1 Getting Started

Although this course will cover programming practices in general, the focus will be on C++. Why not Java?

- C++ is more complex than Java. If you can write good programs in C++, you can transfer this skill to Java; the converse is not true.
- From 1995 until around 2010, Java usage increased and C++ usage decreased. Equality was reached some time during 2006, and now Java is used more than C++. Nevertheless, there is still a need for C++ programmers. Moreover, due to the sale of Java to Oracle, programmers increasingly seem to look for alternatives due to the uncertainties surrounding future developement of Java, which lead to renewed interest in C++ adoption since around 2010.
- Many games and most software for the telecommunications industry is written in C++. Companies that use C++ include: Adobe products; Alias/Wavefront products (e.g., Maya); Amazon; Facebook; Google; JPL/NASA; Mozilla (Firefox, Thunderbird etc.), Microsoft, and Siemens. More recently, algorithms for Deep Learning (e.g., Google's TensorFlow) or crypto-currency mining (e.g., Ethereum) have been implemented in C++. For more applications, see http://www.stroustrup.com/applications.html.
- There are many Java jobs and many Java programmers. There are not quite so many C++ jobs but there are very few *good* unemployed C++ programmers. Strong C++ programmers can find interesting and well-paid jobs.

It is a cliché to say that software is becoming ubiquitous. However, it is noteworthy that programs are getting larger:

Entity	Lines of code
Cell phone:	2×10^6
Car:	20×10^6
Telephone exchange:	100×10^6
Civil aircraft:	10^9
Military aircraft:	6×10^9

The crucial problem for all aspects of software development is ***scalability***. Approaches and techniques that do not work for millions of lines of code are not useful. C++ scales well.

1.1 Hello, world!

2 C++ is the result of a long history of developing programming languages:[1]

1965: BCPL (Martin Richards)

1968: B (Ken Thompson @ Bell Labs) ($8K \times 18b$ PDP/7)

1969: C (Ken Thompson and Dennis Ritchie) ($64K \times 12b$ PDP/11)

1979: C with classes (Bjarne Stroustrup)

1983: C++ named

1990: templates and exceptions

1992: Microsoft C++ compiler

1993: RTTI, namespaces

1994: ANSI/ISO Draft Standard

1995: Java

1998: **C++98**

2009: C++0x Draft Standard

2011: **C++11** Standard (a.k.a. C11, C1X)

2014: C++14 (minor updates to C++11)

2017(?): C++17 aka C++1z (new features)

3 C++ has strengths:

- Low-level systems programming
- High-level systems programming
- Generic programming
- Embedded code
- High performance programming
- Numeric/scientific computation
- Games programming
- General application programming

and weaknesses:

- Legacy of C
- Insecurities
- Complexity
- No standard GUI library

[1] A numbered box in the outer margin indicates that nearby text is used as a slide during a lecture.

Even the lead designer of C++, Bjarne Stroustrup, does not claim that C++ is the ideal programming language: 4

> *C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.*

The C Programming Language (Kernighan and Ritchie 1978), the "classic" text for C, appeared in 1978. Its first example program has set the standard for all successors: 5

```
main()
{
    printf("hello, world\n");
}
```

While C++ originated from C, it has evolved considerably and is now a distinct and significantly more complex programming language. A typical C++ version of the "hello, world" program is shown in Figure 1 (Weiss 2004, page 11).

