Contents

Foreword

The creation of this book has provided me, Mike McGrath, a welcome opportunity to update my previous books on C++ programming with the latest techniques. All examples I have given in this book demonstrate C++ features supported by current compilers on both Windows and Linux operating systems, and in the Microsoft Visual Studio development suite, and the book's screenshots illustrate the actual results produced by compiling and executing the listed code.

Conventions in this book

In order to clarify the code listed in the steps given in each example I have adopted certain colorization conventions. Components of the C++ language itself are colored blue, numeric and string values are red, programmer-specified names are black, and comments are green, like this:

```
// Store then output a text string value.
string myMessage = "Hello from C++!" ;
cout << myMessage ;
```

Additionally, in order to identify each source code file described in the steps a colored icon and file name appears in the margin alongside the steps:

main.cpp

header.h

Grabbing the source code

For convenience I have placed source code files from the examples featured in this book into a single ZIP archive, providing versions for Windows and Linux platforms plus the Microsoft Visual Studio IDE. You can obtain the complete archive by following these easy steps:

1. Browse to **http://www.ineasysteps.com** then navigate to the "Resource Center" and choose the "Downloads" section

2. Find "C++ Programming in easy steps, 4th Edition" in the "Source Code" list, then click on the hyperlink entitled "All Code Examples" to download the archive

3. Now extract the archive contents to any convenient location on your computer

I sincerely hope you enjoy discovering the powerful expressive possibilities of C++ Programming and have as much fun with it as I did in writing this book.

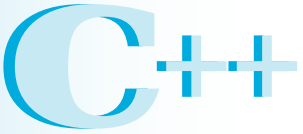
Mike McGrath

1 Getting started

Welcome to the exciting world of C++ programming. This chapter demonstrates how to create a simple C++ program and how to store data within a program.

Introducing C++

C++ is an extension of the C programming language that was first implemented on the UNIX operating system by Dennis Ritchie way back in 1972. C is a flexible programming language that remains popular today and is used on a large number of platforms for everything from microcontrollers to the most advanced scientific systems.



A powerful programming language (pronounced "see plus plus") designed to let you express ideas.

C++ was developed by Dr. Bjarne Stroustrup between 1983-1985 while working at AT&T Bell Labs in New Jersey. He added features to the original C language to produce what he called "C with classes". These classes define programming objects with specific features that transform the procedural nature of C into the object-oriented programming language of C++.

The C programming language was so named as it succeeded an earlier programming language named "B" that had been introduced around 1970. The name "C++" displays some programmers' humor because the programming ++ increment operator denotes that C++ is an extension of the C language.

C++, like C, is not platform-dependent so programs can be created on any operating system. Most illustrations in this book depict output on the Windows operating system purely because it is the most widely used desktop platform. The examples can also be created on other platforms such as Linux or MacOS.

Why learn C++ programming?

Microsoft's free Visual C++ Express IDE is used in this book to demonstrate visual programming.

The C++ language is favored by many professional programmers because it allows them to create fast, compact programs that are robust and portable.

Using a modern C++ Integrated Development Environment (IDE), such as Microsoft's Visual C++ Express Edition, the programmer can quickly create complex applications. But to use these tools to greatest effect the programmer must first learn quite a bit about the C++ language itself.

This book is an introduction to programming with C++, giving examples of program code and its output to demonstrate the basics of this powerful language.

Should I learn C first?

Opinion is divided on the question of whether it is an advantage to be familiar with C programming before moving on to C++. It would seem logical to learn the original language first in order to understand the larger extended language more readily. However, C++ is not simply a larger version of C as the approach to object-oriented programming with C++ is markedly different to the procedural nature of C. It is, therefore, arguably better to learn C++ without previous knowledge of C to avoid confusion.

This book makes no assumption that the reader has previous knowledge of any programming language so it is suitable for the beginner to programming in C++, whether they know C or not.

If you do feel that you would benefit from learning to program in C, before moving on to C++, we recommend you try the examples in "C Programming in easy steps" before reading this book.

Standardization of C++

As the C++ programming language gained in popularity it was adopted by many programmers around the world as their programming language of choice. Some of these programmers began to add their own extensions to the language so it became necessary to agree upon a precise version of C++ that could be commonly shared internationally by all programmers.

A standard version of C++ was defined by a joint committee of the American National Standards Institute (ANSI) and the Industry Organization for Standardization (ISO). This version is sometimes known as ANSI C++ and is portable to any platform and to any development environment.

