function prologue.

A.4.2 Function Names

Functions should be named differently depending upon whether they return a value. A void function is (literally) called by its name, which stands for a group of statements to be executed. Therefore the name of a void function should express the implied action ("do this") by including an imperative verb. Since functions operate on a specific type, the structure "verb + type name" is best suited for a function name. Function names should be capitalized like type names.

Examples: `GetEntry`, `DisplayError`, `PrintAddress`, `GetFirstElement`, `FindName`.

Functions that return a value should contain nouns or adjectives. Again, since these types of functions operate on a specific type, the form "adjective + type name" or "noun + type name" are good choices.

Examples: `GreatestItem`, `CubeRoot`, `LastNode`, `HeadOfList`, `IsEmpty`.

A.4.3 Formatting Function Declarations

The following guidelines apply to function declarations and prototypes. All should be illustrated in Figure A.6.

- When declaring functions, the leading parenthesis and first parameter (if any) are to be written on the same line as the function name. Then, each subsequent parameter should be listed on a separate line to allow for each to be commented.

- Each function parameter is always commented on the same line as the declaration.

- In function declarations and prototypes, a space should appear after the opening parenthesis beginning the parameter list and before the matching closing parenthesis. However, no such spaces should be used in function calls.

- Functions should be separated by at least two blank lines.

A.5 Miscellaneous Guidelines

1. The main() function should *always* be written in the following style:

```
int main ()
{
   < statements >

   return 0;
} // main
```

2. Few constants should appear in your code, other than `0`,`1`, and ‘ ‘. All other constants should be declared and named in a `const` declaration.