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello, world" << endl;
   return 0;
}
```

Figure 1: Hello.1

A modern C++ compiler should compile both programs, because C++ is an extension of C.[2] To be more precise, it was originally the intention that C++ and C should be compatible, but that goal was abandoned when it became clear that C++ would suffer if it was followed rigorously: it is possible to write programs that are legal C but not legal C++. An unfortunate side-effect of this compatibility is that there is now a mixture of C and C++ programming styles, ranging from "classic C" style to "modern C++" style. In this course, we will emphasize the modern style.

The important features of Figure 1 are:

- `#include <iostream>` informs the compiler that the program uses components from the ***stream library*** to perform input and output. (`#include` is similar to Java's `import`.)
- `using namespace std;` a ***using directive*** that allows to use classes from the `std` namespace without a `std::` prefix (similar to the `import` directive in Java).
- `int main()` introduces a function that is usually called the ***main program***. Compilers are quite flexible about the declaration of this function. We will see later, for example, that `main` may take arguments. The initial `int` is important, however: it indicates that

[2] Stroustrup claims that "every example in *The C Programming Language* (2nd Edition) (Kernighan and Ritchie 1978) is also a C++ program." Of course, this does ***not*** imply that every program that a C compiler compiles will also be accepted by a C++ compiler.

the function will return an integer that the operating system may use to choose the next action.

- `cout` is the new version of the old "`printf`" (or `System.out.print` in Java). It is part of the `std` ***namespace*** included above. Because of the `using` directive, we do not have to mention this explicitly with `std::cout`. `cout` (pronounced "see-out") is the name of the ***standard output stream***; when a program runs, text written to `cout` appears on the console window.
- `<<` is an ***operator***, analogous to + or ×. In this case, its right operand consists of stuff to be written to standard output.
- `endl` is the new version of `\n`: it is short for "end line" and outputs a return character. (We can still use `\n`, however.)
- `return 0` sends zero back to the operating system, indicating that the program terminated successfully.

In general, we also note that:

- Statements in C++ are terminated with a semicolon ("`;`").
- Blocks of code are delimited by braces ("`{`" and "`}`").

C has a reputation for being cryptic. The following code for copying a string (Kernighan and
6 Ritchie 1978, page 101) is often quoted:

```
char *s;
char *t;
....
while (*s++ = *t++)
    ;
```

This works for several rather subtle reasons: the post-increment operator `++` has an effect and also returns a value; the assignment operator `=` returns a value; C strings are terminated by the special value '`\0`' (also known as the "null character"); and the null character, interpreted as a boolean, represents `false`. The loop terminates because a C string, by convention, is terminated with a null character ('`\0`').

This style of programming was appropriate when C was introduced. Ritchie and Thompson designed C for systems programming on minicomputers of the early 1970s. They did most of their work on a DEC PDP/11 with 64 Kb of RAM and they wanted a small and simple compiler. The `while` statement above can be compiled into efficient code without any further optimization.

By way of contrast, here is the string copy implemented in "C-for-dummies" style. Note the explicit termination test, comparing `s[i-1]` to `'\0'`. The `do-while` loop is used because the terminator must be copied to the output string.

```
int i = 0;
do
{
   s[i] = t[i];
```

```
    i = i + 1;
} while (s[i-1] != '\0');
```

An experienced C programmer would probably prefer the library call `strcpy(s, t)` to either of these versions.

From a modern perspective, however, the `while` statement is unnecessarily cryptic and potentially ***inefficient*** because it places unnecessary constraints on what the compiler can do. Accurately implementing what the programmer has written requires the use of two registers, separately incremented, and each containing a pointer. At each cycle, the code must test for a null character.

In contrast, a C++ programmer would write something like this: 7

```
std::string s;
std::string t;
....
s = t;
```

In this version, strings are instances (objects) of a standard class `string` rather than "pointers to characters". The copying operation is performed by the library function that implements assignment ("="). This function can be written to be optimal for the architecture for which the library is written: for example, it might use "block moves" rather than copying characters one at a time. It might even avoid copying altogether. The code does not depend on any particular representation for strings. For example, it does not require a null terminating character. In general, a C++ programmer will tend to work at a ***higher level of abstraction*** than a traditional C programmer.

Here is another, much more advanced, example of modern C++. Suppose we want to parse strings according to the following grammar: 8

$$
\begin{array}{lcl}
\mathit{expr} & = & \mathit{term}\ \texttt{`+'}\ \mathit{expr} \\
 & | & \mathit{term}\ \texttt{`-'}\ \mathit{expr} \\
 & | & \mathit{term} \\
\mathit{term} & = & \mathit{factor}\ \texttt{`*'}\ \mathit{term} \\
 & | & \mathit{factor}\ \texttt{`/'}\ \mathit{term} \\
 & | & \mathit{factor} \\
\mathit{factor} & = & \mathit{integer} \\
 & | & \texttt{`('}\ \mathit{expr}\ \texttt{`)'}
\end{array}
$$

Using the Boost Spirit library (Abrahams and Gurtovy 2005), we can write code corresponding to these productions as shown in Figure 2 on the next page (some additional surrounding code is needed to make everything work). Notice how close the C++ code is to the original grammar. 9

That C++ is compatible with C is both good and bad. It is good because at Bell Labs, where C++ was developed, a new language that could not compile existing code would almost certainly have not been accepted. It is bad because C is an old language with many features that would now be considered undesirable; compatibility meant that C++ had to incorporate most of these undesirable features, even though it does quite a good job of covering them up. A well-trained