The examples given in this book conform to ANSI C++. Example programs run in a console window, such as the Command Prompt window on Windows systems or a shell terminal window on Linux systems, to demonstrate the mechanics of the C++ language itself. An example in the final chapter illustrates how code generated automatically by a visual development tool on the Windows platform can, once you're familiar with the C++ language, be edited to create a graphical windowed application.

"ISO" is not an acronym but is derived from the Greek word "isos" meaning "equal" – as in "isometric".

Installing a compiler

C++ programs are initially created as plain text files, saved with the file extension of ".cpp". These can be written in any text editor, such as Windows' Notepad application or the Vi editor on Linux.

In order to execute a C++ program it must first be "compiled" into byte code that can be understood by the computer. A C++ compiler reads the text version of the program and translates it into a second file – in machine-readable executable format.

Should the text program contain any syntax errors these will be reported by the compiler and the executable file will not be built.

If you are using the Windows platform and have a C++ Integrated Development Environment (IDE) installed then you will already have a C++ compiler available as the compiler is an integral part of the visual IDE. The excellent free Microsoft Visual C++ Express IDE provides an editor window where the program code can be written, and buttons to compile and execute the program. Visual IDEs can, however, seem unwieldy when starting out with C++ because they always create a large number of "project" files that are used by advanced programs.

The popular GNU C++ compiler is available free under the terms of the General Public License (GPL). It is included with most distributions of the Linux operating system. The GNU C++ compiler is also available for Windows platforms and is used to compile examples throughout this book.

The terms and conditions of the General Public License can be found online at **www.gnu.org/copyleft/gpl.html**

To discover if you already have the GNU C++ compiler on your system type **c++ -v** at a command prompt then hit Return. If it's available the compiler will respond with version information. If you are using the Linux platform and the GNU C++ compiler is not available on your computer install it from the distribution disc, or online, or ask your system administrator to install it.

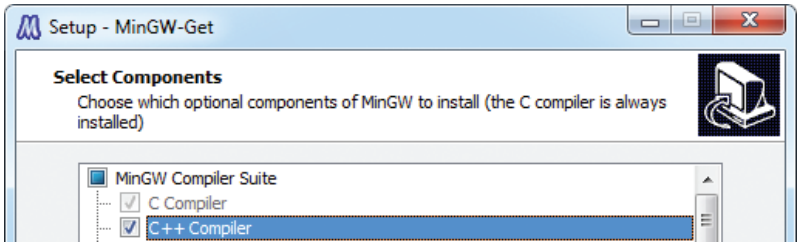
The GNU (pronounced "guh-new") Project was launched back in 1984 to develop a complete free Unix-like operating system. Part of GNU is "Minimalist GNU for Windows" (MinGW). MinGW includes the GNU C++ compiler that can be used on Windows systems to create executable C++ programs. Windows users can download and install the GNU C++ compiler by following the instructions on the opposite page.

...cont'd

1. With an internet connection open, launch a web browser then navigate to **http://sourceforge.net/projects/mingw** and click the "Download" button to get the MinGW installer

2. Launch the installer and do accept the suggested location of **C:\MinGW** in the "Select Destination Location" dialog

3. Be sure to choose the optional "C++ Compiler" item in the "Select Components" dialog then complete installation

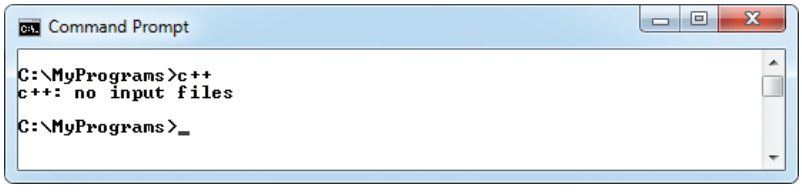


The MinGW C++ Compiler is a binary executable file located at **C:\MinGW\bin**. To allow it to be accessible from any system location this folder should now be added to the System Path:

4. In Windows' Control Panel, click the System icon then select the Advanced System Settings item to launch the "System Properties" dialog

5. In the System Properties dialog, click the Environment Variables button, select the Path system variable, then click the Edit button and add the location **C:\MinGW\bin;**

6. Click OK to close each dialog then open a Command Prompt window and enter the command **c++**. If the installation is successful the compiler should respond that you have not specified any input files for compilation:

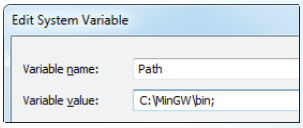


Don't forget

The MinGW installation process may be subject to change, but current guidance can be found at **www.mingw.org/wiki/Getting_Started**.