3. Use the operators ++ and -- only in statements, *never* as part of larger expressions.

A.6 An Example Program

The following program exemplifies the style guidelines outlined in this Appendix.

```
//*******************************************************
// friends.cpp
//     Author:  Joe Student
//     Date:    November 15, 1999
//     Class:   CS 61, Professor King
//     Purpose: This program creates an address book.
//     Input:   (from standard input) first names, last names, phone
//              numbers, and birth dates of a group of people
//     Output:  (to an output file) an alphabetical listing of address
//              book entries
//*******************************************************

#include <iostream.h>
#include <iomanip.h>    // For setw()
#include <fstream.h>    // For file I/O
#include <string.h>     // For strcmp()
#include <ctype.h>      // For toupper()

typedef int  Boolean;
typedef char String8[9];    // Room for 8 characters plus '\0'
typedef char String15[16];  // Room for 15 characters plus '\0'
```

```
const Boolean TRUE = 1;
const Boolean FALSE = 0;
const int     MAX_FRIENDS = 150;  // Maximum number of friends

struct Entry {
    String15 firstName;    // First name of a friend
    String15 lastName;     // Last name of a friend
    int      areaCode;     // Range  100. .999
    String8  phoneNumber;  // Phone number of a friend
    int      month;        // Range  1..12
    int      day;          // Range  1..31
    int      year;         // Range  1900..2100
};

void GetEntry( Entry& );
void Insert( Entry[], int&, Entry );
void WriteEntries( const Entry[], int, of stream& );

int main ()
{
    Entry addressBook[MAX_FRIENDS]; // Array of friends' records
    int   length = 0;       // Number of entries in addressBook
    Entry currentEntry;     // Current record being entered
    char  response;         // Response character from keyboard
    ofstream   friendFile;  // Output file of entries
    char  fileName[51];     // User-specified file name (max. 50 chars)

    // Prompt the user for the name of an output file and open the file

    cout << "Output file name: ";
    cin.get(fileName, 51);
    cin.ignore(100, '\n');

    friendFile .open(fileName);
    if (!friendFile) {
        cout << "** Can't open " << fileName << " **" << endl;
        return 1;
    }

    // Prompt the user for up to MAX_FRIENDS address book entries

    do {
        GetEntry(currentEntry);
        cout << "Is this entry correct? (Y or N) "
        cin >> response;
        if (toupper(response) == 'Y')
            Insert (addressBook, length, currentEntry);
            cout << "Do you wish to continue? (Y or N) ";
            cin >> response;
            cin.ignore(100, '\n');
    } while (toupper(response) == 'Y' && length < MAX_FRIENDS);
```

```
    if (length == MAX_FRIENDS)
        cout << "Address book is full." << endl;

    WriteEntries (addressBook, length, friendFile);

    return 0;
} // main


//*****************************************************
// void GetEntry( Entry& theEntry )
// Purpose:       Builds and returns a complete address book entry
// Input:         (from standard input) a friend's first name,
//                 last name, phone number, and birth date
// Postcondition: User has been prompted for a friend's first name
//                and last name
//                && entry.firstName == input string for first name
//                && entry.lastName == input string for last name
//*****************************************************
void GetEntry( Entry& theEntry )  // Struct being built
{
    // Get a friend's first name

    cout << "Enter person's first name." << endl;
    cin.get(theEntry.firstName, 16);
    cin.ignore(100, '\n');

    // Get a friend's last name

    cout << "Enter person's last name." << endl;
    cin.get(theEntry.lastName, 16);
    cin.ignore(100, '\n');

    // Get a friend's phone number

    cout << "Enter area code, blank, and the number"
         << " (including '-')." << endl;
    cin >> theEntry.areacode;
    cin.ignore(1, ' ');                    // Consume blank
    cin.get(theEntry.phoneNumber, 9);
    cin.ignore(100, '\n');

    // Get a friend's birth date

    cout << "Enter birth date as three integers, separated by"
         << " spaces: MM DD YYYY" << endl;
    cin >> theEntry.month >> theEntry.day >> theEntry.year;
} // GetEntry


//*****************************************************
// void Insert( Entry list[], int& length, Entry item )
```

```
// Purpose:       Inserts item into its proper place in sorted list
// Precondition:  length < MAX_FRIENDS
//                && list[0. length-1] are in ascending order
//                && item is assigned
// Postcondition: item is in list
//                && length == length@entry + 1
//                && list[0..length-1] are in ascending order
//                && IF item was already in list@entry
//                  item has been inserted before the one that was there
//*******************************************************
void Insert( Entry list[],    // List to be changed
             int&  length,    // Length of list
             Entry item  )    // Item to be inserted
{
    int index = 0;        // Position where item belongs
    int count;            // Loop control variable

    list[length] = item;  // store item at position beyond end of list

    // Exit loop when item is found, perhaps as sentinel

    while (strcmp(item.lastName, list[index].lastName) > 0)
        index++;

    // Shift list[index. length-1] down one

    for (count = length - 1; count >= index; count--)
        list [count+1] = list [count];

    list[index] = item;   // Insert item
    length++;             // Increment length of list
} // Insert


//*******************************************************
// void WriteEntries( const Entry addressBook[], int length,
//                    ofstream& friendFile )
// Purpose:       Writes all entries to the file friendFile
// Output:        (to an output file) all address book entries
// Precondition:  length (= MAX_FRIENDS
//                && addressBook[0. length-1] are assigned
// Postcondition: Contents of addressBook[0. length-1] have been
//                output to friendFile
//*******************************************************
void WriteEntries( const     Entry addressBook[], // Array of entries
                   int       length,              // Number of entries
                   ofstream& friendFile )         // File receiving list
{
    int counter;         // Loop counter

    for (counter = 0; counter < length; counter++) {
        friendFile << addressBook[counter].firstName << ' '
```

```
                    << addressBook[counter].lastName << endl;
        friendFile << '(' << addressBook[counter].areaCode << ") "
                    << addressBook[counter].phoneNumber << endl;
        friendFile << setw(2) << addressBook[counter].month << '/'
                    << setw(2) << addressBook[counter].day << '/'
                    << setw(4) << addressBook [counter].year << endl;
        friendFile << endl;
    } // for
} // WriteEntries
```