```
expr =
      ( term[expr.val = _1] >> '+' >> expr[expr.val += _1] )
    | ( term[expr.val = _1] >> '-' >> expr[expr.val -= _1] )
    | term[expr.val = _1]
    ;

term =
      ( factor[term.val = _1] >> '*' >> term[term.val *= _1] )
    | ( factor[term.val = _1] >> '/' >> term[term.val /= _1] )
    | factor[term.val = _1]
    ;

factor =
      integer[factor.val = _1]
    | ( '(' >> expr[factor.val = _1] >> ')' )
    ;
```

Figure 2: Parsing with Boost/Spirit

and conscientious programmer can write secure and efficient programs in C++; an inexperienced programmer can create a real mess.

Figure 2 illustrates another aspect of modern C++. All competent C programmers are essentially equal: they all have a good understanding of the entire language and its standard libraries. With C++, however, a wide gulf separates programmers who can ***use*** the Spirit parser components — which is straightforward — from the programmers who can ***create*** the Spirit components; these programmers form a small minority of C++ programmers. (Parser generators are constructed by ***template metaprogramming***, which few C++ programmers understand, let alone use.)

1.2 Compiling C++

When choosing a C++ compiler, you have to carefully check with version of the language standard it supports. In particular, for code examples using C++11, you have to make sure that your compiler supports this standard.

1.2.1 The compilation process

Compiling a program consists of a number of steps. Errors may occur at any step, and it is important to distinguish the different kinds of error. The compiler processes the program as a number of ***compilation units***. Each compilation unit corresponds to a source code file together with any other files `#include`d with it.

syntax error

1. The compiler parses each compilation unit. This may produce ***syntax errors***.

   If we omit a semicolon in Figure 1 on page 3, writing

   ```
   cout << "Hello, world!" << endl
   ```

the compiler issues a syntax error:[3] 10

```
f:\Courses\COMP345\src\Hello\hello.cpp(32):
   error C2143: syntax error : missing ';' before '}'
```

The "32" is the number of the line on which the error is detected; in this case, it is the line containing `}`, the line ***following*** the line with the error. However, the error message itself is quite helpful: it actually tells you what is wrong with your program.

2. The compiler checks the semantic correctness of the program. For example, it checks that each function is called with arguments of an appropriate type. A program may have ***semantic errors***. semantic error

   If we omit the "`l`" from "`endl`", the compiler issues two semantic errors: 11

   ```
   f:\Courses\COMP345\src\Hello\hello.cpp(31):
      error C2065: 'end' : undeclared identifier
   f:\Courses\COMP345\src\Hello\hello.cpp(31):
      error C2593: 'operator <<' is ambiguous
   ```

   Syntax errors and semantic errors are collectively called ***compile-time errors***. compile-time error

3. The compiler generates object code for the compilation unit.

4. When each compilation unit has been controlled, the ***linker*** is invoked. The linker attempts to link all of the object code modules together to form an ***executable*** (or, occasionally, a library). This may produce ***link-time errors***. link-time error

   Suppose that we declare and use a function `f` but forget to define it. The compiler expects that `f` will be defined somewhere else and so does not generate a semantic error. We do not get an error until the linker discovers that there is no definition: 12

   ```
   Hello error LNK2019:
      unresolved external symbol "void __cdecl f(int)"
      (?f@@YAXH@Z) referenced in function _main
   ```

   The mysterious string `?f@@YAXH@Z` is the name that the compiler has given to the function `f`. The conversion from `f` to `?f@@YAXH@Z` is called ***name mangling***. name mangling

5. When the program has been linked, it can be ***executed*** or, more simply, ***run***. It may run correctly or it may generate ***run-time errors***. run-time error

   If you write `cout` twice by mistake, the compiler accepts the program without any fuss:

   ```
   cout << cout << "Hello, world!" << endl;
   ```

   However, when the program runs, it displays something like: 13

   ```
   0045768CHello, world!
   ```

   Output like this can be disconcerting for beginners: the program has displayed the address of the object `cout` in hexadecimal!

[3] The error messages were produced by VC++. Other compilers produce similar, but not identical, diagnostics.

Although you don't really want any errors at all, you should prefer compile-time and link-time errors to run-time errors, if only because your customers will never see them. Fortunately, C++ compilers perform thorough syntactic and semantic checking (much better than C compilers)
14 and will catch many of your errors.

> ***An event that occurs during compilation is called static. An event that occurs during execution is called dynamic.***

1.2.2 The Preprocessor

C++ inherited from C a particular way of preparing source code for compilation: Before being compiled, source code is transformed by a *preprocessor* that accepts special commands for modifying the source code, like including other code parts, executing macros, or omitting parts
#include of your code (using conditional statements). The `#include` statement is such a directive to the
15 preprocessor.

For now, you do not have to worry about the preprocessor, but simply be aware that the code "seen" by the compiler might be rather different from what you see in your editor/IDE. Depending on the development environment used, you can look at the output of the preprocessor (the actual input to the compiler) using a special command, e.g., for GCC you can type