Beware

Location addresses in the Path statement must end with a ; semi-colon.



Writing your first program

Follow these steps, copying the code exactly as it is listed, to create a simple C++ program that will output the traditional first program greeting:

hello.cpp

1. Open a plain text editor, such as Windows' Notepad, then type these "preprocessor directives"

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

2. A few lines below the preprocessor directives, add a "comment" describing the program

```
// A C++ Program to output a greeting.
```

3. Below the comment, add a "main function" declaration to contain the program statements

```
int main()
{

}
```

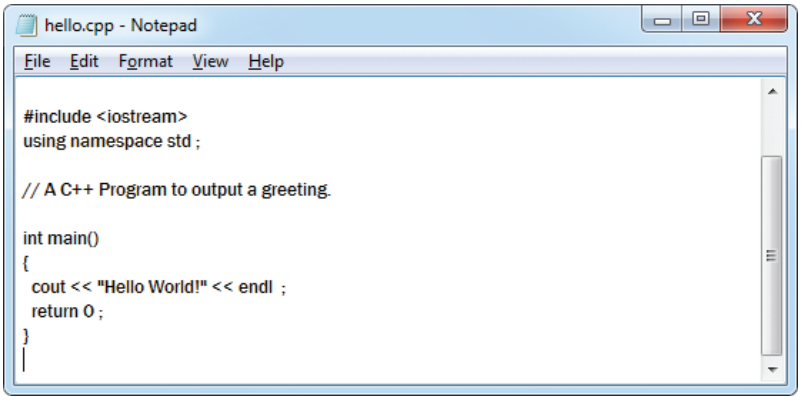
4. Between the curly brackets (braces) of the main function, insert this output "statement"

```
cout << "Hello World!" << endl ;
```

5. Next insert a final "return" statement in the main function

```
return 0 ;
```

6. Save the program to any convenient location as "hello.cpp"- the complete program should look like this:



Comments throughout this book are shown in green – to differentiate them from other code.

After typing the final closing } brace of the main method always hit Return to add a newline character – your compiler may insist that a source file should end with a newline character.

The separate parts of the program code on the opposite page can be examined individually to understand each part more clearly:

- **Preprocessor Directives** – these are processed by the compiler before the program code so must always appear at the start of the page. Here the **#include** instructs the compiler to use the standard C++ input/output library named **iostream**, specifying the library name between **< >** angled brackets. The next line is the "using directive" that allows functions in the specified namespace to be used without their namespace prefix. Functions of the **iostream** library are within the **std** namespace – so this **using** directive allows functions such as **std::cout** and **std::endl** to be simply written as **cout** and **endl**.

- **Comments** – these should be used to make the code more easily understood by others, and by yourself when revisiting the code later. In C++ programming everything on a single line after a **//** double-slash is ignored by the compiler.

- **Main function** – this is the mandatory entry point of every C++ program. Programs may contain many functions but they must always contain one named **main**, otherwise the compiler will not compile the program. Optionally the parentheses after the function name may specify a comma-separated list of "argument" values to be used by that function. Following execution the function must return a value to the operating system of the data type specified in its declaration – in this case an **int** (integer) value.

- **Statements** – these are the actions that the program will execute when it runs. Each statement must be terminated by a semi-colon, in the same way that English language sentences must be terminated by a full stop period. Here the first statement calls upon the **cout** library function to output text and an **endl** carriage return. These are directed to standard output by the **<<** output stream operator. Notice that text strings in C++ must always be enclosed within double quotes. The final statement employs the C++ **return** keyword to return a zero integer value to the operating system – as required by the main function declaration. Traditionally returning a zero value indicates that the program executed successfully.

The C++ compiler also supports multiple-line C-style comments between /* and */ – but these should only ever be used in C++ programming to "comment-out" sections of code when debugging.

Notice how the program code is formatted using spacing and indentation (collectively known as whitespace) to improve readability. All whitespace is ignored by the C++ compiler.

Compiling & running programs

The C++ source code files for the examples in this book are stored in a directory created expressly for that purpose. The directory is named "MyPrograms" – its absolute address on a Windows system is **C:\MyPrograms** and on Linux it's **/home/*user*/MyPrograms**. You can recreate this directory to store programs awaiting compilation:

Hot tip

You can see the compiler version number with the command **c++ --version** and display all its options with **c++ --help**.

