

**Chapter
1**

**GETTING
STARTED**

*Get on the
Fast Track!*



**SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.**

Objectives

You will learn:

- C Terminology.
- C Features and benefits of C++.
- C Definition of class.



1 About C++

The C++ language is a general-purpose programming language known for its efficiency, economy, and portability. While these characteristics make it a good choice for almost any kind of programming, C++ has proven especially useful in systems programming because it facilitates writing fast, compact programs that are readily adaptable to other systems.

Well-written C++ programs are often as fast as assembly-language programs, and they are typically easier for programmers to read and maintain.

C++ is a flexible language. In keeping with this philosophy, C++ imposes few restrictions in matters such as type conversion.

Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave.

1.1 Using C++ as a Better C

Using C++ as a better C is an effective option because, unlike many other programming languages that claim to support object orientation, C++ does not impose object orientation on the programmer.

When using a C++ compiler, the programmer can choose to embrace:

- C a procedural style of programming.
- C an object-oriented style.
- C or something in between.

If there are a selection of C++ class libraries available, several more programming styles, in addition to procedural and object oriented, will be available.

1.2 C++ Features

The important features of C++ include:

- C More type checking at compile time and more type checking at link time.
- C Default function argument values.
- C Inline functions.
- C A type-sensitive replacement for the functions `free` and `malloc`.
- C A type-sensitive replacement for C style I/O.
- C Type-sensitive constant values and Type-sensitive inline functions to replace `#define`.
- C Generic template functions.
- C Operator and function overloading.
- C References (which are the first cousins of pointers).
- C Exception handling.
- C The ANSI standard classes.

1.3 C++ Benefits

A programmer can elect to use the features that he/she likes and understands.

For example, many experienced C++ programmers do not like the new C++ style I/O and prefer to use the `printf` and `scanf` style functions.

Programmers can do this because of the backward compatibility of C++ to C.

Another benefit of C++ is type checking at link time. This benefit has minimal cost to the programmer because it happens automatically.

2 Source Files

A source program can be divided into one or more “source files,” or “translation units.” The input to the compiler is called a “translation unit.”

The components of a translation unit are external declarations that include function definitions and identifier declarations. These declarations and definitions can be in source files, header files, libraries, and other files the program needs. You must compile each translation unit and link the resulting object files to make a program.

2.1 Contents of Source File

A C++ “source program” is a collection of directives, pragmas, declarations, definitions, statement blocks, and functions. To be valid components of a C++ program, each must have the proper syntax. The location of these components in a program does affect how variables and functions can be used in a program.

Source files need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and update when necessary. For the same reason, constants and macros are often organized into separate files called “include files” or “header files” that can be referenced in source files as required.

2.2 Directives to the Preprocessor

A “directive” instructs the C preprocessor to perform a specific action on the text of the program before compilation.

This example uses the preprocessor directive #define:

```
#define MAX 100
```

This statement tells the compiler to replace each occurrence of MAX by 100 before compilation. The C++ compiler preprocessor directives are:

| | | | |
|---------|--------|----------|---------|
| #define | #endif | #ifdef | #line |
| #elif | #error | #ifndef | #pragma |
| #else | #if | #include | #undef |

2.3 Function Declarations and Definitions

Function prototypes establish the name of the function, its return type, and the type and number of its formal parameters. A function definition includes the function body.

Both function and variable declarations can appear inside or outside a function definition.

- C Any declaration within a function definition is said to appear at the “internal” or “local” level.
- C A declaration outside all function definitions is said to appear at the “external,” “global,” or “file scope” level.

2.4 Blocks

A sequence of declarations, definitions, and statements enclosed within curly braces ({}) is called a “block.” There are two types of blocks in C++:

- C The “compound statement,” a statement composed of one or more statements.
- C The “function definition,” consists of a compound statement (the body of the function) plus the function’s associated “header” (the function name, return type, and formal parameters).

A block within other blocks is said to be “nested.”

While all compound statements are enclosed within curly braces, not everything enclosed within curly braces constitutes a compound statement.

For example, although the specifications of array, structure, or enumeration elements can appear within curly braces, they are not compound statements.

3 Preprocessor

The preprocessor is a text processor that manipulates the text of a source file as part of the first phase of translation. The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

You can obtain a listing of your source code after preprocessing by using the appropriate compiler options.

For Microsoft compiler this option is `/E` or `/EP`. Both options cause the preprocessor to be invoked and the resulting text to be output to the standard output device, which, in most cases, is the console. The difference between the two options is that `/E` includes `#line` directives and `/EP` strips these directives out.

3.1 Phases of Translation

C++ programs consist of one or more source files, each of which contains some of the text of the program. A source file, together with its include files (files that are included using the `#include` preprocessor directive) but not including sections of code removed by conditional-compilation directives such as `#if`, is called a “translation unit.”

Source files can be translated at different times — in fact, it is common to translate only out-of-date files. The translated translation units can be kept either in separate object files or in object-code libraries. These separate translation units are then linked to form an executable program or a dynamic-link library (DLL).

The following list describes the phases in which the compiler translates files:

| | |
|-----------------------|---|
| Character mapping | Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase. |
| Line splicing | All lines ending in a backslash (\) and immediately followed by a newline character are joined with the next line in the source file, forming logical lines from the physical lines. Unless it is empty, a source file must end in a newline character that is not preceded by a backslash. |
| Tokenization | The source file is broken into preprocessing tokens and white-space characters. Comments in the source file are replaced with one space character each. Newline characters are retained. |
| Preprocessing | Preprocessing directives are executed and macros are expanded into the source file. The #include statement invokes translation starting with the preceding three translation steps on any included text. |
| Character-set mapping | All source-character-set members and escape sequences are converted to their equivalents in the execution-character set. |
| String concatenation | All adjacent string and wide-string literals are concatenated. For example, "String " "concatenation" becomes "String concatenation". |
| Translation | All tokens are analyzed syntactically and semantically; these tokens are converted into object code. |
| Linkage | All external references are resolved to create an executable program or a dynamic-link library. |

The compiler issues warnings or errors during phases of translation in which it encounters syntax errors.

4 Linking

The linker resolves all external references and creates an executable program or DLL by combining one or more separately processed translation units along with standard libraries.

Linking is necessary because you may not want to compile all of your program at the same time. For instance, you may be writing a large program, with parts of it working and debugged, while other parts are under development. Since it may take a fairly long time to compile a large program, (from seconds to many minutes, depending upon the program size), you may want to recompile each time only those parts under development.

Many compiled languages, with C++ as the primary example, come with library routines that can be added to your program. It is necessary to have a way to link together previously compiled library and program files with recently compiled files to produce a complete program. It is for this reason that the compiler generates an intermediate type of file called an object file.

The linker combines all the necessary object files, both library and user-written, to produce a final executable program.

5 Program Stages

| Program Stage | Description |
|---------------|---|
| editor | Input from terminal. Output is a source code file containing C source code and possibly preprocessor commands. |
| preprocessor | Input is source code file. Output is source code file with macros expanded and other files included as indicated by preprocessor commands. |
| compiler | Input is source code file with macros expanded. Output is object language code. |
| assembler | Input is assembly language source code file. Output is relocatable object code. |
| linker | Input is relocatable object code modules from your programs and from C library. Output is executable code. |

In most systems the preprocessor, assembler, and linker are invoked automatically by a single command.

5.1 Compile Process