```
g++ -E hello.cpp > hello.out
```

to obtain the results of preprocessing step in the file `hello.out`. For the simple "Hello, world" program in Figure 1 on page 3, you might be surprised to see that the file produced for the compiler contains now almost 30 000 lines of code!

1.2.3 Compiling in practice

16 There are a number of platforms available for C++ development.

gcc The Gnu C Compiler is a command-line compiler, not an IDE (see http://gcc.gnu.org/). It is available for several platforms; for Windows, use the MinGW toolchain (see http://www.mingw.org/). For simple programs with all of the source code in one file, just use "`gcc`" (defaults to C) or "`g++`" (defaults to C++) as a command. For more complex programs, it's best to write a `Makefile` or use an IDE.

Code::Blocks is a free (open-source) cross-platform (Linux, Mac, Windows) IDE for C++ that can be used with various compilers (see http://www.codeblocks.org/). One of the binary download packages for Windows includes MinGW, which is the GNU compiler for Windows.

Code::Blocks is a more friendly and easy-to-use IDE than most of the alternatives listed here. Its main disadvantage is libraries: it comes with the standard C++ libraries but, if you want to use any other libraries, you will have to compile them and install them yourself.

Eclipse CDT The CDT (C/C++ Development Tools) Project (see http://www.eclipse.org/cdt/) provides a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform. This is particularly interesting for those who are already familiar with the Eclipse platform (like Java developers). For Linux users, the *Linux Tools Project* (http://www.eclipse.org/linuxtools/) combines the CDT with a number of other useful tools, like Autotools build integration and the Valgrind memory analysis tool.

NetBeans Like Eclipse, NetBeans (see http://netbeans.org/features/cpp/index.html) also fully supports C++ development. It is available for Windows, Linux, MacOS, and Solaris.

Qt Creator Now maintained by Qt after being de-merged from Digia, it was originally developed by TrollTech, which was later bought by Nokia. It is a full-featured IDE for the cross-platform *Qt* framework for C++ development (see http://qt-project.org).

Visual C++ (VC++) is a popular platform for developing C++ programs. It has some disadvantages:

- Early versions did not support modern C++. In particular, *use VC++ 7.1 or later* for this course; earlier versions cannot handle some of the C++ features that we will be using.
- VC++ tries very hard to wrap your programs in MS junk. It is possible, but difficult, to produce the "vanilla", standard C++ programs that we will use in this course.
- Unlike all the other IDEs mentioned above, it is not free/open source software.

1.3 Hello, world! – The Details

Here is the first C++ example program again, this time without the `using` directive: 17

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

"`#include`" is a ***compiler directive***. Strictly speaking, it is not part of the program, but instead is an instruction telling the compiler what to do (see Section 1.2.2 for details). In this case, the directive tells the compiler to include the stream part of the ***standard library***, so that it becomes accessible to the program.

Streams are one of the media that C++ programs use to communicate. The communication is usually external — data is read to or from a device such as a keyboard, screen disk, or network — but may also be internal to the program (that is, to or from memory). A stream is not the same as a file, although a file may be read or written with a stream. streams

An ***output stream*** typically allows the program to send data ***to*** an external device, such as a window. An ***input stream*** typically allows the program to receive data ***from*** an external

device, such as a keyboard. The library component `iostream` provides both: "`i`" for input and "`o`" for output.

The program itself consists of a single ***function definition***. The function's name is `main` and its ***return type*** is `int`. The word `int` is a ***keyword*** of C++, meaning that we cannot use it for anything other than its intended application. In fact, `int` is a ***type*** – the type of ***signed integers***. A value i of type `int` is typically represented using 32 bits and satisfies $-2^{31} \leq i < 2^{31}$. Many modern C++ systems provide 64-bit `int`s.

The ***body*** of the function consists of a sequence of ***statements*** (two in this example) enclosed in braces (`{ ... }`). The last character of each statement is a semi-colon (`;`).

The return type of the function (`int`) is related to its last statement: `return 0`. The program behaves ***as if*** the function `main` is invoked by the operating system, executes its two statements, and returns the integer 0 as its result. The operating system interprets 0 as "normal termination". We can return other values if we want to tell the operating system that something went wrong.

The only thing left to discuss is the line beginning with `std::cout`. This illustrates an important general principle of C++. The text