1. Move the "hello.cpp" program source code file, created on page 12, to the "MyPrograms" directory on your system

2. At a command prompt, use the "cd" command to navigate to the "MyPrograms" directory

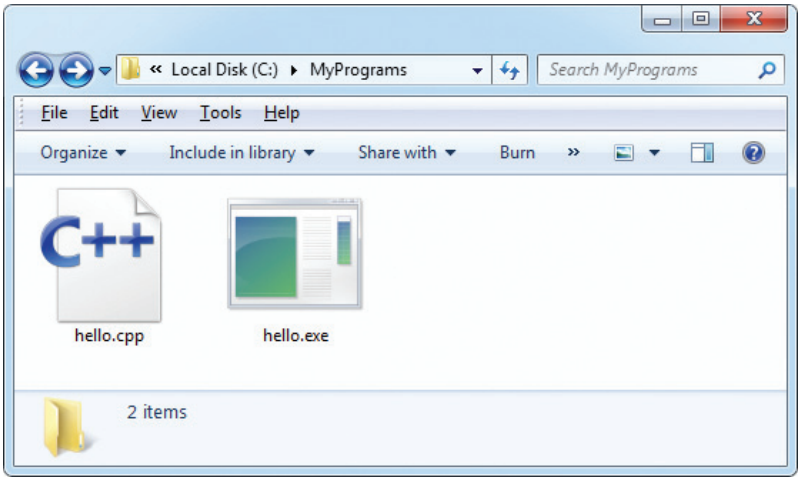
3. Enter a command to attempt to compile the program
   c++ hello.cpp

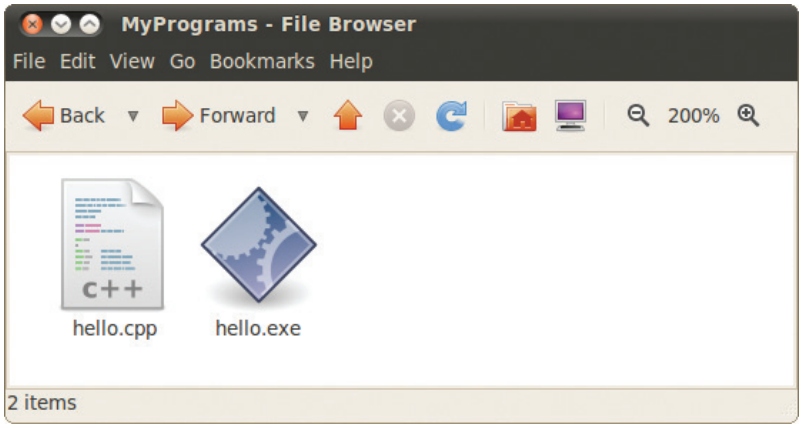
When the attempt succeeds the compiler creates an executable file alongside the original source code file. By default the executable file is named **a.exe** on Windows systems and **a.out** on Linux. Compiling a different source code file in the same directory would now overwrite the first executable file without warning. This is obviously undesirable so a custom name for the executable file should be specified when compiling programs, using the compiler's **-o** option in the compile command.

4. Enter a command to compile the program, creating an executable file named "hello.exe" alongside the source file
   c++ hello.cpp -o hello.exe

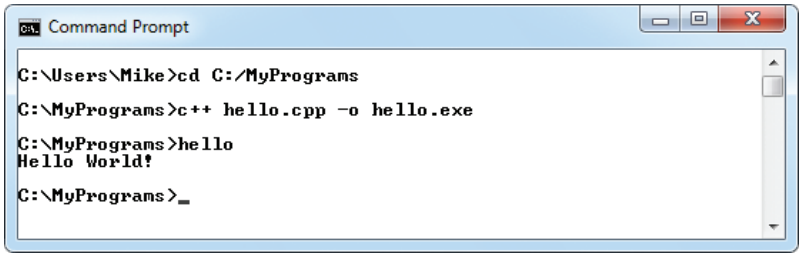
Don't forget

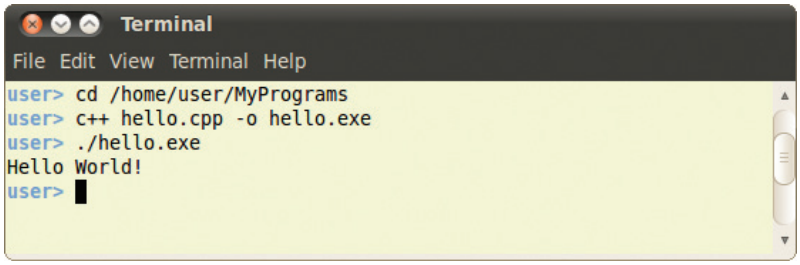
The command **c++** is an alias for the GNU C++ compiler – the command **g++** can also be used.





5 To run the generated executable program file in Windows simply enter the file name at the prompt in the "MyPrograms" directory – optionally the file extension may be omitted. In Linux the full file name must by used, preceded by a ./ dot-slash – as Linux does not look in the current directory unless it is explicitly directed to do so:





All command line examples in this book have been compiled and tested with the GNU C++ compiler (4.5.2) and with the Microsoft C++ compiler from Visual C++ 10.0 – they may not replicate exactly with other compilers.

Creating variables

A "variable" is like a container in a C++ program in which a data value can be stored inside the computer's memory. The stored value can be referenced using the variable's name.

The programmer can choose any name for a variable providing it adheres to the C++ naming conventions – a chosen name may only contain letters, digits, and the underscore character, but cannot begin with a digit. Also the C++ keywords, listed on the inside cover of this book, must be avoided. It's good practice to choose meaningful names to make the code more comprehensible.

To create a new variable in a program it must be "declared", specifying the type of data it may contain and its chosen name. A variable declaration has this syntax:

data-type variable-name **;**

Multiple variables of the same data type can be created in a single declaration as a comma-separated list with this syntax:

data-type variable-name1 **,** ***variable-name2*** **,** ***variable-name3*** **;**

The five basic C++ data types are listed in the table below, together with a brief description and example content:

Data Type:	Description:	Example:
char	A single byte, capable of holding one character	**'A'**
int	An integer whole number	**100**
float	A floating-point number, correct to six decimal places	**0.123456**
double	A floating-point number, correct to ten decimal places	**0.0123456789**
bool	A boolean value of **true** or **false**, or numerically zero is false and any non-zero is true	**false** or **0** **true** or **1**

Variable declarations must appear before executable statements – so they will be available for reference within statements.

Names are case-sensitive in C++ – so variables named **VAR**, **Var**, and **var** are treated as three individual variables. Traditionally C++ variable names are lowercase and seldom begin with an underscore as some C++ libraries use that convention.

Character values of the **char** data type must always be enclosed between single quotes – not double quotes.

When a value is assigned to a variable it is said to have been "initialized". Optionally a variable may be initialized in its declaration. The value stored in any initialized variable can be displayed on standard output by the **cout** function, which was used on page 12 to display the "Hello World!" greeting.

vars.cpp

1. Start a new program by specifying the C++ library classes to include and a namespace prefix to use

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

2. Add a main function containing a final return statement

```
int main()
{
  // Program code goes here.
  return 0 ;
}
```

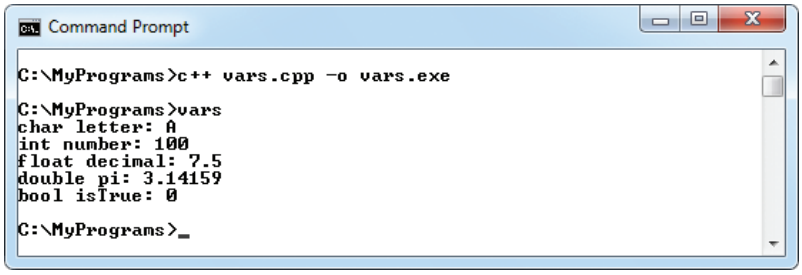
3. In the main function, insert statements to declare and initialize variables of various data types

```
char letter ;       letter = 'A' ;      // Declared then initialized.
int number ;        number = 100 ;      // Declared then initialized.
float decimal = 7.5 ;                   // Declared and initialized.
double pi = 3.14159 ;                   // Declared and initialized.
bool isTrue = false ;                   // Declared and initialized.
```

4. Now insert statements to output each stored value

```
cout << "char letter: "  << letter << endl ;
cout << "int number: " << number << endl ;
cout << "float decimal: " << decimal << endl ;
cout << "double pi: "   << pi << endl ;
cout << "bool isTrue: " << isTrue << endl ;
```

5. Save, compile, and run the program to see the output



Always begin boolean variable names with "is" so they are instantly recognizable as booleans. Also, use "lowerCamelCase" for all variable names that comprise multiple words – where all except the first word begin with uppercase, like "isTrue".

Employing variable arrays

An array is a variable that can store multiple items of data – unlike a regular array, which can only store one piece of data. The pieces of data are stored sequentially in array "elements" that are numbered, starting at zero. So the first value is stored in element zero, the second value is stored in element one, and so on.