```
std::cout << "Hello, world!" << std::endl
```

is an ***expression*** that yields a value. By putting a semicolon at the end of this expression, we discard the value and turn the expression into a ***statement***.

Expressions consist of ***operands*** and ***operators***. For example, $x + 5$ is an expression with two operands (x and 5) and one operator, +. We say that + is a ***binary operator*** because it takes two operands: x is its ***left operand*** and 5 is its ***right operand***. An expression can be ***evaluated*** to yield a ***result***. If x has the value 2, then evaluating $x + 5$ yields 7.

In C++, `<<` is a binary operator, usually called "insert". Its left operand is an output stream; its right operand may be a ***string***, a ***manipulator***, and various other things. Its result is a stream. When an expression contains more than one insertion operator, they are evaluated from left to right. The first step in evaluating

```
std::cout << "Hello, world!" << std::endl
```

is to evaluate

```
std::cout << "Hello, world!"
```

This has the effect of appending the string `"Hello, world!"` to the standard output stream and yields the updated output stream, say `uos`. The next step is to evaluate

```
uos << std::endl
```

which has the effect of appending a new line to the standard output stream (and a bit more, discussed below) and yields the updated output stream. In the program, this expression is followed by a semicolon, which throws away the final value. `cout` is a persistent object, however, and it still exists in its updated state.

There is a small, but significant, difference between the statements

```
std::cout << "Hello, world!" << std::endl;
```

and

```
std::cout << "Hello, world!" << "\n";
```

which could also be written as

```
std::cout << "Hello, world!\n";
```

The difference is that `std::endl` writes the `\n` and also ***flushes the output buffer***. What does this mean?

It would be inefficient for a program to go through all of the operations of transferring data for every character in a stream. To save time, the program stores characters in a temporary area called a ***buffer*** and performs the actual transfers only when the buffer is full. If the program is writing data to a file, this behaviour is undetectable to the user. But, if the program is writing to a window, buffering makes a lot of difference. If we use `\n`, a significant amount of output may be generated by the program before we actually see it displayed. Using `std::endl` ensures that each line of output will be displayed as soon as it has been computed.

Here is yet another way of writing "Hello, world!" followed by end-of-line:

```
std::cout << "Hello, world!"
             "\n";
```

When C++ sees a quote (`"`) at the end of a string, followed by white space (blanks, tabs, and line breaks), followed by a quote at the beginning of a string, it erases quotes and the white space and treats the result as a single string. This is useful for writing long strings or strings that run over several lines.

1.4 Namespaces

As we noted above, we write `std::cout` to tell the compiler that the name `cout` comes from the namespace `std`. The name `cout` is a ***simple name*** (or just a ***name***), `std::cout` is a ***qualified name***, and `std::` is a ***qualifier***. "`::`" is called the ***scope operator***.

We can use this convention but, as programs get longer and more complex, the frequent appearance of "`std::`" becomes irritating, as well as being tedious to type. An alternative is to say to the compiler "When I write `cout`, I mean `std::cout`". This is the purpose of the `using` directive, as shown in Figure 3 on the next page. In general, we write one `using` directive at the beginning of the source file for each name that we intend to use. 18

We can go further, saying to the compiler "When I write any name that belongs to `std`, take it from there". The form of the `using` directive that does this is shown in Figure 4. Note that, in both Figure 3 on the following page and Figure 4 on the next page, we use `cout` and `endl` without the qualifier `std::`. 19

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 3: Hello.2

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 4: Hello.3

Some people will tell you that the point of namespaces is to make the source of names explicit; they will tell you that version on page 9 is the best way to write programs, Figure 3 is acceptable, and Figure 4 is bad – the kind of code produced by lazy programmers who should be fired. This attitude is ***wrong***. The actual situation is more complex.

Namespaces were introduced into C++ to prevent name clashes in programs that use more than one library. Suppose that you are writing a program that uses a library `Cowboy` and another library `Artist`. Both libraries provide a function called `draw`. When you try to use `draw`, the compiler complains that it cannot tell which library to take it from. To avoid this problem, the libraries wrap their names in namespaces – perhaps `namespace cowboy` and `namespace artist`. The programmer can then write `cowboy::draw` or `artist::draw`, depending on which function is needed.

Most of the time, however, name clashes do ***not*** occur. In particular, it is very unlikely that a library writer would use a well-known name such as `cout`. Consequently, it is quite acceptable to use the form of Figure 4, while being aware that it may one day be necessary to use explicit qualification when a name clash actually occurs.

1.5 Strings

20

Figure 5 on the next page (Koenig and Moo 2000, page 9) shows a program that asks for the
21 user's name and then greets the user. Running it produces a dialogue like this:

```
What is your name? Nebuchadnezzar
Hi, Nebuchadnezzar!
```

Much of this program should already be familiar because it is similar to `Hello, world!`. The new features are the class `string` and the input stream `cin`.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "What is your name? ";
    string name;
    cin >> name;
    cout << "Hi, " << name << "!" << endl;
    return 0;
}
```

Figure 5: Greeting.1

The directive `#include <string>` informs the compiler that we will be using class `string`. Since the names provided by `string` are included in the namespace `std`, we can write simply "`string`" in the program rather than "`std::string`".

The second statement of the main program,

```
string name;
```

is both a ***declaration*** and a ***definition***. (We will discuss the distinction between declarations and definitions in detail later. For now, note that all definitions are also declarations, but a declaration is not necessarily a definition.) It introduces a new variable, `name`, into the program. The type of `name` is `string`. We could also say that `name` is a new ***object***, and an ***instance*** of class `string`.

The value of a `string` object is a string of characters, such as `"Hello, world!"`. After the declaration, the value of `name` is actually the ***empty string***, written `""`.

There are various ways of putting characters into a string. In this program, the statement

```
cin >> name;
```

reads data from the ***standard input stream***, `cin` (pronounced "see-in"). By default, `cin` gets data from the keyboard. In other words, whatever text you enter in response to the prompt `"What is your name?"` gets stored in the variable `name`.

Actually, this description is not quite accurate: `cin` reads only until it encounters white space (a blank, tab, or ENTER) and then stops. This explains the following dialogue:

```
What is your name? King Kong
Hi, King!
```

After the program has stored a value in `name`, it can use this value, as in the second `cout` statement.

There is a subtlety in Figure 5 on the preceding page that is worth noting. On the basis of the discussion at the end of Section 1.3 on page 9, you would be right to wonder why the statement

```
cout << "What is your name? ";
```

produces any output at all. Why is the data not left in the buffer until `endl` is output in the second `cout` statement? The answer is that the streams `cin` and `cout` are ***linked***. Any use of `cin` causes the buffer for `cout` to be flushed. This ensures that programs that implement a dialog in which the user must respond to displayed text will work as expected.

Memory management is an important aspect of C++ programming. In Figure 5 on the preceding page, memory for the string `name` is allocated on the run-time stack when the definition `string name` is evaluated. The class `string` manages memory when characters are put into the string (e.g., by `cin`) or the string is changed in other ways. At the final closing brace ("`}`") of the program, any memory used by `name` is de-allocated.

A definition, of a variable or other named entity, occurs within a ***scope***. The scope consists either of the closest enclosing braces or, if there are no enclosing braces, the entire source file. In the latter case, we say that the definition is "at file scope". Uses of the name must follow its definition. In other words, we cannot refer to a variable earlier in the program text than its definition: this is called the "definition before use rule". The name ceases to be accessible at the end of its scope.