Array numbering starts at zero – so the final element in an array of six elements is number five, not number six.

An array is declared in the same way as other variables but additionally the size of the array must also be specified in the declaration, in square brackets following the array name. For example, the syntax to declare an array named "nums" to store six integer numbers looks like this:

```
int nums[6] ;
```

Optionally an array can be initialized when it is declared by assigning values to each element as a comma-separated list enclosed by curly brackets (braces). For example:

```
int nums[6] = { 0, 1, 2, 3, 4, 5 } ;
```

An individual element can be referenced using the array name followed by square brackets containing the element number. This means that **nums[1]** references the second element in the example above – not the first element, as element numbering starts at zero.

Arrays can be created for any C++ data type, but each element may only contain data of the same data type. An array of characters can be used to store a string of text if the final element contains the special **\0** null character. For example:

```
char name[5] = { 'm', 'i', 'k', 'e', '\0' } ;
```

The entire string to be referenced just by the array name. This is the principle means of working with strings in the C language but the C++ string class, introduced in chapter four, is far simpler.

Collectively the elements of an array are known as an "index". Arrays can have more than one index – to represent multiple dimensions, rather than the single dimension of a regular array. Multi-dimensional arrays of three indices and more are uncommon, but two-dimensional arrays are useful to store grid-based information, such as coordinates. For example:

```
int coords[2] [3] = { { 1, 2, 3 } , { 4, 5, 6 } } ;
```

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6

1. Start a new program by specifying the C++ library classes to include and a namespace prefix to use

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

2. Add a main function containing a final return statement

```
int main()
{
  // Program code goes here.
  return 0 ;
}
```

3. In the main function, insert statements to declare and initialize three variable arrays

```
// Declared then initialized.
float nums[3] ;
nums[0] = 1.5 ; nums[1] = 2.75 ; nums[2] = 3.25 ;

// Declared and initialized.
char name[5] = { 'm', 'i', 'k', 'e', '\0' } ;
int coords[2] [3] = { { 1, 2, 3 } , { 4, 5, 6 } } ;
}
```

4. Now insert statements to output specific element values

```
cout << "nums[0]: " << nums[0] << endl ;
cout << "nums[1]: " << nums[1] << endl ;
cout << "nums[2]: " << nums[2] << endl ;
cout << "name[0]: " << name[0] << endl ;
cout << "Text string: " << name << endl ;
cout << "coords[0][2]: " << coords[0][2] << endl ;
cout << "coords[1][2]: " << coords[1][2] << endl ;
```

5. Save, compile, and run the program to see the output

```
Command Prompt

C:\MyPrograms>c++ arrays.cpp -o arrays.exe

C:\MyPrograms>arrays
nums[0]: 1.5
nums[1]: 2.75
nums[2]: 3.25
name[0]: m
Text string: mike
coords[0][2]: 3
coords[1][2]: 6

C:\MyPrograms>_
```

arrays.cpp

Beware

Where possible variable names should not be abbreviations – abbreviated names are only used in this book's examples due to space limitations.

Hot tip

The loop structures, introduced in chapter three, are often used to iterate array elements.

Employing vector arrays

A vector is an alternative to a regular array and has the advantage that its size can be changed as the program requires. Like regular arrays, vectors can be created for any data type and their elements are also numbered starting at zero.

In order to use vectors in a program the C++ **vector** library must be added with an **#include <vector>** preprocessor directive at the start of the program. This library contains the predefined functions in the table below, which are used to work with vectors:

Function:	Description:
at(*number* **)**	Gets the value contained in the specified element number
back()	Gets the value in the final element
clear()	Removes all vector elements
empty()	Returns true (1) if the vector is empty, or returns false (0) otherwise
front()	Gets the value in the first element
pop_back()	Removes the final element
push_back(*value* **)**	Adds a final element to the end of the vector, containing the specified value
size()	Gets the number of elements

Individual vector elements can be referenced using square brackets as with regular arrays, such as **vec[3]**.

A declaration to create a vector looks like this:

vector < *data-type* > *vector-name* (*size*) ;

An **int** vector will, by default have each element automatically initialized with a zero value. Optionally a different initial value can be specified after the size in the declaration with this syntax:

vector < *data-type* > *vector-name* (*size* , *initial-value*) ;

The functions to work with vectors are simply appended to the chosen vector name by the dot operator. For example, to get the size of a vector named "vec" you would use **vec.size()** .