> ***The definition of a name must textually precede its use.***

In many cases, there are actions connected with the definition and the end of the scope. In Figure 5 on the previous page, which is typical, the object `name` is ***constructed*** at the point of its definition and ***destroyed*** at the end of the program. What actually happens is this: when an object definition is processed, a ***constructor*** for the object is called and, at the end of a scope, the ***destructors*** for all stack-allocated objects are called.

constructor (ctor)
destructor (dtor)

Some C++ programmers (and books) use the abbreviations "ctor" and "dtor" for constructor and destructor, respectively.

1.6 A Pretty Frame

22 The greeting produced by Figure 5 on the preceding page is dull and boring. The next version, in Figure 6 on the next page (Koenig and Moo 2000, page 12), produces dialogues like this:

```
Please enter your first name: Ferdinand

**********************
*                    *
* Hi there, Ferdinand! *
*                    *
**********************
```

23
24

The new feature in this program is `const string`. There are four definitions of the form

`const string` ⟨*name*⟩ `....` ;

The keyword `const` tells the compiler that the values of the names are not going to change. We cannot declare `name` as `const` because its initial value (`""`) gets changed when we use `cin` to copy characters into it.

When we introduce a `const` name, we must provide a value for it in the same declaration. The value can be a simple value, as in this example:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
cout << "Please enter your first name: ";
string name;
cin >> name;

const string greeting = "Hi there, " + name + "!";
const string spaces(greeting.size(), ' ');
const string second = "* " + spaces + " *";
const string first(second.size(), '*');

cout << endl;
cout << first << endl;
cout << second << endl;
cout << "* " << greeting << " *" << endl;
cout << second << endl;
cout << first << endl;

return 0;
}
```

Figure 6: Greeting.2

```
const string WELCOME = "Welcome to COMP 345!";
```

Figure 6 on the preceding page shows several other ways of providing an initial value using features of class `string`:

- `greeting = "Hi there, " + name + "!"`

  The operator `+` concatenates strings (i.e., joins them together). After `name` has received the value `"Ferdinand"`, the definition gives `greeting` the value `"Hi there, Ferdinand!"`.

  Although `greeting` is `const` and `name` is not `const`, we can use `name` as part of the value of `greeting`. The `const` qualifier says that `greeting` will not be changed later; it does not say that the value is known at compile-time.

- `spaces(greeting.size(), ' ')`

  The expression `greeting.size()` makes use of a ***member function*** of class `string`. Member functions are called with the "dot notation":

  ⟨*object name*⟩ `.` ⟨*function name*⟩ `(` ⟨*parameters*⟩ `)`

  In this case, the function name is `size` and it returns the size of (i.e., the number of characters in) the object `greeting`. Thus `greeting.size()` is a ***number*** (in this case, it is 20).

  Class `string` has a constructor which expects two arguments: a numeric value (type `int`) and a character value (type `char`). By writing `spaces(n, ' ')` we are saying: construct an instance of class `string` with name `spaces` that contains `n` characters where each character is `' '` (that is, a blank).

  Thus the effect of this particular definition is to define a constant `string` object `spaces` with value `"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"`. (␣ denotes a blank.)

- `second = "* " + spaces + " *"`

  The definition of `second` uses the definition of `spaces` and the concatenation operator, `+`.

- `first(second.size(), '*')`

  The definition of `first` is similar to that of `spaces`. It has the same number of characters as `second`, but each character is an asterisk, `'*'`. Note that C++ distinguishes between ***strings***, which have zero or more characters between double quotes (`"...."`) and ***characters*** (instances of type `char`), which have exactly one character between single quotes (`'.'`).

Having defined the strings `greeting`, `spaces`, `second`, and `first`, the program uses them to generate the desired output.

Although this program is trivial, it illustrates two important points about programming in general:

> ***Avoid unnecessary assumptions.***

The program is not written for people with five-letter names but for people with names of any (reasonable!) length.

> ***Make the program do the work.***

The strings needed for the display are ***computed*** (as much as possible). This makes it easy to generate a display whose size matches the name of the user.

It is not necessary to call `cout` once for each line of output. The six calls of `cout` in Figure 6 on page 15 could be replaced by a single call: 25

```
cout <<
   endl <<
   first << endl <<
   second << endl <<
   "* " << greeting << " *" << endl <<
   second << endl <<
   first << endl;
```

References

Abrahams, D. and A. Gurtovy (2005). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley.

Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language*. Prentice-Hall.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.